VALIDATION OF
FUNCTIONAL BEHAVIOR SPECIFICATIONS OF
DISTRIBUTED OBJECT FRAMEWORKS

Thèse N° 1742

PRÉSENTÉ À LA SECTION DE SYSTÈME DE COMMUNICATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ES SCIENCES

PAR
DOMINIQUE SIDOU
Titulaire du Diplôme d'Études Approfondies (DEA) de l'Université Paul Sabatier de Toulouse.
de nationalité Française

Composition du Jury :
Prof. Jacques Labetoulle, rapporteur
Prof. Christian Bonnet, corapporteur
Ing. ETH. Rolf Eberhardt, corapporteur
Prof. Jean-Pierre Hubaux, corapporteur
Prof. Henri Nussbaumer, corapporteur

Sophia Antipolis, Eurécom 1997

Validation of Functional Behavior Specifications of Distributed Object Frameworks

Dominique Sidou

Submitted to the Section of Communication Systems.
Swiss Federal Institute of Technology (EPFL),
for the degree of docteur ès sciences.

Abstract

Modeling in the context of distributed object frameworks is a difficult problem because it requires the integration of three research areas : (1) object oriented modeling, (2) formalization, and (3) distributed processing. The open distributed processing (ODP) reference model defines the theoretical framework to realize this integration. This thesis proposes an original approach to this problem through a pragmatic instantiation of this integration in the perspective of ODP.

By deliberately limiting the scope of the work to functional models, and to information and computational viewpoint issues of the ODP reference model, it is shown how a very abstract and expressive behavior specification framework can be devised. The specification framework is based on a declarative specification of actions. Actions in the system are declared with respect to all the functionalities to be modeled. The behavior specification template is a variant of the well known event-condition-action (ECA) rule model, extended by the specification of safety properties or assertions, that are used during validation. With a declarative approach control issues in the system that typically overspecify a functional model can be abstracted away to a maximal extent. In addition, the specification framework makes possible the integration of any high level aggregation structures typically proposed by most of the object oriented analysis and design methods. In particular, roles and relationships are used to provide very expressive modeling. Note well that the specification framework is generic in the sense that it does not depend on any particular distributed computing notation. In particular both OSI systems management (GDMO, ASN.1) and CORBA (IDL) notations can be integrated. Each element specified in such notations is integrated according to the role played with respect to either the information or the computational ODP viewpoint.

Validation is based on execution, therefore an important part of the thesis consists of defining how the declarative specification of actions model can be made executable. To this end a precise operational semantics is defined and implemented by an algorithm called the behavior propagation engine (BPE) algorithm. Since the proposed declarative specification framework follows the very general ECA-rule model, the execution semantics can interestingly be based on principles already stated in the context of active database management systems (ADBMS). The result is a transition system, defined by a set of transition functions that exercise their processing on a user-level control structure called the behavior execution tree (BET). The BET gives a complete representation of the control state at each execution step. The BET can be visualized and be used for user interaction and debugging purposes. In addition, execution backtracking is a powerful problem analysis facility that has been incorporated in the interactive execution environment. This facility is supported by undo functions and the use of continuations.

Finally, the validation environment incorporates reduced state space exploration algorithms based on a combination of partial order and state caching techniques. Jointly, interactive execution and state space exploration are very complementary validation tools. State space exploration is typically used to detect problems in a model, and interactive execution performs very well for problem analysis.

The concrete result of this thesis is a powerful tool-set that is in addition expected to be usable by distributed object computing systems engineers in their daily work in specification. A tool allows specifiers to get immediate and concrete feedback about their work. In addition, engineers are forced to get a detailed knowledge of the models considered. Though a significant amount of work may be needed to get a model complete, the result is a better understanding of models. Finally, the corresponding specification can be implemented faster. The tool-set was developed in the context of the TIMS project. TIMS stands for TMN-based Information Model Simulator. It is a joint project between “Institut Eurécom” and “Swisscom”. In the thesis references are made about the application of the tool-set in the context of the TIMS project.

Validation of Functional Behavior Specifications of Distributed Object Frameworks

Dominique Sidou

Thèse présentée à la Section de Système de Communication.
École Polytechnique Fédérale de Lausanne (EPFL),
pour le titre de docteur ès sciences.

Résumé

La modélisation des composants basés objects distribués est un problème difficile parce qu'elle necessite l'intégration de trois domaines de recherches : (1) la modelisation orientée objet, (2) la formalisation et (3) le traitement distribué. Le modèle de reference du traitement distribué ouvert (ODP) définit le cadre théorique pour la réalisation de cette intégration. Cette thèse propose une approche originale à ce problème à travers une instantiation pragmatique de cette intégration dans la perspective de l'ODP.

En se limitant de façon déliberée à des modèles fonctionels, et aux points de vue information et traitement du modèle de reference ODP, il est montré comment un cadre très abstrait et expressif pour la spécification de comportement peut être obtenu. Le cadre de spécification de comportements se base sur une spécification déclarative d'actions. Les actions du système sont déclarées suivant les fonctionalités à modéliser dans un système donné. Le gabarit de spécification de comportement est une variante des regles evénement-condition-action (ECA), étendu par la spécification de propriétés de sûreté ou assertions qui sont utilisées durant la validation. L'aproche déclarative permet de faire abstraction des aspects contrôle dans le système qui typiquement sur-spécifient un modèle fonctionel. De plus, le cadre de spécification permet l'intégration des structures d'aggrégation de haut niveau proposées par la plupart des méthodes de d'analyse et de conception orientées objet. En particulier, les rôles et les relations sont utilisées pour fournir une modélisation très expressive. Il faut noter que le cadre de spécification est générique dans le sens qu'il ne dépend pas d'une notation particulière pour le traitement distribué. En particulier, les notations pour la gestion de système OSI (GDMO, ASN.1), ainsi que CORBA-IDL peuvent être intégrées. Chaque élément spécifié dans de telles notations est intégré suivant le role qu'il joue par rapport soit au point de vue de traitement soit au point de vue information du modele ODP.

La validation s'effectue par exécution, c'est pourquoi une part importante de cette thèse définit comment le modèle de spécification déclaratif d'actions peut devinir exécutable. Dans ce but, une sémantique opérationelle précise est définie par un algorithme, appelé le moteur de propagations de comportements (BPE). Comme le cadre de spécification déclaratif proposé suit le modèle général des règles ECA, la sémantique d'exécution peut de façon très intéressante s'appuyer sur les principes déjà établis dans le contexte des bases de données actives (ADBMS). Le résultat est un système de transitions, défini par un ensemble de fonctions de transitions qui s'appuient sur une structure de contrôle au niveau utilisateur appelée l'arbre d'exécution de comportement (BET). Le BET donne une représentation complète de l'état du contrôle à chaque pas d'ecécution. Le BET peut être visualisé et être utilisé pour intéragir avec l'utilisateur pour le déverminage. De plus, le retour arrière d'execution, une facilité très puissante pour l'analyse de problèmes, a été incorporé à l'environement d'exécution intéractif. Cette facilité est supportée par l'utilisation de fonctions "undo" et de continuations.

Finalement, l'environement de validation est doté d'algorithmes d'exploration de l'espace d'état reduits, basés sur une combinaison de techniques d'ordres partiels et de cache d'états. Conjointement, l'exécution intéractive et l'exploration de l'espace d'états sont très complémentaires. L'exploration de l'espace d'états peut être utilisé pour détecter les problèmes dans un modèle, et l'exécution intéractive est très performante pour l'analyse des problèmes.

Le résultat concret de cette thèse est un outil puissant qui de plus vise à être utilisé par des ingénieurs telecom dans leur travail journalier relatif à la spécification. Un outils permet aux spécifieurs d'obtenir un retour immédiat et concret sur leur travail. De plus, les ingénieurs sont contraints à se donner une connaisance approfondie des modèles traités. Bien qu'une quantité significative de travail soit nécessaire pour obtenir un modèle complet, le résultat est une compréhension bien meilleure des modèles. Finalement, les spécifications correspondantes peuvent être implantées plus rapidement. L'outil lui même a été développé dans le contexte du projet TIMS. TIMS signifie "TMN-based Information Model Simulator". C'est un projet conjoint entre "l'Institut Eurécom" et "Swisscom". Dans la thèse, les références sont données sur l'utilisation de l'outil dans le cadre du projet TIMS.

Acknowledgments

First of all I would like to thank Jacques Labetoulle, who gave me the opportunity (i) to lead the TIMS project at Institut Eurécom, and (ii) to allow me to do a PhD Thesis in the context of this project.

I am grateful to Rolf Eberhardt and Swisscom because this project would not have even existed without the original idea and motivation of Rolf Eberhardt and the funding from Swisscom. Rolf gave us interesting problems and let us find suitable solutions, in the end that is probably where the success of the TIMS project resides. In addition, I am also grateful to Marco Randini who has worked in Swisscom on case studies using the TIMS tool-set. This has resulted in very useful feedback.

I will always be indebted to Prof. Henri Nussbaumer who was head of EPFL-LIT laboratory. During my stay as an assistant from 1990 to 1993 in Lausanne, me an others add the opportunity to study the field of networking and distributed computing in general, in an excellent atmosphere.

I am grateful to Prof. Claude Petitpierre who is head of the EPFL-LTI laboratory to have accepted to be the president of the jury; I am also grateful to the other members of the jury, Christian Bonnet and Jean-Pierre Hubaux.

It has been a great pleasure for me to work closely with Sandro Mazziotta during this last three years. Sandro is an excellent co-worker and more importantly a friend.

I am grateful to Olivier Festor for the help he gave us in the TIMS project during its stay in Institut Eurécom in 1995, and for the careful reading he spent of my thesis document.

I am grateful to Kateel Vijayananda for his careful reading of early drafts of my thesis document.

I am grateful to Guy Genilloud for the very important discussions we add in the field of ODP where his expertise is very important.

Thanks to Remy Giaconne who is charge of student affairs at Institut Eurécom, who has been very helpful in all the administrative work around the thesis.

Finally, this thesis would not have been possible without the moral support of my wife Farzy and my daughter Guilda, for sure, this thesis was more difficult for them.

Contents

1 Introduction **1**
- 1.1 Issues . . . 2
  - 1.1.1 Distributed Object Frameworks . . . 2
  - 1.1.2 Formal and Object Oriented Modeling . . . 3
  - 1.1.3 Pragmatism w.r.t. Existing Notations Used in Distributed Computing . . . 3
  - 1.1.4 Generic Behavior Modeling Approach and ODP . . . 3
  - 1.1.5 Pragmatism w.r.t. Validation Approach . . . 3
  - 1.1.6 TIMS Project with Swisscom . . . 4
- 1.2 Plan of the Thesis . . . 4
  - 1.2.1 Behavior Modeling Issues . . . 4
  - 1.2.2 The Behavior Model, Static and Dynamic Semantics, ODP Mapping . . . 5
  - 1.2.3 Precise Execution Semantics (BPE Algorithm) . . . 6
  - 1.2.4 High Level and Powerful Execution Support . . . 6
  - 1.2.5 Improved Validation . . . 6
- 1.3 Miscellaneous Points . . . 8
  - 1.3.1 Reader's Background . . . 8
  - 1.3.2 Language Mapping in *Scheme* . . . 8
- 1.4 Other Relevant Publications . . . 8

2 Behavior Modeling Issues **9**
- 2.1 Introduction . . . 9
- 2.2 Distributed Object Computing . . . 10
  - 2.2.1 Distribution Technologies TMN, CORBA, SNMP-SMI, Java-RMI, etc . . . 10
  - 2.2.2 The Role of ODP . . . 11
  - 2.2.3 Access Transparency . . . 11
  - 2.2.4 Importance of Semantics, Behaviors Issues . . . 13
  - 2.2.5 Higher Level Services, CORBA Object Services, OSI-SM Systems Management Functions . . . 13
  - 2.2.6 Distributed Object Frameworks . . . 13
- 2.3 Scope of Behavior Issues (Functional vs. non-Functional Models) . . . 16
- 2.4 Usefulness of ODP . . . 16
  - 2.4.1 ODP Concepts for Behavior Modeling . . . 16
  - 2.4.2 ODP Viewpoints . . . 18
  - 2.4.3 Scope of Behavior Issues w.r.t. ODP Viewpoints . . . 21
- 2.5 Powerful Information Modeling Abstractions . . . 22
  - 2.5.1 Change Management . . . 23
  - 2.5.2 High Level Aggregation Structures . . . 26
  - 2.5.3 Losing High Level Aggregation Structures in Existing OO Methods . . . 26

2.5.4 Keeping High Level Aggregation Structures 27
2.6 Role / Relationship-based Behavior Formalization 27
2.6.1 Separation of Concern 28
2.6.2 Polymorphism and Dynamic Subtyping 28
2.6.3 Configuration of Objects 28
2.6.4 Relationships 29
2.6.5 Our Approach 29
2.7 Specification and Validation 30
2.7.1 Correctness and Validation 30
2.7.2 Specification-based validation 31
2.7.3 Executable vs. non-Executable Specifications 33
2.8 Specifications for Distributed and Reactive Systems 35
2.8.1 Nondeterminism 35
2.8.2 Transition Systems 36
2.8.3 Using Control Abstractions (Process Oriented Model – POM) 36
2.8.4 Declarative Specification of Actions (Data Oriented Model – DOM) 36
2.8.5 Some Existing Specification Approaches 37
2.8.6 Reasoning Problems in the Context of Reactive Systems 40
2.9 Conclusion 41

3 Behavior Model Instantiated : Static and Dynamic Semantics, ODP Issues 43
3.1 Introduction 43
3.2 Behavior of Actions and the ECA-rule Model 44
3.3 Notation for the Behavior Language 45
3.4 Necessity for a Precise Execution Semantics 46
3.5 Triggering Event Messages and Behavior 46
3.5.1 Principle 46
3.5.2 Event Message Levels with respect to ODP, IVP & CVP 47
3.5.3 Generic Set of IVP Messages Used 48
3.5.4 Behaviors on IVP Messages to Check for GRM Conformance 48
3.5.5 CVP Messages, Behaviors and IVP Mapping 49
3.6 Rule Processing Semantics 52
3.6.1 *recognize-act* Cycle Algorithm (Weak Event / Rule Execution Coupling) . . 52
3.6.2 Conflict Resolution Approaches 53
3.6.3 Towards Stronger Event / Rule-Execution Coupling 53
3.7 Behaviors and Reaction Semantics 53
3.7.1 The **is-trigger** Reaction Semantics 54
3.7.2 2-Phase Fetching 54
3.7.3 Coupling Mode 54
3.7.4 The **exec-rules** Clause 55
3.8 Behavior Fetching / Dispatching (Scoping and Filtering) 56
3.8.1 Introduction 56
3.8.2 Execution Semantics of Behavior Fetching 56
3.8.3 Behavior Scoping Functions (BSF) and Behavior Execution Contexts (BEC) 57
3.8.4 Typical BEC for Role / Relationship-based Behavior Formalization (RBF) . . 57
3.8.5 Advantages of Behavior Scoping 58
3.8.6 Behavior Scoping Illustrated 59
3.8.7 Behavior Filtering 60
3.8.8 Behavior Filtering Illustrated 60

3.8.9 Whole Behavior Fetching Algorithm 62
3.9 Conclusion 64

4 Behavior Propagation Engine (BPE) 67
4.1 Introduction 67
4.2 Behavior Propagation – Forward Search 67
4.3 Atomic Transitions / Execution Steps 68
4.4 States / Configurations of the System 68
4.5 Behavior Execution Node 69
4.6 Trigger Execution Control 71
4.7 Transition Functions 71
4.7.1 BEN:STEP-CHECK-PRE 72
4.7.2 BEN:STEP-BODY 73
4.7.3 BEN:STEP-CHECK-POST 77
4.8 BPE Algorithm 78
4.8.1 Bootstrapping a Behavior Propagation 79
4.8.2 Function BET:WALK 79
4.8.3 BEN:EXEC 80
4.9 Atomic Execution Support 80
4.9.1 Principle 80
4.9.2 Syntax 81
4.9.3 Implementation Support 81
4.10 Conclusion 82

5 Execution and Debugging Environment 83
5.1 Introduction 83
5.2 System Visualization 83
5.2.1 Object View 84
5.2.2 Execution View 85
5.2.3 Scheme Shell Text View 86
5.2.4 Whole Execution Environment 86
5.3 Dynamic Behavior Analysis 88
5.3.1 Principle 88
5.3.2 Execution / Walk Modes 88
5.4 Debugging Support 90
5.4.1 Basic Debugging Support 90
5.4.2 Improved Debugging with Backtracking 91
5.5 User Interface 94
5.6 Discussion 96
5.7 Conclusion 97

6 Improving Validation with eXhaustive Dynamic Behavior Analysis 99
6.1 Introduction 99
6.1.1 Classical State-full Search 100
6.1.2 State Representations 101
6.2 BPE State-less Search / Full Walk Algorithm 102
6.2.1 Principle 102
6.2.2 Inefficiency of a State Less Search 103
6.3 Operational Semantics and Labeled Transition Systems (LTS) 103

6.4 Traces and Partial Order Methods 104
6.4.1 Principle for an Efficient Detection of Termination States 106
6.4.2 Principle for an Efficient Detection of Assertion Violations 106
6.5 Independence Relation on Transitions 106
6.5.1 D Instantiated on (IR, BET) Configurations 107
6.5.2 D_{op} : the Dependency Relation on Basic IR Operations 107
6.5.3 Refining D_{op} 108
6.5.4 Introducing More Dependency to Keep more Behavior 109
6.6 Selective Search / Exploration 109
6.6.1 Selective Search / Exploration Principle 109
6.6.2 Selective Search / Exploration Techniques 109
6.6.3 Principle of Selective Search based on Sleep-sets 110
6.6.4 Properties of Sleep-sets 112
6.7 An Efficient State-less Search Algorithm 113
6.8 State Caching Combined with Sleep-Sets 113
6.8.1 Cache Representation 114
6.8.2 State Caching + Sleep-Sets Full Walk Algorithm 115
6.9 Using eXhaustive Dynamic Behavior Analysis 116
6.9.1 Detecting Confluence / Divergence 119
6.10 Discussion 121
6.10.1 Partial Order vs. Proof Methods 121
6.10.2 Partial Order and Incomplete Checking 121
6.10.3 Partial Order and Too Big / Infinite State Spaces 121
6.10.4 Closed Systems under Scenarios Steps 121
6.11 Conclusion 122

7 Conclusion **125**
7.1 Summary and Contributions 125
7.2 Related Work (ITU-SG 4 [G851 0196]) 126
7.3 Further Issues 127

A Behavior Language (BL) **137**
A.1 Introduction 137
A.1.1 Behavior Specification Files : 137
A.1.2 Some Typographical Conventions : 137
A.2 The Generic Part of the Behavior Language 137
A.2.1 **`define-behavior`** syntax 138
A.2.2 Behavior Label 138
A.2.3 Enabling Condition (guard) 138
A.2.4 **`exec-rules`** syntax 139
A.2.5 **`pre`** 142
A.2.6 **`body`** 142
A.2.7 **`post`** 145
A.2.8 **`bec`** 145
A.2.9 Sending Messages to the BPE 149
A.2.10 Atomic Execution Syntax 151
A.2.11 Message Buffering Syntax 152

B Open Distributed Processing (ODP) 153
B.1 Introduction . . . 153
B.2 Basic Object and Behavior Concepts . . . 154
B.3 Viewpoints . . . 154
B.3.1 Viewpoint Languages and Notations . . . 155
B.3.2 Enterprise Viewpoint Language . . . 155
B.3.3 Information Viewpoint Language . . . 155
B.3.4 Computational Viewpoint Language . . . 155
B.3.5 Engineering Viewpoint Language . . . 155
B.3.6 Technology Viewpoint Language . . . 156
B.4 Other Architectural Concepts . . . 156
B.4.1 Distribution Transparencies . . . 156
B.4.2 ODP Functions . . . 157

C Spanning Tree Case Study 159

D Continuations in *Scheme* 161
D.1 Implementation Perspective . . . 161
D.1.1 Tail Recursion . . . 161
D.1.2 The Continuation Chain . . . 162
D.1.3 Capturing continuations (call/cc) . . . 163
D.1.4 Syntax of call/cc . . . 163
D.2 Utilization Perspective . . . 163

E Acronyms 165

F Summary of Case Studies performed in the TIMS Project 169
F.1 TIMS Simple Case Study . . . 169
F.2 SDH Multiplex Section Protection (MSP) . . . 169
F.3 V5 Interface Management . . . 170
F.4 Metropolitan European TRAnsport Network (METRAN) . . . 170

G Summary of the V5.1 Case Study and Ensemble 173
G.1 Introduction . . . 173
G.2 Scope of the Ensemble . . . 173
G.3 Management Context . . . 174
G.3.1 Functional architecture . . . 174
G.3.2 Transport bearer service, resource and management requirements . . . 176
G.3.3 Functional TMN Architecture and Management Domain . . . 176
G.4 Models . . . 182
G.4.1 Resource model . . . 182
G.4.2 Topological model . . . 184
G.5 Functions . . . 185
G.5.1 Resource Configuration & Provisioning Management (a2, a4) . . . 185
G.5.2 Service Configuration & Provisioning Management (a3) . . . 186
G.5.3 Configuration & Provisioning at the resource level (a1) . . . 187
G.6 Management Information Model . . . 188
G.6.1 Overview of the Management Information Model . . . 188
G.6.2 Service fragment . . . 190
G.6.3 Topology fragment . . . 190

G.6.4 Equipment & cabling fragment . . . 191
G.6.5 Transport access network fragment . . . 191
G.6.6 Digital section fragment . . . 191
G.6.7 V5-based transport bearer service fragment . . . 193
G.6.8 Narrow-band leased line transport bearer service fragment . . . 194
G.7 Example of MIBs and Fragments . . . 194
G.8 Scenarios . . . 195
G.8.1 InitialMIB . . . 197
G.8.2 InsertEquipment-Pstn . . . 198
G.8.3 InsertV5Interface . . . 201
G.8.4 Establish-Connection . . . 203
G.8.5 De-Establish-Connection . . . 207
G.9 Conclusion . . . 211

List of Figures

1.1 Distributed Object Frameworks, Positioned Between Basic Object Libraries and Distributed Applications. . . . 2

2.1 OMA Object Framework . . . 14
2.2 Strategy Problem. . . . 24
2.3 Strategy Pattern. . . . 24

3.1 IVP and CVP Messages Levels. . . . 47
3.2 IVP Messages. . . . 48
3.3 LCCapacity Relationship in GRM. . . . 49
3.4 LCCapacity Relationship Behaviors Checking for GRM Conformance. . . . 50
3.5 Behavior Checking for LC Role Binding Support. . . . 50
3.6 Trigger Reaction Phases. . . . 55
3.7 Behavior Repository Indexed with Scopes . . . 59
3.8 Simple Instance Repository. . . . 60
3.9 Scoping Function **`immediate / phase-i`**. . . . 61
3.10 Behavior Fetching Process. . . . 63

4.1 Behavior Execution Node FSM. . . . 70
4.2 Atomic Executions. . . . 81

5.1 Object View. . . . 84
5.2 Directed Graph 1 for the Spanning Tree. . . . 85
5.3 Trace of an Optimal Spanning Tree Execution. . . . 86
5.4 BET for an Optimal Spanning Tree Execution. . . . 87
5.5 Trace of a Non-Optimal Spaning Tree Execution. . . . 87
5.6 BET of a Non-Optimal Spanning Tree Execution. . . . 88
5.7 TIMS Session. . . . 89
5.8 BEN Record. . . . 91
5.9 User Interface (Execution Control Panel). . . . 94

6.1 Independence of Transitions. . . . 105
6.2 Sleep Set Principle. . . . 111
6.3 Directed Graph 2 for the Spanning Tree. . . . 117

6.4 Directed Graph 3 for the Spanning Tree. 118
6.5 Divergence Detection in the Spanning Tree Executions (for graph 6.3). 120

A.1 A Behavior with a **`when`**, **`pre`** and **`post`**. 140
A.2 Trigger Reaction Phases. 141
A.3 Simple Instance Repository. 142
A.4 Behavior for Testing **`exec-rules`**. 143
A.5 **`exec-rules`** Test. 144
A.6 A Behavior with a **`body`**. 145
A.7 Behavior for Testing BEC Fields Accessors. 147
A.8 BEC Fields Accessors Test. 148
A.9 Behavior for Testing **`msgsnd`** Result. 150
A.10 **`msgsnd`** Result Test. 151

D.1 Continuation Chain. 162

G.1 Access Network boundaries ([G.902 fig 1]) 175
G.2 Example of functional architecture of an AN ([G.902/fig.3]) 175
G.3 Interworking management domain and reference points covered in this model 177
G.4 TMN Architecture for the Access Network with the example of isdnBA 179
G.5 Interaction Models for co-ordination (both service-id and resource-id based) 180
G.6 Example process for connectivity service provisioning 181
G.7 Resources relevant for the Connectivity service provisioning Ensemble 183
G.8 Sample mapping of the physical/logical representation of a FITL access network onto the topological model 185
G.9 Information Model overview 189
G.10 Topological fragment 190
G.11 Equipment fragment 191
G.12 Transport access network fragment 192
G.13 Connection Points from the Service Node to the Access Network 192
G.14 Digital section relationships 193
G.15 V5 transport bearer service fragment 193
G.16 Leased Line transmission bearer service fragment 194

List of Tables

2.1 Examples of Design Patterns and Intended Variations. 23

5.1 Mapping of Execution Control Panel Buttons to *Scheme*. 95

6.1 Full Walks on Spanning Trees. 119

G.1 Examples of SPF and UPF ([G.902/p.11]) 175
G.3 General format of layer 1 for the designation of international routes 184

List of Miscelaneous Things

Behavior 3.5.1 `moind-set` agent behavior 51
Algorithm 3.6.1 *recognize-act* cycle 52

Behavior 3.7.1 `ivpmsg-set` **`is-trigger`** reaction semantics 54
Algorithm 3.8.1 Behavior Fetching 63
Algorithm 4.2.1 Forward Search Inference Engine 68
Algorithm 4.7.1 BEN step check pre 72
Algorithm 4.7.2 BEN step body 73
Algorithm 4.7.3 BPE Message Sending 75
Algorithm 4.7.4 BEN Gather Triggers Results 75
Algorithm 4.7.5 BEN build trigger execution controls 76
Algorithm 4.7.6 TEC build children 76
Algorithm 4.7.7 TEC next phase switching 77
Algorithm 4.7.8 BEN check post-condition 78
Algorithm 4.8.1 Behavior Propagation Bootstrapping 79
Algorithm 4.8.2 Behavior Propagation Engine 80
Algorithm 4.8.3 BEN Execution 80
Algorithm 6.1.1 Classical State-full Search (Full Walk) 100
Algorithm 6.2.1 State-less Search / Full Walk 102
Definition 6.3.1 Labeled Transition System (LTS) 104
Definition 6.4.1 Concurrent alphabet of transitions 104
Definition 6.4.2 Independence of transitions 105
Definition 6.4.3 Trace 105
Definition 6.4.4 Partial Order 105
Definition 6.4.5 Linearization of a partial order 105
Theorem 6.4.1 Linerizations and partial order 105
Theorem 6.4.2 Traces and state reachability 106
Algorithm 6.7.1 State-less Sleep Set Search / Full Walk Algorithm 113
Algorithm 6.8.1 State Caching combined with Sleep Set Full Walk Algorithm . . . 116
Behavior A.2.1 Step of the Spanning Tree 151
Behavior C.0.1 The Simple Spanning Tree Improvement Step Behavior 159

Chapter 1

Introduction

Distributed Processing Environments (DPE) have become an available distribution technology to support current and future distributed applications. Satisfactory solutions exist for communication oriented aspects based on protocols, services, and on top Interface Definition Languages (IDL) (e.g. RPC systems, CORBA systems, OSI systems management, SNMP framework).

Thanks to existing DPEs and IDLs a satisfactory level is achieved in terms of communication interoperability. This guarantees e.g. that a server is able to receive and decode a request sent by a client. However, this is far from being sufficient to ensure full interoperability. One problem is that people from the networking communities who designed interface definition languages are primarily concerned with communication aspects. Full interoperability requires also state and behavior / semantics aspects to be taken into account.

With GDMO / ASN.1 the OSI-SM community has tried to fill this gap. GDMO / ASN.1 provide some description of the state of managed objects. On the other hand behavior is poorly considered using unstructured prose. For most of the IDLs there is no support at all associated to the specification of state and behavior. The IDL is limited to the specification of interfaces. A typical example is CORBA-IDL (note that this is a deliberate choice). The lack of specification for state and behavior inevitably results in ambiguous and incomplete specifications and in inter-working problems between implementations of clients and servers involved in distributed applications. There is no precise specification of how the server should understand a request coming from a client.

The community of people the more concerned with behavior and semantics is the community of formalization and formal description techniques (FDTs). Naturally it is very important to take into account FDTs and results from this community.

Another important aspect is software engineering. In effect, distributed applications are complex softwares that require powerful methodologies to support the development life-cycle. In this context object oriented modeling techniques play also an important role.

Therefore good result are likely to be obtained to model distributed applications only if both object oriented modeling, formalization and distributed processing are considered altogether in an integrated way.

Fortunately this problem has already been recognized and considered at least on the paper by ISO and ITU-T in the context of the open distributed processing (ODP). ODP provides mainly a conceptual framework where all the concepts for object orientation, behavior and distribution are defined in a consistent way. An important concept identified in ODP to model distributed applications in the concept of viewpoints. ODP viewpoints allow one to partition the modeling activity. Thus the scope of each part is clearly determined, specifications becomes more manageable and can be written, in each part, using the more adapted modeling abstraction(s).

The objective of the thesis is to propose a suitable behavior modeling framework that, in the per-

spective of the ODP, integrates both formality, object orientation and distributed processing. Note that in our mind, modeling covers both specification and validation issues. The first chapter of the thesis is devoted to the analysis of behavior modeling in general, and one important result is that it is difficult to find a behavior modeling framework that covers all issues. In particular it is difficult to consider together functional[1] and non-functional issues. At least it is shown that limiting the scope to functional issues is interesting because in this context a behavior modeling approach that is both abstract and expressive can be obtained.

It is important to note that the scope of the thesis is limited to modeling issues (in particular how to obtain model as much correct as possible). As a consequence, issues related to the design, the implementation, the generation of code are beyond our concerns. Similarly, issues related to the correct refinement of specifications are also beyond the scope of this thesis.

1.1 Issues

1.1.1 Distributed Object Frameworks

One important point is to find the more suitable place where behavior modeling can be done. The level of interface specification or information objects (as they exist in GDMO) is not suitable because behavior tend typically to depend on application issues, and behavior is typically concerned with the context of utilization of the involved interfaces and information objects. On the other side, it is not desirable to specify behavior at the level of the distributed application itself because too much details are typically present at this level. Therefore, as shown in figure 1.1, some intermediate level between object libraries and distributed applications is needed to facilitate modeling.

Distributed Applications	Built from DOFs
Distributed Object Framework (DOF) NMF-Ensembles OMA Object Frameworks	Functions (e.g. path mgmt) - IO+ - CO+ - Behavior / Semantics - Use cases - Test cases
Computational + Information Object Libraries (basic objects)	GDMO, ASN.1 GRM CORBA-IDL UML Java SNMP-SMI etc

IO+, CO+ are DOF level objects extending basic objects through inheritance and aggregation.

Figure 1.1: Distributed Object Frameworks, Positioned Between Basic Object Libraries and Distributed Applications.

In fact, this level has already been identified by standardization bodies in the form of NMF-

[1]Functional is not considered in the sense of model of computation as it is the case in the context of functional programming languages. Here, functional is used in the more common sense of functionality (exercised in the system). This covers mostly all kind of state transitions that can be observed in the behavior of the system. On the other side, non-functional behavior issues cover the quantitative aspects such as timing, real-time and performance issues.

Ensembles and OMA Object Frameworks. In this thesis, the term of *distributed object framework* (DOF) is used to represent such components. The various forms of DOFs are the suitable units of modeling for distributed application. At the level of an object framework not only communication issues **but also** behavior issues can be captured.

1.1.2 Formal and Object Oriented Modeling

Distributed application are typically data intensive applications that thus require strong OO modeling facilities. This may include the use of powerful data modeling abstractions, such as roles and relationships. Classical object oriented modeling techniques (e.g. OMT, Booch) typically propose such facilities, but do not perform as well w.r.t. rigorous behavior modeling. One approach to reach formality is to use a FDT with OO modeling concepts embedded, e.g. an object oriented Z approach [Stepney et al.92], SDL'93 [Bartocci et al.93, Mazaher et al.93], etc. Another approach consists of starting from OO modeling and of adding formal behavior modeling. Note that an approach is considered to be formal as soon as a semantics is precisely defined. The ongoing work on the unified modeling language (UML [Partners97]) follows this approach. UML tries to propose a complete modeling approach covering both state, interfaces and behavior issues. Note that UML is pushed by the OMG, and naturally the goal in CORBA is to adopt UML to support state and behavior modeling. This approach is easier to understand and use by people involved in analysis and design of applications, and tool support is more likely to become available in this context.

1.1.3 Pragmatism w.r.t. Existing Notations Used in Distributed Computing

It is important to note that DOFs are based primarily on OO models typically written using existing distributed object computing (DOC) notations that specify object state and interfaces. Domain experts such as telecommunication engineers are familiar with such notations. A lot of libraries written with such notations already exist, they are typically reused each time a new DOF is considered. Pragmatism lead to allow one to be able to incorporate directly such models in the form they are given and thus to accommodate the universe of discourse of domain experts.

1.1.4 Generic Behavior Modeling Approach and ODP

The goal is not targeted to a particular distributed object computing (DOC) framework, i.e. trying to extend an existing DOC notation such as GDMO to include formal behavior specification is possible but is not advocated in this thesis because of limited applicability. In fact, in the context of the Telecommunication Management Network (TMN), future applications will typically use other DOC technologies than the OSI-SM (in particular CORBA). This has already be recognized by [Tinac94] and [G851 0196]. Therefore a generic approach is preferred. ODP concepts and in particular ODP viewpoints provide the basis to devise such a generic behavior modeling framework. In order to cope with existing models written using DOC notations a mapping has to be defined on the generic approach and thus with respect to ODP viewpoints.

It is important to note here that though the cases studies where our work was applied, concern mostly OSI-SM models, this is always done in a generic way, i.e. in the ODP perspective onto which any DOC technology can be mapped, and in particular the OSI-SM DOC technology.

1.1.5 Pragmatism w.r.t. Validation Approach

With respect to validation pragmatism typically leads to an approach based on executability and even system prototyping rather than formal reasoning. For executable specifications, formality requires a precise operational semantics to be defined that determines how behaviors are to be exercised.

1.1.6 TIMS Project with Swisscom

The TIMS (TMN-based Information Model Simulator) project has been launched by Rolf Eberhardt at Swisscom in 1994. The original idea was to improve the standardization process of TMN interfaces thanks to simulation and prototyping. Most of the concepts considered in this thesis have been devised and experimented in the TIMS project. From the beginning, this project was the opportunity to confront the behavior modeling ideas with actual case studies.

Four significant case studies were performed during the TIMS project. In appendix F, a summary of each case study is given. The more significant one has been developed in Swisscom by Rolf Eberhardt and Marco Randini. The case study has considered configuration and provisioning for the V5.1 management model. This case study has been developed to support the writing of an Ensemble document that is in the standardization process to become an ETSI standard [Etsi97]. Results on this case study have also been published in an internal Swisscom report [Eberhardt et al.97a]). The V5 case study and the Ensemble document are explained more in details in appendix G. This appendix gives some credit on the usefulness and effectiveness of the tool-set resulting from the approach, proposed in the thesis, for behavior modeling and validation. This proof of concept is given as an appendix just because the case study itself was not developed by the author of the thesis. However, the fact that the case study was performed by people external to the development of the tool-set gives in some way more credit with respect to the applicability of the tool-set.

The role of case studies has played a key role in the TIMS project, Very quickly the confrontation with actual case studies has lead us to solutions that “work” both in terms of behavior modeling and tool support. It has to be noted here that though the case studies performed in the context of the TIMS project are based on TMN information models given in GDMO / ASN.1, the approach advocated in this thesis for behavior modeling is not dedicated to TMN-systems. In fact the approach is generic by following the modeling concepts proposed by ODP and by mapping TMN modeling concepts on the generic ODP framework. An equivalent mapping for CORBA models has also been proposed in the context of the TIMS project.

1.2 Plan of the Thesis

1.2.1 Behavior Modeling Issues

Chapter 2 introduces the problem of behavior modeling for DOFs. Its objective is to determine what features are required to obtain a suitable behavior model. This concerns both specification and validation issues. This chapter introduces first the context of our work, i.e. distributed object computing (DOC) and it emphasizes the role played by the open distributed processing (ODP) reference model, in terms of conceptual framework that has been used to guide our work. The contribution of chapter 2 is to state and justify that a suitable behavior model for functional issues in the context of DOFs can be obtained by observing the following statements :

1. The behavior model should follow an approach based on data oriented modeling, and the declarative specification of actions. That is from an OO-model including powerful information modeling such as roles and relationships, (functional) behaviors can be specified in an abstract and expressive way. In addition a declarative approach facilitates evolution and allows incremental specification.

2. In the ODP perspective both information and computational ODP viewpoints specifications can be used in a complementary and optimal way in terms of the involved efforts. The information model defines the state (static schema) and transitions (dynamic schema) in the system. Whereas the computational model defines the interfaces of the objects in the system, and thus

the interfaces of the system with its environment. With both models a complete description of the system can be envisioned (at least for the purpose of functional modeling).

3. Between inspection and reasoning, executable specifications provide a satisfactory compromise, especially in the context of distributed and reactive systems where simulation is particularly appreciated.

A summary of these results have been published in [Sidou97b].

1.2.2 The Behavior Model, Static and Dynamic Semantics, ODP Mapping

The contribution of chapter 3 is to show how the expected features of the behavior model identified in chapter 2 can be concretely instantiated :

1. The static semantics corresponding to the declarative specification of actions is defined. The resulting behavior notation template includes both pure specification aspects (e.g. assertions) and execution / simulation aspects. However, each issue is given using well identified and separate behavior clauses.

2. The principles of the dynamic semantics are given in the form of an algorithm :. the behavior propagation engine (BPE) algorithm. The BPE is implemented as a message processor. This allows some kind of decoupling between events and behavior. The link can be realized in a flexible way using a separate behavior fetching process. One important contribution of chapter 3 is to define the principles of the dynamic execution semantics adapted to the declarative specification of actions framework that is adopted. The difficulty comes from the fact that usually executable specification languages for distributed and reactive systems are rather imperative, i.e. languages where the execution of actions is driven in a somewhat rigid way using control abstraction such as automata or processes. In the context of declarative languages, the research community has came up to significant results in terms of execution semantics concepts and principles. These results have been used to build rule production systems such as expert systems or active database management systems (ADBMS). Even if these applications are not oriented towards pure modeling activities, the fundamental principles and concepts used to define corresponding execution semantics have been found particularly useful. A contribution of this chapter is to show how well stated ADBMS principles can be incorporated to define the execution rules that control both the fetching and the execution of behaviors.

3. The integration of ODP computational and information viewpoint issues is instantiated. This is based on the definition of two message levels (CVP and IVP messages). Moreover, mappings are defined between computational viewpoint (CVP) messages and information viewpoint (IVP) messages. As a result behaviors in the CVP model typically reuses behaviors specified in the IVP model.

4. Another important contribution in chapter 3 is the behavior fetching function. It is divided into a behavior scoping and a behavior filtering phase. The behavior scoping phase is fully customizable by allowing the incorporation of behavior scoping functions obeying to a well defined interface. This flexibility provides a complete decoupling between event messages and behaviors. This can be used to define powerful dynamic behavior dispatching. In particular it is shown how dynamic behavior dispatching based on role and relationship abstractions (the RBF paradigm) is possible.

Part of the materials presented in chapter 3 have been published in [Sidou97a]. This addresses mostly the execution semantics based on ADBMS principles.

1.2.3 Precise Execution Semantics (BPE Algorithm)

The objective of chapter 4 is to give the precise definition of the execution semantics in its final form, that is the behavior propagation engine (BPE) algorithm. One important point is to use user level data structures to represent the processing state of the BPE. These user level data structures are based on the concept of behavior execution node (BEN). It follows that behavior executions appear as trees of BENs, which lead to the concept of behavior execution tree (BET). The BPE algorithm itself is presented as a set of transition functions exercising their effect on the BET. In addition, such transition functions define the atomic execution steps of the resulting transition system.

1.2.4 High Level and Powerful Execution Support

Validation is primarily user driven because it is performed using test cases that are given by the user as scenarios and executed thanks to the execution environment. Therefore the availability of a powerful execution environment is a key point for the validation environment. The first contribution of chapter 5 is to show how such a powerful execution environment can be provided based on the fact that the control state in the BPE is represented by user level data structures (the BENs and the BET) that allow the user to visualize directly anything occurring during a behavior execution. This procures a powerful high level execution support, i.e. a powerful execution support at the behavior level. On the other hand, the low level execution support is provided by a programming language implementation. The *Scheme* [Clinger et al.91] programming language has been selected, more details about *Scheme* are given in section 1.3.2.

The second contribution of chapter 5 is to show how powerful debugging facilities based on execution backtracking can be integrated into the behavior execution engine. These facilities are very useful because when an error occurs during the processing of a behavior propagation, they allow the user to analyze precisely the execution path that was followed, to go backwards (backtrack) and to analyze the system's state at any intermediate step. Then most of the time it is possible to interactively correct the error and replay forward the execution – e.g. step by step – and to check that the problem has been fixed. Execution backtracking is decomposed in data and control backtracking issues. Undo functions working on user level data structures support data backtracking and part of control backtracking. The remaining is dependent on control structures of the low level execution environment and is based on *Scheme* continuations. Part of the materials presented in this chapter have been published in [Sidou96].

1.2.5 Improved Validation

To obtain more formality in a framework whose semantics is operationally defined, an approach consists of using state space exploration techniques. In chapter 6 it is shown how reduced state space exploration techniques can be incorporated in the behavior execution environment. The contribution of chapter 6 is to show how this can be done with minimal effort. In fact this is done by reusing the existing behavior execution environment unchanged and by only adding some extensions. The important part of the behavior execution environment that is used to support the exhaustive exploration is naturally the execution backtracking facilities developed for debugging. In fact obtaining an exhaustive state space exploration algorithm from the existing execution environment is immediate. But this results in a naive, or brute force exhaustive state space exploration that is most of the time totally inefficient. Note that the state space of interest represents all the possible behavior propagations that can be observed from the system in a given state following a given solicitation, i.e. a step in a scenario. So we do not consider the infinite behavior of the system along all its life-cycle. This would be completely infeasible for systems such as distributed object frameworks. A pragmatic approach is thus to work based on scenarios, and to consider that the system is closed, i.e. that the model includes all that

is required about the system of interest itself and about its environment. Naturally all the properties checked are dependent on the fact that this hypothesis of work can be observed. This is completely under the user's control. However, even when working heuristically with scenarios and the closed system assumption, the state space resulting from an exhaustive exploration is still too large. To solve this problem, reduced state space exploration techniques can be used to explore only a part of the complete state space that is sufficient to check for some desired properties. One of the more common requirement is to allow to detect all termination states. This gives an important result about the whole behavior of the system w.r.t. properties such as convergence or divergence. In addition since the behavior notation template is an assertional behavior language, a very important point is to allow the detection of all the violations of assertions. Not all reduced state space exploration techniques can be incorporated directly (and above all with minimal effort) into the existing behavior execution environment. So one important part of chapter 6 is to identify which technique is the most adapted and then to show how it can be integrated. The main technique identified is the *sleep-set* method that is based on fundamental results coming from trace semantics and partial order methods [Godefroid95]. This technique is suitable because (i) it can be used to detect both deadlocks and assertion violations, and (ii) it can be incorporated with minimal effort because it is based on dynamic properties exercised by the system at execution time rather than techniques such as *persistent-sets* based on static properties that are to be deduced statically from the text of the specification itself.

State caching is also introduced in the reduced state space exploration algorithm because it is known as a very interesting complement to partial order methods such as the sleep-sets technique. State caching requires additional work to be incorporated. This work consists mostly of defining a cache representation of visited states during the search.

Naturally, performance and scalability obtained with the proposed exhaustive dynamic behavior analysis are not comparable to what can be achieved using dedicated verification tools such as CAESAR [Fernandez et al.96a], COSPAN [Hardin et al.96], CWB [Cleaveland93], SPIN [Holzmann91], etc. However, our point is that from user-driven dynamic behavior analysis the transition to exhaustive search is immediate. This allows to perform more intensive validation from the same specification as the one that was used in the simple user-driven dynamic behavior analysis phase. In addition, if a problem is detected in the model when performing exhaustive search, it can be analyzed using the powerful dynamic behavior analysis tools provided (e.g. improved debugging facilities present in the execution environment). So our approach is somewhat integrated and provides for, in a complementary way, a powerful execution environment and exhaustive search. Typically model validation can be performed as follows :

1. The powerful execution environment is used to quickly obtain a working model. The model is tested by the user / specifier based on the execution of some scenarios.

2. Then exhaustive search can be performed on this working model to detect more subtle errors along all execution paths. This has to be done on the basis of each scenario step. In general only the more critical scenarios are checked using exhaustive search techniques.

3. At the point where an error is detected, it is very interesting to be able to go back to the powerful execution environment (step 1), where these errors can be analyzed and fixed much more easily.

This cycle is repeated until a satisfactory degree on confidence about the correctness of the model is reached. As usual, this degree of confidence is defined subjectively by the people involved, that define and do the actual work. They just benefit from better tools to perform better.

1.3 Miscellaneous Points

1.3.1 Reader's Background

This document assumes a reasonable knowledge about distribution technologies, in particular with respect to the OSI Systems Management framework and CORBA. In fact, OSI-SM and CORBA are the distribution technologies that were used in the context of the TIMS project. So experimentations and softwares were developed for OSI-SM and CORBA models. However, the concepts, and techniques presented are discussed in the much more general context of open distributed processing (ODP). The more relevant aspects of ODP with respect to our work are presented as they are needed. This concerns mostly concepts related to behavior, defined in the foundations document of RM-ODP [Rm odp2]. In addition, a general presentation summarizing all the issues considered by the ODP reference model is given in appendix B.

1.3.2 Language Mapping in *Scheme*

To provide for low level execution support, the proposed behavior notation template is mapped on an existing programming language. The behavior language is in fact directly embedded into the target programming language. The important advantage is that no low level or basic language design and implementation support has to be done. Such an effort would be required for most of the behavior clauses of the behavior template. Basic low level language support typically ranges from the use of logical expressions, to complex algorithmic description that include usual control structures (conditionals, loops etc), variable notation, assignment, input output, etc. The programming language that has been chosen to provide this language mapping is *Scheme*. *Scheme* is a clean, simple and small but still very powerful programming language. *Scheme* is standardized [Clinger et al.91] and many implementations are available, including free-ware implementations. In addition both compiled and interpreted implementations of the *Scheme* programming language are available. On one hand, an interpreter is very useful to provide an interactive development environment. On the other hand, *Scheme* compilers allow to transform *Scheme* code into a faster representation, e.g. to C and then to machine code, or to a byte-code representation. For all these reasons the *Scheme* programming language revealed as a good compromise to instantiate concepts and techniques presented in this thesis. However, it should be noted that *Scheme* is only used to provide concrete mapping for the proposed behavior language and low level execution support. Though this has not been envisioned in the context of our work, many other languages could have been used as well.

1.4 Other Relevant Publications

[Sidou et al.95b, Sidou et al.96a, Eberhardt et al.97b] are survey papers about the behavior modeling approach used in the TIMS projects and the various case studies onto which the behavior model was confronted.

Chapter 2

Behavior Modeling Issues

2.1 Introduction

The objective is to define and elaborate on the problem of the specification and validation of distributed object frameworks with respect to distributed object computing in general and in the light of architectural concepts coming from ODP. It is shown that by limiting the scope of modeling to functional issues, an abstract and expressive specification approach can be obtained, The approach is based on the declarative specification of actions. Then validation is discussed. Execution and reasoning are the two alternatives that can provide some assistance in the validation process. An approach based on execution is advocated because it seems more easy to apprehend by users as long as a suitable execution environment is also provided.

The plan of this chapter is the following :

- Section 2.2 focuses on distributed object computing (DOC), i.e. the application context of our work, and defines in this context the concept of distributed object framework (DOF). Open distributed processing (ODP) is also considered here, simply because ODP provides a very general framework to deal with all the relevant issues in the context of DOC systems. The scope of our work can be easily delimited using ODP viewpoints. In addition, most of the underlying concepts needed to tackle behavior issues in the context of DOC systems are properly defined by ODP.

- In section 2.3 the scope of behavior issues that one can consider when modeling distributed object frameworks is discussed. A clear distinction is made between functional and non-functional behavior issues.

- Section 2.4 discusses ODP concepts worth to be considered. In particular, concepts related to behavior issues are recalled. In addition the important notion of viewpoints is discussed with respect to the problem of functional modeling. In this context the important role of information modeling is emphasized.

- Due to the importance of information modeling, section 2.5 considers the problem of finding suitable and powerful information modeling abstractions. This is naturally a basic software engineering problem, that can be reduced to the fundamental problem of managing change, i.e. how to make possible behavior evolution, customization, etc. This has both to be done statically (by changing libraries or specifications to accommodate new application requirements), or dynamically because behavior may also have to change at runtime. It is first shown that the commonly used static class-based inheritance scheme reveals limited to support effectively change management. That is the reason why new paradigms have been proposed to circumvent

this problem. In particular, this is illustrated with some implementation level solutions that have been found, such as design patterns. All these solution have all in common that they introduce some form of dynamic aggregation structure to evolve the system's state. Finally, the distinction between high level and low level dynamic aggregation structures is made. In this context it is interesting to see how popular OO-methods are positioned.

- In section 2.6 the role / relationships-based behavior formalization (RBF) paradigm is discussed. it is shown that it provides a powerful dynamic aggregation structure. Finally, it is shown why the GRM is a suitable notation. In addition the restrictions made in our work on the usage of the GRM notation are considered.
- Section 2.7 discusses specification and validation in general. Executable vs. non-executable approaches are considered and compared.
- DOC systems being a particular case of distributed and reactive systems, section 2.8 discusses validation in the context of distributed and reactive systems.
- In section 2.9 to conclude this chapter, the main features identified of a suitable functional behavior model for distributed object frameworks are listed.

2.2 Distributed Object Computing

Distributed Object Computing encompasses all the distribution technologies that have been devised in order to ease the development of all the components of distributed applications, i.e. both clients and servers. There is in fact no particular reason to emphasize on the object oriented character in the DOC acronym. Indeed, object orientation is used in distributed computing systems for the same reasons it is used in other application domains, i.e. because object orientation is considered to be a good software engineering technique generally useful. It is generally useful because it is based on the more fundamental modeling features, i.e. *encapsulation* and *abstraction*. Distributed applications require as all complex applications good software engineering techniques. Anyway, such (object oriented) distribution technologies are middle-wares providing appropriate and standardized APIs. These APIs provide a much more comfortable level of work compared to the basic and low level APIs for network and system programming provided by the operating system. In addition, DOC technologies usually include commonly useful services (e.g. naming service), and functions (e.g. management, security, etc) that are most of the time needed for the deployment of distributed applications.

2.2.1 Distribution Technologies TMN, CORBA, SNMP-SMI, Java-RMI, etc

Many DOC technologies have been devised. They correspond sometimes to very different technical problems and solutions. However they also reflect many redundancies and confusion. The main reason is that they originally correspond to different communities of people.

- Two network management communities have come up with two different management frameworks :
    1. In ITU-T, the TMN community has devised the OSI-SM [Osi sm] framework and MIBs to manage telecom network equipments, services, etc.
    2. In the IETF, the internet community has devised the SNMP framework and the SNMP-SMI [Schoffstall et al.90, Rose et al.90] to manage internet protocols, services and applications.

- CORBA [Omg corba96] is the distribution technology devised by the object management group (OMG). OMG is a group of industrial companies that came together to a standardized and general purpose DOC technology.

- Java remote method invocation (RMI) [Java rmi] technology has recently been introduced by Sun Micro-Systems. Java-RMI provides a very easy development of distributed applications using the Java programming language.

One important point that has to be clearly stated here is that the need for good specifications of distributed object frameworks is not a problem related to a particular distribution technology. This is a general problem that manifests itself as long as one is interested in the correct specification of distributed applications or more generally distributed application components.

2.2.2 The Role of ODP

In the context of distributed object computing, ODP [Rm odp1] can be viewed as an appropriate framework to speak about DOC systems. Of particular interest is the concept of ODP viewpoints, which identifies properly different aspects into which a DOC system can be described. A minimal overview of ODP (goals, structure and concepts) is given in appendix B. ODP has also identified the need for generic functions such as management and security. In addition, the concept of distribution transparencies is also important. Distribution transparencies identify the different generic facilities to ease the development of distributed applications. The most basic distribution transparency is *access transparency*. Access transparency mostly addresses communication as[ects. It is discussed in section 2.2.3. Other distribution transparency issues are persistence, migration *etc*. Access transparency is a mandatory feature, that is the reason why it is the transparency issue that has received the more attention in the past. The different distribution technologies used today can be primarily distinguished based on the way they take access transparency into account.

2.2.3 Access Transparency

In the context of distributed object computing (DOC), a significant degree of maturity has been reached with respect to *access transparency* issues. Access transparency is defined by ODP as follows :

> Access transparency masks differences in data representations and invocation mechanisms to enable inter-working between objects.

Access transparency is mostly concerned with the basic communication mechanisms, e.g. communication protocols, services, APIs, marshaling and unmarshaling, etc.

2.2.3.1 Static vs. Dynamic Interfaces

With respect to inter-object communication APIs, dynamic and static families of APIs can be distinguished. This results in dynamic and static interfaces. A dynamic interface can be used to send with the same interface any kind of message. In contrast, with a static interface, a dedicated interface is used to send each kind of message. Static interfaces are typically more simple and easy to use, whereas dynamic interfaces are more complex and difficult to use. They are required when building generic applications such as browsers, gateways etc. Static interfaces are also called *stubs* (on the client side) and *skeletons* (on the server side).

2.2.3.2 Access Transparency in OSI-SM

The OSI-SM [Osi sm] framework is used mostly to build TMN-systems applications. CMIS/P [Cmis, Cmip] provide the means to properly invoke and convey the systems management messages. CMIS defines the first level of the interface, the inner levels or the details are dependent on the particular managed objects used in a given management applications. This part is defined using GDMO [Gdmo] and ASN.1 [Asn188]. Note that an API defined directly on the definition of CMIS services is a typical example of a dynamic interface, indeed this is a good example of a complex dynamic interface.

2.2.3.3 Access Transparency in CORBA

In the context of CORBA-based systems, both static interfaces (stubs and skeletons) and dynamic (DII and DSI) interfaces are defined to convey messages related to a given CORBA-IDL [Omg corba96] specification. Language mappings are also available with respect to each kind on interface. Language mappings ensure the portability of distributed applications at the level of source code. This was the main objective of the version 1 of CORBA. This can be used to port (compile and link) one application form one ORB to another, without difficulty. The main objective of the version 2 of CORBA is to provide ORB interoperability. So, a generic inter-ORB protocol (GIOP) has been defined, as well as some mappings to concrete protocols. In particular the more popular mapping is the internet inter-ORB protocol (IIOP). It is defined by the mapping of the GIOP on TCP/IP.

2.2.3.4 Access Transparency in SNMP

SNMP obeys to a similar setting as the OSI-SM, except it was designed in much more simple terms. So, as in OSI-SM, SNMP services are usually made available through dynamic interfaces. However that is not considered as a problem because even dynamic, such interfaces are still very simple. As in OSI-SM, only the first level of the interface is defined, the details depend on the particular SNMP MIB definitions used in a given management application. This part is defined using the SNMP-SMI [Rose et al.90]. The SNMP-SMI is based on ASN.1, but on a very restricted subset of ASN.1 that avoids all the intricate aspects of this data definition language.

2.2.3.5 Access Transparency in Java-RMI

The remote method invocation (RMI) mechanism maps directly interfaces defined in the Java programming language to corresponding static interfaces. A specific serialization mechanism has been defined to send and receive RMI message to and from the wire. The transport of RMI messages is done on top of TCP/IP. Java and RMI are very attractive because thanks to RMI, the transition from a local Java object to a networked Java object is immediate. This provides an ideal framework for application prototyping. For industrial applications where legacy components are typically to be integrated, it is less easy to work with Java and RMI. By the way, that is the reason why more general mechanisms based on the separation of interface definitions from programming languages have been defined. However it should be noted that Sun Micro-Systems has committed to provide RMI on top of IIOP. This may be a very important step for Java and RMI to be adopted in industrial applications.

2.2.3.6 Summary

To summarize, the current status is as follows : object interoperability issues are defined by object interface specifications using some kind of object interface definition language (OIDL, e.g. OMG-IDL, GDMO, ASN.1, SNMP-SMI or even Java interface specifications). The question is now : is that sufficient to guarantee interoperability ? Obviously no !

2.2.4 Importance of Semantics, Behaviors Issues

Object interfaces only guarantee that syntactically correct messages are conveyed between client and server objects. Interoperability goes far beyond the point of syntactic issues. Of particular importance is to guarantee that the semantics of messages are properly implemented by server objects, i.e. the behavior of server objects when processing incoming messages is the expected one. Though it seems possible to consider the specification of interfaces at the level of objects alone, it turns out that considering the specification of behavior at the level of objects is in general very limited. The main reason is that the behavior of an object is typically inclined to reflect application oriented issues. As a consequence, the objective is to obtain specifications of components directly usable to build distributed applications and which are typically directly in relation with end users or with the business of the enterprise. For instance, the semantics of components required to build network management agents and applications are of the highest importance for telecom companies making their business by providing network links, services etc. Finally, application oriented issues are at least as important as communication oriented ones. Indeed they are more important because of their direct link to, and influence on, the enterprise business.

2.2.5 Higher Level Services, CORBA Object Services, OSI-SM Systems Management Functions

In OSI-SM and CORBA, the need for a set of general purpose and generic services has been identified. These services are built on top of the basic communication services, and aim at the procurement of higher level abstractions. In OSI-SM, this set of higher level services is called the system's management functions (SMF). In CORBA, one rather speaks in terms of object services. Note that neither SNMP and nor Java-RMI have identified such higher level services. The important point, with respect to object services is that they are only concerned with interface specification of generic services, and that is all. There is no more consideration with respect to semantics and behavior issues other than conventional object interfaces used to provide basic object communication.

2.2.6 Distributed Object Frameworks

The need to think in application oriented terms to specify semantics and behavior has already been identified. However, it is not desirable to specify behaviors that are completely application specific. So, what is needed is an intermediate level, where distributed application components can be considered. In the context of distributed object computing we have chosen to call such distributed application components *distributed object frameworks* (DOF) or simply *object frameworks* as it is proposed by the OMG (see below).

2.2.6.1 Distributed Object Frameworks in Standardization

Standardization activities have already recognized that dealing only with access transparency issues was far from sufficiency to guarantee interoperability. Therefore, attempts have been made to include semantics / behavior issues and the related notions to capture application requirements. The more significant achievement is the concept of *object framework* that has been introduced by OMG in a recent specification of the object management architecture (OMA) [Omg oma]. Before, the TMN community and more precisely the network management forum (NMF) has also introduced the concept of *Ensemble*. OMA object frameworks and NMF Ensembles are described more precisely in sections 2.2.6.2 and 2.2.6.3 respectively.

2.2.6.2 OMA Object Framework (OMA-OF)

The concept of *object framework* has been introduced in a recent revision of the OMA reference model [Omg oma]. An object framework is a configuration of objects providing a service or a set of services. Such an object configuration forms a higher level component providing a functionality of direct interest to end user applications. The configuration is a collection of objects able to interact at their interfaces and the specification of the configuration determines the set of objects involved in each interaction. Figure 2.1 shows all the basic constituents of an object framework as it is defined by the OMA. The outer circle of an object represents the types of interface supported, and the inner circle represents the functional core supporting those interfaces.

Object Frameworks are collections of cooperating objects categorized into Application, Domain, Facility, and Service Objects. Each object in a framework supports (for example, by virtue of interface inheritance) or makes use of (via client requests) some combination of Application, Domain[1], Common Facility, and Object Services interfaces. A particular Object Framework may contain zero or more Application Objects, zero or more Domain Objects, zero or more Facility Objects, and zero or more Service Objects. Service Objects support Object Services (OS) interfaces; Facility Objects support interfaces that are some combination of Common Facilities (CF) interfaces and potentially inherited OS interfaces; Domain Objects support interfaces that are some combination of Domain Interfaces (DI) and, potentially, inherited CF and OS interfaces; and so on for Application Objects. Thus, higher level components and interfaces build on and reuse lower level components and interfaces.

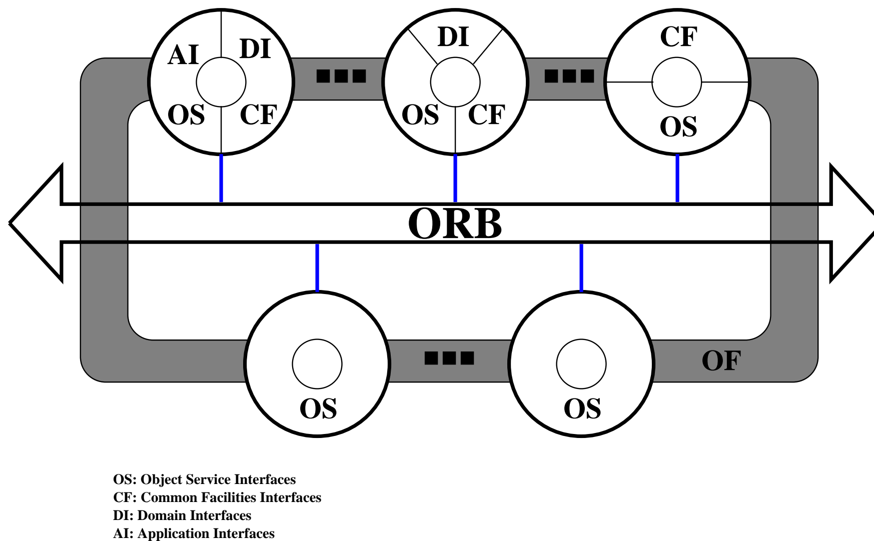


Figure 2.1: OMA Object Framework

The OMG "green" paper [Wood97] lists some requirements about the way object frameworks should be specified. In particular, a template and guidelines for the specifications of object frameworks should make possible the specification of :

- the services provided by an object framework to be offered by a number of the objects that comprise it.
- the assumptions about how the objects framework is to be used.

[1] Domain Interfaces are domain-specific interfaces for application domains such as Finance, Health-care, Manufacturing, Telecom, Electronic Commerce, and Transportation.

- the *environment contract* for an object framework. The environment contract may include QoS, usage, management and security constraints. Environment constraints can describe both :
  - requirements placed on an object's environment for the correct behavior of the object.
  - constraints on the object's behavior in the correct environment.

This same "green" paper defines also what is the specification of an object framework itself. The specification of an object framework :

1. defines the objects that make up the object framework.
2. defines the type, interfaces, operation sequencing and QoS for each object.
3. identifies the object framework interfaces (those object interfaces that support access to functionality provided by the object framework).
4. defines the patterns of interactions between the objects in the object framework, and the semantics of those interactions, in support of the object framework functionality.

Items 2 and 4 in the list above are themselves very general and difficult problems, e.g. how is defined the type of an object or its QoS ? However the "green" paper is only a basis for a future request for proposals in OMG about the specification of object frameworks, where the submissions are supposed to detail their approach with respect to these complex problems.

2.2.6.3 NMF Ensemble

The Network Management Forum (NMF) *Ensemble* concept is defined in [Nmf ensco92] as follows :

> An Ensemble is the application of an Information Model in a particular Management Context. An Ensemble draws together all the various strands of a solution to a management problem by linking the management context with an information model and a set of scenarios.
>
> Ensembles may reference or use other Ensembles. The management context sets the scene for a particular Ensemble by stating the management problem as requirements and constraints expressed in terms of resources, functions and the level of abstraction of the management problem.
>
> The information model is a general solution and can be applied in many management contexts. For example, managed object definitions are usually defined with optional components using conditional packages, so that they can be used as widely as possible. To be used in a particular Ensemble the information model is constrained to fit that Ensemble.

Note that the concept of Ensemble does not actually include a direct description of the semantics behind the distributed objects involved in the Ensemble. Rather what is given is an indirect description of these semantics by a set of scenarios along with their expected outcome. So, the concept of Ensemble is only a first step in the direction of the concept of object framework as it is considered in the OMG-OMA. In any case, an Ensemble is still a good complement to the general description of behaviors given in prose at the level of Managed Objects in GDMO. That is the reason why the concept of Ensemble is quite successful in the community of TMN systems.

2.3 Scope of Behavior Issues (Functional vs. non-Functional Models)

There is a distinction that can be made between functional and non-functional behavior issues. Note that functional is not considered in the sense of model of computation as it is the case in the context of functional programming languages. Here, functional is used in the more common sense of functionality (exercised in the system). This covers mostly all kind of state transitions that can be observed in the behavior of the system. So, a functional behavior specification does not cover the quantitative aspects such as timing, real time and performance issues. Such quantitative aspects are referenced as non-functional behavior issues. Both kind of behavior issues have their importance. Functional properties are the properties that tend to be primarily considered in a model of a distributed system. This reflects the need to get first an overall idea about the functions accomplished in the system. In some applications such as multimedia applications real time constraints and thus non-functional behavior issues play also an important role.

For practical reasons due to the fact that the concrete case studies envisioned in the context of the TIMS project were exclusively concerned with functional modeling. Therefore our primary interest goes naturally towards the development of functional models.

2.4 Usefulness of ODP

ODP defines a set of architectural concepts to describe and build distributed computing systems in general. Appendix B gives an overall view on the whole set of the ODP architectural concepts. This section focuses only on the more relevant aspects of the ODP reference model with respect to our objective, i.e. functional modeling in the context of distributed object frameworks. In section 2.4.1 it is shown how ODP has defined properly all the basic concepts for object oriented behavior modeling. These concepts are recalled and naturally reused, just because there is no advantage to try to find better definitions. In section 2.4.2, the concept of ODP viewpoint is presented. This concept is useful to delimit properly the scope of behavior issues that are finally intended to be covered.

2.4.1 ODP Concepts for Behavior Modeling

In this section, the more relevant definitions from the foundations document of RM-ODP [Rm odp2] concerning behavior issues are recalled. The definitions present in the foundations document of RM-ODP are not dependent on any viewpoint. They constitute the common substrate that is reused for all the viewpoint languages. The definitions in this section are directly taken from RM-ODP2.

2.4.1.1 Object

An object is a model of an entity. An object is encapsulated and provides abstraction. An object is characterized by its behavior and dually by its state. *Encapsulation* means that changes in the state can only occur as a result of internal actions or interactions. *Abstraction* implies that the internal details on an object are hidden from other objects, so that they can be implemented in different ways. An object has an *identity*, that gives a non-ambiguous way to refer to the object in a model, independently of its state.

2.4.1.2 Action, Interaction, Internal Action

An action is merely defined as something which happens. Every action of interest for modeling purposes is associated with at least one object. The set of actions associated with an object is partitioned into internal actions and interactions. An internal action always takes place without the participation

of the environment of the object. An interaction takes place with the participation of the environment of the object. It is important to note that :

1. "Action" means "action occurrence". Depending on context, a specification may express that an action has occurred, is occurring or may occur.

2. The granularity of actions is a design choice. An action needs not be instantaneous. Actions may overlap in time.

3. Interactions may be labelled in terms of cause and effect relationships between the participating objects. The concepts that support this are discussed in [Rm odp2] section 13.3.

4. An object may interact with itself, in which case it is considered to play at least two roles in the interaction, and may be considered, in this context, as being a part of its own environment.

5. Involvement of the environment represents observability. Thus, interactions are observable whereas internal actions are not observable, because of object encapsulation.

Though an action is a very general concept, its definition is not ambiguous or imprecise. Of particular importance is that all levels of granularity can be used, from atomic actions (e.g. an non-interruptible piece of code executing locally in some process) to non-atomic actions (typically occurring in a distributed application) that may overlap over time.

2.4.1.3 Behavior of an Object

The behavior of an object is defined as the set of actions the object is involved in. Two consequences of this statement are that :

1. Modeling object behavior involves modeling of actions.

2. Because an interaction may involve more than one object, behavior may be defined on collection of objects.

The behavior of an object may also includes the set of constraints that determine when its actions may occur. Constraints may include for example sequentiality, non-determinism, concurrency or real-time constraints.

2.4.1.4 State of an Object

The state of an object determines the set of actions the object can take part. Because of nondeterminism knowledge of state does not necessarily make possible to predict the sequence of actions that will actually occur. State changes are a consequence of the occurrence of actions; hence a state is somewhat determined by the previous actions in which the object took part. That is the reason why state and behavior are qualified as dual concepts.

2.4.1.5 Interface of an Object

An interface is a subset of interactions in which an object can participate. An object can have several interfaces that define a partition of the set of its interactions.

2.4.2 ODP Viewpoints

Being able to build different models of a system according to different levels of abstraction is important. This is exactly the objective that ODP viewpoints are intended to deserve in the context of distributed computing systems. The general objective of viewpoints is to provide different perspectives into which distributed computations can be described. The idea of viewpoints and especially the five viewpoint identified in RM-ODP have been widely accepted by the distributed computing community. The main objective of this section is to consider the specification of functional models with respect to ODP viewpoints. Of particular importance are the information (section 2.4.2.1) and computational (section 2.4.2.2) viewpoints that are introduced more in details because they are identified as a suitable combination for the purpose of functional modeling. A summary on the others viewpoints as well as the other architectural concepts included in the ODP reference model are given in appendix B. In section 2.4.3 the actual analysis of behavior issues (i.e. functional and non-functional) with respect to ODP viewpoints is done, and section 2.4.3.4 concludes the section by summarizing the approach observed in our work.

2.4.2.1 Information Modeling

An information specification focuses on the semantics of information and on information processing. This can be used to get a clear understanding of what is done in the system for the purposes of a given distributed application. An information specification is complete with respect to functional issues. In addition, a functional model at the information viewpoint can be specified in an abstract way. Details related to the computational model are hidden, i.e. the identification of potential units of distribution, their interfaces. In addition, details related to the engineering model are hidden, i.e. the physical distribution, the communication channels, the threads of execution, and protocols used. A specification of the information processed in the system defines (i) an important part of the system's state (static schema), and (ii) from the specification of state transitions between system's state (dynamic schema) can be specified. For these two important reasons, it is clear that information modeling plays a key role in the treatment of semantics and behavior issues in the context of distributed object frameworks. In fact a large part of behavior issues can be considered at the information viewpoint. That is the reason why, it is interesting to see the attention paid to information modeling by different technologies of distribution. This topic is discussed below, note that the discussion addresses only issues related to the static schema.

Information Modeling with respect to CORBA CORBA-IDL, which is the main notation used in CORBA-based systems, is only a computational language, i.e. an IDL specification gives only operation signatures. Thus CORBA as such does not provide any support with respect to information modeling. However, within OMG the need to consider state and behavior / semantics issues has also been recognized, and consequently the need for information modeling support. The current trend in the OMG with respect to information modeling is to use the unified modeling language (UML) [Partners97].

Information Modeling with respect to OSI-SM Separation between information and computational modeling within the OSI-SM framework is rather controversial. The reason is that the OSI-SM framework is a somewhat biased model. Some people see a MO as a computational object because as in CORBA-IDL when an attribute is defined in a managed object class (MOC) in GDMO, what is in fact defined is e.g. a GET and a SET operation that can be invoked on MOs of that MOC. What is clear is that an attribute in a MOC defines one or more operation interfaces. However, the problem lies in the identification of the computational object, to which this interfaces are to be associated. In

effect, what actually occurs is communication between a manager and an agent, so in the pure ODP perspective the computational object is the agent and not the MO. All the arguments to justify this view are presented in [Genilloud96]. The same observations are also stated in the TINA-C computational modeling concepts document [Tinac cmc94] and in the TINA-C information modeling concepts document [Tinac imc94]. So, such people consider MOs to be the information objects in the agent (which is a computational object). Where the GDMO model is biased is in the fact that GDMO is a super-set of what is actually needed for an information specification language. And in fact when GDMO MIBs are specified people do not always think in terms of information modeling but rather in terms of interfaces to MOs, indeed in terms of agent interfaces (even if this is not done consciously). Though GDMO can be used for information modeling, by discarding the computational issues from an existing GDMO MIB, this leads to an information model but may be an incomplete one and probably not the best one. The main reason is that due to the computational bias with which the GDMO MIBs are written, the resulting information model may not be so satisfactory. In fact, starting from scratch (with only an information modeling objective) would probably lead to something different. For instance some attributes may appear in the pure information model that were not identified in the original GDMO MIB, or that were identified in another way. Because of the computational bias of GDMO, other groups have defined specific information languages. ITU-SG15 [G851 0196] have defined the GDIO (Guidelines for the Definition of Information Objects) notation, GDIO is merely GDMO where computational issues have been removed. Within TINA-C the quasi-GDMO+GRM notation [Tinac imc94] have been defined to make a clear distinction between computational an information viewpoint issues.

Information Modeling with respect to SNMP and SNMP-SMI There is not much to say here because the situation is comparable to what has been said in the context of OSI-SM. That is a SNMP-MIB purged from the computational aspects can be interpreted as an information modeling notation, although it was though in terms of an interface to a SNMP agent. It should be noted that the resulting information model would based only on the very restricted subset of ASN.1 used to define SNMP MIBs.

Information Modeling with respect to Java-RMI One can see the information model to be obtained in the same way as the specification of the object interfaces, i.e. by using the directly the notation of the Java programming language, just like one can use `C` or `C++` as an information viewpoint notation. Thus, the information model would result from the instance variables that are defined in each object. The data description facilities available in the Java programming language would be directly used as the information notation.

2.4.2.2 Computational Modeling

The computational viewpoint addresses a significant part of the distribution problem. A computational model decomposes the system into *computational objects* interacting which each other at specified interfaces in order to facilitate distribution. The interfaces defined by the computational viewpoint define the maximum level of distribution that may be supported. The final decision on the actual level of distribution supported is defined in the engineering viewpoint specification. A computational viewpoint specification includes a detailed specification of interfaces provided by each object, their operational signatures and their behavior. The computational language is often presented as the core of the RM-ODP because it establishes most of the characteristics of ODP systems. The definition of objects (as opposed to only interfaces) can be used to specify the interactions between interfaces, as well as the explicit number and types of interfaces (e.g. client / server) to be supported by a single object. This provides a complete definition of the application related to the object. That is the reason

why CORBA has been primarily considered with respect to computational issues only, i.e. the IDL notation. The computational language imposes constraints which ensure that computational objects can effectively be distributed transparently. Considerations such as the ones related to the location of computational objects, i.e. processor, storage, and communication resources are abstracted away.

A point upon which the computational language is very prescriptive are computational interfaces that are intended to support object interactions. Three types of computational interfaces have been defined in ODP :

- *operation interfaces* are either *client operation interfaces*, which allow object to use service provided by other objects, or *server operation interfaces* that offer an abstract service. Service operation interfaces are defined by a set of operations to be invoked by an object with a client operation interface. An object may have several client and / or server operation interfaces, and even several instances of a same client / server operation interfaces. Operations that take place at operation interfaces can be of two types :

  – *Announcements* are operations for which no outcome is reported to the invoker.

  – *Interrogations* are operations for which an outcome is always reported to the invoker, this includes errors caused by a failure at the level of the engineering infrastructure.

  Operations define asynchronous interactions between objects, and because of communication delays the reception of the outcome can not be considered as instantaneous. However, it is an implementation issue to decide whether an interrogation may be blocking (the thread that has invoked the operation waits for its outcome) or non-blocking (the thread executes something else, and is then notified when the outcome is received or it makes an explicit request to the outcome when it is ready to do so).

- *signal interfaces* define atomic shared and one-way interactions between only two objects.

- *stream interfaces* define continuous flows e.g. multimedia streams.

In the applications we have been considering, signal and stream interfaces were not used. In fact neither in GDMO nor in CORBA-IDL the definition of such kind of interfaces is possible. Only the operation interfaces can be specified. That is the reason why signal or stream interfaces are not considered further below. An important point with respect to behavior issues is that the computational language does not constrain the behavior of computational objects. RM-ODP documents do not advocate any particular concept related to behavior issues in the computational viewpoint language. Naturally common behavior concepts coming from the the ODP foundations document can be used. In particular, in the computational viewpoint the behavior of an object is typically defined by the specifications of its interactions that occur on one of its computational interfaces. Since at the computational level, operation interfaces are precisely specified, the specification of interactions (that define the behavior) can be based on such precise specifications. Though it is useful to specify behavior to have a precise specification of operation interfaces, this is far from being sufficient to provide a complete specification of the corresponding interactions. One possibility is to use the information model and to see the computational model as a refinement of the information model. This approach is used in [G851 0196], which is an application of RM-ODP to the management of the transmission network[2]. So computational objects can be defined as a specific view of information, as it is defined

[2]In this recommendation, the objective is to define a general network level model. This model is intended to work first with the usual OSI-SM infrastructure that is classically adopted in TMN applications. However, the model is done such that the evolution in the direction of other infrastructures (e.g. CORBA-IDL and ODP Functions) will be possible as the corresponding standards become available. So, the methodology proposed in [G851 0196] enterprise, information and computational viewpoint specifications are independent of the underlying (distribution) infrastructures.

by information objects, for a specific application purpose. Concretely, interactions can be specified (i) by reference to computational interfaces (ii) by reference to data that was specified in the information objects (static schema), and (iii) by reference to conditions that were specified in the information viewpoint (dynamic schema, such conditions can typically be used to specify pre- or post-conditions of interactions). In section 3.5.2, it is shown how in the context of the proposed behavior model this approach is concretely instantiated.

2.4.3 Scope of Behavior Issues w.r.t. ODP Viewpoints

As indicated in section 2.3, we are primarily interested in functional modeling. In this section, both functional and non-functional modeling issues are considered in the perspective of ODP viewpoints. A very important point is to understand the relation existing between viewpoint models with respect to functional and non-functional behavior issues.

2.4.3.1 Non-functional Issues

The more details about the underlying infrastructure are available, the more non-functional issues can be treated precisely. Note that these details are available in an engineering viewpoint model. Certain time bound (e.g. for multimedia applications) typically depend on the knowledge about the network links (speed, capacity, reliability etc), processing units (speed, storage capacity). However that does not mean that some real time constraints can not be checked using more abstract viewpoints models, i.e. in the information or in the computational viewpoints. Computing time bounds may also be based on knowledge about potential network interactions, i.e. given by a computational model. What may be important for certain kind of timing constraints is to know only where interactions on the network can occur. The resulting time bounds are, in that case, less dependent on the physical infrastructure. Finally, even an information model may be suitable to check time bounds that are in that case totally independent on the underlying distribution, i.e. delays may be associated to actions in the system based only on application domain knowledge. So, depending on the application and the kind of real time properties all the viewpoint specifications may be relevant. It should be noted that non-functional modeling can be done on a fine grain only at the engineering viewpoint. However, for coarse grained non-functional models working in more abstract viewpoints is probably more interesting.

2.4.3.2 Functional Issues and Information Models

For functional issues the situation is rather different. A functional model can be completely specified from the information model. And what is given in the computational and engineering models are only additional details. In fact, it may not be interesting to include such additional details. It is shown in section 2.8.4.1 that engineering issues typically involve unwanted constraints. That is the reason why, it seems inappropriate to use a behavior specification framework oriented towards engineering abstractions for the purpose of functional modeling. Our position w.r.t. computational issues is clarified just below.

2.4.3.3 Functional Issues and Computational Models

A point of discussion exists about the use of the computational model for functional modeling. Though an information model is complete, it focuses primarily on the system itself, i.e. information objects, state invariants on the information objects and state transitions on the information objects. To understand fully the functions performed by a system some kind of link with the environment is necessary. To this end one can introduce a notion of interface to information objects (as it is mentioned in [Genilloud96]) that are typically intended to link the environment with the state transitions. Another

possibility is to use the specification of interfaces as they are defined in the computational model. This results in fact in the methodology advocated by [G851 0196] where the behavior is mostly specified at the information viewpoint level. But in the end things are gathered at the level of the interactions occurring on the computational interfaces by systematically referencing the information viewpoint schemas. This approach is attractive because it minimizes in some way the effort involved. The only inconvenient is that the resulting functional model may be biased by engineering issues. This is not a problem of the approach itself, but rather a problem of available computational notations that are typically dependent on engineering concerns. For instance, in the case of GDMO / ASN.1, there is a strong dependence on CMIS/P and on encoding rules. That is the reason why [G851 0196] has introduced a notion of communication independent (and communication dependent) computational viewpoint specification. In particular, in a communication independent computational model a restricted subset of ASN.1 is advocated and a way of defining operation interfaces is defined that uses a simplified and truly abstract form of the ASN.1 notation. Indications are given to map the resulting (communication independent) computational models on the existing and intended (communication dependent) computational models, that is true GDMO + ASN.1 and CORBA-IDL. Though in theory this approach seems promising, in particular for the development of future specifications, it has to be judged on practical work and on its acceptance by the community.

2.4.3.4 Our Approach

In conclusion using the complementarity of both information and computational viewpoint specifications to provide functional models of distributed object frameworks is a valuable approach. However practical considerations have lead us to follow this approach in a rather pragmatic way, i.e by using existing computational viewpoint specifications that may be biased by engineering issues (as it is the case with GDMO / ASN.1). In fact, the requirement to define something that is engineering viewpoint independent is not mandatory. It exists only if one is actually interested to map the model on several protocols, e.g. CMIS/P and CORBA. If this is not required, the effort involved in using a communication independent computational notation is not necessary and probably not worth to be done. This is worth to be done if the model is intended to run on several services and protocols, e.g. CMIS/P and CORBA.

2.5 Powerful Information Modeling Abstractions

It has been shown in the previous sections that when building functional models most of the work can be done at the information viewpoint. In this context, this section focuses on information modeling abstractions that are intended to help as much as possible the specification of such information models. Since information modeling is concerned with modeling data and the processing of the data, information modeling is a direct application of the results of the software engineering community, that nowadays advocates objected oriented modeling. In section 2.5.1.1, it is shown that because of its limited support w.r.t. change management the original static class-based inheritance paradigm is not a sufficient software engineering technique. That is the reason why other approaches that have all in common to be based on some form of dynamic aggregation have been envisioned. In section 2.5.1.2 design and implementation solutions based on low level dynamic aggregation are introduced. These low level aggregation structures correspond to the level of aggregation structures available in usual programming languages such as C++. In section 2.5.2, higher level dynamic aggregation structures are considered. A special attention is paid on role based modeling which is a powerful information modeling abstraction with very interesting features w.r.t. change management.

2.5.1 Change Management

The ability to support and manage customization and evolution, in a word *change*, is the fundamental problem that each software engineering paradigm strives to attain. This can be explained by the fact that the software engineering work required when building software applications is mostly concerned with customization of or evolution of (i.e. changing) some existing stuff. Sadly, application development is more related to the reworking of existing work than to be a creative activity (at least the creative aspects are not where expected at first glance). In this context, the production of highly reusable and adaptable components is seen as a fundamental step in the effort to successfully manage change. Such components typically identify clearly (i) an aspect of a system that can be fixed and thus reused; and (ii) other aspects of the system that allow variations and the way to introduce variations. For instance, *design patterns* have been recently introduced [Gamma et al.94] and have been very helpful in recognizing common forms of evolution for objects in an application domain. Each design pattern acknowledges the evolutionary property of software in terms of both the structure of an object, its operations, and the corresponding behavior. In addition, while all that has been widely recognized as valuable implementation techniques, specification has received less attention.

Pattern	Variations
Adapter	Object Interface
Bridge	Object Implementation
Decorator	Object Responsibilities
Iterator	Composite Object Traversal
Observer	Inter-Object Dependency
State	State-Dependent Object Behavior (e.g. finite state machine)
Strategy	Algorithm
Visitor	Task-Specific Behavior During Traversal

Table 2.1: Examples of Design Patterns and Intended Variations.

Table 2.1 (taken from [Seiter et al.96]), lists some design patterns from [Gamma et al.94], along with the variation each is intended to support.

2.5.1.1 Problems of Static (Class-based) Inheritance Models

To allow for reuse and variation a usual approach is the static class based inheritance model that is available in programming languages such as C++. Let us reason on the *Strategy* problem, that is a very simple and recurring problem, to see why the static class based inheritance model reveals very limited. The strategy problem is a generic formulation of the very common problem that consists of permitting several algorithms to be defined for a single interface. This is illustrated in figure 2.2. A $Receiver$ class has a method m.

According to a given strategy either $ConcreteReceiver_1$ or $ConcreteReceiver_2$ is activated. It turns out that the following problems are difficult to support in the classical static class-based inheritance model :

1. There may be many algorithms of the $Receiver$ class that should be varied, as a result it becomes undesirable to define subclasses (of the $Receiver$ class), to cover all the variations.

2. If for the $Receiver$ class several methods like m need multiple implementations according to different strategies, what has to be done ? Defining subclasses to cover all the permutations between variations is again more undesirable.

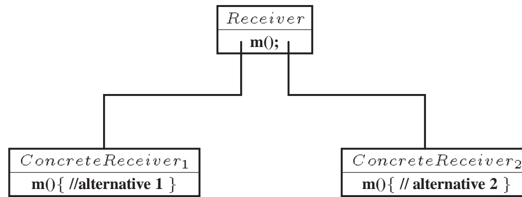


Figure 2.2: Strategy Problem.

3. In addition, it may be necessary to vary at runtime which strategy is used for a particular instance of the $Receiver$ class. That means that the method m of the $Receiver$ class can exhibit *dynamic behavior*.

In conclusion it becomes apparent that using only static class inheritance is not sufficient to encode such rather common application requirements. Note that the problem to have many behavior variation being necessary arises very easily, just because both state and behavior of objects can change either in a static way (at development time) or in a dynamic way (at runtime) to accommodate requirements of applications.

2.5.1.2 Introducing Dynamic Aggregation

All the patterns in [Gamma et al.94], including the strategy pattern which gives a solution to the strategy problem mentioned above, are described using constructs such as static inheritance extended with aggregation relations, and the examples are given in C++. Figure 2.3 describes the strategy pattern using aggregation and inheritance relations. As before the $Receiver$ class has a method m. A $Strategy$ class hierarchy is defined to provide alternative implementations of the method m, with each concrete strategy subclass representing a particular variation. An instance of the $Receiver$ class references a strategy object. The implementation of the method m for the $Receiver$ class simply delegates the request to its strategy object, passing its self reference along as an argument. The method m defined in the concrete receiver strategy is executed to perform the correct variation (currently active) of the algorithm for the receiver object.

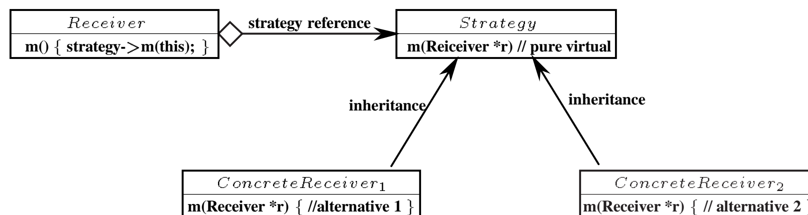


Figure 2.3: Strategy Pattern.

So, the interesting point is that using aggregation to represent the relation between a receiver and a strategy (rather than using static inheritance to define $Receiver$ subclasses and implementing the variations of the method m in the subclasses), solves the problems mentioned above : (i) many algorithms of method m can be varied by defining subclasses of the $Strategy$ class only and not the whole $Receiver$ class, (ii) if another method n of the class $Receiver$ need variations, a $Strategy_n$ class has to be defined that encapsulates the abstract definition of n, and then allows the variations of n to be defined as concrete subclasses of $Strategy_n$, and (iii) all this variations can be exercised dynamically (at runtime) by updating the references to the strategy objects in a receiver object.

2.5.1.3 Related Work

In this section, other approaches are listed, that also introduce high level dynamic aggregation structures to manage behavior evolution at the level of implementation. So, unlike the strategy pattern [Gamma et al.94] that is written in C++, these approaches typically propose new language constructs for dynamic aggregation intended to be easy to use.

Context Relations Though the strategy pattern fulfills all the requirements with respect to customization and evolution of strategies, it has two inconvenients : (i) In a language like C++, requiring explicit delegation with the receiver object being passed as an argument requires the $Strategy$ class to be made a friend of the $Receiver$ class, which violates class encapsulation, (ii) relying on the inheritance relation to group the variations under an abstract $Strategy$ class always requires to define this additional $Strategy$ class whose purpose is only to define a common interface for its subclasses that duplicates the $Receiver$ method that needs variations. For these reasons, in [Seiter et al.96], a dedicated language construct is proposed to define a more general mechanisms : *context relations*.

Context relations can be used to link directly the concrete strategy classes to the $Receiver$ class, as they provide alternative implementations of a method of the receiver class. Context relations are still based on dynamic aggregation but the delegation is made implicit thanks to the underlying language support, i.e. it has not to be encoded by the programmer by using explicit delegation along object references. At the design level context relations are drawn using a new form of arrow between classes, at the implementations level a small extension of C++ has been experienced. The basic idea is that if class C is context related to a base class B, then B-objects can get their functionality dynamically altered by C-objects. The advantage of adding context relations to the design and implementation languages is that many well known design patterns can be realized in a better way. Note that, the mechanism of context relations is an extension of the *Demeter adaptive software model* [Lieberherr96].

Contracts A *Contract* [Helm et al.90, Holland93] models a set of object interactions at the implementation level. The participants to the contract have a certain number of *obligations* (attributes and behavior) that they must support. A contract defines a template of participants and obligations of a task, and is reused by instantiating and enabling the contract in a dynamic way (at runtime). Each participant role is attached to an object and obligations are attached to specific properties of the object. So, with contracts it is necessary to specifically attach (activate, enable) a participant role to a particular object in order to modify its runtime behavior. When an object is sent a message, its active contracts determine its behavior. While an object may participate in several contracts simultaneously, it may not participate in multiple contracts that define the same method. Additionally, the method defined in the base class hides anything defined in a contract. Thus it is not possible to dynamically alter an existing class method.

Dynamic Inheritance in Prototype-based OO Languages Prototype-based languages such as SELF [Ungar et al.91] offer a new paradigm for object-orientation. Unlike classical object-oriented languages the concepts of class and instance variable have been discarded. In a prototype-based language an object is created by cloning (copying) a prototype object. Each object has slots which encapsulate state or behavior. Interaction with objects is done only on message passing basis. A special slot is the *parent* slot, it represents the dynamic inheritance or delegation slot that allows the behavior of an object to evolve dynamically by modifying its parent relation to other objects. If an object does not have an implementation for a message it receives, it is implicitly forwarded to its parent and forwarded along the parent chain until the message is understood. While dynamic inheritance avoids some of the problems of using static aggregation and inheritance to model behavior evolution,

prototype-based languages may reveal difficult to use because (i) change impact (of parent relations) may be difficult to understand, and (ii) basically behavior overriding is not possible (dynamic inheritance may be used to add behavior to an object, but not to override existing behavior defined for an object).

Explicit Relationships in Object Oriented Development In [Noble et al.95] a relationship oriented design and implementation approach is presented. The key idea is to use extra objects to represent relationships explicitly. Behavior which is associated with the relationship can then be put into the relationship object – behavior is promoted. Several implementation techniques are proposed to exercise such promoted behavior. In [Noble et al.95] the techniques of : methods in relationship objects, relationship dependents, update description propagation and monitoring abstract updates are presented. These techniques allow programmers to ensure that functionality associated with relationships is carried out at the appropriate times, and result in less complex systems. In addition, such systems should be easier to extend and maintain.

2.5.2 High Level Aggregation Structures

The previous section has shown that aggregation structures were needed for behavior evolution at the implementation level. In object oriented analysis methods, such as OMT [Rumbaugh et al.91] and Booch [Booch94], high level structures are used to specify the composition of objects to form composite objects. All these structures are no more than *aggregation relationships*. An *association relationship* is a specific kind of aggregation relationship between two objects that is very commonly used. An association relationship indicates that one object makes uses of another object in some way. Besides aggregation relationships, component objects are usually specified using class templates that encapsulates attributes and methods; and usual static inheritance relationships may exist between such object classes. These inheritance relationships are also called *generalization relationships*.

Thus, generalization and aggregation are the two fundamental concepts used to build object oriented models. From the perspective of the ODP information viewpoint, generalization and aggregation relationships are typically used as static schema modeling techniques, which in the end this defines the state of the objects. Association relationships are actually a low level aggregation structure, they allow a direct and straightforward mapping on reference pointers. Reference pointers are the only dynamic aggregation facility available in usual programming languages such as C++. Other approaches for high level dynamic aggregation structures are based on the concept of *role* and (optionally) *relationship*. Roles and relationships are considered in section 2.6.

2.5.3 Losing High Level Aggregation Structures in Existing OO Methods

An important point to note is that generalization relationships can usually be kept unchanged during design and implementation, this allows seamless development that promotes in its turn *trace-ability*, so that an object in a program can be easily traced back to an object in the design and in the analysis. This is perhaps one of the essential benefits of object oriented development because this facilitates change management. However, it turns out [Noble et al.95] that aggregation relationships are lost during design and implementation, because they are generally refined into low level structures such as object references, losing most of the meaning identified during analysis. Since behavior is highly dependent on state, it seems not desirable to work on the behavior on a low level description of state when a higher level form is available. However that is the way object oriented development is performed today. The main reason is that behavior is not actually considered at analysis time but more likely at design and implementation and using low level representations for the state of objects. The objective is to go fast to design and implementation. For these pragmatic reasons, popular OOA&D

methods are inclined to accommodate the requirements of the target implementation languages typically used (e.g. C++, Ada, etc). That is also the reason why aggregation structures used for state representation are not the more abstract. The more commonly used are the ones supporting a direct mapping in C++ such as low level associations. In popular OOA&D method, when behavior is considered at analysis time people tend to do that in a rather informal way, e.g. using graphical notations such as interaction diagrams, message sequence charts, use cases, etc. The objective is in fact to obtain high level descriptions of functionalities, that are very useful for people involved to communicate (a valuable objective in its own right). These high level descriptions may also be useful as informative documents during design and implementation. Note this does not mean that no appropriate support exists (e.g. in OMT), to specify complete models at analysis time. In fact such support exists. It is typically based on the use of some form of state-charts [Harel et al.96]. In addition, the effort supported by the OMG, in trying to unify OMT and Booch in the direction of the unified modeling languages (UML) [Partners97] is also intended to cover behavior issues in a more effective way. Note that UML is currently under development, so the future will tell us about its capabilities w.r.t. a modeling language and a development methodology. However, it should be noted that though behavior becomes formal, this is still done based on low level associations. In addition as mentioned in [Harel et al.96] the objective of people of using state-charts in e.g. UML is also to be able to do automatic code generation. So it seems that two camps can be distinguished. The first camp is the one of people primarily interested to go fast to design and implementation. Such people are ready to pay the price of using low level aggregation structures at analysis time, if this facilitates design or if that enables code generation from an analysis model. The second camp is the one of people primarily interested in modeling for the purpose of validation, behavior analysis, test generation, etc. Naturally, such people are typically interested in high level aggregation structures making a model easier to write, to understand, etc. In the middle, two kind of approaches are tried to conciliate these two camps. The first approach (the language approach) tries to go upwards by introducing into programming languages higher level aggregation structures enabling a direct correspondence with analysis structures. Section 2.5.1.3 is in fact an overview of the different alternative that have been devised in the research community. The second approach is a pragmatic approach, that consists of using well chosen higher level aggregation structures in analysis allowing direct mapping on lower level structures such as associations and then object references. The OOram method proposed in [Reenskaug96] based on role modeling is an example of such a pragmatic approach.

2.5.4 Keeping High Level Aggregation Structures

The objective in this thesis is concerned with modeling and not with design. So there is no good reason to lose any high level aggregation structure, especially if such an aggregation structure is suitable to capture application requirements. The important point is that it becomes easier to write and use a specification based on higher level abstractions because application level terms (e.g. role labels) are kept unchanged is the model, or in other words, the model directly reflects the thinking of the analyst.

2.6 Role / Relationship-based Behavior Formalization

In this section, high level aggregation structures based on role and relationship abstractions are considered. A role represents a view of an object for a particular purpose or utilization context. So roles, by specifying utilization contexts capture directly and in a natural way application requirements. The advantages of role based modeling are : (i) roles encourage separation of concern (section 2.6.1), roles allow easy specification of polymorphic behavior (section 2.6.2), and (iii) roles implicitly define object configurations (section 2.6.3). Finally, in section 2.6.4 the use of relationships in addition to roles

is considered, and the use of the generic relationship model (GRM) [Grm94] notation is considered more in details.

2.6.1 Separation of Concern

The role model facilitates the separation of concerns : to each different purpose corresponds a different utilization context and to each utilization context a role can be associated. Thus roles are a very powerful modeling abstraction allowing to cope one by one with the different situations an object may be involved in, by modeling each different issue using a separate role. Note that this defines a purely declarative modeling framework. Thus the modeling process can be partitioned as required and even be exercised incrementally. Advantages of this separation of concerns is that more manageable models are obtained. On one hand, object specifications are limited to their core functionalities, and on the other hand, application or context-sensitive issues are encapsulated into roles (interestingly in a declarative way). The disadvantage is that this can lead to a fragmented description of large problems, since each model describes a limited aspect of the whole system. In particular a difficulty resulting from this fragmented modeling is model composition which is a non trivial issue.

2.6.2 Polymorphism and Dynamic Subtyping

Objects typically exhibit polymorphism. Polymorphism can be generally defined as the ability for objects to handle messages differently depending on their individual characteristics, i.e. state. Note that the strategy problem (section 2.5.1.1) is a generic problem involving polymorphism, and the strategy pattern (section 2.5.1.2) is a generic implementation level solution for this problem. It turns out that roles give a suitable modeling abstraction for object variants or object with polymorphic behavior. A particular object may play several roles. At the application level, this reflects the fact that an object may be used for different purposes. At the level of behavior specification, each role tells that the object fulfills certain characteristics, or equivalently that the object state is extended in some way. As a result, depending on the role, the behavior exercised when a given message is received on one of the object interfaces can be adapted according to these state extensions, and polymorphic behavior is achieved. In addition, an object playing a role defines a subtype [Kilov et al.96], and since roles may be added and deleted at runtime, roles define in fact dynamic subtyping relationships.

2.6.3 Configuration of Objects

Inherent to the concept of role is the notion of configuration of objects[3]. The concept of role typically captures very well the dispatching of functions, responsibilities etc, of objects within configurations. In fact role models provide by themselves a suitable way to specify object configurations. To each object playing a role (e.g. *client* role) other roles are always associated (e.g. *server* role), that define clearly the object configuration and optionally interactions between roles. A typical example is the object oriented role analysis and modeling (OOram) software engineering method [Reenskaug96]. In OOram, object configurations are described using the *role model collaboration view*. This view shows the roles, their attributes and their collaboration structure. The collaboration view can be used to specify which messages are understood by which role, which messages can be sent by which role and optionally role interactions, i.e. messages that one role may send to another. Note that from the collaboration view based on roles, the “make use of” associations become apparent. At design time, this can be used to determine easily what low level object references will be actually used (in OMT

[3]Configuration is not especially considered in the sense of physical configuration of objects. It is considered more generally in the sense of functional configuration, i.e. a coherent set of interacting objects with functions, responsibilities clearly dispatched between the members of the configuration. One can also speak in terms of object communities, or object collaborations.

this would be typically done with associations). So, the OORam method is a good compromise in the sense that on one hand, by providing high level modeling abstractions, roles perform well for pure modeling issues, and on the other hand the collaboration view defines a straightforward low level mapping for pure design and implementation issues.

2.6.4 Relationships

The concept of relationship follows directly from the need to support configurations of objects inherently present in role models. However, as it is shown by the OOram method, this is not mandatory. In OOram, no reference to any notion of relationship is made because it is deliberately chosen to keep the emphasis on roles. Roles capture purpose of direct interest w.r.t. application issues. In contrast relationships may be viewed, at first glance, as a facility for object grouping (though relationship also identify roles). However, an important point is that a pure object grouping facility is also useful because in the context of specification it is desirable to specify properties whose scope is a whole relationship. That is a property that concern a whole object configuration and not a particular role or role interaction. A typical example is the specification of relationship invariants.

2.6.4.1 Relationships in GRM

The GRM notation was proposed by ISO and ITU-T to compensate the lack for a general relationship facility in the context of the OSI-SM standards notations, i.e. GDMO + ASN.1. GRM standard and corresponding templates can be divided in two distinct parts (in fact these two parts could have been provided as two distinct documents, but unfortunately all the issues are merged altogether in the current GRM document). The first part defines a notation template allowing to specify relationships at an abstract level. This abstract level can typically be used to specify relationships for pure information modeling purposes. For this purpose, a relationship class template is defined. It provides a precise specification of roles in terms of label, cardinality constraints and dynamic role binding support. Compatibility of role members with respect to specific object classes can be specified. Additional attribute properties can be included that define properties of the relationship itself. The relationship class template can be used to specify static inheritance between relationship classes, and the corresponding static semantics is precisely defined. This part constitutes a suitable notation to specify relationships. As such it can be used in more general application context than TMN-systems applications based on the OSI-SM. In fact it has been proposed in the semantics working group at OMG to be used as a general information modeling notation suitable for the specification of relationships [Kilov et al.97, Kilov97]. Note that this part of GRM can also be used to define abstract operations on relationships or roles in a relationship.

2.6.4.2 GRM Mappings

The second part of GRM is concerned with design and implementation issues. This part consists of defining how the abstract level can be mapped on GDMO / ASN.1 and CMIS. On one hand, relationship mapping templates specify how the design of the abstract relationships can be done on a GDMO / ASN.1 object model. On the other hand, operation mapping templates specify how abstract operations can be realized with CMIS operations on the underlying managed objects supporting the relationship mapping.

2.6.5 Our Approach

In our approach the GRM notation is used as an information viewpoint notation to specify relationships and roles. So, only the abstract GRM notation template is used. Note that this ap-

proach is also used in [G851 0196]. TINA-C, in the document related to information modeling concepts [Tinac imc94] also follows the same approach. The quasi-GDMO+GRM notation is an information modeling notation based on GDMO and GRM notation templates slightly simplified. As mentioned above, the use of GRM has also been proposed in the context of the OMG. Using GRM has the advantages of a role based modeling approach, i.e. separation of concerns, declarative and incremental specification, polymorphism, dynamic subtyping, etc. In summary, using GRM provides powerful behavior modeling abstractions supporting customization, evolution or more generally change management. Other details about our usage of relationships are :

- the notion of GRM abstract operations is not used.
- relationships are not directed, i.e. from a role any other object participating in the relationship can be attained. Undirected relationships can be used as a navigation facility among the object graph defined by relationships and roles.
- role interactions appear in the behavior specification themselves. So, there is no additional notation used to specify explicitly role interactions (as it is the case in the OOram method with the role model collaboration view).
- As already mentioned, using relationships can be used to specify properties on whole relationships, i.e. based on object groups or collections with high level or application meaning. Such properties are typically safety properties such as assertions and invariants on relationship. Note that it is (obviously) also possible to specify role invariants.

2.7 Specification and Validation

Previous sections in this chapter have considered application domain features pertaining to DOFs. This has established the language or universe of discourse that is typically used hereafter to model DOFs. The remaining sections focus on validation issues. First validation issues in general are discussed, then the validation of distributed and reactive systems is considered. Distributed and reactive systems can be considered as a generalization of DOFs.

2.7.1 Correctness and Validation

In general, correctness of a system means correctness with respect to the requirements, i.e. with respect to explicit and implicit users intentions and needs. Validation is any task performed to ensure the correctness of a system. In fact validation is most of the time partial and what is actually provided by validation is only some degree of confidence with respect to correctness. That is the reason why several levels and approaches have been proposed to exercise validation activities. Though often opposed, the different approaches are in fact complementary depending on the application context. Two validation levels are identified below : implementation-based validation and specification-based validation. Because implementation-based validation is not relevant in the scope of our work, validation approaches are discussed only at the level of specification-based validation. But first, in section 2.7.1.1 the role of specifications is discussed, and in section 2.7.1.2 the important point of the involvement of users is emphasized.

2.7.1.1 Roles of Specification

Specifications are intended to capture in a formal way the original user requirements which are informally stated. So, specifications are primarily an intermediate representation typically used by

designers and developers to build actual implementations of systems. In general, validation can be done at different levels, i.e. either working from the implementation or from the specifications. So, another role played by specifications is validation. Note that specifications can also play many other roles, e.g. test generation, documentation, etc.

2.7.1.2 Involvement of Users

For any system, an important point is that in any case, users must be involved in the validation of the specification of the system's requirements, since in the end only users actually know what they want. This remark has very important consequences with respect to any specification and validation framework. It is clear that the more usable the specification and validation framework is, the more users or application domain experts are able to work by themselves. In some cases, mixed teams composed of application domain experts and of specification and validation experts can work together. However, because of the costs involved, this situation is rather an exception than the rule. Note that even in that case, the usability of the specification and validation framework facilitates the work. The important point to keep in mind is that most of the time usability is a mandatory feature. As a matter of fact, the specification and validation of distributed object frameworks is typically performed by a small group of application domain experts, e.g. one or two TMN system engineers.

Note on Implementation-based validation To build an entire working system, e.g. a complete distributed application, it is mandatory to do a part of the validation based on the implementation. Indeed, errors related to the design, coding and integration can not be detected before implementation. However, concerning specification errors, it is widely recognized that it is better to detect them as early as possible :

1. this avoids wasting the costly development efforts.
2. because (i) users involvement is important and (ii) users are more likely to be application domain experts usually not interested (if not confused) by implementation details; implementations do not offer an appropriate level for user involvement.

In any case, and in addition to the inherent problems caused by implementation-based validation, we are in fact not concerned with the actual implementation of systems, and thus neither concerned with implementation-based validation. At best, the objective is to build prototypes of specifications of distributed object frameworks. So, the scope of this work is naturally limited to specification-based validation.

2.7.2 Specification-based validation

Unlike implementations, specifications are more prone to the involvement of users, since at least implementation details are not present[4]. Specification-based validation means checking the correspondence between informal requirements and formal specifications. Three techniques have been identified to check this correspondence :

1. inspection,

[4]However, users involvement even at the specification level is not so easy to obtain. In effect, a specification notation can be very general and low level, e.g. set theory and predicate logic for Z. This implies that mappings between a given application domain, e.g. TMN, and the notation used for specifications has to be done by the users at any time. In all cases, the existence of such mappings is a problem with respect to user friendliness because users always prefer to think in problem oriented terms.

2. reasoning also referenced as static checking and

3. execution also referenced as dynamic checking.

2.7.2.1 Inspection

Inspection is something that is always useful and that can always be done. People read or cross read specifications in order to check their common understanding of the intended system's behavior. Inspection is probably the technique the most commonly used, in particular in standardization. A first problem with inspection is that this process is manual. In addition, inspection is directly limited by the reasoning capacity of humans. This implies that problems caused by complex interactions between system's components are not prone to be detected simply by inspecting the specification. Because of these limitations, reasoning and execution are the two other techniques used to validate a specification.

2.7.2.2 Reasoning / Static Checking

Reasoning means proving properties of a specification by examining the specification itself and using the meaning of the specification notation, i.e. its semantics. Reasoning is also referenced as static checking and as static behavior analysis. The *static* qualifier is due to the fact that checking is based on the static specification of the system and not on any executing instantiation as it is the case for dynamic checking described below.

2.7.2.3 Execution / Dynamic Checking

Execution consists of running an executable specification on the basis of a set of tests. This can be used to check that no problem occurred during the executions and to check that the expected behavior is exhibited. Executions are performed according to the semantics of the specification. To be executable, the semantics must be given operationally. Execution is also referenced as dynamic checking or as dynamic behavior analysis.

2.7.2.4 Static vs. Dynamic Checking

Static and dynamic approaches have been opposed for ages, not only in the context of specification, but also in the context of programming languages. In the context of programming languages typing information can be specified for the data processed, function interfaces, etc. Static type checking is often advocated because the typing information can be used by the compiler to produce safer and more efficient compiled code. A static type checker prevents the compiler from generating programs with type errors. In contrast, a dynamic type checker halts the program at runtime as it is about to make a type error. In general, there is no perfect and universal checking mechanism. Static checking is more robust but may reveal too constraining because some restrictions have to be assumed in order to allow a compiler to check for correctness. Even then, a complete form of correctness is never achieved. However, in many cases type correctness is sufficiently useful that paying the penalty in terms of accepting some restrictions is reasonable. That is the reason why some statically typed languages have become popular, e.g. ML [Ullman94] in the the research community and Java in more industrial contexts.

2.7.2.5 Interaction and Dynamic Checking

It is clearly useless to have a dynamic checking system for a program for which any type errors that occur would be meaningless to the user of the program. However, when interaction with a user or with

the environment in general is important, type errors can be used to check such interactions. Because all that is done at runtime, dynamic type checking has to be performed.

Generic Applications Dynamic type checking is used for generic applications, such as database browsers, where the user is guiding the application through the data browsed, here dynamic type checking is reasonable because, by signaling that wrong actions have been tried, type errors have direct meaning to the user.

Fast Prototyping Another context where dynamic typing is used is when interactive and incremental development is performed. This kind of development is typically used to perform fast-prototyping of applications. In that case an interpreted language is used to avoid compilation because compilation is considered to be tedious in such a context.

2.7.3 Executable vs. non-Executable Specifications

In the context of specifications and validation dynamic checking is done using executable specifications. In contrast, non-executable specifications can be used to do static checking. Hayes and Jones [Hayes et al.90] give two main arguments against executable specifications :

1. Executable specifications are less expressive and less abstract than non-executable specifications. Requiring a specification notation to be directly executable restricts the form of specifications that can be produced. In addition artificial constraints may be added to ensure executability. For instance an executable specification is prone to include implementation or algorithmic details. The result is over-specification because more than just the core properties of the system are specified. An important problem arises if the additional implementation or algorithmic details included for the purpose of executability, add also unwanted constraints. For instance the nondeterminism may be reduced and this will in its turn unnecessarily constrain some choices made related to the implementation.

2. Executable specifications are less powerful with respect to validation. Though executing individual test cases is useful, it is less powerful than proving more general properties. This problem is in fact related to any approach based on testing, the degree of correctness is limited to the individual test cases performed.

Fuchs has refuted [Fuchs92] one by one the argumentation of Hayes and Jones against executable specifications. It turns out that when using declarative specification languages (e.g. functional or logic based specification languages) executable specifications result in almost the same level of abstraction and expressiveness as a non-executable form. From the perspective of validation, the important advantage of executable specifications is that they result in a greater involvement of users or domain experts as opposed to experts in modeling techniques. Domain experts can directly participate in the formulation of the specification and in the immediate validation. The involvement of domain experts and the immediate reflection on the consequences of the specification back to the people involved are very interesting features. Though validation performed through the execution of test cases can still be considered as less powerful as reasoning, it should be noted that reasoning is a more difficult way to perform validation, in particular in the context of distributed and reactive systems. So, it is also more difficult to get users involved in reasoning based validation activities. Because involvement of users is the more important thing, this constitutes a major obstacle to the applicability of reasoning based validation. In conclusion, an executable and declarative specification is probably, at least for the time being, the best compromise, in terms of both specification and validation issues.

2.7.3.1 Property-oriented vs. Model-oriented Specifications

A distinction is made between property-oriented specifications and model-oriented specifications. A property-oriented specification defines the behavior of a system indirectly by a set of properties in the form of axioms that the system must satisfy. In contrast, a model-oriented specification defines the behavior of a system directly by constructing a model of the system. To be the more abstract as possible, one may be tempted to think that a specification should be only property oriented. Though, a specification should be as abstract as possible, it should not be more abstract than what is actually required, i.e. not more abstract than the requirements themselves. For instance, if a specific algorithm is required, this algorithm must be specified. Other constraints may be considered in a specification because the requirements impose to use given data structures, APIs, tools, etc, existing in the real world. Model-oriented specifications are useful because it is not easy to specify all the behavior of a system only by using property-oriented specification. Some application domains make model-oriented specification more appropriate than property-oriented specification. In particular, it is important to note that many of the existing frameworks for the specifications of distributed and reactive systems are model-oriented. For example, (i) automata based languages such as SDL [IT93], Estelle [Estelle89], (ii) process algebra based languages such as CCS [Milner89], CSP [Hoare85], Lotos [Lotos87], and (iii) Petri-Nets [Reisig85] are all model oriented specification frameworks that have been used intensively in the context of distributed and reactive systems.

2.7.3.2 From Models to Executable Specifications and Prototyping

Because of the constructive nature of a model-oriented specification, a natural trend is to use a model as an executable specifications (and execution based validation). Through execution the concrete work related to validation is expected to be easier than through reasoning (Note that some behavior specification and validation frameworks, such as Lotos [Lotos87] are intended to support both execution and reasoning). An advantage of an executable model is that it may be used to build a prototype of the system by reusing the execution / simulation environment as-is or somewhat extended. Though executable specifications can be used as a basis for prototypes, on the other hand, a prototype can not be considered as an executable specification. Prototypes typically serve to explore ideas and to verify decisions. A prototype requires usually only a part of the system to be generated in whatever way that seems appropriate. Executable specifications, on the other side, form the basis for design and implementation. And as such, they must describe the system functionalities in a rather complete way. So, an executable specification can be viewed as an unusual way to build a prototype of a system through rigorous modeling and simulation activities and not by e.g. hacking a quick implementation as it is possible in general. Hayes and Jones tend to consider that executable specifications are always prototypes, and that because it is relevant to draw a distinction between specification and prototyping, it is difficult to consider an executable specification as a true specification.

> According to [Hayes et al.90] : an executable specification is very useful to give users a feel for the system. Here executability is important but the function is that of a prototype rather than a specification. The executable prototype is typically considerably more detailed in describing how to compute, as opposed to the specification of what to compute, and that perhaps much of what is described in the literature as executable specifications would be better classified as rapid prototyping – a valuable area in its own right. The plea is that the positive advantages of specification should not be sacrificed to the separable objective of prototyping.

We tend to agree with the statement that a clear separation between specification and prototyping is to be observed. However, that does not mean that these issues are incompatible in the sense that they

could not be considered within a same behavior specification and validation framework. Our point is merely that issues related to specification and issues related to execution, simulation and prototyping should not be merged and / or confused. Indeed to build a prototype one may typically include some algorithmic descriptions exhibiting a lot of details not relevant in the context of a specification. So, if the distinction is clearly made and observable between the different issues (e.g. apparent in the behavior notation) it seems that there is no problem to use executable specifications and then to build prototypes from these executable specifications. It will appear more clearly in what follows and in particular in chapter 3 that our work typically fits into this hypothesis of work.

2.8 Specifications for Distributed and Reactive Systems

Any modeling approach is always designed to capture a specific family of real world phenomena. To this end some mental constructions or abstractions are used to fit this phenomena best. The systems we are interested in, i.e. distributed object frameworks can be classified in the family of distributed and reactive systems. They are obviously distributed systems. In addition their behavior is of reactive nature because such systems must continually react depending on their state to stimuli coming from their environment. Reactive systems can be opposed to transformational systems where the behavior can be specified only in terms of the transformation from inputs to outputs. It turns out that reactive systems are notoriously more difficult to handle than transformation systems. They are typically composed of distributed and thus independent parts executing concurrently. This implies that there is no global control that is exercised on the system. Distributed and reactive systems can in fact be considered in the larger family of *concurrent systems*. The mathematical theory of how to model and reason about concurrent systems is still alive and unsettled. In fact, because the field is so large, it is doubtful whether a single *unified* theory of concurrency is possible. In section 2.8.1 the important feature of nondeterminism in distributed and reactive systems is emphasized. Historically, Dijkstra language of guarded commands was one of the first models for concurrent computation [Dijkstra76]. Then, Hoares's communicating sequential processes (CSP) [Hoare85] and Milner's calculus of communicating systems (CCS) [Milner89] can be viewed as extensions or refinements of the original framework of Dijkstra's guarded commands. From these first frameworks devised to model distributed and concurrent computation, we can already make the important distinction between two families of models. Those that put the emphasis on data (section 2.8.4) and based on the declarative specification of actions. And those that put the emphasis on control (section 2.8.3) and based on control and communication abstractions, e.g. processes, automata, channels, etc. Note that it is interesting to see this distinction in the light of ODP viewpoints, and with respect to the fact that we are interested in an executable and functional behavior model. This allows in section 2.8.4.1 to conclude about our preferred approach. Finally for the sake of completeness, section 2.8.5 lists some well known models that were examined because of their relevance with respect to the specification and validation of distributed and reactive systems.

2.8.1 Nondeterminism

Nondeterminism is a natural consequence of distribution. In a distributed system, actions can execute concurrently (concurrency), actions may execute in an undetermined order (unordering), or actions can be selected for execution in a non-determined way (choice). As a result, the overall behavior of the system can not be uniquely determined a priori. Nondeterminism is a fundamental property of distributed and reactive systems, any behavior model in the context of distributed object frameworks must support the modeling of nondeterminism. Two approaches are possible : (i) Process / Control Oriented Modeling that typically uses explicit control abstractions (e.g. concurrency constructs) to introduce nondeterminism, or (ii) the Declarative Specification of Actions Model that directly models the nondeterminism and thus does not require the use of explicit control abstractions. Before

considering these two approaches, it is worth to introduce the concept of *transition system* that is a conventional choice to represent the low level operational semantics for any specification language, as soon as its semantics is defined operationally. This is obviously the case for executable specification.

2.8.2 Transition Systems

In this section the notion of *transition* and *state* are defined. Putting this two notions together defines in its turn the concept of *transition system* which can be viewed as a low level representation for the behavior of any distributed system. Transitions in the system are its atomic actions, i.e. steps that can be observed in a single and coherent phase. The configurations of the system between its atomic transition steps define naturally its states. The overall behavior of the system can be defined either by all the sequences of transitions or states the system can go through from a given initial state s_0. This defines the concept of *transition system*. Though the behavior of a system can always be given as a transition system, such a representation is rather impractical because it is too low level. Most of the time transition systems are used by verification tools as a back-end representation [Fernandez et al.96a]. Because specification is a human activity the availability of higher level abstractions is mandatory to allow the specification of state and transitions – even if validation is typically performed at the low level representations based on transition system.

2.8.3 Using Control Abstractions (Process Oriented Model – POM)

In many notations dedicated operators are introduced to model the causes of nondeterminism, e.g. concurrency or nondeterministic choice operators. These operators are applied to some rather sophisticated control abstractions. For instance, processes are used in process calculi such as [Lotos87], automata are used in automata based specification languages such as SDL [IT93] and Estelle [Estelle89], and synchronization structures are used in Petri-Nets [Reisig85]. Such control structures are intended to give a suitable partitioning of the control state of the system into well identified and manageable pieces. Actions are typically organized on such control structures, i.e. actions are specified based on the local (control) states that are defined in each process / automata in the system. Following [Jarvinen et al.91] such models can be qualified as process oriented models (POM), just because transitions are defined and organized based on some notion of process. In fact, a more general terminology would be to use the term *control oriented model*, since the emphasis is put on control abstractions in general. Note that the work consisting to add new abstractions to capture new real world phenomena (and in particular control and communication issues) as they become relevant to model has been pursued. Since mobility is nowadays becoming an important technology in distributed applications, new modeling frameworks have been devised to allow to model and to reason about mobility. For instance, the π-calculus [Milner91] is intended to support the modeling of interacting agent systems whose configuration is continually changing.

2.8.4 Declarative Specification of Actions (Data Oriented Model – DOM)

In this approach, each action is declared one by one. Each action is self contained in the sense that it contains both a specification of the conditions required for its activation, a specification of its effects, and a specification of any other constraint that it may observe. It is important to note the key role played by the underlying data state of the system. Effects are typically specified as data state changes, and constraints are assertional conditions on data states that have to be verified at well identified places, e.g. pre- and post-conditions or general assertions. The enabling condition is based on a condition related to the data state. However it may also include a triggering event used to model interactions of the systems with its environment. For instance, a client that makes a request

on a server object, or more basic things such as data state changes. Since the configuration of the system is defined only w.r.t. data abstractions, such models are also referenced to as data oriented models [Jarvinen et al.91]. Following Dijkstra's language of guarded commands, two interesting examples of data oriented models have been devised especially for formal reasoning : the temporal logic of actions (TLA) [Lamport94] and the Unity logic [Chandy et al.88]. Such models are limited to the specification of transitions, i.e. to the atomic actions that are allowed to occur in the system. First order predicate logic is used with a minimal number of temporal logic extensions to support the specification of fairness constraints.

2.8.4.1 DOM, POM and Executable Specifications

It is important to note that DOM and POM are not to be opposed. A DOM is merely more abstract and is intended to capture a minimum of constraints with respect to the control issues. In contrast, a POM aims precisely at capturing such control issues simply because at some time it is relevant to specify control issues. Because a DOM is mostly concerned with the data in the system and processing on this data, it is clear that a DOM can typically be used in the context of ODP to specify information models. In contrast, POMs are typically oriented towards engineering issues, i.e. control flows in the system, communication channels, physical distribution, mobility issues, etc. For reasoning based validation both POMs and DOMs have received and are still receiving a significant attention from the research community. Interestingly, for execution based validation most executable specification frameworks have been traditionally based on POMs. This is probably justified by the fact that from a POM an execution model is straightforward to derive. In fact, POMs very often support code generation, e.g. code generation tools exist for both SDL and Estelle. Another reason is also that control abstractions such as finite state machines are simple, well known and well accepted abstractions with significant tool support. Therefore, they are often the default abstraction used, even if it is not the more adapted. In particular, to model functional issues it has been shown that a DOM was more adapted. Because of the declarative nature of the resulting specification, a higher degree of abstraction and expressiveness is obtained. Thus, a DOM does not over-specify functional requirements with unwanted control and communication issues (or engineering viewpoint issues in general). However, to support execution, it is clear that a DOM requires additional work that consists of defining a precise execution semantics. From the perspective of the underlying semantics, there is a direct correspondence between DOMs and POMs. Both define an abstract machine, i.e. some form of transition system (states and possible transition between states). The difference is only a matter of presentation. A DOM typically defines for each action which changes upon state can be expected, whereas, a POM gives for each state which actions are possible. Note that this reflects the duality that has been already observed in section 2.4.1 between *state* and *behavior* as stated in the ODP foundations [Rm odp2].

2.8.5 Some Existing Specification Approaches

This section describes some well known specification approaches typically used in the modeling of distributed and reactive systems.

2.8.5.1 Dijkstra Guarded Command Language

Dijkstra guarded command language (GCL) [Dijkstra76] is probably one of the oldest specification framework for concurrent systems. It uses a nondeterministic construction to help free the programmer from over-specifying a method of solution. GCL has basic arithmetic ($a \in \mathbf{A}$) and boolean expressions ($b \in \mathbf{B}$), as well as two syntactic sets : the set of commands ($c \in \mathbf{C}$), and the set of guarded commands ($gc \in \mathbf{GC}$). The abstract syntax is given by these rules [Winskel94] :

$$c ::= \textbf{skip} \mid \textbf{abort} \mid X := a \mid c_0; c_1 \mid \textbf{if}\ gc\ \textbf{fi} \mid \textbf{do}\ gc\ \textbf{od}$$

$$gc ::= b \longrightarrow c \mid gc_0 \mid\mid gc_1$$

The constructor used to form guarded commands $gc_0||gc_1$ is called alternative (or “fatbar”). The guarded command typically has the form :

$$(b_1 \longrightarrow c_1) \mid\mid \cdots \mid\mid (b_n \longrightarrow c_n)$$

In this context the boolean expressions are called guards, the execution of the command body c_i depends on the corresponding guard b_i evaluating to true. If no guard evaluates to true, at a state, the guarded command is said to *fail*. Otherwise, the guarded command executes nondeterministically as one of the guarded commands c_i whose associated guard b_i evaluates to true. $gc_0 \mid\mid gc_1$ introduces nondeterminism, i.e. such a guarded command can execute like gc_0 or like gc_1. As an example the Euclid’s algorithm for the greatest common divisor of two numbers allows a particularly concise formulation using Dijkstra GCL :

$$\begin{array}{l} \textbf{do} \\ X > Y \longrightarrow X := X - Y \\ \mid\mid \\ Y > X \longrightarrow Y := Y - X \\ \textbf{od} \end{array}$$

2.8.5.2 TLA and Unity

The temporal logic of actions (TLA [Lamport94]) and the Unity logic [Chandy et al.88] are approaches especially targeted towards formal reasoning about specifications of distributed algorithms. In TLA and Unity, rigorous reasoning is considered as the only way to avoid subtle errors in concurrent algorithms. The reasoning is intended to be as simple as possible, by making the underlying formalisms simple. In TLA, both distributed algorithms and properties to check for their correctness are specified by formulas in a single logic. The approach to check the correctness consists to check that the algorithm formula implies the property, where *implies* is ordinary logical implication. Temporal logic is a well known way to specify and reason about sequences of states (a state is defined by an assignment of values to variables), and the semantics of temporal logic is based on infinite sequences of states (also called *behaviors*). Because the execution of an algorithm can be viewed as a sequence of atomic steps, each step producing a new step by changing the values of one or more variables, an execution is thus the resulting sequence of states, and the semantic meaning of an algorithm is the collection of all its possible executions. This makes temporal logic a suitable way to specify and reason about algorithms. Note that both TLA and Unity can be classified as data oriented models, their specification framework used is based on the declarative specification of (atomic) actions. They are probably the more significant examples, of data oriented models with a validation support based on formal reasoning.

2.8.5.3 CCS, CSP, Lotos

CCS [Milner89], CSP [Hoare85] are the two classical theories to specify concurrent systems based on communicating processes. Lotos [Lotos87] results from the coupling of CCS and CSP. Lotos has inherited from the operational nature of CCS. This allows to reuse the concept of bisimulation equivalence, which can be used for instance to check that a refined model is bisimilar with a specification.

Different languages for processes and different notions of equivalence relations have lead to different process algebras. In the context of CCS, Milner has defined two important equivalence relations : the observational equivalence and the bisimulation equivalence. CCS is an algebraic language used to describe the synchronous behavior of processes as terms of a calculus. The semantics of CCS is defined by inference rules representing how transitions in the system can be performed. These rules allow to define (i) equivalence relations between behaviors and (ii) a calculus allowing to reason about processes based on a set of equations. This link between an equational theory and a behavior equivalence relation, based on an operational semantics, was first established by Milner. The equational laws are sound in the sense that if $p_1 = p_2$ is proved using these laws, then indeed they are behaviorally equivalent i.e. $p_1 \equiv p_2$, with respect to e.g. bisimilarity. These laws are complete for finite state processes. The equational laws are the basis for the constitution of the process algebra. In addition there exists logical characterizations of these equivalence relations, that is the *Hennessy-Milner Logic*. Two processes are bisimilar if and only if they satisfy the same assertions in a little modal logic.

2.8.5.4 SDL

SDL [IT93] belongs to the family of automata based specification languages. Processes are specified using the extended finite state machine (EFSM) abstraction. SDL is one of the more mature formal methods used in the telecommunications domain. There are commercial tool-sets available for SDL that facilitate design, debugging and maintenance. If the descriptions of the EFSM-actions are written in a C-like language, then the specification may be compiled into C. There are many ways to describe a SDL model for a given behavior. Process state are represented by an explicit state identifier and by other persistent data store present in the process. Similarly, events have an explicit identifier and optional data parameters. With both state and events, information can be built into either the identifier or the associated data. This flexibility can be used to make explicit (by using state and event identifiers) a part of the model not only to meet formal goals but also to meet informal goals such as making the model more intuitively understandable to the target community.

2.8.5.5 Petri-Nets

The basic Petri nets specification framework [Reisig85] is based on graphs of places and transitions interconnected by arrows. Tokens move from place to place according to well defined rules. When no more move is possible the system is said to have reached a deadlock state. Petri nets are relatively simple to understand and well suited to the precise description of the control flow issues. In particular, Petri nets are a good framework to specify and check for the correctness of synchronization problems in the system's processing. However, their use for general requirements specification is less obvious. In particular, Petri nets are probably too low level for most applications. Many extensions of the basic place and transition model have been devised. In particular extensions increase the modeling power of Petri nets by adding data modeling abstractions. For instance structured tokens become complex data structures based on abstract data types definitions. Additional structuring capabilities have also been proposed [Biberstein et al.96] in the form of object oriented extensions to structure the specification of complex systems allow reuse, etc.

2.8.5.6 π-calculus

The π-calculus represents the improved next generation of process oriented models devised because of the failure of previous generations of process calculi (e.g. CCS) to support algebraically full mobility among processes. The π-calculus is a way of describing and analyzing systems consisting of agents which interact among each other, and whose configuration or neighborhood is continually changing.

So, the π-calculus fits in the family of formalisms for processes without the restriction of a finite fixed initial connectivity. A tutorial on the π-calculus can be found in [Milner91].

2.8.5.7 Esterel

ESTEREL [Berry et al.92] is a language with a precisely defined mathematical semantics, for programming the class of input-driven deterministic reactive systems, those that wait for a set of possibly simultaneous inputs, react to the input by computing and producing outputs, and then wait for new inputs. ESTEREL is based on the "synchrony hypothesis", which states that every reaction of the system to a set of inputs is assumed to be instantaneous with respect to the environment. In practice, this comes to the same thing as requiring that the environment of the system is invariant during every reaction, or equivalently that reactions are atomic. The programming model in ESTEREL is the specification of components, or modules, that run in parallel and that can be built along hierarchical structures. Modules communicate with each other and with the outside world through *signals*, which are broadcast and may carry values of arbitrary types. Consistent with the synchrony hypothesis, the emission and the reception of signals is considered to be instantaneous. Pre-emption operators and nesting permits coding behaviors such as interrupts and timeouts, and provide for precise and unambiguous description of complex responses to input events. ESTEREL allows only deterministic behaviors to be specified : the inputs to any reaction and the current values of variables fully determine the outputs emitted in each reaction. Instantaneous broadcast communication, parallelism and pre-emption, present in most synchronous languages, are structuring tools for programming convenience and simplify reasoning about reactive systems. They preserve determinism and do not incur any run time overhead (the compiler automatically performs the complex interleaving between parallel modules and all internal communication is compiled away). ESTEREL fully supports "logical concurrency" : logical concurrent tasks can be described as parallel modules, while the implementation produced by the compiler is purely sequential. The ESTEREL compiler generates a single deterministic finite state machine that yields a predictable and efficient implementation. ESTEREL has also been found very interesting because it can be used to check for real time properties [Jagadeesan et al.95].

2.8.5.8 Reactive Systems Specification in Z

The **Z** notation [Spivey89] combines abstract data modeling and a mathematical toolkit based on the set theory and on first-order predicate logic. **Z** features very modest structuring conventions. Because of the very restricted structuring conventions, any practical use of **Z** typically adds some structuring and naming conventions. These may be used to specify system states and valid state changes for instantaneously responding reactive system. For modeling reactive behavior in **Z**, a conventional approach [Ardis et al.96] is to model atomic events using separate **Z** schema, with pre-conditions guarding against illegal operations, post-conditions specifying the outcomes. For simple specifications some properties can be derived by hand, and directly, from the specification. **Z** is primarily a requirement language, for abstract data types and does not offer a modeling languages; it therefore scores poorly on the criteria of simulation / execution.

2.8.6 Reasoning Problems in the Context of Reactive Systems

As noted in [Ardis et al.96], while technology is available for reasoning with **Z**, such verification is not a process that can be routinely automated. More sophisticated environment based on **Z** and related notations are becoming available (e.g. the B-Method [Abrial95]). These offer support for both reasoning and implementation, but the limited industrial use of such technology for reactive systems makes it difficult to evaluate their utility for large-scale projects. The work on TLA performed by Lamport and others [Lamport94] lead to the same conclusions about reasoning techniques in the

context of distributed and reactive systems : The formal verification of large specifications is a difficult and rarely attempted task; it seems premature to draw any conclusion about its practicality. In addition, the work on mechanically verifying, e.g. TLA formulas is preliminary. So far, only simple examples have been completed. Recent work have concentrated on provers, and in particular in the development of convenient user interfaces to manage proofs [Paulson93, Dowek et al.91]. This is felt as the main prerequisite for practical systems to become available. As an conclusion it can be stated that for the specification and analysis of distributed and reactive systems, execution / simulation still appears to be an essential technique.

2.9 Conclusion

This chapter has introduced the problem of behavior modeling for distributed object frameworks. Its contribution is an analysis of the problem, proposed approaches, etc.. From this analysis, the salient features of the proposed functional behavior modeling framework can be listed as follows :

1. The behavior model should follow an approach based on data oriented modeling, and declarative specification of actions. That is from an OO-model including powerful information modeling such as roles and relationships, (functional) behaviors can be specified in an abstract and expressive way. In addition a declarative approach facilitates evolution and allows incremental specification.

2. In the ODP perspective both information and computational ODP viewpoints specifications can be used in a complementary and thus optimal way in terms of the effort involved.

3. Between inspection and reasoning, executable specifications provide a satisfactory compromise. In the context of distributed and reactive systems simulation is particularly appreciated.

This analysis and resulting behavior modeling framework have been addressed in the following publications where the author has contributed : [Sidou96, Sidou et al.96a, Eberhardt et al.97b, Sidou97b, Sidou97a, Sidou et al.95b, Sidou95, Sidou et al.96b, Sidou et al.95a].

Though we are more concerned with the modeling of distributed object frameworks based on information and computational viewpoint specifications, this problem can also be envisioned in the enterprise viewpoint level or as considered in the OMG on the basis of business object models. The following note discusses our thinking with respect to this issue.

Note on Enterprise Viewpoints Models and Business Object Models Enterprise models focus on the *role* of the distributed system within the organization. So an *enterprise model* defines the purpose, scope, and policies of an ODP system. It is an object-based model, and as such it consists of a set of interacting objects (in that case enterprise objects) participating in actions. As such there is no obstacle against the suitability of the proposed generic behavior modeling framework to cope with enterprise viewpoint models.

Chapter 3

Behavior Model Instantiated : Static and Dynamic Semantics, ODP Issues

3.1 Introduction

The behavior model can be described starting from the very general concept of *action* as it is defined by ODP [Rm odp2]. An action models something that can occur in the system. The action concept procures a suitable unit of specification because it can be used to decompose the behavior of a whole system into well identified and manageable pieces. It can also be viewed as a suitable unit of work and dialogue between involved individuals, e.g. analysts, users, implementors. The objectives of this chapter are :

1. to show how actions can be specified declaratively to allow for expressive, abstract and dynamic modeling of functional behavior. The resulting behavior language in simply called BL.
2. to describe the basic principles used to obtain a precise operational semantics of BL. All these basic principles are gathered altogether in chapter 4, where the *behavior propagation engine* (BPE) algorithm is entirely described.

To this end the plan of the chapter is decomposed as follows :

- Section 3.2 presents the concept of behavior of actions. A first link to the well known ECA-rule model is made. This link will be reinforced as the instantiation of the other issues of the behavior model (especially issues pertaining to the execution semantics) will be considered.
- Section 3.3 gives an overall view of the surface syntax used to specify behaviors in the proposed behavior language (BL). Then, a more abstract description of a behavior as it can be viewed internally in the system (i.e. the behavior execution engine) is presented. This form is the more useful to explain issues of the behavior model.
- Section 3.4 emphasizes on the need for a precise execution semantics.
- In section 3.5 the important concept of *triggering event message* (TEM) is introduced. This concept is important because behavior are always considered as the reaction to the occurrence of such TEMs. Section 3.5.2 considers TEMs with respect to ODP viewpoint issues. In particular, it is shown how by using on one hand information viewpoint messages (IVP-TEM) and on the other hand computational viewpoint messages (CVP-TEM), it is possible, in a first step, to consider separately information viewpoint issues. In a second step, computational viewpoint behaviors can be considered as a refinement of information viewpoint behaviors. To illustrate

out point, section 3.5.4 shows some behaviors triggered by IVP messages working on information viewpoint relationship objects. These behaviors correspond to the relationship and role invariants that are automatically generated from a GRM specification. CVP-TEMs and CVP behaviors as well as an example of mapping that can be performed on IVP-TEMs and behaviors is illustrated in section 3.5.5.

- Section 3.6 discusses rule processing semantics in general. The different approaches with respect to the issue of coupling between event occurrences and rule executions are considered.

- In section 3.7 the different reaction semantics that have been identified are considered. Following the ECA-rule model, the 2-phase behavior fetching principle, and the concept of coupling modes are re-instantiated. Finally, the concrete syntax (**`exec-rules`** syntax) within which these issues are specified in BL is presented.

- Section 3.8 describes how behavior fetching is done. Behavior fetching is a very important issue in the behavior model that has to be designed carefully. It does not consist of a mere execution of flat expressions representing conditions of activation for behaviors. Rather it should be considered as the place where behavior dispatching occurs, i.e. the place where the link between event messages and behavior is made. The dispatching process is typically the place where suitable abstractions to cope with polymorphic and dynamic specification of behavior are considered. In particular we show in this section how the role / relationships-based behavior formalization (RBF) paradigm is instantiated. The dispatching mechanism is intended to be flexible and extensible to incorporate if the need arises other abstractions. To this end, it is explained why the behavior fetching is separated into two phases (behavior scoping and filtering), each phase fulfilling a well identified role. In particular customization is done in the behavior scoping phase by injecting specific behavior scoping functions. The important concept of behavior execution context (BEC), which results from the behavior scoping phase is also established in this section.

3.2 Behavior of Actions and the ECA-rule Model

Because the behavior model follows a decomposition into actions, specified behaviors are behaviors of actions or action behaviors and a behavior specification is a set of action behaviors given declaratively, i.e. one by one in an independent way. Declarative specification is very useful because it avoids to consider the complexity of the whole system, especially at early stages of its requirements analysis. Indeed, it can be viewed as a way to make this analysis feasible. Each action is described by its behavior properties. Several kind of behavior properties have been identified :

- The enabling condition specifies the conditions under which an action is to occur. It can make reference or not to a triggering event. Indirectly, enabling conditions specify the liveness properties of the system, i.e. they specify the condition under which actions should eventually occur. The enabling condition is decomposed in two parts : a scope part and a guard part (see details below).

- The behavior block is composed of assertions, i.e. a pre, a post and a body in between :
  - The assertions (pre- and post-conditions) capture system requirements in terms of safety properties recorded by the specifier. A pre-condition ensures that a certain condition is met before the action / behavior body is executed. By doing so a certain degree of confidence can be asserted about its future execution. A post-condition ensures that a certain condition is met after the action / behavior body is executed. By doing so, a certain degree

of confidence can be asserted about the fact that the intended effects of the action were actually performed. Because action behaviors are potentially highly distributed, it is allowed to have several actions executing concurrently. In this context the role of assertions is very important, even simple assertions can reveal very useful in order to detect interference between actions.

  – The emulation code (behavior body) is not actually intended to capture system requirements, it is rather an informative algorithmic description of what has to be executed between the pre- and the post-conditions. In addition, it is important to note that behavior bodies are required in order to perform behavior simulations.

- Execution rules specify how the evaluation of enabling conditions and of behavior blocks are to be scheduled. In addition the execution of behavior bodies can be specified to be exercised in a coupled or uncoupled way (see details below).

This declarative specification framework naturally follows the *event-condition-action* ECA-rule model that has emerged as a consensus in the community of production systems. Expert systems [Brownston et al.85] and active database management systems (ADBMS) are two instantiations of production systems. For instance, in an ADBMS rules are used to support the active functionality.

Basically, the ECA-rule model consists of an event expression, one or more conditions, an action to be performed, and a set of additional attributes. Explicit specification of these components of a rule provides maximum flexibility and expressiveness, although these components have been packaged in different ways in the literature [Hanson et al.92]. It is clear that the ECA-rule model and its corresponding well established basic constituents is sufficiently general and complete so that it is naturally retrieved in any declarative rule specification model, including our behavior model. However, it is worth to note the following specificities of BL with respect to with the general ECA-rule model :

- Action in the ECA-rule model corresponds to the behavior body in our behavior model.

- An additional use of assertions is made that is required for the purposes of specification and validation.

- The set of additional attributes is dedicated to the specification of execution rules, i.e. fetching phase and coupling mode. Execution rules play a very important role as soon as a precise semantics has to be defined, which is a prerequisite in order to accomplish any kind of validation based on executable specifications.

3.3 Notation for the Behavior Language

Here is the notation used in our behavior model[1] :

$$define-behavior \triangleq (\texttt{define-behavior}\ \begin{array}{l} label-spec \\ scope \\ when \\ exec-rules \\ pre \\ body \\ post) \end{array}$$

[1] This notation is parenthetic because the language mapping and the underlying execution environment are based on the *Scheme* programming language.

This notation is user oriented and is typically used by the behavior specifier when a behavior definition is entered into the system. Another representation is the internal representation that is used by the behavior propagation engine at execution time. It can be described with the following record :

$$behavior \triangleq \ll label, scope, when, pre, body, post, exec-rules,$$
$$def-beh-lno, pre-lno, body-lno, post-lno, src-file \gg$$

Note that the fields after the $exec-rules$ field are not very important with respect to the behavior model. They are used in the execution environment for debugging purpose. Their importance will appear more clearly in chapter 5.

3.4 Necessity for a Precise Execution Semantics

A declarative framework allows to specify behavior in a piecewise way. As a result, independent parts of a specification may be done independently. Moreover, additions and modifications to the specification can be done incrementally. This flexibility is a very important feature for specification. However, the price to pay to this flexibility is that difficulties may appear for models of significant size. The difficulty is to ensure consistency of a whole specification. In other word, one important point of the validation phase is to check that isolated action behaviors compose smoothly when considered altogether, and that finally user requirements about the whole system are met. Since the approach used to perform validation is based on executable specifications, a very important point is that the way behavior executions are performed is clearly stated. In other words, the definition of a **precise execution semantics** is essential. The reason is that even for relatively small behavior specifications, behavior executions can be complex and unpredictable. In chapter 4 this semantics is defined operationally by an algorithm called the behavior propagation engine (BPE) algorithm. Basically, the BPE is a forward search inference engine, i.e. it performs behavior execution steps until saturation, i.e. nothing remains to be done. Note that because of the declarative nature of BL, it is natural to base the operational semantics of the BPE on previous work done in the context of rule production systems, e.g. expert systems and in particular active database management systems (ADBMS). The rule production system community has produced the more advanced operational semantics for declarative rule based specification languages. The proposed behavior propagation engine reuses and adapts such well established concepts. Survey papers about rule processing semantics in particular in the context of ADBMSs are [Hanson et al.92, Dittrich et al.95]. A significant ADBMS implementation where such a sophisticated execution semantics is used is the SAMOS system [Geppert et al.95]. An again more sophisticated execution semantics is the one used in HiPAC [Chakravarthy et al.89] from which the BPE is the more inspired. Though the HiPAC system has only been partially implemented, probably because it is difficult to implement the semantics in the context of an ADBMS, the principles of the HiPAC rule execution semantics has been found particularly adapted to implement the BPE.

3.5 Triggering Event Messages and Behavior

3.5.1 Principle

The notion of event message is very important because a behavior is always a reaction to the occurrence of an event. That is the reason why event messages are also called triggers. Events are represented as messages that are sent to the system either from its environment or from itself, i.e. from a behavior body. For instance, each step of a scenario consists of sending one or several event messages to test the system. Such original input stimuli can be used to model a client ordering for

a service, a state change, a fault occurring in a device, etc. In the end, the BPE can be viewed as a message processor.

3.5.2 Event Message Levels with respect to ODP, IVP & CVP

Different levels of event messages can be distinguished. This distinction follows the separation already identified by ODP viewpoints, in particular with respect to information and computational viewpoint issues. In ODP, the computational language is very prescriptive with respect to the specification of interfaces. In contrast, it does not constrain the behavior of computational objects. This behavior may be specified by any appropriate means [Najm et al.95]. One possibility is to consider each computational object as a system and to apply the viewpoints recursively to it. A computational object is decomposed into smaller computational objects, until basic constituents are reached. It turns out that information objects tend to reappear naturally as basic constituents of computational objects. So in the end, though this is not mandatory, the behavior of computational objects is specified with information models. Interestingly, this allows to reuse information specifications, and correspondences between information and computational model of a system become more apparent. A typical illustration of this approach is [G851 0196], where at the computational viewpoint level the proposed semi-formal behavior notation template makes explicit references to static and dynamic schemas specified at the information viewpoint level.

3.5.2.1 IVP and CVP Messages

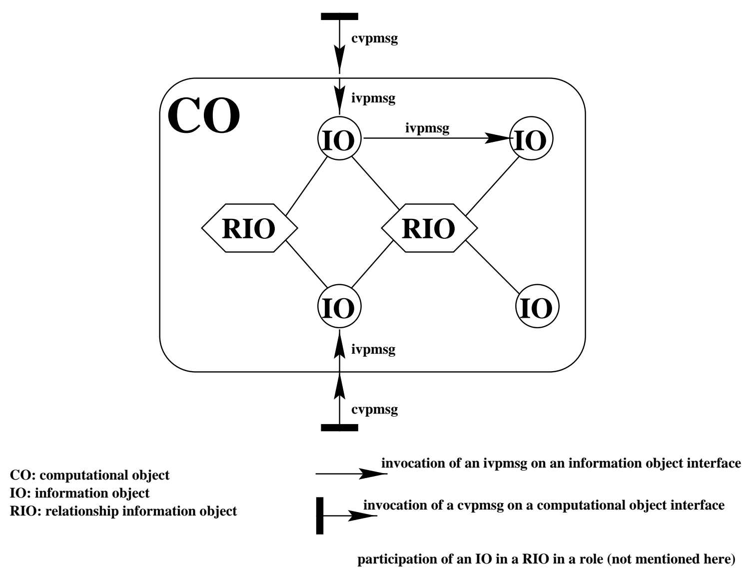


Figure 3.1: IVP and CVP Messages Levels.

Information viewpoint (IVP) messages model the set of basic operations needed to manipulate the underlying information of the system. Computation viewpoint (CVP) messages represent the messages exchanged between the potential units of distribution. Typical examples are CMIS or CORBA-IIOP messages as they can be send and received between a manager and an agent or (respectively) between a client and a server. The important point to note here is that most of the behaviors can be specified as reactions to IVP messages. It is at the IVP level that the core semantics of the system can be captured, e.g. what are the state transitions that can occur on the underlying information objects. Behaviors

associated to reactions to CVP messages are less interesting, they consist either to send CVP messages to composite computational objects or to send IVP messages to constituent information objects. Figure 3.1 illustrates such mappings between the CVP and IVP message levels.

3.5.3 Generic Set of IVP Messages Used

A generic set of IVP messages have been defined. These IVP messages should be sufficiently general be able to manipulate information objects comforming to any information model. At least they have been used for information objects and relationships specified with TMN information models. That is with GDMO and GRM templates. As shown in figure 3.2, the generic set of IVP messages used is a set of records with corresponding fields.

$$ivpmsg - create \triangleq \ll inst, inits \gg$$

$$ivpmsg - delete \triangleq \ll inst \gg$$

$$ivpmsg - set \triangleq \ll inst, attr, val \gg$$

$$ivpmsg - add \triangleq \ll inst, attr, val \gg$$

$$ivpmsg - remove \triangleq \ll inst, attr, val \gg$$

$$ivpmsg - update \triangleq \ll inst, attr, fct \gg$$

$$ivpmsg - establish \triangleq \ll ri, role - inits, inits \gg$$

$$ivpmsg - terminate \triangleq \ll ri \gg$$

$$ivpmsg - bind \triangleq \ll ri, inst, role \gg$$

$$ivpmsg - unbind \triangleq \ll ri, inst, role \gg$$

Figure 3.2: IVP Messages.

3.5.4 Behaviors on IVP Messages to Check for GRM Conformance

The conformance with respect to a particular information model is realized simply by plugging into the system a fixed set of behaviors. The role of this fixed set of behaviors is to ensure that the manipulations performed on the information object are done according to the specifications of information objects used. For instance, such behaviors check that role or relationship cardinality constraints specified in GRM are never violated. To illustrate this point, an example of a GRM specification of the relationship `"LCCapacity"` is shown in figure 3.3.

Then behaviors in BL are automatically generated by the system. Among the set of the generated behaviors, some of them are listed in figure 3.4 for the sake of illustration. For instance, according to the GRM specification of the relationship `"LCCapacity"`, the behavior `"LCCapacity-dlc--bind-card"` checks the cardinality of the role `"dlc"` after a bind. In addition, before the bind op-

```
1   LCCapacity RELATIONSHIP CLASS
      BEHAVIOUR LCCapacityBehavior;
      SUPPORTS ESTABLISH, TERMINATE;

5     ROLE lc
        COMPATIBLE-WITH mLinkConnection
        PERMITTED-ROLE-CARDINALITY-CONSTRAINT INTEGER(1)
        REQUIRED-ROLE-CARDINALITY-CONSTRAINT INTEGER(1)
        PERMITTED-RELATIONSHIP-CARDINALITY-CONSTRAINT INTEGER(1)
10    REGISTERED AS { metranRole 1 }

      ROLE dlc
        COMPATIBLE-WITH mDelivLinkConnection
        PERMITTED-ROLE-CARDINALITY-CONSTRAINT INTEGER(0..MAX)
15      REQUIRED-ROLE-CARDINALITY-CONSTRAINT INTEGER(0..MAX)
        BIND-SUPPORT
        UNBIND-SUPPORT
        PERMITTED-RELATIONSHIP-CARDINALITY-CONSTRAINT INTEGER(1)
      REGISTERED AS { metranRole 2 };
20
```

Figure 3.3: LCCapacity Relationship in GRM.

eration occurs, the behavior `"LCCapacity-dlc-bind-support"` ensures that the role `"dlc"` supports the bind operation.

According to the GRM specification of the relationship `"LCCapacity"`, the role `"lc"`, does not support dynamic binding, so the behavior `"LCCapacity-lc-bind-support"` (figure 3.5) prevents any bind operation for the role `"lc"` to occur.

3.5.5 CVP Messages, Behaviors and IVP Mapping

The CVP messages used are based on dynamic invocations and server interfaces available to communicate with computational objects through a CMIS or a CORBA-IDL interface.

3.5.5.1 CVP Messages Used for OSI-SM (CMIS/P)

As an example let us consider the CVP message corresponding to the CMIS M_Set [Cmis] request :

$$moreq - set \triangleq \ll msd, invoke, moc, moi, mode, \\ scope, filter, access, sync, attrvals \gg$$

The field $attrvals$ is a list of $moinv - cmis - set - attr$ record. This record is defined as follows :

$$moinv - cmis - set - attr \triangleq \ll attrid, attrval, modify \gg$$

The corresponding indication message received on the agent side is exactly the same except that it is called *moind-set*. At this point it is worth to show how the agent behavior that is executed when the system receives a *moind-set* message. In particular, it is interesting to see the mapping on IVP messages that is realized. This is shown on behavior 3.5.1, note that the details about the code in *Scheme* are not important. The `map` function prepares the list of the IVP messages corresponding to

```
1    (define-behavior
       "LCCapacity-dlc-bind-card"
       (scope (ri "LCCapacity") (role "dlc") (msg "ivpmsg-bind"))
       (when)
5      (exec-rules (fetch-phase ii) (coupled after-trigger))
       (pre)
       (body)
       (post
        (and (<= 0 (length (ir:fetch-insts (ri) '"dlc")) 1024)
10           (<= 1
                 (length (ir:fetch-ris (inst) '"dlc" '"LCCapacity"))
                 1))))
```

Figure 3.4: LCCapacity Relationship Behaviors Checking for GRM Conformance.

The post-condition checks that the length of the list of object instances participating in the role `"dlc"` is in the interval delimited by the minimum and maximum bounds defining the cardinality constraints of this role in any instance of the relationship `"LCCapacity"`.

```
1    (define-behavior
       "LCCapacity-lc-bind-support"
       (scope (ri "LCCapacity") (role "lc") (msg "ivpmsg-bind"))
       (when)
5      (exec-rules (fetch-phase i) (coupled before-trigger))
       (pre #f)
       (body)
       (post))
```

Figure 3.5: Behavior Checking for LC Role Binding Support.

The pre-condition stops the execution as soon as this behavior is fetched. This prevents any bind operation to be done on the role `"lc"` of any relationship instance of the relationship `"LCCapacity"`.

the *moind-set* CVP message. A *moind-set* CVP message, can result in $ivpmsg - set$, $ivpmsg - add$ or $ivpmsg - remove$ IVP messages. The `let` statement assigns the list of messages built to the local variable `msgs`. Then, altogether the list of messages is sent to and thus executed by the system through the `msgsndl` statement.

Behavior 3.5.1 `moind-set` *agent behavior :*

```
(define-behavior "mset-ind"
  (scope (msg "moind-set"))
  (when)
  (exec-rules (fetch-phase i) (coupled is-trigger))
  (pre)
  (body
   (let ; here the IVP Messages are built.
       ((msgs
         (map
          (lambda (attrval)
            (cond
             ((eq? (moinv-cmis-set-attr:modify attrval) 'replace)
              (ivpmsg-set:make (msg-> moi)
                               (moinv-cmis-set-attr:attrid attrval)
                               (moinv-cmis-set-attr:attrval attrval)))
             ((eq? (moinv-cmis-set-attr:modify attrval) 'add)
              (ivpmsg-add:make (msg-> moi)
                               (moinv-cmis-set-attr:attrid attrval)
                               (moinv-cmis-set-attr:attrval attrval)))
             ((eq? (moinv-cmis-set-attr:modify attrval) 'remove)
              (ivpmsg-remove:make (msg-> moi)
                                  (moinv-cmis-set-attr:attrid attrval)
                                  (moinv-cmis-set-attr:attrval attrval)))
             ((eq? (moinv-cmis-set-attr:modify attrval) 'set-default)
              (ivpmsg-set:make (msg-> moi)
                               (moinv-cmis-set-attr:attrid attrval)
                               (mor:find-default-value
                                (msg-> moc)
                                (moinv-cmis-set-attr:attrid attrval))))))
          (msg-> attrvals))))

     ;; here the ivpmsgs are sent and executed together.

     (msgsndl msgs)

     ;; here the morsp-set is sent to the manager.

     (msgsnd (morsp-set:make2
              `(msd ,(msg-> msd))
              `(invoke ,(msg-> invoke))
              `(moc ,(msg-> moc))
              `(moi ,(msg-> moi))
              (list
               'attrvals
               (map (lambda (av)
                      (mores-cmis-set-attr:make2
                       `(attrid ,(moinv-cmis-set-attr:attrid av))
                       `(attrval ,(Get (msg-> moi)
                                       (moinv-cmis-set-attr:attrid av)))
                       `(errorid 0)))
                    (msg-> attrvals)))))
     )
   )
  (post))
```

Note well that this behavior assumes that the moind-set message is correct, i.e. that no error will occur during its processing. This behavior also assumes single object selection, i.e. no managed object scoping and filtering is used.

3.5.5.2 CVP Messages for CORBA

The CVP message corresponding to the CORBA-DII Request [Omg corba96] is :

$$request - send \triangleq \ll target, request - header, request - body \gg$$

with the value taken by the field $request - header$ being itself a record defined as :

$$\begin{aligned} request - header \triangleq \ll & idl - full - ifname, service - context, request - id, \\ & response - expected, object - key, operation, \\ & requesting - principal \gg \end{aligned}$$

3.6 Rule Processing Semantics

The semantics of a production rule language determines how rule processing will take place at run-time once a set of rules has been defined. Even for relatively small rule sets, the resulting behavior can be complex and unpredictable, so a precise execution semantics is essential. In this section, the general approaches used to define execution semantics of rule-based languages are described. There are a number of alternatives for rule execution, and different rule systems have taken different approaches. These different approaches depend on two key design choices :

1. *conflict resolution* defines the rule firing strategy, i.e. if several behaviors have been fetched, what exactly has to be executed.

2. *rule triggering* defines the way rules are intended to interact with the arbitrary operations[2] that are submitted by users, application programs and rules themselves. More precisely the key point is the *level of integration* between events being sent to the system and related rule executions. One can also speak in terms of *coupling level* between events and executions of rules.

Both points are important, together they determine the resulting rule processing algorithm. If the coupling is weak, the resulting semantics define different forms of *recognize-act cycle* algorithms. This algorithm is very often used in the context of expert systems (e.g. OPS5 [Brownston et al.85]).

3.6.1 *recognize-act* Cycle Algorithm (Weak Event / Rule Execution Coupling)

Algorithm 3.6.1 recognize-act *cycle :*

RECOGNIZE-ACT-CYCLE()

1 $match()$ *// initial match: test rule conditions*
2 **while** $some - rule - match$
3   **do** $conflict - resolution()$ *// select a subset of the triggered rule*
4 $act()$ *// execute the selected rule's action*
5 $match()$ *// test rule conditions*

The event-rule coupling level is said to be weak because rule triggering (which is determined by the testing of rule conditions) is done in a completely independent phase with respect to event occurrences sent when rule actions are executed ($act()$ phase in algorithm 3.6.1).

[2]Note that the abstraction used to represent these arbitrary operations is the notion of triggering event messages presented in section 3.5.

3.6.2 Conflict Resolution Approaches

In terms of conflict resolution different approaches can be used. For instance, in the *recognize-act* cycle algorithm, in the *match* phase, rule patterns are matched against data in the working memory to determine which rules are triggered and for which instantiations (of variables, objects in the working memory). The entire set of triggered rule instantiations is called the *conflict set*. Then, all the variants of the recognize-act cycle algorithm are defined according to which subset of the rules and instantiations are selected for execution in the *act* phase :

- The firing single rule, single instantiation : in the *act* phase only one selected rule is executed for only one matched instantiation.
- The firing single rule, all instantiations : in a single *act* phase the selected rule is executed for all the matched instantiations of the rule.
- The firing all rules, all instantiations also called *set-oriented* firing consists to execute in a single *act* phase all rules for all instantiations. Here one can say that conflict resolution is no more used.

Note well that though the different conflict resolution techniques have been considered in the context of the *recognize-act* cycle algorithm, they remain applicable also in the context of rule processing algorithm with a stronger coupling between events and rule executions. The only difference is that conflict resolution may occur more frequently, just because rule matching occurs more frequently, e.g. at each event occurrence.

3.6.3 Towards Stronger Event / Rule-Execution Coupling

The need for stronger coupling between an event occurrence and rule executions was felt in particular in the context of ADBMSs. In effect, in ADBMSs the database activity (e.g. queries, updates, transactions) tend to be fully integrated with rule executions. This makes a weak event / rule-execution coupling such as the pure *recognize-act* cycle algorithm not appropriate.

Another problem with the decoupling of events and rule executions in the recognize-act cycle algorithm is that cause-effect relationships between execution of rules is more difficult to follow at runtime. The reason is that the causality links are indirectly supported by the data being updated, and not by the explicit sending of events. To summarize there is a need to trigger reactions not only with respect to patterns about the data present in a database or in a working memory, but also on explicit events. In addition, rules are used not only to specify indirect reactions to incoming events. Especially in ADBMSs, rules are used to define the core semantics of higher level events, i.e. transactions. For behavior modeling, behaviors are typically used in the same way to specify the semantics of high level interactions between objects. That is CVP messages such as CORBA-IDL operations, CMIS M_Action, etc.

3.7 Behaviors and Reaction Semantics

As introduced before, a behavior corresponds to a reaction to an event message sent to the system. That is the reason why hereafter the terms reaction and behavior are used interchangeably. Independently of the message level, i.e. IVP or CVP, different semantics can be identified for such reactions. Two important issues are used to characterize reaction semantics : (i) the *fetching phase* and (ii) the *coupling mode*. These issues are specified using the **`exec-rules`** clause of the proposed behavior notation template. Note that, similar concepts can be retrieved in the ADBMS literature [Hanson et al.92]. The fetching phase is called the event-condition (EC) coupling mode and

our concept of coupling mode is called the condition-action (CA) coupling mode. But before these issues can be described, it is first required to introduce a specific kind of reaction semantics : the **`is-trigger`** reaction semantics.

3.7.1 The `is-trigger` Reaction Semantics

Because there is no builtin semantics pre-defined for any event message[3], a first type of reaction semantics is used to define what has to be actually performed when a given event message is sent to the system. Very often this consists of doing simple and basic things. For instance, the execution semantics of a `ivpmsg-set` message consists of making some changes in the data store of the system for a given object and attribute to a given value (see behavior 3.7.1). This reaction semantics is called the **`is-trigger`** semantics just because it defines the direct semantics that can be associated to a triggering event message.

Behavior 3.7.1 *`ivpmsg-set` **`is-trigger`** reaction semantics :*

```
(define-behavior "ivpmsg-set"
  (scope (msg "ivpmsg-set"))
  (when)
  (exec-rules (fetch-phase i) (coupled is-trigger))
  (pre)
  (body (ir:inst-prop! (msg-> inst)
                       (msg-> attr)
                       (msg-> val)))
  (post))
```

3.7.2 2-Phase Fetching

The fetching phase specifies when behaviors are candidate for fetching with respect to the occurrence of the triggering event message. Fetching can be specified to occur immediately (**`immediate / phase-i`**), in that case fetching is done as soon as the trigger is sent to the system. Otherwise, fetching can be deferred (**`deferred / phase-ii`**). One kind of deferred fetching[4] has been defined. A behavior whose fetching phase that is specified as deferred is candidate for fetching only when all the behaviors executing with the **`is-trigger`** reaction semantics have completed. Note that completion means that the execution of the entire behavior block – i.e. **`pre`**, **`body`** and **`post`** – has terminated.

3.7.3 Coupling Mode

Once a behavior has been fetched, another important issue is to determine how its behavior block has to be executed. Here two types of reaction semantics can be distinguished :

[3] That means that the default reaction to an event message sent to the system is to do nothing.

[4] One could imagine other ways to do deferred fetching, e.g. at the end of the other execution phases of figure 3.6 (**`before-trigger`** and **`after-trigger`**). However it is not clear whether such deferred fetching possibilities would actually be useful. In fact, no need has been felt by users of the behavior model for such deferred fetching possibilities. In addition the benefits that would be obtained in terms of simplification of behaviors (in terms of expressions in **`when`** clauses) is difficult to foresee. In addition there is no mention to such deferred fetching possibilities in the production systems litterature.

1. The **`coupled`** reaction semantics specifies that the execution of the behavior block is coupled with the caller, i.e. the behavior that has sent the trigger. An important consequence is that the calling behavior block execution is blocked until all the coupled reactions terminate. Another important consequence is that this results in a cascaded behavior execution model, as soon as new event messages are sent from a behavior body.

2. The **`uncoupled`** reaction semantics specifies that the execution of the reaction is to be done in a thread independent of the execution of the calling thread. The ability to model such reaction semantics is very useful in a distributed environment, because distribution results naturally in independent threads of execution.

To completely define a coupled reaction, one has to specify when the execution of the behavior block is expected to initiate and terminate. For an uncoupled reaction, only the initiation phase is relevant. For both the coupled and uncoupled cases such parameters are specified relatively to the **`is-trigger`** reaction semantics. The execution phase during which behaviors with the **`is-trigger`** reaction semantics are executing is called the **`during-trigger`** phase. Intuitively, this allows to specify what processing is directly associated to the occurrence of an event. Then, it is possible to specify what behaviors are allowed to execute before, during and after the direct processing. As shown in figure 3.6, it follows that the time during which the reactions to a trigger are being executed can be partitioned into three phases : the **`before-trigger`**, the **`during-trigger`**, and the **`after-trigger`** phases.

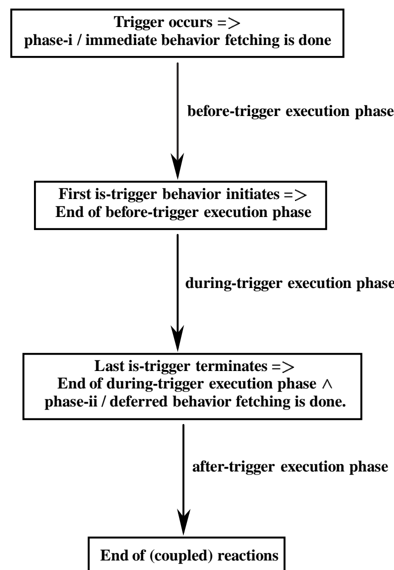


Figure 3.6: Trigger Reaction Phases.

3.7.4 The `exec-rules` Clause

In conclusion, the different reaction semantics are defined using two parameters : the fetching phase and the coupling mode. As indicated before, they are specified in the **`exec-rules`** clause as follows[5]:

[5]In the coupled mode, if only one execution phase is given, that means that the behavior block initiates and terminates in that phase. Note also that **`(coupled is-trigger)`** defines a behavior with the **`is-trigger`** execution semantics.

$$
\begin{aligned}
exec-rules \triangleq\ & (\texttt{\textbf{exec-rules}}\ fetch-phase\ coupled)\ |\\
& (\texttt{\textbf{exec-rules}}\ fetch-phase\ uncoupled)
\end{aligned}
$$

$$
\begin{aligned}
fetch-phase \triangleq\ & (\texttt{\textbf{fetch-phase immediate}})\ |\\
& (\texttt{\textbf{fetch-phase i}})\ |\\
& (\texttt{\textbf{fetch-phase deferred}})\ |\\
& (\texttt{\textbf{fetch-phase ii}})
\end{aligned}
$$

$$
\begin{aligned}
coupled \triangleq\ & (\texttt{\textbf{coupled before-trigger}})\ |\\
& (\texttt{\textbf{coupled before-trigger during-trigger}})\ |\\
& (\texttt{\textbf{coupled before-trigger after-trigger}})\ |\\
& (\texttt{\textbf{coupled during-trigger}})\ |\\
& (\texttt{\textbf{coupled is-trigger}})\ |\\
& (\texttt{\textbf{coupled during-trigger after-trigger}})\ |\\
& (\texttt{\textbf{coupled after-trigger}})
\end{aligned}
$$

$$
\begin{aligned}
uncoupled \triangleq\ & (\texttt{\textbf{uncoupled before-trigger}})\ |\\
& (\texttt{\textbf{uncoupled during-trigger}})\ |\\
& (\texttt{\textbf{uncoupled after-trigger}})
\end{aligned}
$$

3.8 Behavior Fetching / Dispatching (Scoping and Filtering)

3.8.1 Introduction

Inherent to every object model is the mapping of invocations (i.e. triggering event messages) on objects to their behavior. In the context of object oriented systems, *dispatching* is the process of finding the appropriate behavior for a particular invocation. The dispatching mechanism in almost all object models is fixed. In contrast in our behavior model, a radically different approach has been chosen : dispatching is isolated and treated separately from the kernel of the behavior propagation engine. So the dispatching process is viewed as something that has to be explicitly introduced into the system. In other words in the behavior model it is possible to customize the behavior dispatching process. This section describes the principles of the generic behavior dispatching machinery, and also how it has been used. In particular it has been used to provide better and higher level abstractions for behavior integration, such as the Role-based Behavior Formalization introduced in section 2.6. This explicit customization of the dispatching process is based on the concepts of (i) behavior scoping functions (BSF) and (ii) behavior execution contexts (BEC). Both concepts are detailed in section 3.8.3.

3.8.2 Execution Semantics of Behavior Fetching

Behavior fetching always occurs atomically, i.e. always in a single and coherent execution step. Behavior fetching is decomposed into two consecutive sub-steps : a behavior scoping sub-step and a

behavior filtering sub-step. In the proposed behavior notation template scoping is given with the **`scope`** clause, whereas filtering is given in the **`when`** clause. **`Scope`** and **`when`** clauses together define the enabling condition or guard for a behavior to be executed. Though, all could be merged in the **`when`** clause, the scoping phase is clearly separated because it corresponds also (in addition to be part of the enabling condition) to the computation of a key concept of the behavior model : the behavior execution context (BEC). A similar notion exists in the ADBMS literature with the recognized concept of *binding context* of active rules [Hanson et al.92] when rule fetching occurs. This concept has been extended in some way by explicitly associating to it a dedicated sub-step of the behavior fetching phase. This behavior fetching sub-phase is called *behavior scoping*. Another important notion is the associated concept of *behavior scoping functions*. Behavior scoping functions are the functions responsible for the computation of BECs during the execution of the behavior scoping sub-step. Because the set of scoping functions is not hard coded into the system, rather it can be redefined, or just extended in order to define or customize the required behavior dispatching.

3.8.3 Behavior Scoping Functions (BSF) and Behavior Execution Contexts (BEC)

As soon as behavior fetching occurs, a determined set of behavior scoping functions is called. There are two sets of behavior scoping functions, one is intended to be used in the **`immediate / phase-i`** behavior fetching phase. And the other works in the **`deferred / phase-ii`** behavior fetching phase. The reason is that, in the immediate fetching phase, scoping functions compute BECs from a triggering event message sent to the BPE. In contrast, in the deferred fetching phase, behavior scoping functions work from existing BECs, containing the result of the execution of a behavior with the **`is-trigger`** reaction semantics. A behavior execution context (BEC) contains the references to all the required information used during all the execution of a behavior. In particular a BEC references precisely the set of information objects concerned with the execution of a behavior. It is important to note here that :

1. **`when`**, **`pre`**, **`body`** and **`post`** behavior clauses are directly executable clauses implemented as functions of a single argument : the scoped BEC. All the references to objects made by such functions at execution time have to be reachable from the contents of the BEC.

2. In addition BECs and the BPE machinery are completely independent issues. The BPE sees BECs as opaque values, indeed several kind of BECs can be used. In the end, BECs can be used to customize the generic BPE machinery. Or conversely, the BPE machinery is independent to the underlying information model and the information modeling abstractions used. To make this possible the system allows new scoping functions to be injected into it.

3.8.4 Typical BEC for Role / Relationship-based Behavior Formalization (RBF)

Because our universe of discourse (or information model) is concerned with objects and roles / relationships, the implemented behavior scoping functions are naturally brought to compute BECs including references to such entities. However if the need arises, other forms of BECs can also be envisioned. Currently, a BEC is represented as a record defined as :

$$bec \triangleq \ll msg, inst, role, ri, rel, locals, res, err \gg$$

The behavior scoping functions in the current implementation with this BEC definition, use its fields in different ways. The `msg` field is the message sent that caused the BEC to be fetched, this field has always a meaningful value in a BEC. From now on all the other fields are optionally filled in. The `inst` field is what was found to be the target instance of the message. The `role` field is the role

played by the target instance, in a relationship instance (`ri` field) of a given relationship class (`rel` field). The `locals` field can be used to store local variables during the execution of behavior blocks. This may be used to pass information between behavior clauses, e.g. from the `pre` or `body` to the `post`. The `res` field may be used to store a result. The `err` field may be used to indicate that an error occurred and to include informations about it.

3.8.5 Advantages of Behavior Scoping

Here is a summary of the main arguments justifying that the use of BECs, behavior scoping functions and corresponding **`scope`** clause in the proposed behavior notation template is a "good" feature of the behavior model :

- Scopes provide more meaningful specifications of enabling conditions and of the execution context of behaviors. e.g. the role / relationship scope appears more clearly in the separate scope clause. Otherwise all this information would be merged in the **`when`** clause, in a less obvious way. For instance a typical RBF scope is the one of the form :

  ```
  (scope (ri <rel-label>) (role <role-label>) (msg <msg-label>))
  ```

  In a **`when`** clause the same expression would be specified as :

  ```
  (and (<msg-label>:isa? (msg)) ; tests that the message is a <msg-label> record
       (record:features? (msg) 'inst) ; tests that the field "inst" is present
       (not (null? (FetchRis (msg-> inst) <role-label> <rel-label>))))
  ```

  Note that the exact details in the equivalent expression are not important. The important point is that the scope expression gives an immediate idea about the intended parts of behavior enabling conditions. In addition this expression does not include anything for the construction of the BEC that is to be done to evaluate the other behavior clauses, i.e. the pre-, the body and the post-. In this example the BEC should be composed of at least of the message `(msg)`, the target instance `(msg-> inst)`, and a relationship instance `(ri)` such as :

  ```
  (ri) ∈ (FetchRis (msg-> inst) <role-label> <rel-label>)
  ```

- Expressions in **`when`** clauses are simplified : in effect, scopes can be designed to offer more abstract and synthetic expressions for the equivalent conditions that would otherwise be specified in **`when`** clauses. The previous example shows clearly that enabling condition in addition to be more meaningful are also much more simple. In fact they focus more efficiently to the application specific tests of behavior enabling conditions, e.g. values of application oriented fields in the message, values of properties in other objects of a relationship instance, etc.

- Behavior fetching is more efficient : without scopes, behavior fetching would require the evaluation of all the **`when`** clauses of all the behaviors in the behavior repository. Such a repository would be completely flat. With scopes, the behavior repository is organized and can be viewed as a table indexed by scopes as shown in figure 3.7.

- Finally scopes define proper encapsulation units for behaviors. Identification of such encapsulation units is crucial in the perspective of behavior composition, i.e. for the design of reuse artifacts (generic behavior libraries) or any other form of behavior composition, e.g. overriding, sequencing, etc. Though this issue has not been investigated in this work, the principles of scopes, behavior execution contexts and behavior scoping functions could be a good basis for further study in this direction.

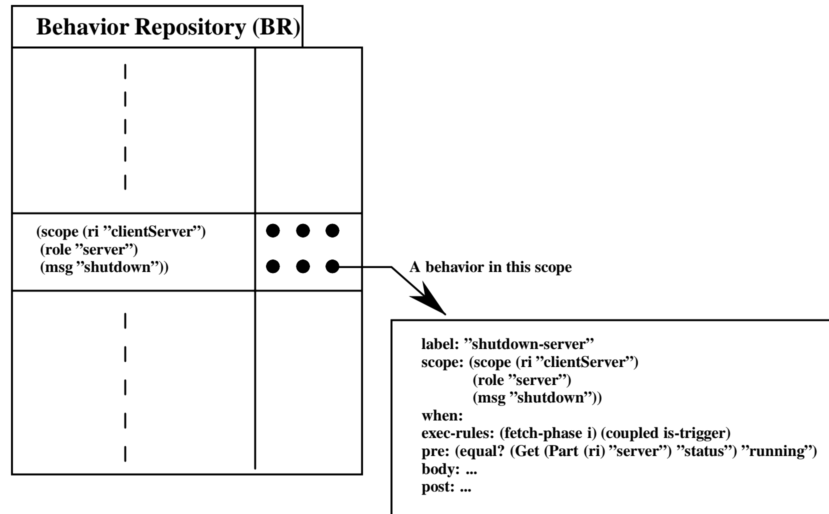


Figure 3.7: Behavior Repository Indexed with Scopes

3.8.6 Behavior Scoping Illustrated

Figure 3.9 illustrates on an example the behavior scoping process that is implemented. The example is based on the simple spanning tree case study presented more in details in appendix C. The initial configuration is shown in figure 3.8. The directed graph is only composed of two nodes and a directed link between them. This is sufficient for the intended illustration about behavior scoping. Let's see the result of behavior scoping (**`immediate / phase-i`**) on the initial message sent to trigger the computation of the spanning tree. The message used is `msg0` defined as follows :

```
> (define msg0 (ivpmsg-set:make 2 "dist" 0))
> (printf "msg0 value is %a\n" msg0)
msg0 value is (%%record ivpmsg-set (inst 2) (attr dist) (val 0))
```

This message is used to initiate the spanning tree algorithm by indicating that the distance of node 2 to the termination node is set to "0". Here node 2 is the termination node. To test and check that behavior scoping **`immediate / phase-i`** works well, one can use the function `bpe:bscope--phase-i`. The result is a list of all the potentially useful `(scope BEC)` pairs (see figure 3.9 line 2) that are to be used for the remaining of the behavior fetching process, i.e. behavior filtering :

- The first scope is based on the message type label only :
  `(scope (msg "ivpmsg-set"))`

  To this scope are associated all the behaviors related to a reaction to the occurrence of a `"ivpmsg-set"` event message. As shown in figure 3.9, line 4, the scoped BEC that will be used in the behavior filtering process contains only a reference to the event message sent.

- The other scopes are more interesting because they illustrate the use of RBF.

  `(scope (ri "linkRel") (role "dstRole") (msg "ivpmsg-set"))`

  – In addition to the message record label, this scope is based also on the target instance that has to fulfill the role `dstRole`. As shown in figure 3.9, line 6 the BECs found by the scoping process with this scope contains a reference to a relationship instance of class `relClass` into which the targeted instance participates in the role `dstRole`.

```
(scope (ri "linkRel") (role "dstRole"))
```

– This scope specifies that the processing of any kind of message is to be promoted in the context of a relationship instance of class `linkRel` provided the target instance of the message fulfills the role `dstRole` in that relationship instance (see figure 3.9, line 11).

```
(scope (ri "linkRel"))
```

– This scope specifies that the processing of any kind of message is to be promoted in the context of a relationship instance of class `linkRel`, without any constraint on the role fulfilled by the target object instance (see figure 3.9, line 16).

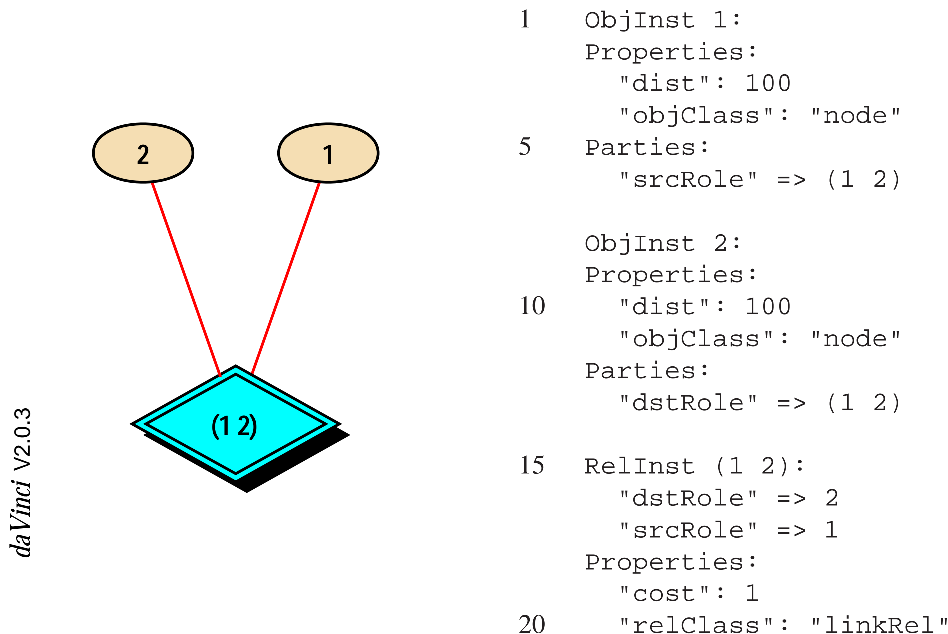


Figure 3.8: Simple Instance Repository.

3.8.7 Behavior Filtering

Once scopes and BECs have been determined by the behavior scoping phase, the filtering phase accesses the behavior repository to get the behaviors corresponding to the scopes found. To facilitate this access the behavior repository is indexed on scopes. This indexing on scopes also improves the efficiency of the retrieval of behaviors. For each behavior its **`when`** clause is evaluated to test additional conditions on anything that is in the scope of the BEC, i.e. anything that is referenced in the BEC. This may concern (i) the event message itself that is usually included in the BEC by the scoping function, and (ii) the state of some underlying data in the system, i.e. scoped objects and relationships referenced in the BEC. Note that the **`when`** clause is evaluated with the scoped BEC as argument.

3.8.8 Behavior Filtering Illustrated

Behavior filtering is illustrated following the sample example based on the simple spanning tree case study used to illustrate behavior scoping. Since behavior filtering is the second and last processing step of behavior fetching, the result of the the behavior filtering is the result of the whole behavior

```
1    > (pretty-print (bpe:bscope-phase-i msg0))
2    (((scope (msg "ivpmsg-set"))
3      (%%record bec (msg (%%record ivpmsg-set (inst 2) (attr "dist")
4                                               (val 0)))))
5     ((scope (ri "linkRel") (role "dstRole") (msg "ivpmsg-set"))
6      (%%record bec (msg (%%record ivpmsg-set (inst 2) (attr "dist")
7                                               (val 0)))
8                    (inst 2) (role "dstRole") (ri (1 2))
9                    (rel "linkRel")))
10    ((scope (ri "linkRel") (role "dstRole"))
11     (%%record bec (msg (%%record ivpmsg-set (inst 2) (attr "dist")
12                                              (val 0)))
13                   (inst 2) (role "dstRole") (ri (1 2))
14                   (rel "linkRel")))
15    ((scope (ri "linkRel"))
16     (%%record bec (msg (%%record ivpmsg-set (inst 2) (attr "dist")
17                                              (val 0)))
18                   (inst 2) (role "dstRole") (ri (1 2))
19                   (rel "linkRel"))))
20
```

Figure 3.9: Scoping Function **`immediate / phase-i`**.

fetching process. This can be tested using the functions (i) `bpe:bfetch-phase-i` that takes a message as argument, and (ii) `bpe:bfetch-phase-ii` that takes a BEC as argument. The result of **`immediate / phase-i`** behavior fetching on `msg0` can be obtained as follows :

```
1    > (define fbres-i (bpe:bfetch-phase-i msg0))
     #<unspecified>
     > (pretty-print fbres-i)
     (((%%record behavior
5        (label "ivpmsg-set")
         (scope (scope (msg "ivpmsg-set")))
         (when #<CLOSURE (bec) #t>)
         (pre #<CLOSURE (bec) #t>)
         (body #<CLOSURE (bec) ...>)
10       (post #<CLOSURE (bec) #t>)
         (exec-rules (phase-i (during-trigger is-trigger during-trigger)))
       (%%record bec (msg (%%record ivpmsg-set (inst 2)
                                               (attr "dist")
                                               (val 0)))))))
15
```

Only one behavior is fetched in the immediate behavior fetching phase. It corresponds to the basic processing of the message `ivpmsg-set`, i.e. that operates the actual update of the attribute `dist` of node 2. To fetch behavior reactions corresponding to the processing of the spanning tree algorithm, **`deferred / phase-ii`** behavior fetching has to be considered because the reactions to `msg0` related to the processing of improvement steps for the spanning tree algorithm, are defined in the deferred behavior fetching phase. A BEC that can be used is stored in the *Scheme* variable `bec0`. Naturally a BEC is never constructed by the user as it is shown below, since BECs are usually computed by behavior scoping functions and then used by behaviors all along their execution. Note that BECs are however available to the user during behavior executions and between behavior execu-

tion steps, so that deferred behavior fetching can effectively be tested as shown just below. Here an example of BEC is constructed "by hand" just for the purpose of the illustration :

```
> (define bec0 (bec:make2 `(msg ,msg0)))
> (pretty-print bec0)
(%%record bec (msg (%%record ivpmsg-set (inst 2)
                                        (attr "dist")
                                        (val 0))))
```

In fact in this example the BEC corresponds to the resulting BEC of the immediate behavior scoping phase. From this BEC, deferred behavior scoping can be performed using the function `bpe:-bscope-phase-ii`. (the result of the scoping phase is not shown because it is identical to what was obtained in **`phase-i`**). In contrast, behavior filtering is more interesting. To test deferred behavior filtering, one can proceed as follows :

```
1    > (define fbres-ii (bpe:bfetch-phase-ii bec0))
     > (pretty-print fbres-ii)
     (((%%record
        behavior
5       (label "dg-st-first-step")
        (scope (scope (ri "linkRel") (role "dstRole") (msg "ivpmsg-set")))
        (when #<CLOSURE (bec) ...>)
        (pre #<CLOSURE (bec) #t>)
        (body #<CLOSURE (bec) ...>)
10      (post #<CLOSURE (bec) #t>)
        (exec-rules (phase-ii (after-trigger)))
       (%%record bec
        (msg (%%record ivpmsg-set (inst 2) (attr "dist") (val 0)))
        (inst 2)
15      (role "dstRole")
        (ri (1 2))
        (rel "linkRel"))))
```

The resulting behavior fetched is the behavior that updates for the first time the distance of the `srcRole` of the link to the termination node. See appendix C to understand how the attribute `iter` is used to identify each occurrence of the execution of the spanning tree algorithm on the directed graph of nodes and links.

3.8.9 Whole Behavior Fetching Algorithm

The whole behavior fetching process is illustrated in figure 3.10. The algorithmic details are described in algorithm 3.8.1.

Algorithm 3.8.1 *Behavior Fetching :*
BPE:BFETCH$(msg-or-bec, scoping-functions, fetch-phase)$
1 *// Behavior Scoping*
2
3 **if** $fetch-phase =$ *phase-i*
4 **then** $scoping-functions \leftarrow bpe : *scoping-functions-phase-i*$
5 **else if** $fetch-phase =$ *phase-ii*
6 **then** $scoping-functions \leftarrow bpe : *scoping-functions-phase-ii*$
7
8 $scope-bec-list \leftarrow ()$

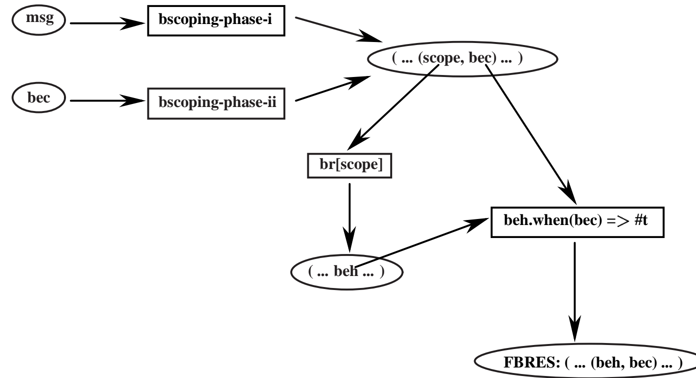


Figure 3.10: Behavior Fetching Process.

9 **for each** $scoping - function$ **in** $scoping - functions$
10 **do** $scope - bec - list \leftarrow \text{Append}(scope - bec - list,$
11 $scoping - function(msg - or - bec, fetch - phase))$
12
13 *// Behavior Filtering*
14
15 $beh - bec - list \leftarrow ()$
16 **for each** $(scope, bec)$ **in** $scope - bec - list$
17 **do** $beh - list \leftarrow br[scope]$
18 **for each** beh **in** $beh - list$
19 **do if** $beh.fetch - phase = fetch - phase \wedge beh.when(bec)$
20 **then add** (beh, bec) **to** $beh - bec - list$
21 **return** $beh - bec - list$

In algorithm 3.8.1 behavior scoping and filtering is done in sequence. At line 3, the set of scoping functions to be used is selected according to the fetching phase. Two sets of scoping functions are available. One for **`immediate / phase-i`** behavior fetching, the scoping functions are stored in the global BPE variable $bpe : *scoping - functions - phase - i*$. And one for **`deferred / phase-ii`** behavior fetching, the scoping functions are stored in the global BPE variable $bpe : *scoping - functions - phase - ii*$. At line 9, the selected scoping functions are scanned and called one by one with the message or BEC given as argument. The important agreement between scoping functions and the BPE is that each scoping function must return a list, of scope-bec pairs, i.e. something of the form :

$$((scope_1, bec_1)(scope_2, bec_2) \cdots)$$

Each such results are appended together to form the whole result of the scoping process in variable $scope-bec-list$. At line 16, the filtering process scans each such $(scope, bec)$ pair. At line 17, the list of behaviors registered with this scope are retrieved from the behavior repository (br). Then this list of behaviors is scanned. In this list only behaviors specified to be fetch-able in the given fetching phase are considered. Then if the **`when`** clause of such a behavior applied on the scoped BEC evaluates to $true$, this behavior along with the BEC are added to the list of results of the behavior fetching process. In the end, the result of behavior fetching, called a FBRES, is a list of (beh, bec) pairs, i.e. something of the form :

$$((beh_1, bec_1)(beh_2, bec_2)\cdots)$$

3.9 Conclusion

The overall contribution of this chapter is to define the principles of the static and dynamic semantics of the proposed behavior modeling framework. This chapter also shows how the approach is generic. ODP viewpoints can be integrated using different levels of triggering event messages. Roles and relationships can be integrated using the generic behavior dispatching mechanisms provided. Details about the contributions of this chapter can be listed as follows :

Static semantics : The static semantics corresponding to the declarative specification of actions is defined. The resulting behavior notation template includes both the pure specification aspects (e.g. assertions) and execution / simulation aspects. An important point is that each issue is given using well identified and separate behavior clauses.

Triggering Event Messages : The important concept of triggering event message (TEM) has been identified. All the processing in the system occurs because a TEM was sent to the system, and behaviors define this processing. Since different levels of processing exist in the context of distributed object frameworks, different categories of TEMs and corresponding behaviors can be distinguished.

For instance, ODP information models can be specified by using IVP-TEMs and behaviors triggered by IVP-TEMs. Then the computational viewpoint model can be defined using CVP-TEMs and behaviors triggered by CVP-TEMs. Typical examples of CVP-TEMs are CMIS or CORBA-GIOP messages. The CVP model can be defined as a refinement of the IVP model by defining in behaviors associated to the processing of CVP-TEMs a mapping on IVP-TEMs. Then, these IVP-TEMs trigger the actual information processing as specified in the IVP model. This approach minimizes the effort required to develop CVP models. That is the reason why it has also been advocated in e.g. [G851 0196].

Dynamic semantics : Another contribution of this chapter is to show that concepts about rule execution semantics that were well established in the context of active database management systems can be incorporated to define precisely the dynamic semantics of the behavior model. As expected a satisfactory, powerful and comprehensive semantics for the declarative behavior language is obtained.

In summary concepts of the dynamic semantics are : 2-phases (immediate and deferred) behavior fetching, reaction semantics for behaviors, **`is-trigger`** reaction semantics, trigger execution phases (**`before-trigger`**, **`during-trigger`** and **`after-trigger`**), coupled / uncoupled reaction semantics.

Behavior dispatching : Another important contribution of this chapter is the generic behavior dispatching mechanisms that is instantiated by the concepts of of behavior execution context (BEC) and behavior scoping function (BSF). In the ADBMS litterature people speak in terms of the *binding context* of active rules to reference the concept of BEC. This concept is intimately linked to the concept of behavior scoping functions (BSF) because BSFs typically compute BECs in the behavior scoping phase. BSFs realizes the link between triggering event messages and behaviors. The important point to note is that this process is deliberately separated from the core behavior model. One can define any appropriate BEC structure and corresponding BSFs and plug them into the system. This allows one to customize the behavior dispatching process as

required. In particular in this chapter, a BEC typically oriented towards role /relationship based formalization (RBF) has been defined. In addition, the way the corresponding BSFs work has been illustrated. Note that from the syntactic viewpoint, to allow this flexibility the syntax of the scope clause in the BL template does not impose any constrain on scope expressions. So scope expressions can be chosen as synthetic and meaningful as wished. In the context of RBF a scope typically references high level dynamic aggregation structures such as roles and relationship labels. Finally a detailed description of the whole behavior fetching process has been presented. The objective was to show more in details how the behavior fetching phases work together and the data structures involved. The behavior scoping phase computes scope-bec pairs. Each scope-bec pair is taken as input by the behavior fetching phase to compute finally a FBRES structure that is a list of behavior-bec pairs. Naturally the behavior represent a fetched behavior and the BEC its execution context that is supposed to contain all the references (e.g. to information objects) necessary during all of its execution.

The principles of the static and dynamic semantics of the proposed behavior modeling framework has been addressed in the following publications where the author has contributed : [Sidou95, Sidou96, Eberhardt et al.97b, Sidou97a, Sidou et al.95b].

Reuse and Composition There is no widely accepted agreement on any reuse artifact that is considered to work universally. The problem finds its roots in the fundamental way people model systems, i.e. by considering on one hand the static schema (state model) and on the other hand the dynamic schema (behaviors). The state model is typically considered first and then behaviors follow. In fact, people do not know how to do otherwise. At the state level, reuse or composition artifacts such as static class-based inheritance have been devised and work without any difficulty because there is no actual problem occurring at the state level, i.e. problem that would break the system. However, at the dynamic level no behavior composition (compatible with state composition) principle has been found. It turns out that at the behavior level the composite systems built often reveals broken, e.g. behavior components are not executed in the right sequence, some synchronization has to be added in an add-hoc manner, etc. This well-known problem is referenced in the litterature as the *inheritance anomaly* [Matsuoka et al.93]. Because of the duality and thus interdependence existing between state and behavior, no valuable approach may emerge as long as no-one finds a compatible way to manage change in both directions (or another modeling approach is found). That is a reason why we have not tried to propose reuse and composition artifacts hard coded in the behavior model, i.e. a syntax, and corresponding static and dynamic semantics. So the core behavior model is not polluted with arbitrary choices committed w.r.t. reuse and composition issues. However, that does not mean that such issues can not be taken into account. Indeed it is possible to define reuse and composition artifacts using the flexibility deliberately provided for behavior dispatching in the behavior scoping phase. This flexibility can be used to define any appropriate BEC and corresponding behavior scoping functions. Though this is beyond the objective of this thesis, this flexibility could probably be used to instantiate most of the reuse and composition artifacts one can think of. Note that in the current implementation simple reuse and composition based on relationship inheritance hierarchy is available.

Chapter 4

Behavior Propagation Engine (BPE)

4.1 Introduction

In this chapter the concepts previously introduced in chapter 3 are integrated altogether. This results in a complete and precise semantics for the proposed behavior notation in the form of an algorithm : the behavior propagation engine (BPE) algorithm. The plan of this chapter can be described as follows :

- In section 4.2, the overall BPE algorithm is described as a classical forward search inference engine.
- In section 4.3, the execution steps or transitions that are performed atomically in a forward search are introduced.
- From section 4.4 to section 4.6 the state / configuration of the system is defined. Such configurations characterize entirely the state of processing of the BPE between each atomic execution step. Only control state issues are considered, i.e. only the data structures used to drive the control flow of the BPE. In particular, these data structures encapsulate the important concepts of behavior execution node / behavior execution tree (BEN / BET), and of trigger execution control (TEC). Note that issues related to the data state are not taken into account because they can be treated and implemented separately from the BPE machinery.
- In section 4.7 the precise processing performed by the set of transition functions on the BPE control structures (i.e. BEN and TEC) is algorithmically described.
- Finally, in section 4.8 the whole BPE algorithm and associated functions are described in their final form. This includes (i) the bootstrapping of behavior propagations (section 4.8.1) from the initial stimuli and the behavior propagation algorithm itself (section 4.8.2). In section 4.8.3, it is shown how the BPE algorithm calls all the functions responsible to execute each behavior execution step that were described in section 4.7.
- Section 4.9 describes the mechanism to allow the user to control more precisely the atomic execution steps of behavior bodies. This includes both the required syntaxes in the behavior language and the required implementation support in the BPE.

4.2 Behavior Propagation – Forward Search

The overall structure of BPE algorithm follows the principle of a classical forward search inference engine, i.e. it performs behavior execution steps until saturation, i.e. nothing remains to be done.

This defines the concept of *behavior execution / propagation*[1]. A *behavior propagation* is a sequence of execution steps performed from the initial state to a state reached at saturation. Let S be the configuration of the system or its state at each step. Let $enabled_steps(S)$ be the function giving the set of enabled steps that are ready to be executed from state S. The overall algorithm of a forward search inference engine is :

Algorithm 4.2.1 *Forward Search Inference Engine :*
BPE:WALK(S)
1 **while** $enabled - steps(S) \neq \emptyset$
2 **do** $S := execute - step(es \in enabled - steps(S), S)$

The basic principle of the BPE algorithm follows algorithm 4.2.1. It is now needed to define more precisely the important notions of execution step and system's configuration. This is done on one hand by describing the data structures that are used to represent the system configuration; and on the other hand by describing what processing is performed by the different execution steps of the BPE.

4.3 Atomic Transitions / Execution Steps

Transitions / execution steps are defined by the atomic pieces of execution. Basically, two kinds of execution steps have been identified :

1. assertion evaluation step is a `pre` or a `post` clause that is evaluated.
2. behavior body evaluation steps correspond to the algorithmic pieces of behavior body code delimited by statements sending event messages. Note that several event messages can be sent. A point in the execution of a behavior body where event messages are sent is called a *blocking point*. At a blocking point, the execution of the behavior body is blocked until all the coupled reactions to the event messages sent are terminated. A point where all such coupled reactions terminate is called a *resume point*. So, in the end an evaluation step in the execution of behavior bodies is either :
   (a) the execution from its beginning to the first blocking point.
   (b) an execution from a resume point to the next blocking point. It is important to note that this includes the evaluation of the behavior fetching corresponding to the triggering event messages sent at the next blocking point.
   (c) the execution from the last resume point to the end of the behavior body.

4.4 States / Configurations of the System

The state or configuration of a system is usually partitioned into a *data state* and a *control state*. Here it is important to note that the data state can be abstracted away. The reason is that the BPE machinery is totally independent on any underlying model used to represent the data state and its associated transitions. Indeed the data state can be represented / implemented by any appropriate means, e.g. a database, a repository in main memory, etc. References to the data state are only needed to implement behavior scoping functions and to code behaviors themselves. Both things are outside of the kernel of the BPE machinery. Therefore in this section, only issues related to the state of control

[1]Hereafter, terms like behavior execution, behavior propagation and behavior walk are use interchangeably.

are considered, i.e. the structures used to represent the state of control and the processing performed upon such structures by transitions. The important structure used to represent the control state is based on the concept of *behavior execution node* (BEN). A behavior execution node encapsulates a behavior being executed along with the associated information needed at execution time. Then the concept of *behavior execution tree* follows naturally from the fact that each time a behavior sends one or several triggering event messages during the execution of its body code, to the reactions of the system are associated new behaviors that are typically being fetched for further execution. To each newly fetched behavior is associated in its turn a BEN that is identified as a child of the original BEN (the message(s) sender BEN). This reflects the causality relationship existing between behavior executions. As a consequence, a natural representation for behavior propagations is a graph. More precisely it is a tree, called the *behavior execution tree* (BET).

4.5 Behavior Execution Node

The actual structure used to represent a BEN is a record with the following fields :

$$\begin{aligned} ben \triangleq \ll\, & id, children, parent, \\ & behavior, bec, state, \\ & ccrc, tecs, parent-tec-id, \\ & cont, bbody-src-lno, bbody-conts-stack, \\ & undo-info-stack, bbody-src-lnos-stack, ops-seqs \gg \end{aligned}$$

Here is a brief description of the interesting[2] fields in the BEN record :

1. The id field is a unique identifier for a BEN. It is an integer. Each time a new BEN is created, a global counter is incremented and its value is stored in the id field of the newly created BEN. So this field can be interpreted as the birth date of a BEN. If the behavior execution tree is visualized, using this field one can easily get an indication on how the behavior execution tree was developed.

2. The $children$ and $parent$ BEN fields implicitly define the behavior execution tree (BET), in effect from the set of developed BENs and the value of this field the BET can be easily constructed :

   (a) The $children$ field is a list of BEN references to all the children that were developed from this BEN, i.e. in all previous steps in the execution of its behavior body.

   (b) The $parent$ field is a reference to the BEN that has triggered the development of this BEN, i.e. the behavior encapsulated by this BEN.

3. The $behavior$ field is a reference to a behavior record where all the features of a behavior are stored as defined in section 3.3.

4. The bec field is the scoped behavior execution context. The BEC is determined when the behavior was fetched. Then the BEC is used during the execution of the behavior clause that are evaluated (i.e. **`pre`**, **`body`** and **`post`**).

5. The $state$ field indicates the execution state of the behavior. The possible states and transitions are described in figure 4.1. Underlined node labels indicate states from which an execution step is possible. A particular state is the $wait-ccrc=0$ state, in this state the BEN is blocked

[2] The fields not described here are considered in section 5.4 (improved debugging) and in section 6 (improved validation).

because its behavior body has sent event messages and at least one coupled reaction has been fetched. When the $state$ field has this value, the BEN state is completed by the fields $ccrc$ and $tecs$ described below. That means that these complementary fields have a meaningful value only when the $state$ field has the value $wait - ccrc = 0$.

6. The *coupled children remaining counter* $ccrc$ field is the counter for the number of coupled children whose behavior execution (behavior block) is not yet completed.

7. The *triggers execution controls* ($tecs$) field is a set of *trigger execution control* (TEC) elements. Each TEC is itself a record used to store all the information required to control the execution of all the behaviors associated to one triggering event message that was sent. The number of elements in the $tecs$ set is the number of event messages sent by the behavior body of this BEN at its last blocking point.

8. The $cont$ field is a continuation [Reynolds93]. Briefly speaking[3], a continuation is a powerful control structure that can be used to capture a state of processing, stop it and resume it later on. It turns out that continuations are particularly adapted to the processing of execution steps in behavior bodies. Note that the $cont$ field has a meaningful value if the $state$ field is either $ready$ or $wait - ccrc = 0$. A behavior body continuation is captured (and stored in this field) as soon as message sending occurs in a behavior body (blocking point). At a blocking point, if coupled behavior reactions are fetched, the BEN enters the state $wait - ccrc = 0$. When coupled reactions terminate their execution the BEN enters the state $ready$, and the $cont$ field can be used to resume the execution of the behavior body, i.e. this performs its next step. Note that since the low level execution environment used is based on *Scheme* and that continuations are a feature of *Scheme* , it was natural and easy to implement the blocking and resume points using continuations. In other execution environments without continuations it is probably possible to work with other facilities, e.g. threads or coroutines.

9. The $parent - tec$ field is a reference to the TEC controlling the execution of the behavior in this BEN. This TEC is in the set of TECs of the parent BEN that triggered this BEN and that is currently blocked.

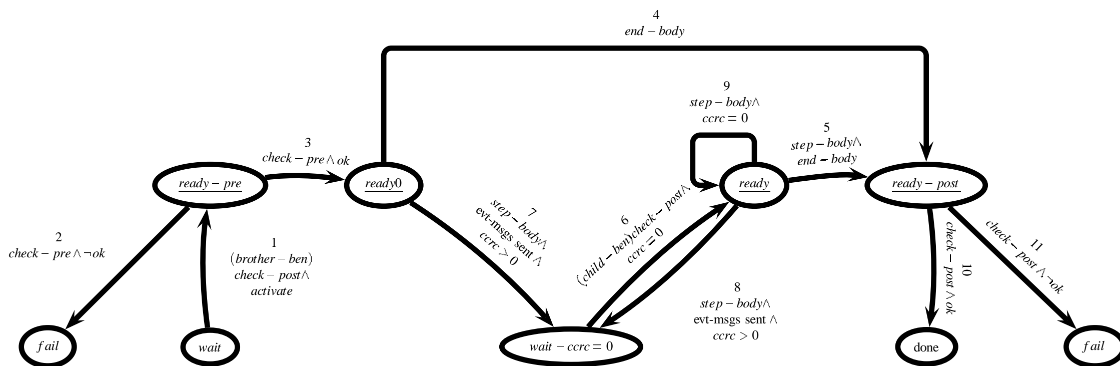


Figure 4.1: Behavior Execution Node FSM.

[3] A more detailed description of this concept is given in appendix D.

4.6 Trigger Execution Control

The *trigger execution control* (TEC) is used to schedule the reactions that have been fetched for a given triggering event message. Note well that this concerns only coupled behavior reactions. Such schedule consists of going through the different trigger reaction phases that have been identified in figure 3.6. A TEC is a record with the following fields :

$tec \triangleq \ll children, ccrc, itccrc, phase \gg$

Here is a description of the fields present in a TEC record :

1. The $children$ field is the list of references to the children BENs, associated to the trigger.

2. The $phase$ field is the trigger execution phase as described in figure 3.6. It is either **`before-trigger`**, **`during-trigger`** or **`after-trigger`**. For each trigger, the behavior reactions that are specified to terminate in the **`before-trigger`** execution phase are executed first (in an arbitrary order). Then the behavior reactions that are specified terminate in the **`during-trigger`** execution phase are executed. This includes the behaviors specified with the **`is-trigger`** reaction semantics. Finally, the behavior reactions that are specified to terminate in the **`after-trigger`** execution phase are executed. Concerning uncoupled behavior reactions to this same trigger, they are activated as soon as the TEC execution phase they are specified to initiate is entered.

3. The $ccrc$ field in the TEC record plays the same role as the $ccrc$ field in the BEN record, but it counts the number of coupled children for the trigger in the current trigger reaction phase. When this counter is decremented to zero, phase switching occurs as described in figure 3.6.

4. The *is-trigger coupled children running counter* ($itccrc$) field is used in the **`during-trigger`** phase. It counts the number of children still running an **`is-trigger`** reaction semantics. This counter is necessary to detect when **`deferred / phase-ii`** behavior fetching has to occur, i.e. when this counter is decremented to zero.

4.7 Transition Functions

Following the atomic BEN execution steps identified in section 4.3, to implement the corresponding transitions three BEN transition functions have been implemented :

1. The $check - pre$ function simply evaluates the pre-condition and stops the behavior propagation in case of failure.

2. The $step - body$ function executes behavior body steps. Two alternative are possible. The first step in the execution of a behavior body is identified by the fact that the $state$ field has the value $ready0$. In this state the behavior body step is executed simply by calling the behavior body function with the BEC as argument. Following steps are identified by the fact that the $state$ field has the value $ready$. In the $ready$ state steps in behavior bodies are executed by resuming the continuation stored in the $cont$ field. An execution step in a behavior body terminates either because the behavior body itself completes of when the next blocking point is encountered, at this point a new continuation is created. This continuation is used later to resume the execution of this behavior body. In the same coherent execution step behavior fetching (**`immediate / phase-i`**) occurs and the newly fetched BENs are registered as children of the BEN from which the original behavior body execution step was executed.

3. The $check - post$ transition function is probably the more complex, because in this function side-effects on the parent BEN and brother BENs are done to implement :
   (a) trigger execution phase switching that may occur according to figure 3.6,
   (b) **`deferred / phase-ii`** behavior fetching may be needed, and
   (c) the execution of the parent BEN may have to be resumed.

   Last but not least, this function also evaluates the post-condition and stops the behavior propagation in case of failure.

The algorithms used to implement the identified transition functions are described below. Concretely, these transitions are implemented with the language used to implement all the execution environment. So to be exact the execution semantics should be defined in terms of the semantics of the supporting language. However, the supporting language itself has very few importance. Indeed, the essence of the execution semantics can be captured algorithmically by showing only how the BET is developed. That is how the records used to represent the control state of the system (BENs and TECs) are manipulated. The execution environment is based on *Scheme*. *Scheme* was used because it is a basic programming language well designed without superfluous features. It is sufficient in terms of the programming language features needed for the behavior language, e.g. usual control flow structures, variable notation, etc. In addition, it is powerful because it features in the language itself one of the most advanced control flow structures, i.e. continuations. Continuations are used to allow intermediate execution steps to be performed in the execution of behavior bodies. In the algorithms below, we use the *Scheme* form of continuations, i.e. the very basic $call/cc$ *Scheme* construct described more in details in appendix D. Note that continuations are only a suitable mechanism to implement the intended execution semantics, the same execution semantics could be implemented (maybe with more difficulties and in a less clear way) using other control flow mechanisms such as coroutines or non-preemptive threads.

4.7.1 BEN:STEP-CHECK-PRE

Algorithm 4.7.1 describes the BEN:STEP-CHECK-PRE transition function. Like all the transition functions, it is given a single argument that is the BEN onto which the transition is to be exercised. This transition function performs transitions 3 (the pre-condition is ok) and 2 (the pre-condition is not ok) in figure 4.1. This function is called only for a BEN in the state $ready - pre$.

Algorithm 4.7.1 *BEN step check pre :*
BEN:STEP-CHECK-PRE(ben)
1 **if** $ben.behavior.pre(ben.bec)$
2 **then** $ben.state \leftarrow$ *ready0*
3 **else** $ben : check - pre - failed(ben)$ // *the behavior propagation is stopped*

We take advantage of the simplicity of this function to explain the notation used to access all the required fields from the BEN record given as argument (formal argument ben). In line 1, $ben.bec$ is used to access the bec field of the BEN (variable ben). Still at line 1, $ben.behavior.pre$ accesses first the behavior record associated to this BEN and then the pre-condition inside the behavior record itself. The pre-condition is a function of one argument, and is called (line 1) on the BEC of the currently being executed BEN. Naturally the result of this call is interpreted as a boolean value telling if the pre-condition was ok or failed.

4.7.2 BEN:STEP-BODY

Algorithm 4.7.2 describes the BEN:STEP-BODY transition function. This transition function performs transitions 7, 4, 8, 9 and 5 in figure 4.1. This function can be executed for a BEN in the state $ready0$ or $ready$.

Algorithm 4.7.2 *BEN step body :*

BEN:STEP-BODY(ben)

```
1  bpe : *beh − body − cont* ← #f
2  bbody − step − res ← call/cc(λ(cont){
3                                    bpe : *cont* ← cont
4                                    if ben.state = ready0
5                                      then ben.behavior.body(ben.bec)
6                                      else if ben.state = ready
7                                              then resume ben.cont with result
8                                                   BEN:GATHER-TRIGGERS-RESULTS(ben)
9                                    resume bpe : *cont * with result * unspecified*
10                                   }
11                                  )
12 if bpe : *beh − body − cont* ≠ #f
13   then ben.cont ← bpe : *beh − body − cont*
14        fbres − list ← map⁴( bpe:bfetch-phase-i , bbody − step − res)
15        BEN:BUILD-TECS(ben, fbres − list)
16        for each tec in ben.tecs
17            do TEC:NEXT-PHASE(tec, ( before-trigger , during-trigger , after-trigger ))
18        if ben.ccrc > 0
19          then ben.state ← wait-ccrc=0
20          else if ben.ccrc = 0
21                  then ben.state ← ready
22   else ben.state ← ready-post
```

This function makes use of two global variables to capture two continuations :

- $bpe : *cont*$ captures the control flow of the BPE itself before the behavior body step is launched. This is done at line 3. This global variable is used by the behavior body to resume the BPE processing as soon as a blocking point is reached, i.e. event messages are sent in the execution of the behavior body. This is done in the function BPE:MSGSND described in algorithm 4.7.3. Note that this function is systematically called when a message is sent in a behavior body. Another important point to note here is that BPE:MSGSND uses the $bpe : *cont*$ variable to resume the BPE processing, but at the same time this defines the value returned by $call/cc$ to be the event messages sent at the next blocking point encountered in the execution of the behavior body. This value is stored in the variable $bbody - step - res$ at line 2.

- $bpe : *beh - body - cont*$ captures the control flow of the behavior body at its next blocking point i.e. at the point where new event messages are sent in the execution of the behavior body.

[4] map is used in the same way as it is defined in standard *Scheme*, i.e. a function and a list are given as arguments, the result is a list of the application of the function to each element of the list. To work the function has to be a function of one argument. Example :

```
(map (lambda (x) (+ x 1)) '(1 2 3)) => (2 3 4)
```

Naturally, if the behavior body completes, the BPE:MSGSND function is not called and the $bpe : *beh - body - cont*$ global variable is not set just because no continuation has been captured. So by initializing this variable to false (noted $\#f$) at line 1, then at line 12, just after the execution of the behavior body step, it can be tested if the behavior body has reached a blocking point or has completed.

There are two ways of launching the execution of a behavior body step.

- If the BEN is in the state $ready0$, that means that the behavior body is at its beginning. So it is merely launched like a pre- or post-condition, i.e. using the function stored in $ben.behavior.body$ called on the BEC ($ben.bec$) at line 5.

- If the BEN is in the state $ready$, things are a bit more complex. The behavior body is resumed using the behavior body continuation that was stored in the BEN when its previous behavior body step was executed. However, the behavior body is not only resumed. In addition, a result has to be returned to this behavior body. For the user, this is important because this represents the result of the processing of the event messages sent. This result is obtained by calling the function BEN:GATHER-TRIGGERS-RESULTS described in algorithm 4.7.4. Note that the call to BPE:MSGSND does not appear in any algorithm presented here just because it is done in a behavior body specified by the user. In fact, BPE:MSGSND call is not done directly, it is encapsulated in a more meaningful syntax of the behavior language.

We are now at the point to describe what is done after the behavior body execution step has been performed, and the control flow has returned back to the BPE. This point corresponds to the statement after the call to $call/cc$, i.e. line 12. At this point two alternatives are possible :

- if bpe:*beh-body-cont* is $\#f$ the behavior body has completed, and the BEN can switch to the $ready-post$ state (line 22). This corresponds also to transition 5 or to transition 4 in figure 4.1. Note that transition 4 corresponds to the case where the BEN switches from the state $ready0$ directly to the state $ready - post$, without any intermediate transition step in the $ready$ state. In other words, the behavior body has not send any event message.

- if bpe:*beh-body-cont* is not $\#f$, it represents the continuation captured to resume the behavior body when its next execution step will have to be performed. So to make it available at this time, it is stored in the field $cont$ of the BEN at line 13. Then because new event messages have been sent that may trigger new behaviors, they are processed by the BPE as follows :

  – at line 14, behavior fetching is done for each message. The result is a list of $fbres$, each $fbres$ being itself a list of (beh, bec) pairs. Each such pair is thus composed of a behavior and a BEC. This corresponds to each new behavior that has been fetched for further execution along with the fetched execution context that will have to be used when its actual execution will occur. Each $fbres$ gives the list of (beh, bec) pairs that have been fetched for each event message sent. The separation of (beh, bec) pairs with respect to event messages is necessary because then one trigger execution control record is built for each $fbres$ as explained below.

  – at line 15, trigger execution control records are built, one for each $fbres$ or message sent. The function BEN:BUILD-TECS is described in algorithm 4.7.5.

  – at line 16, TECs execution phases are initialized by calling the function TEC:NEXT-PHASE. This function is described in algorithm 4.7.7.

– finally at line 18, the coupled children running counter is tested to determine if coupled child behaviors have been fetched. In that case the current BEN is blocked until their completion, and is switched to the state $wait - ccrc = 0$. Otherwise, the BEN is not blocked and is set to the $ready$ state. Note that the $ccrc$ BEN field is initialized consistently with the number coupled child BENs that is determined when the TECs are initialized (by calling the function TEC:NEXT-PHASE).

4.7.2.1 BPE:MSGSND

Algorithm 4.7.3 *BPE Message Sending :*
BPE:MSGSND$(msgs)$

```
1  call/cc(λ(cont){
2            bpe : *beh − body − cont* ← cont
3            resume bpe : *cont * with result msgs
4            }
5          )
```

The function BPE:MSGSND implements message sending in the execution of a behavior body. At line 2, the behavior body control flow is captured and stored in the global variable $bpe : *beh - body - cont*$. Note that this will allow the BPE to execute the next step of this behavior body by resuming this continuation. Finally, the BPE control flow is resumed at line 3 by resuming its continuation stored in the global variable $bpe : *cont*$, it was captured at line 3 in algorithm 4.7.2. The list of messages at this blocking point is sent to the BPE for behavior fetching, etc.

4.7.2.2 BEN:GATHER-TRIGGERS-RESULTS

Gathering results consists to build the list of BECs that are currently stored in the children BEN having a behavior with the **`is-trigger`** reaction semantics. In fact what is returned is a list of such lists, one for each TEC. This corresponds to get the resulting interesting BECs giving the result of the processing of each trigger. Note that because the only communication mechanism between the BPE and behaviors is the BEC, the BPE can not return anything more meaningful to the behavior body. Remember that a BEC is in fact a totally opaque data structure for the BPE. BECs are only stored in BENs, to (in addition to gather triggers results) call behavior functions representing executable clauses of the behavior language, i.e. **`when`**, **`pre`**, **`body`** and **`post`** BL clauses.

Algorithm 4.7.4 *BEN Gather Triggers Results :*
BEN:GATHER-TRIGGERS-RESULTS(ben)

```
1  res ← ()
2  for each tec in ben.tecs
3      do res2 ← ()
4          for each child − ben in tec.children
5              do if is − trigger?(child − ben.behavior)
6                      then add child − ben.bec to res2
7          add res2 to res
8  return res
```

This function is based on two nested loops, At line 2, the first one scans each TEC in the BEN. The second loop scans each child BEN in each TEC. At line 4, the second loop scans child BENs

within each TEC, i.e. related to a same message. At line 5, the test performed tells that only the child BENs having a behavior with the `is-trigger` execution semantics are considered in the gathering process. At line 6, the BEC of such child BEN is simply added to the list of results corresponding to a same message.

4.7.2.3 BEN:BUILD-TECS

Algorithm 4.7.5 *BEN build trigger execution controls :*

BEN:BUILD-TECS$(parent - ben, fbres - list)$
1 $parent - ben.tecs \leftarrow ()$
2 **for each** $fbres$ **in** $fbres - list$
3   **do** $tec \leftarrow tec : make()$
4   **add** tec **to** $parent - ben.tecs$
5   TEC:BUILD-CHILDREN$(tec, fbres, parent - ben)$

At line 3, one TEC for each $fbres$ or message sent is created. At line 5, the function TEC:BUILD-CHILDREN is called to create the child BENs for this TEC and to link each child to the TEC and to its parent. The function TEC:BUILD-CHILDREN is described in algorithm 4.7.6.

4.7.2.4 TEC:BUILD-CHILDREN

Algorithm 4.7.6 *TEC build children :*

TEC:BUILD-CHILDREN$(tec, parent - ben - tec, beh - bec - list, parent - ben)$
1 **for each** (beh, bec) **in** $beh - bec - list$
2   **do** $child - ben \leftarrow ben : make()$
3         $child - ben.behavior \leftarrow beh$
4         $child - ben.bec \leftarrow bec$
5   **add** $child - ben$ **to** $tec.children$
6         $child - ben.parent - tec \leftarrow parent - ben - tec$
7         $child - ben.parent \leftarrow parent - ben$
8   **add** $child - ben$ **to** $parent - ben.children$

This function builds the child BENs fetched for a given message. Each child BEN can be built from the behavior and BEC fetched. In the child BEN, a reference to the TEC record it belongs to is established at line 6. Some child BEN fields are also initialized, i.e. the beh and bec fields that come from the fetching process. Finally to represent properly the structure of the BET, reciprocal references are set up between the parent BEN and each created child BEN.

4.7.2.5 TEC:NEXT-PHASE

Algorithm 4.7.7 *TEC next phase switching :*

TEC:NEXT-PHASE$(tec, remaining - phases)$
1 **for each** $phase$ **in** $remaining - phases$
2   **do if** $\exists$ $child - ben$ **in** $tec.children$ **such that** $child - ben.beh.term_phase = phase$
3   **then** $tec.phase \leftarrow phase$
4   **for each** $child - ben$ **in** $tec.children$
5   **do if** $child - ben.state =$ *wait* $\wedge phase \in child - ben.beh.phases$
6   **then** $ben.state \leftarrow$ *ready-pre*
7   **if** $is - coupled?(child - ben.beh)$

```
8                  then child − ben.parent.ccrc ← child − ben.parent.ccrc + 1
9                    if phase = child − ben.beh.term_phase
10                     then tec.ccrc ← tec.ccrc + 1
11                   if phase = during-trigger ∧ is − trigger?(child − ben.beh)
12                     then tec.itccrc ← tec.itccrc + 1
13               if child − ben.state ≠ done ∧ phase = child − ben.beh.term − phase
14                 then tec.ccrc ← tec.ccrc + 1
15        return
```

Remaining phases are scanned one by one, in the order in which they are given that is assumed to be in ascending order. As shown at line 2, the next phase is the first one found where some behavior in the TEC must terminate. In fact for a given TEC, execution phases depend on the execution rules specified in the fetched behaviors. For instance, that means that if there is no behavior fetched that is specified to terminate **`before-trigger`**, there will be no **`before-trigger`** TEC phase observed. As a result the first TEC phase encountered will be the **`during-trigger`** phase. Once the next phase (variable $phase$ in algorithm 4.7.7) has been found, the TEC switches to it (line 3). Then, it is the time to wake up the behaviors in the TEC that are waiting and are supposed to be active in $phase$. This test is performed at line 5. The wake up is done at line 6, by setting the BEN state to $ready - pre$. If a waken behavior is coupled, then the coupled child running counter of its parent is incremented at line 8. The same counter is incremented for the TEC, only if this behavior is intended to terminate in $phase$. The $ccrc$ counter at the TEC level is used for TEC phase switching. Finally, special care is taken if the waken behavior has the **`is-trigger`** execution semantics. In that case, the is-trigger counter child running counter ($itccrc$) field in the TEC has to be incremented at line 12. As the name stands, this field counts the number of **`is-trigger`** behaviors in this TEC. When it reaches 0, i.e. when the last behavior in this set executes its post-condition, that means that the triggering event message associated to this TEC has completed, and it is time to perform **`deferred / phase-ii`** behavior fetching. All that is done in function BEN:CHECK-POST described in algorithm 4.7.8. Note that once the next phase is found the other ones in the list $remaining - phases$ are not scanned, the function TEC:NEXT-PHASE returns (line 15).

4.7.3 BEN:STEP-CHECK-POST

This transition function performs transitions 10 (if the post-condition is ok) and 11 (if the post-condition is not ok) in figure 4.1. This function is called only for a BEN in the state $ready - post$.

Algorithm 4.7.8 *BEN check post-condition :*

```
BEN:STEP-CHECK-POST(ben)
1   if ben.behavior.post(ben.bec)
2     then if is − coupled?(ben.beh)
3            then ben.parent.ccrc ← ben.parent.ccrc − 1
4                 tec ← ben.parent − ben − tec
5                 if ben.beh.term − phase = tec.phase
6                   then tec.ccrc ← tec.ccrc − 1
7                 if is − trigger?(ben.beh)
8                   then tec.itccrc ← tec.itccrc − 1
9                        if tec.itccrc = 0
10                         then TEC:BUILD-CHILDREN(tec,
11                                                 ben.parent − tec,
12                                                 bpe : bfetch − phase − ii(tec),
```

```
13                                    ben.parent)
14           if tec.ccrc = 0
15             then if tec.phase = before-trigger
16                   then TEC:NEXT-PHASE(tec,
17                                  ( during-trigger , after-trigger ))
18                   else if tec.phase = during-trigger
19                         then TEC:NEXT-PHASE(tec, ( after-trigger ), ben.parent)
20           if ben.ccrc = 0
21             then ben.parent.state ← ready
22       ben.state ← done
23   else ben : check − post − failed(ben) // the behavior propagation is stopped
```

At line 1, the post-condition is called with the BEC of the currently being executed BEN as the single argument. Naturally the result of this call is interpreted as a boolean value telling if the post-condition was ok or failed. Then, most of the processing (form line 3 to line 21) is done if the behavior is coupled. This processing consists of taking into account that the the completion of a coupled behavior may imply updates both in the TEC record it belongs to and in the parent BEN record. Typically at line 21, if the completion of this BEN corresponds for the parent-BEN the last completion of a coupled execution, then the parent BEN is re-activated.

For a coupled execution that completes, the coupled children running counters ($ccrc$ field) are decremented both for the parent BEN (line 3) and for the TEC the BEN belongs to (line 6). The TEC is obtained from the field $parent - ben - tec$ (line 4). This field references a TEC that is stored in the set of TECs of the parent BEN. This set of TECs is currently being used to control the execution of the child behaviors that have been fetched because a triggering event messages has been sent in the behavior body of the parent BEN.

In the coupled behavior processing, special care is taken if the behavior is in addition doted with the **`is-trigger`** execution semantics. If the behavior being completed is the last behavior with the **`is-trigger`** execution semantics in this TEC, then deferred (line 12) behavior fetching can occur on all the BECs corresponding to the **`is-trigger`** behaviors in this TEC. Once deferred behavior fetching has been done, new child BENs are being built from the fetched behaviors and BECs using the function TEC:BUILD-CHILDREN described in algorithm 4.7.6. The construction of child BENs fetched in the deferred fetching phase is done in the same way as it was done in the case of BENs fetched in the immediate behavior fetching phase.

In the coupled behavior processing, another important part is the implementation of TEC phase switching according to figure 3.6. This is performed from line 14 to 19. TEC phase switching is done if the behavior being completed is the last coupled behavior in the current trigger execution phase. This is tested at line 14. Thus if the current phase is $before - trigger$ we try to switch to the $during - trigger$ phase or to the $after - trigger$ TEC phase. If the current phase is $during - trigger$, we try to switch to the $after - trigger$ phase. In each case this is done by calling the function TEC:NEXT-PHASE described in algorithm 4.7.7. Finally at line 22, and independently of the execution semantics of the behavior, the BEN goes to the state $done$ just because the execution of its behavior block (pre, body, post) has been completed.

4.8 BPE Algorithm

In this section, the overall forward search inference engine described in section 4.2.1 is refined with respect to the data structures used to represent the control state and the transition functions defining

the atomic execution steps. The BPE algorithm presented below describes how a behavior propagation is developed. However, before the development itself a bootstrapping has to be done.

4.8.1 Bootstrapping a Behavior Propagation

A behavior propagation is typically caused by an initial solicitation, i.e. one or several messages being sent to the system. This is typically specified by the user that wishes to test some features of the model. Such initial solicitation can be grouped to form scenarios representing complete sequences of interactions between the system and its environment. As shown in algorithm 4.8.1, to bootstrap the behavior propagation the initial solicitation is encapsulated into an artificial behavior called the *root* behavior. The root behavior is itself encapsulated into an artificial BEN called the $root$ BEN (the root of the BET).

Algorithm 4.8.1 *Behavior Propagation Bootstrapping :*
BPE:BPINIT$(msgs)$
1 $root - ben \leftarrow ben : make()$
2 $root - ben.beh \leftarrow behavior : make()$
3 $root - ben.beh.label \leftarrow "root"$
4 $root - ben.beh.pre \leftarrow \lambda(x)\{$**return** $\#t\}$
5 $root - ben.beh.body \leftarrow \lambda(x)\{printf($ *"Hello, root behavior body."* $)$
6 BPE:MSGSND$(msgs)$
7 $\}$
8 $root - ben.beh.post \leftarrow \lambda(x)\{$**return** $\#t\}$
9 $root - ben.beh.exec - rules \leftarrow ((fetch - phase$ **unspecified** $)$
10 $(uncoupled$ **unspecified** $))$
11 $root - ben.beh.bec \leftarrow$ **unspecified**
12 $root - ben.beh.state \leftarrow$ *ready-pre*
13 $root - ben.beh.parent \leftarrow$ **unspecified**

The behavior is created with pre- and post-conditions that always return true. Its body just sends the messages representing the initial solicitation. The fetch phase of the root behavior is unimportant because this behavior is implicitly forced to be fetched and active. It is specified to be uncoupled, this way when it is completed, no try is made to do side-effects on its parent BEN or TEC. This would be totally irrelevant for the root BEN because it has no parent BEN and TEC it belongs to. In addition, its initiating trigger execution phase is unimportant because the behavior is implicitly active. The value of the BEC is not used and is thus also unimportant. Finally, the root BEN state is set to $ready - pre$, this way it is active.

4.8.2 Function BET:WALK

From the bootstrapped root BEN, the behavior propagation is performed by developing the behavior execution tree. The function BET:WALK described in algorithm 4.8.2 describes this process.

Algorithm 4.8.2 *Behavior Propagation Engine :*
BET:WALK(tr)
1 $ready - bens \leftarrow \{ben \in BET : ben.state \in \{$ *ready-pre* , *ready0* , *ready* , *ready-post* $\}\}$
2 **if** $ready - bens = \emptyset$
3 **then** // *termination state*
4 $dump - trace(tr)$

5 **else** $ben \leftarrow select - ben(ready - bens)$
6 $ben : exec(ben)$
7 BET:WALK(**add** ben **to** tr)

BET is referenced as if it was a global variable storing the behavior execution tree of developed BENs. What is important is to be able to find ready BENs in the the set of BENs that have been developed (line 1). A clear identification of the BET itself is not necessary. Indeed, the BET is only defined implicitly by the BENs that can be accessed from the root BEN and following the parent-children links between BENs. The function BET:WALK is called with an argument tr that represents the trace collected from the initial state of the system to the current one. Initially BET:WALK is called with an empty trace. Such a trace can be represented as the sequence of BENs that have been selected at each step for execution. Note that this trace is not used for the development of the BET itself, it is just used to record the way the BET was developed. In particular at saturation, i.e. when no more BENs are active, a dump of the trace can be displayed (line 4). This provides a complete description on how the BET was developed. At line 5, the BEN from which the next execution step is performed is selected among the ready BENs. Depending on the strategy used for this selection, the BET is developed according to different kinds of paths. The different selection strategies result in developments following either a *depth first walk*, a *breadth first walk* or a *random walk*. At line 6, the selected BEN is executed by calling the BEN:EXEC function that is described in algorithm 4.8.3. Finally, from the new state of the system reached after the selected BEN has been executed, BET:WALK is recursively called with a trace augmented with a reference to the lastly executed BEN.

4.8.3 BEN:EXEC

Algorithm 4.8.3 is quite simple. Depending on the BEN state, it just calls the transition function associated to the current state of the BEN to be executed.

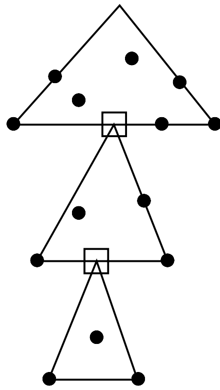
Algorithm 4.8.3 *BEN Execution :*
BEN:EXEC(ben)
1 **if** $ben.state =$ *ready-pre*
2 **then** BEN:CHECK-PRE(ben)
3 **else** **if** $ben.state =$ *ready0* $\vee$ $ben.state =$ *ready*
4 **then** BEN:STEP-BODY(ben)
5 **else** **if** $ben.state =$ *ready-post*
6 **then** BEN:CHECK-POST(ben)

4.9 Atomic Execution Support

4.9.1 Principle

Note that in the execution semantics of the behavior model presented in this chapter `when`, `pre` and `post` clauses are evaluated atomically. For behavior bodies the execution steps are delimited by message sending present in behavior bodies. This unit of atomic processing provides a low level granularity of execution steps in behavior bodies. This emulates concurrency in the system by interleaving the execution of such execution steps among all the enabled behaviors. However, to fulfill some application requirement this low level and fine grain interleaving may have to be made more

coarse. For instance in a behavior, one may be interested to specify that two attribute values require to be updated together in a single and coherent phase, e.g. a shared buffer and its length. In the end a mechanism has to be defined so that pieces of behavior bodies can be evaluated in a single and coherent phase, with respect to the other behavior execution steps enabled in a given state. This is the objective of the atomic execution support. The principle is illustrated in figure 4.2, when from a BEN an atomic execution section is entered (square in top sub-BET), the development of the other BENs that were enabled in this state (BEN represented by bullets above the square) is temporarily blocked until the end of the atomic section is reached. Naturally it is possible to obtain nested atomic execution sections. This arises if within an atomic section a message is sent that triggers a behavior having as well an atomic section in it. Note well also that atomic execution is delimited only by the coupled behavior executions, i.e. the atomic section ends when all the coupled behavior executions in the atomic sub-BET have completed.



The filled bullets represent ready BENs in the whole BET. The squares represent ready BENs that have entered atomic sections. When entered only the sub-BET rooted at such nodes is developed. Atomic executions can occur in a nested way.

Figure 4.2: Atomic Executions.

4.9.2 Syntax

The syntax used to introduce an atomic execution section in a behavior body consists merely of enclosing the section with the keyword **`atomic`** as shown in this example :

```
(atomic
  (Set buf "length" (+ (Get buf "length") 1))
  (Set buf "contents" (cons new-elt (Get buf "contents")))
  )
```

4.9.3 Implementation Support

Two functions are defined for entering and leaving atomic execution sections. These functions are called `bpe:begin-atomic` and `bpe:end-atomic`. At load time when behavior definitions are entered in the system, behavior body with **`atomic`** syntax is expanded as follows :

```
(begin
  (bpe:begin-atomic)

  (Set buf "length" (+ (Get buf "length") 1))
  (Set buf "contents" (cons new-elt (Get buf "contents")))
```

```
(bpe:end-atomic)
)
```

The way `bpe:begin-atomic` and `bpe:end-atomic` can be implemented is not interesting as such. In fact there are several options. The principle is to record in a stack (because of the possibility to have nested atomic processing) the set of BENs that are blocked in `bpe:begin-atomic` and to reactivate these BENs in `bpe:end-atomic`. Note that atomic execution support has no effect on the remaining of the BPE machinery that was presented in this chapter.

4.10 Conclusion

The contribution of this chapter is the precise specification of the execution semantics of the proposed behavior language. Details about the contributions of this chapter can be listed as follows :

Behavior propagation engine : The execution semantics is given operationally in terms of the behavior propagation engine (BPE) algorithm. The BPE can be viewed as a forward search inference engine performing execution steps until saturation.

Transition system : To describe the actual processing performed during a behavior propagation, the BPE was presented as a transition system, which is classically defined by states and transitions. States are defined using data structures representing the configuration of the system. Transitions represent the atomic execution steps. They are defined algorithmically by a set of transition functions. These transition functions exercise their processing on the data structures used to represent the system's configuration.

A conventional choice is to partition the configuration of a system into a control part and into a data part. An important point is that the transition functions can be described only based on the control configuration of the system. The data configuration (also called the information repository (IR) in this thesis) is used in the processing of behavior clauses that are evaluated (i.e. **`when`**, **`pre`**, **`body`** and **`post`**) and in the processing of behavior scoping functions. All these issues are outside the core part of the behavior model. In the end what is described in this chapter is a generic control semantics that can be used on top of any form of data configuration. Concretely, the information repository can be implemented in any way appropriate, e.g. in memory, on disk, in a database, etc.

User level control structures : Another important point is that the control structures used to support the control semantics are deliberately based on user-level structures and not on language implementation control structures such as the runtime stack. In summary, the user-level structures defined are the BEN record and the TEC record. The whole BET is obtained from any BEN by following parent-children references. Interestingly, the BET provide a graphical representation of the control flow that has been followed in a behavior propagation. In particular the nondeterminism exhibited by the system can be easily visualized and understood by the people involved. In chapter 5 more details on this aspect are given when considering the execution environment.

because it is very interesting to provide a powerful execution environment.

The dynamic semantics of the proposed behavior modeling framework has been addressed in the following publications where the author has contributed : [Sidou97a, Sidou96, Sidou95, Eberhardt et al.97b, Sidou et al.95b].

Chapter 5

Execution and Debugging Environment

5.1 Introduction

Performing validation based on executable specifications imposes to build an execution environment. It is also desirable that the execution environment is powerful. Otherwise the behavior modeling framework itself is not usable with respect to a validation approach based on simulation. Most of the clauses used in the behavior language, (i.e. **`when`**, **`pre-`**, **`body`** and **`post-`** clauses) need to be executable and typically make use of usual programming language constructs such as control flow structures (e.g. loops, conditionals . . .), declaration of variable, assignment, etc. To avoid reinventing such basic programming language features, a pragmatic approach is to embed the behavior language into an existing programming language. The *Scheme* programming language has been chosen, because it is simple, clean and powerful enough. So, the first level of execution environment available to perform behavior simulations is the basic *Scheme* read-eval-print interpretation loop. It provides a simple and low level command line interface to the simulation environment. On top of the *Scheme* interpreter, two useful features have been identified and implemented to make the execution environment more usable : (i) system visualization, and (ii) debugging. In section 5.2, it is shown how system visualization facilities can be used to monitor the different parts of the state of the system at runtime. In section 5.3.2, the basic dynamic behavior analysis facilities are described. This consists of the different modes of execution available to develop a behavior propagation, i.e. user driven, depth first, breadth first and random. In section 5.4, basic and improved debugging facilities are described. In section 5.5, a summary of the different functions of the execution environment available to the user are summarized along with their corresponding interface. Both graphical and command line interfaces are given.

5.2 System Visualization

System visualization is required in order to properly monitor the state of the system at runtime. The state configuration of the system is classically partitioned into a data part and a control part. This defines two monitoring views available for the user : an object view, and a control view. Note that a complementary view is the low level text view provided by the *Scheme* interpreter. It is used to display raw text such as the list of attributes and corresponding values of an object as requested on the graphical user interface of the object view. The execution view is naturally based on the notion of behavior execution tree that was described in section 4.4. The execution view is detailed in section 5.2.2 below. The next section focuses on the object view.

5.2.1 Object View

The information repository contains the information object instances existing in the system. According to the way the information repository is structured, e.g. with objects and relationships, it can be visually represented as a graph. Behaviors being executed can be seen as rewriting rules on the information repository. Each time something occurs in a behavior, its effects can be reflected on the underlying information repository. So, any change on this graph can be highlighted which naturally lends to animation during behavior propagation. Figure 5.1 gives an example of an Object View snapshot[1]. This example is taken from a TMN case study. So, the object and relationship graph is instantiated according to TMN information models : GDMO for Managed Objects (drawn as ovals), and GRM for relationships (drawn as rhombuses). With the graph visualization tool daVinci, any part of the object view can be hidden and restored as needed.

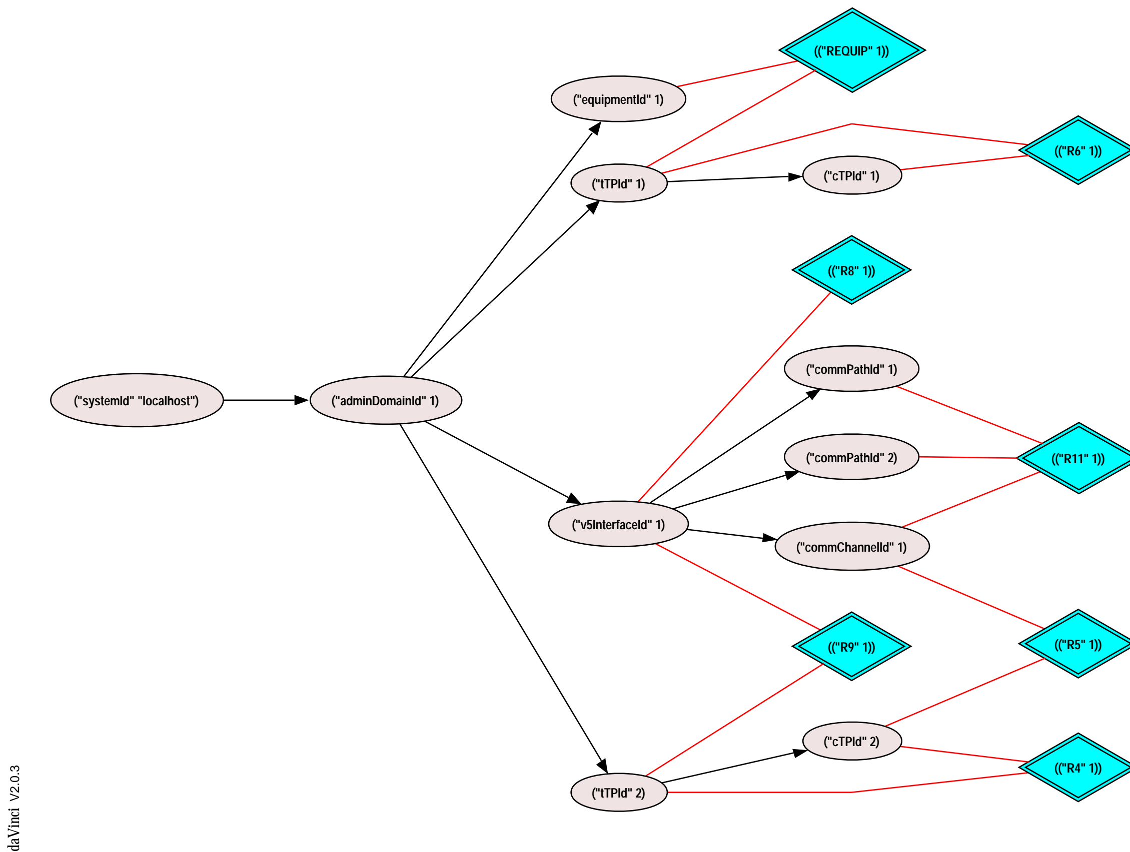


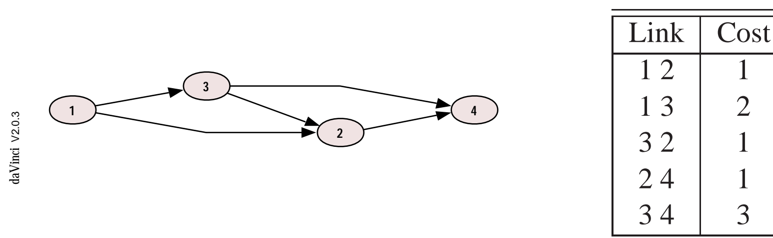
Figure 5.1: Object View.

The object view is not a passive view. It allows interaction with the user since any clicking on a node in this view triggers the display of its features on the text view. Typically this concerns object attributes along with their corresponding values.

[1] The graph visualization tool *daVinci* [davinci] is used to provide the object view. This tool is also used to provide the execution view (see section 5.2.2).

5.2.2 Execution View

This view is simply obtained by visualizing the behavior execution tree (BET) defined in section 4.4. The BET is a complete representation of the control state of the system. The BET can be used to provide for a visual description of a complex behavior propagation. The BET gives also a visual representation of the way the operational semantics is exercised. Since the operational semantics used is based on an interleaving execution model, execution steps of executed behaviors can be interleaved in any arbitrary order. For a given behavior propagation, the user can get from the visual representation of the BET an immediate feeling on the actual interleaving that was performed. Concretely, the user can monitor and control at each step in a behavior propagation all the possible alternatives that the execution can go through. This can be done by identifying all the BENs that are currently in a form of *ready* state, i.e. *ready-pre*, *ready0*, *ready*, *ready-post*. Ready BENs are typically colored in green so that they can be easily identified on the BET. This is especially useful in the user driven execution mode described in section 5.3.2, where the user has a full control about which execution branch of the BET should be explored. Figure 5.4 gives an example of a behavior execution tree. Each node is documented with a unique identifier accompanied with the label of its corresponding behavior and its state : $\ll i, beh - label, state, \cdots \gg$. The different values for the BEN state field were shown in figure 4.1. BEN ids can be interpreted as a birth date. This gives an immediate view about how the BET was developed. More precisely, this gives the exact order of creation of the BENs. To get a visual representation, BENs are labeled and colored differently according to their state. Additional attributes accessible from each BEN can also be added by the user to customize what is actually displayed for each BEN developed, e.g. the label of the corresponding behavior.



Link	Cost
1 2	1
1 3	2
3 2	1
2 4	1
3 4	3

Figure 5.2: Directed Graph 1 for the Spanning Tree.

In figures 5.4 and 5.6 two execution views are shown. They represent two possible executions of the spanning tree algorithm for the directed graph described in figure 5.2. This execution of the spanning tree algorithm computes shortest paths of any node in the directed graph to the termination node 4. The spanning tree case study is introduced in appendix C with its correponding behavior. Each behavior execution node in the two execution views are represented with their id, their behabior label, and the link fetched in their BEC.

When the execution of the spanning tree algorithm is initiated for node 4, two alternatives are possible for the next execution step, i.e. update either node 2 or node 3. This corresponds to select BEN 3 or BEN 4 (respectively) in figures 5.4 and 5.6. If node 2 is updated first, we obtain the optimal execution of figure 5.4. A trace of this execution showing the sequence of node updates is given in figure 5.3. The shortest path of each node in the directed graph is obtained with a minimal number of execution steps. On the other hand if node 3 is updated first, the first steps of the behavior propagation are in fact useless because they are improved by subsequent steps. This subsequent execution steps correspond to the update of node 2 that finally lead to the result. A trace of this non-optimal execution showing the sequence of node updates is given in figure 5.5. Note that in the end both executions lead to the same solution because in the trivial directed graph proposed in figure 5.2 there is only one solution to the problem. However in general, different executions can lead to different results because

several shortest path may exist between two nodes.

```
1    st 2.next=> 4
2    st 2.dist=> 1
3
4    st 1.next=> 2
5    st 1.dist=> 2
6
7    st 3.next=> 2
8    st 3.dist=> 2
9
10   Result: ((1 next=> 2 dist=> 2)
11            (2 next=> 4 dist=> 1)
12            (3 next=> 2 dist=> 2)
13            (4 next=> ??? dist=> 0))
14
15   Trace: (1 2 3 5 6 8 6 7 9 7 3 4 1)
16
```

Figure 5.3: Trace of an Optimal Spanning Tree Execution.

Optimal execution of the spanning tree algorithm. Node 2 is updated first, then node 1 and node 3. The trace gives the sequence of BEN execution steps performed to develop the BET in figure 5.4.

5.2.3 Scheme Shell Text View

This complementary view is the low level text view provided by the *Scheme* interpreter. It can be used to display the details about information displayed graphically on the object view or on the execution view :

- if the user clicks on an object on the object view, the details about each attribute and corresponding value are displayed in raw text mode on the *Scheme* text view. Also displayed is the list of roles and relationships, this object instance is participating.
- still on the object view, if the user clicks on a relationship the details about each participants in each role are displayed along with any additional qualifier attribute that is stored in this relationship.
- if the user clicks on a BEN of the behavior execution tree displayed on the execution view, the details about the current execution state of the node are displayed on the *Scheme* text view. The more useful information is probably the current value of the behavior execution context, the BEC associated to this BEN.

Finally, the *Scheme* shell provides a low level command line interface that may be used to query any information maintained in the system.

5.2.4 Whole Execution Environment

The overall execution environment is integrated with *Emacs* [Stallman87]. *Emacs* provides facilities to integrate inferior processes that compose the TIMS tool set. In addition *Emacs* provides for free a

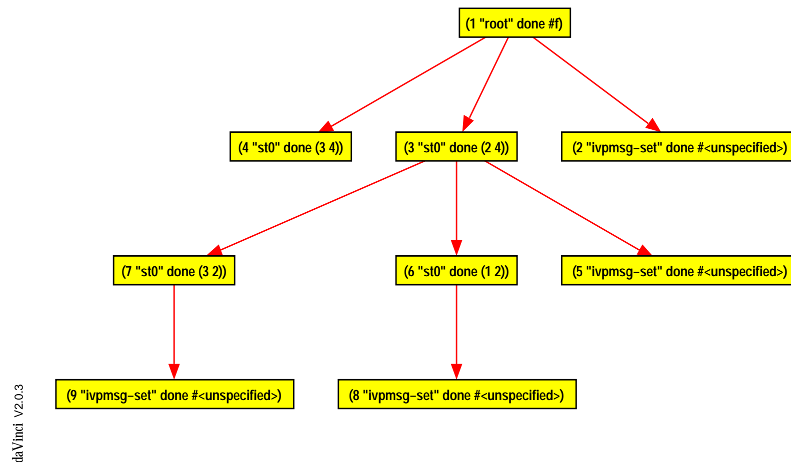


Figure 5.4: BET for an Optimal Spanning Tree Execution.

```
1     st 3.next=> 4
2     st 3.dist=> 3
3
4     st 1.next=> 3
5     st 1.dist=> 5
6
7     st 2.next=> 4
8     st 2.dist=> 1
9
10    st 1.next=> 2
11    st 1.dist=> 2
12
13    st 3.next=> 2
14    st 3.dist=> 2
15
16    Result: ((1 next=> 2 dist=> 5)
17             (2 next=> 4 dist=> 1)
18             (3 next=> 2 dist=> 2)
19             (4 next=> ??? dist=> 0))
20
21    Trace: (1 2 4 5 6 3 8 9 10 12 10 11 7 9 3 6 4 1)
22
```

Figure 5.5: Trace of a Non-Optimal Spaning Tree Execution.

Non-optimal execution of the spanning tree algorithm. Node 3 is updated first, then node 1. Finally node 2 that re-triggers the update to the final value for node 1 and node 3. The trace gives the sequence of BEN execution steps performed to develop the BET in figure 5.6.

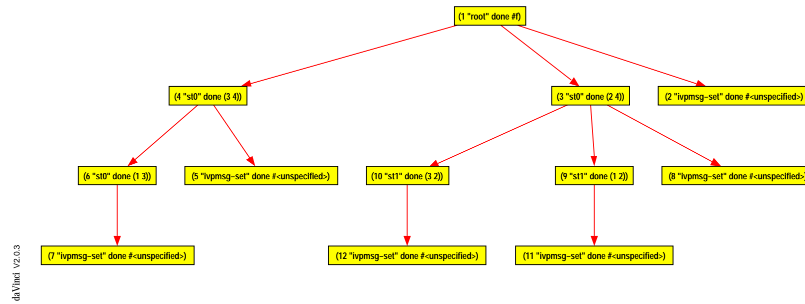


Figure 5.6: BET of a Non-Optimal Spanning Tree Execution.

documented and sophisticated set of file browsing and obviously editing facilities.

In figure 5.7 a screen snapshot shows a typical working session on a case study using the tool set that was developed in the context of the TIMS project. The window on the background is the *Emacs* window, the buffer on the top of the window is the *Scheme* shell text view. At the bottom right corner the object view window is displayed. An object instance has been selected on this view and its properties are displayed on the text view. At the top right corner the execution view window is displayed. A behavior execution node has been selected on this view, and the code of the associated behavior is given in the *Emacs* buffer at the bottom of the *Emacs* window. In addition, in this buffer, an arrow indicates the current execution position in the behavior code. This illustrates one of the facilities inherited for free from the *Emacs* environment, i.e. the possibility to invoke file browsing and line displaying.

5.3 Dynamic Behavior Analysis

5.3.1 Principle

As already stated, the approach chosen for validation is based on executable specifications. This approach consists of submitting scenarios or test cases to the system, to execute the corresponding behavior propagation and to observe the outcome. Because of the execution based nature of the approach, it can also be qualified as a *dynamic behavior analysis*, in opposition to a *static behavior analysis* that is typically based on reasoning directly on the specification without any requirement for an execution support. Because of the nondeterminism typically present in the behavior of a distributed system, it is worth to have different ways to perform executions, so that one can get a chance to observe the different behaviors of the system. To this end, various execution or walk modes are described in the next section.

5.3.2 Execution / Walk Modes

To perform dynamic behavior analysis, several execution or walk modes are available :

- The **user driven** mode enables the user to develop the BET in a fully controlled way.
- The **random walk** policy enables to develop the BET randomly, and reach one of the possible terminating states.
- The **fixed walk** modes enable the user to develop the BET according to a fixed strategy. Two fixed walk modes are available :

Figure 5.7: TIMS Session.

1. in the depth first walk mode the BEN selected for execution is always the one that became ready the more recently. And one of the more recently activated, if there are more than one such BEN.
2. in the breadth first walk mode the BEN selected for execution is always a one among those that are ready for the larger number of execution steps.

Fixed walk modes are useful for early stages of debugging, to force the execution to follow always the same path. This avoids to be annoyed by the problems caused by nondeterminism that one may prefer to fight at a later stage. Though the definitions above for depth first and breadth walk modes seem a bit tricky, in practice fixed walk execution modes are very easy to implement. For instance if ready BENs are pushed on the top of a list, the depth first walk mode consists merely of selecting the BEN at the head of the list, whereas in the breadth first walk mode the BEN at the end of the list is selected.

5.4 Debugging Support

Since validation is based on execution, debugging support is a very important feature. The next section briefly summarizes the basic debugging support, whereas section 5.4.2 describes the improved debugging support features. Improved debugging features are particularly useful in order to ease the analysis of root causes of problems, e.g. an assertion violation. This constitutes the basis for a powerful explanation tool for problems, that may be used between the people involved in the specification and validation process.

5.4.1 Basic Debugging Support

Basic debugging support is merely based on stepping the execution steps in a fully controlled way. This consists to benefit from the features of the user driven walk mode. At each step, the user selects a ready BEN from which the next step is to be executed.

In this mode, it is also possible to terminate the execution of a BEN according to one of the policies previously described. This is useful to complete the development of a sub-BET if one is not interested in the details of the development of this particular sub-BET. This allows to go faster to interesting parts, i.e. the parts that require careful analysis and debugging. One important basic debugging feature is source level debugging. In particular, when the user performs execution steps one by one it is pleasant to see also the behavior source code reached by the executed BEN. To make this possible behaviors (more exactly behavior specification files) are instrumented so that when they are loaded extraneous debugging information concerning line numbers in behavior specification files are recorded in the system. Two levels of behavior instrumentation are required, :

- the first level of instrumentation concerns behavior clauses. Here what is needed is to provide for each behavior the line position in the source file for each behavior clause. At load time theses line positions are stored in additional fields present in the record used to represent behaviors internally in the system. Let us recall the definition of the behavior record already given in section 3.3 to focus on the additional debugging fields.

$$behavior \triangleq \ll label, scope, when, pre, body, post, exec-rules,$$
$$def-beh-lno, pre-lno, body-lno, post-lno, src-file \gg$$

The field $src-file$ gives the full pathname of the source file. The field $pre-lno$, $body-lno$, and $post-lno$ give the line position in the source file where the pre-condition, the body and

the post-condition clauses are beginning. This fields are respectively used when the BEN state is $ready - pre$, $ready0$ and $ready - post$.

- The second level of instrumentation concerns intermediate execution steps within behavior bodies. Here a rather *dynamic* form of instrumentation is required. Intermediate steps also called blocking points in section 4.3 are delimited by the statements that send messages. So, to instrument behavior bodies, it is just necessary to encapsulate such statements with instrumentation code. At runtime, and thus in a dynamic way, the instrumentation code communicates the line position of each message sending statement to the BPE. The communication is done e.g. by updating a global variable. At the end of each execution step, if the BEC has reached a blocking point in the behavior body, such a global variable has been updated and its value is the corresponding position of the blocking point in the behavior source file. This variable is stored in an additional BEN field, the $bbody - src - lno$ field. This field is meanningful when the BEN state is $ready$, $wait - ccrc = 0$.

Let us recall in this chapter the definition of the BEN record already given in section 4.5 to focus on the additional debugging fields. This record definition will be used again more extensively to describe the other debugging fields required for the purpose of improved debugging support.

$$
\begin{aligned}
ben \triangleq \ll\; & id, children, parent, behavior, bec, state, \\
& ccrc, tecs, parent - tec - id, cont, bbody - src - lno, bbody - conts - stack, \\
& undo - info - stack, bbody - src - lnos - stack, ops - seqs \gg
\end{aligned}
$$

Figure 5.8: BEN Record.

5.4.2 Improved Debugging with Backtracking

5.4.2.1 Motivation

Given an erroneous behavior propagation that has failed to produce the desired outcome, how does one proceed to find where it went wrong. The input data, the behavior source and the erroneous execution are the information available to the user. It should be noted that the display snapshot of the BET yet provides a good trace of what happened in the system. For instance, from the BET it is easy to see if the proper behaviors where fetched, and how (i.e. with which BEC). One can also see if a behaviors was not fetched. Basic source level debugging support do not prevent from wasting precious debugging time in setting break-points in backward order and re-running the program, or in stepping over the whole execution flow, until the erroneous code is reached. However, in order to find out the root causes of problem it turns out that improved debugging facilities such as execution backtracking mechanisms in interactive source level debuggers allows users to mirror their though processes while debugging [Agrawal et al.90a]. This enables to work backwards from the location where an error is manifested and determine the conditions under which the error occurred. Such a facility also allows a user to change program characteristics and re-execute from arbitrary points within the program under examination (a “what-if” capability). In our context, changing program characteristics typically consists of fixing bugs in behavior code. Then using the interpreted execution environment available, the fixed behavior code can be dynamically re-loading and re-executed to check that the problem was fixed. This results in a very interactive and incremental execution environment. That is the reason why execution backtracking support applied to the behavior execution environment is considered below.

5.4.2.2 Incremental vs. Non-Incremental Approaches

Several approaches are possible for backtracking. In particular, one can distinguish between the incremental and non-incremental approaches. In the incremental approach backtracking is based on the ability to undo the effects of each execution step individually. The non-incremental approach, in contrast, is based on the ability to go backwards at a fixed execution step, e.g. the initial state of the system, and then to replay the execution step until the previous step. The non-incremental approach is typically used when the incremental approach is not possible. It is obviously less efficient than the incremental one. Efficiency is not that important for interactive debugging. However efficiency is more important for the purpose of exhaustive exploration of all the behavior executions. As shown in chapter 6, this can be achieved by reusing the execution backtracking facility. That is the reason why the incremental approach has been selected to implement execution backtracking.

5.4.2.3 Control vs Data Backtracking

In dynamic debugging environments [Agrawal et al.90a], backtracking is usually partitioned into **control backtracking** and **data backtracking** issues. This partition comes mostly from the fact that data backtracking is related to user level structures of a running program, e.g. user variables. In contrast, control backtracking is related to runtime support structures, e.g. the runtime stack. The distinction is useful because the techniques used to undo for user level structures are quite different from the techniques used to undo runtime support structures. Undo on user level data structures is usually based on the existence of reverse operations for all the operations than can be used on data variables in a program. For instance, to reverse an assignment, it is just necessary to make an assignment using the old value. In contrast, undoing a function call requires more elaborated undo functions that work (at least in a language such as `C`) on the runtime stack. In the `C` language and under UNIX one can use `setjmp` and `longjmp` `C` library functions to go backwards function calls.

5.4.2.4 Backtracking Applied to (IR, BET) Configurations

This distinction between data and control backtracking has to be slightly re-considered for the purpose of undoing behavior execution steps as they were defined in section 4.7. First let us recall that the configuration of a behavior execution at each step is defined by the contents of the (IR, BET) pair. Since the IR is pure user level data, undoing IR operations is based on usual principles previously mentioned for such a purpose. On the other hand, the BET is based on both user level and runtime support structures. Indeed, BENs are based on normal data structures comparable to what is used for IR data. So, part of undo functions for BET updates can be based on usual user level data undoing techniques. The only part in the BET dependent on runtime support structures are *continuations* that are general purpose control structures available in *Scheme*. They are typically used to capture the control state of processing in a behavior body as soon as new event messages are sent in its execution. This defines the intermediate blocking points in the execution of behavior bodies (see section 4.3 for the details).

5.4.2.5 Continuations & Control Backtracking

Pure control backtracking is only necessary in (IR, BET) configurations to go backwards *Scheme* processing in behavior bodies, is made trivial thanks to *Scheme* continuations. It is just needed to store the history (rather than just the last one) of continuations encountered during the execution of each behavior body in a stack present in each BEN. Then it is possible to restore any previous control state in the execution of a behavior body just by popping the BEN continuations stack and calling back the

corresponding continuation. As shown in figure 5.8, the BEN field used as a stack to maintain the history of continuations is the field $bbody - cont - stack$.

5.4.2.6 Data Backtracking on (IR, BET) Configurations

Data backtracking is another issue. This requires to be able to restore the effects on the IR and on the BET of each behavior execution step. Data backtracking consists of defining data undoing function for each possible update operation on user level data structures, such as the IR and the BET. In fact the problem consists of providing update functions that in addition to performing the actual update compute the required undo information and store this information at an agreed location, e.g. a global variable. When a behavior execution step has completed, the BPE has just to move the contents of the agreed location in a suitable location in the BET, i.e. in the BEN to which the last execution step was applied. In fact, what is needed is a stack of such undo information. As shown in figure 5.8, the BEN field used for this purpose is the field $undo - info - stack$.

Now that the overall interface between update functions and the BPE is somewhat fixed, the form used to store each individual undo information can be described. Many forms are possible, in particular in an interpreted language like *Scheme*. The simple and general form chosen consists of storing each item of undo information (corresponding to each update operation performed) as a list. The first element is the function to call to undo the update operations, the remaining elements are the arguments that will have to be passed to the function when undoing occurs. Naturally the whole undo information is a list of such undo information items. This form of undo information is enough general because in *Scheme* any kind of function can be built dynamically at runtime in the form of λ expressions.

Now that a general format for each individual undo information has been defined, the particular undo information used to undo operations performed on the IR and the BET can be defined according to this format. The IR and thus the corresponding update operations are in fact something that is somewhat outside of the kernel of the system. However, an in memory implementation of the IR and its operations is used. It is based on a general form of repositories of record entries, that is also used to store the BET. If another storage form was used for the IR, e.g. on disk storage or a general database, a corresponding undo support would have to be provided. As stated above, this consists of updating functions that compute additionally undo information according to the specific storage used. With the generic in memory storage based on repositories of record entries, undo support can be realized at this level both for the IR and the BET contents, and in a generic way.

At the level of generic repositories of record entries, the update operations are reduced to three functions :

1. $(set_record_field!\ record\ field\ value)$. The corresponding undo information is :

   $(set_record_field!\ record\ field\ old_value)$

2. $(set_repository_entry!\ key\ value)$. If this repository entry exists, the corresponding undo information is :

   $(set_repository_entry\ key\ old_value)$

   If this repository entry does not exist, the corresponding undo information is :

   $(delete_repository_entry!\ key)$

3. $(delete_repository_entry!\ key)$. If this repository entry exists, the corresponding undo information is :

   $(set_repository_entry!\ key\ old_value)$

   If this repository entry does not exist, no undo information is necessary.

Finally, it turns out that data undoing is not difficult to implement generically on such basic repositories of record entries.

5.4.2.7 Behavior Propagation State

An execution trace is also needed besides the BET. The backtracking process has to follow this trace in backward direction. This trace can be simply implemented as a stack of BEN identifiers, from which execution steps can be retrieved in backward order. This trace is in fact stored in a global variable that is a record gathering a collection of information about the current behavior propagation. This record type is of type *bpstate*. The global variable is called `*bpstate*`. The field of the `*bpstate*` used to store the trace followed by the current behavior propagation is the field $trace$. For the sake of optimization, the set of ready BENs is also stored in the `*bpstate*` in the field *nowait-set*. The complete record definition for a $bpstate$ is :

$$\begin{aligned} bpstate \triangleq \ll\ & trace, depth, \\ & nowait - set, nowait - set - stack, \\ & atomic - exec - lev, ben - id - ctr, \\ & test - trace, ops - seqs, \\ & ir - updates, bet - update \gg \end{aligned}$$

5.5 User Interface

Figure 5.9 is a screen snapshot of the control panel available to control the execution of behavior propagations and debug the system. Table 5.1 gives for each button and function involved the corresponding *Scheme* command that could be used instead in the *Scheme* interpreter.

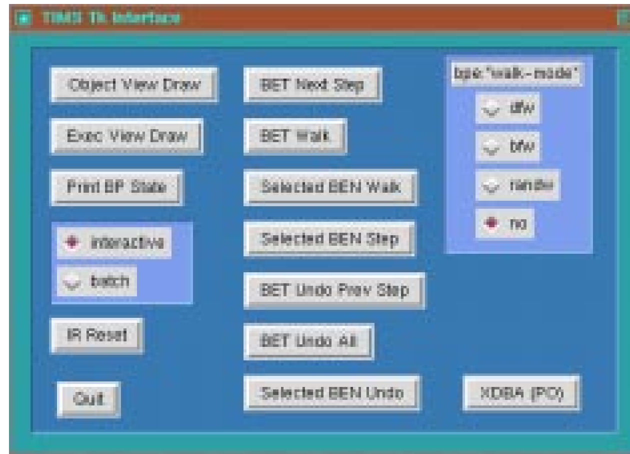


Figure 5.9: User Interface (Execution Control Panel).

Here is a brief descriptive of the functions accomplished :

- Object View Draw : triggers the drawing of the object view.
- Exec View Draw : triggers the drawing of the execution view.
- Print BP State : prints the state of the current behavior propagation, i.e. `*bpstate*` the global variable that comprises the trace followed, ready BENs, etc.

Function / Button Label	*Scheme* Interface
Object View Draw	`(bpe:*irdraw*)`
Exec View Draw	`(bpdraw)`
Print BP State	`(pp *bpstate*)`
IR Reset	`(bpe:*irreset!*)`
BET Next Step	`(bet:walk-step)`
BET Walk	`(bet:walk)`
Selected BEN Walk	`(ben:walk bpe:*selected-ben-id*)`
Selected BEN Step	`(ben:walk-step bpe:*selected-ben-id*)`
Undo Step	`(bet:undo)`
Undo All	`(bet:undo-all)`
Selected BEN Undo	`(ben:undo bpe:*selected-ben-id*)`
Walk mode: DF	`(set!  bpe:*walk-mode* 'depth-first-walk-mode)`
Walk Mode: BF	`(set!  bpe:*walk-mode* 'breadth-first-walk-mode)`
Walk Mode: random	`(set!  bpe:*walk-mode* 'random-walk-mode)`
Walk Mode: NO	`(set!  bpe:*walk-mode* #f)`

Table 5.1: Mapping of Execution Control Panel Buttons to *Scheme*.

- IR Reset : calls the function to reset the information repository. This function is in fact referenced by a BPE global variable that can be changed according to the desire of the user with respect to the initial state required for the information repository.
- BET Next Step : executes the next step of the current behavior propagation according to the current walk mode.
- BET Walk : executes the current behavior propagation until saturation and according to the current walk mode.
- Selected BEN Walk : executes the sub-BET under the selected BEN in the current behavior propagation until saturation and according to the current walk mode.
- Selected BEN Step : executes the next step given by the selected BEN. Here the user develops the BET in a fully controled way. Naturally the selected BEN has to be in a form of ready state.
- Undo Step : undo the last step in the current behavior propagation.
- Undo All : undo all the steps performed in the current behavior propagation.
- Selected BEN Undo : Undo the development of the steps performed by the BENs under the selected BEN. Naturally since undo has to be performed in the backward sequence of the executed steps, undoing a sub-BET may undo execution steps in any other part of the BET.
- Walk mode: DF : sets BPE depth first walk mode.
- Walk Mode: BF : sets BPE breadth first walk mode.
- Walk Mode: random-walk-mode : sets BPE random walk mode.
- Walk Mode: NO : sets BPE NO walk mode. That means that the behavior propagation is stopped until a valid walk mode is selected. This is typically used to initiate a behavior propagation without performing its development.

5.6 Discussion

Simulator = Favorable Environment It should be noted that a simulation environment is a framework where it is much more easy to build improved debugging features such as execution backtracking. In more conventional programming environments debugging support is limited to basic debugging (e.g. trace, breakpoints and stepping). The main reason is that to support backtracking it is necessary to be able to undo the effects of all the statements in a program. Though this is easy for effects that remain internal to a process, this is much more difficult, if not impossible when the program interacts with its environments, e.g. makes systems calls to interact with peripherals or to make communications on a network. So, since providing execution environments where it is possible to undo disk operations or packet sent on a network interface is far from trivial it is clear why improved debugging features are usually not available. In contrast a simulation environment is quite an ideal context for improved debugging support, just because everything about the real world and its environment is virtually represented in structures that are completely under the control of the simulator.

Dynamic Program Slicing There exist other improved debugging features that could also be integrated in the behavior execution environment. In particular, *dynamic program slicing* is a very interesting feature [Agrawal et al.90b] that is an excellent complement to execution backtracking. *Program slicing* consists of finding all those statements that directly or indirectly affect the value of some variable or object. The set of such statements is the *slice* of the program with respect to a given object. If the slices are obtained independently of any execution, that is based only on the program code, they are called *static slices*. However, rather than having the set of statements that can potentially affect an object, it is more interesting to have the set of such statements for a given execution. This is called a *dynamic slice*, and the task consisting to collect information about dynamic slices is naturally called *dynamic program slicing*. Dynamic program slicing is useful and is complementary to execution backtracking because it may help a lot to identify the potential pieces of code that can be suspected to be the root cause of a problem. Indeed, the dynamic slice may give very useful insight about the place to backtrack and from which the problem analysis has to be initiated. Adding dynamic program slicing to the behavior execution environment would not be a big problem. In fact, the collection of the information required to build dynamic program slices is already available. As described in chapter 6, this task is performed for a completely different purpose, i.e. to compute a dependency relation between behavior execution steps. To summarize, at each execution step a trace of all effects performed on information repositories entries is kept in the appropriate BEN field. So, only the user interface to program slices remains to be implemented on top of the dynamic program slicing information already available.

User vs. Language Level Execution Backtracking Support The concept of reverting program state while debugging is not new. In [Agrawal et al.90b] a reference to the use of such a technique for Fortran goes back to the late 60's. Several attempts have been made to incorporate an execution backtracking facility within the programming language itself (e.g. [Teitelman et al.81]). The objective was not to support interactive debugging but to provide a general builtin support to implement backtracking algorithm in general. However the underlying implementation support is the same and thus can be used for both things. The proposed execution backtracking mechanism is provided at the user level[2], i.e. based on undo functions defined on user level data structures (very general repositories of record entries). This choice has been made to keep the implementation of the behavior execution environment as much as possible independent to the underlying programming language used.

[2]Except the only one pure control backtracking required to backtrack the execution control of steps in behavior bodies that is based on *Scheme* continuations.

The Space Problem Since at least execution backtracking performed on user level data structures requires to record the execution history of update statements and the corresponding undo information (e.g. previous values). Hence, for long running programs, e.g. with loops, the execution history + undo information can become very large and can not be determined in advance because it depends on each individual execution. This problem is referenced as the *space problem*. To overcome this problem a usual approach consists to use bounded histories, e.g. on the size of the history list or on the size of the space occupied by the history list. When the history list is beyond the bound, the oldest undo information are discarded as new undo information are pushed on the history. Thus returning to points arbitrarily far back in the execution becomes limited by the physical storage allocated to the undo information. Note that we should not be confronted to this problem because behavior propagations correspond to the execution of scenarios steps that typically remain of manageable size.

5.7 Conclusion

The contribution of this chapter is the execution environment that has been designed and implemented to run the BPE. This environment allows the user to exercise models based on the execution of scenarios or test cases. Details about the contributions of this chapter can be listed as follows :

System monitoring views : Graphical system visualization facilities allow to monitor both data and control issues of the system during behavior propagation. The control visualization facility takes advantage of the fact that the control state is available in the form of normal user level data structures, i.e. the BET.

Improved debugging support : Powerful dynamic debugging features based on execution backtracking have been designed and implemented. Interestingly, the powerful execution backtracking facility has been easily implemented based on *Scheme* continuations and on undo functions working on basic repositories of record entries. The resulting execution environment allows to trace and analyze all that is happening during a behavior propagation. It has been felt quite useful and suitable to develop significant case studies.

Open environment : In addition the execution environment is open in the sense that it can be customized in many ways. The experienced user can define more suitable ways to draw the object and the execution views, as well as the information presented in these views. All these customizations require a very small amount of work. Finally a very important point is that each component of the execution environment is implemented separately from the others. That means that the *Scheme* execution engine, the graph browser, and the execution control panel are distinct applications that can in fact be run independently. They interact together only when each component is launched as an *Emacs* sub-process. To make this possible *Emacs* can be easily customized to filter the standard output of each sub-process and dispatch invocations according to an agreed string interface between interacting components. The main benefit of this separation is that the coupling between components is minimal. The core functionality of each component is well identified. In this context it is trivial to change the implementation of one component because the impact on the other ones is minimal. For instance the graph browsers can be replaced by an implementation performing better, etc. This remark is also valid for the behavior execution engine. Since the behavior execution engine is coded in pure standard *Scheme* [Clinger et al.91] without any dependence on any system, network or GUI function, one can use the more suitable *Scheme* implementation compliant with the standard. One opportunity is to use a *Scheme* implementation with a powerful development environment to proceed in the first phase of a case study. Another opportunity is to use a *Scheme* implementation with less goodies but oriented towards efficiency, e.g. doted with a compiler. The resulting behavior

execution engine can be launched in a standalone mode in order to avoid any overhead that would be caused by a load of the whole execution environment. In this context intensive simulations can be envisioned to allow the support of improved validation techniques such as the one considered in chapter 6.

The execution environment is mentioned in the following publications where the author has contributed : [Sidou et al.96a, Eberhardt et al.97b].

Chapter 6

Improving Validation with eXhaustive Dynamic Behavior Analysis

6.1 Introduction

State-space exploration is one of the most successful strategies for analyzing the correctness of concurrent reactive systems [Holzmann91]. It consists of exploring a directed graph, called the state space, representing the combined behavior of all concurrent components in a system. Such a state space can be computed automatically from a description of the concurrent system specified in a modeling language such as the behavior language proposed in chapter 3. Many properties of a model of a system can be checked by exploring its state space : deadlocks, violations of user-specified assertions, etc. The objective of the chapter is to show what can be achieved in terms of state space exploration from the proposed behavior language, and its operational semantics i.e. the BPE algorithm.

In the remaining of this section the classical state space exploration principle is recalled. Then the particular situation of an exhaustive search based on the BPE presented in chapter 4 is described. The more immediate way to incorporate exhaustive search is to assume in a first step an implicit representation of the system's state and a state-less depth first search algorithm.

With an implicit representation of states, a first level of reduction can be obtained by using the technique of sleep-sets. To obtain more reduction and thus to verify more complex systems, a cache representation can be defined. An with a cache representation state caching can be introduced in combination to sleep-sets.

The plan observed in this chapter is as follows :

- Section 6.2 presents the state-less state space exploration principle and algorithm. The algorithm is instantiated on the structures used by the BPE algorithm that were presented in section 4.8.

- Section 6.3 establishes the rather obvious link between the state-less state space exploration algorithm and labeled transition system (LTS). This does not constitute any significant result as such. This is just worth mentioning that from the proposed behavior model LTSs can be derived. Since LTSs are often used to define a basic / low level operational semantics of modeling languages for concurrent systems, it is normal that this is also applicable to systems specified in the behavior language. The added value of a low level semantics defined in terms of a LTS is that a lot of theoretical results and tools become potentially available.

- Section 6.4 introduces the theory of traces and partial order methods. This theoretical setting is a well known basis for the optimization of state space exploration.

- Since traces semantics and partial order methods are based on a notion of independence between transitions, section 6.5 instantiates this notion with respect to our behavior model.
- Section 6.6 presents the selective state space exploration principle and techniques. It is shown here that the more suitable technique that can be used is the technique based on sleep sets [Godefroid95]. This section also shows how the sleep set selective search principle works.
- Section 6.7 incorporates the use of sleep sets into the basic state-less state space exploration algorithm presented in section 6.2.
- Section 6.8 shows how a cache representation for the system's states can be derived from the existing representation of states that is used to develop the search. This representation can be used to combine sleep-sets (the suitable partial order technique previously incorporated in algorithm 6.7) with state caching. The objective is to retrieve the results already stated in [Godefroid95], being that these two methods are complementary, and thus the reduction of the state space exploration is improved when they are used in combination.
- Section 6.9 illustrates on the simple spanning tree case study the results obtained with the different algorithms available for state space exploration.
- Finally section 6.10 discusses the exhaustive dynamic behavior analysis algorithms obtained in sections 6.7 and 6.8 with respect to validation issues in general.

6.1.1 Classical State-full Search

The global state, denoted by Q is usually computed by performing a search of all the states that are reachable from an initial state s_0. Algorithm 6.1.1 represents a classical way to perform such a search.

Algorithm 6.1.1 *Classical State-full Search (Full Walk) :*

```
CLASSICAL-STATE-FULL-FULL-WALK ()
 1  Init :
 2          Dev ← {s0}
 3          Cache ← ∅
 4  while Dev ≠ ∅
 5      do take s out of Dev
 6          if s ∉ Cache
 7             then enter s in Cache
 8                   ts ← enabled − transitions(s)
 9                   for each t in ts
10                       do s' ← succ (s) after t
11                           add s' to Dev
```

This algorithm recursively explores all successor states of all states encountered during the search, starting from the initial state s_0, by executing all enabled transitions in each state (lines 8-9). The main data structures used are a set Dev to store the states whose successors still have to be explored, and a set $Cache$ to store all the states that have already been developed during the search. This set behaves as a cache of developed states. At each step the presence of a visited state in the cache of developed states is tested. If the state is present in the cache that means that it has already been developed in a past search, and it is not re-developed. The set of all transitions enabled in a state s is denoted by "$enabled - transitions(s)$". The state reached from a state s after the execution of a transition t is

denoted "**succ**(s)**after** t". It is easy to prove that, if Q the global state space is finite, all the states of Q are visited during the search performed by the algorithm 6.1.1.

6.1.2 State Representations

One important thing to note is that algorithm 6.1.1 assumes that each visited state has some kind of representation, that can be stored in the data structures Dev and $Cache$ during the search. An again more important thing is that this representation of states has to support state comparison (line 6) and state development (line 10) to be performed on it.

In fact, after a closer look at algorithm 6.1.1 one should note that at least two kind of representations can be distinguished. A first representation is the one used to store states in Dev. This representation is typically used to develop the search, i.e. to compute successors of visited states using the function "**succ**". We qualify this representation as the *development* representation. Another representation that can be identified is the one used to store states in $Cache$. This representation is naturally called the *cache* representation.

One important point is that different storage models are possible to store both development and cache representations of system's states. In particular a distinction can be made in terms of "user level" structures or in terms of "machine level" structures.

A user level structure is a data structure that can typically be defined using the programming language supporting the implementation of the search algorithm, e.g. basic type, array, structure, hash table, etc. The important point is that a user level structure does not depend on the low level execution support. On the other hand, "machine level" structures are typically supported by the low level execution environment. For instance, the state representation can be defined by the state of the process executing the search in a given operating system and machine architecture. That means that the system state is directly represented using facilities provided by a programming language and the underlying runtime support, e.g. C variables referencing memory locations on the runtime stack, the process heap, etc.

Depending on the system, the semantics and the implementation support used for the search algorithm, not all representations are suitable for caching and development. It turns out that user level structures tend to be used for cache representations because cache representation requires state comparison operation to be available, which is usually straightforward to implement on user level data structures. Though it may be possible for a given system to rely on user level data structure for the development representation, for other systems, this may not be appropriate if not impossible. Thus machine level structures such as a runtime stack, the heap may provide immediate support for the development representation. Since in conventional programming languages machine level structures are usually not manipulable by the user, it is difficult if not impossible to use a machine level representation as a cache representation (just because it is impossible to define the state comparison operation on it). In addition a development representation typically appears in a completely implicit way. That means that in the end, the set of operation supported may be limited to the successor and the predecessor operations. As a consequence, the set Dev in algorithm 6.1.1 does not actually exist. Or it may be viewed as a set with a single element that is the current state in the search. In this context, the exhaustive search has to be performed depth first. This is called a *depth first search* (DFS).

This hybrid situation corresponds to our context of work in terms of development representation for our system state. User level data structures support the information repository and the behavior execution tree (BET) composed of behavior execution nodes (BENs). However, one part is implicitly defined by low level execution structures. As presented in chapter 4 the intermediate execution steps in behavior bodies are based on continuations. Continuations are a powerful control structure available in the current execution support based on the *Scheme* programming language.

Note that to provide state caching a mapping has to be defined from the development representation

to a cache representation. Since this mapping appears only because of the problem of the exhaustive search, this is something that is not immediately available from the existing behavior execution and debugging environment described in chapter 5. Thus a suitable definition of a cache representation requires additional design and implementation effort. Section 6.8.1 shows how it is possible to define a cache representation of visited states in our execution model. However, in a first step since we are interested to provide exhaustive search with minimal effort, the available departure point that is assumed is an implicit representation of state and a depth first search. This can be described using the state-less depth first search algorithm presented in the next section.

6.2 BPE State-less Search / Full Walk Algorithm

6.2.1 Principle

Algorithm 6.2.1 gives the principle for the exploration of all behavior propagations that the BPE algorithm presented in algorithm 4.8.2 can go through from the initial state $s_0 = (bet_0, ir_0)$.

Algorithm 6.2.1 *State-less Search / Full Walk :*
BPE:STATE-LESS-FULL-WALK(tr)
1 $ready - bens \leftarrow \{ben \in BET : ben.state \in \{$ *ready-pre* , *ready0* , *ready* , *ready-post* $\}\}$
2 **if** $ready - bens = \emptyset$
3 **then** // *S is a termination state*
4 $dump - trace(tr)$
5 **else for each** ben **in** $ready - bens$
6 **do** BEN:EXEC(ben)
7 **push** $*undo - info*$ **on** $ben.undo - infos$
8 BPE:STATE-LESS-FULL-WALK$(tr.t)$
9 $undo($**pop** $ben.undo - infos)$

In line 6, a next step of an active BEN is executed from a system configuration $s = (bet, ir)$ by calling the function BEN:EXEC. As a side effect to the execution of this function, undo information is stored in the global variable $*undo - info*$. This undo information includes all the necessary information that is needed to undo this processing step, i.e. all the side effects exercised on the underlying BET and IR representing the complete configuration of the system. At line 7, the undo information stored is recorded in the BEN by pushing it in the $undo - infos$ field that is managed as a stack. Once the search from that new state (and hence the corresponding call to BPE:STATE-LESS-FULL-WALK) is completed, the exploration of the other BENs selected to be explored from $s = (bet, ir)$ is done. The system is then brought back to the global state s in line 9, by popping the corresponding undo information from the $undo - infos$ stack field.

From algorithm 6.2.1 it appears clearly that a depth first search is performed using an implicitly representation of the system state. The only operations available on this development representation are a successor operation and a predecessor operation. The successor operation is done with the BEN:EXEC function. The predecessor operation is supported by the undoing function.

Note that undoing support is immediately available by reusing the dynamic debugging mechanisms presented in section 5.4. These debugging mechanisms allows to perform incremental undo. As suggested in [Godefroid97], one can think of undoing the last processing step by reinitializing the behavior propagation and re-executing the sequence of transitions in the behavior execution followed. Though possible, this way of undoing might be time consuming when used in the context of a full walk.

This algorithm is a natural starting point because it can be directly implemented by using the execution backtracking facilities already implemented for the purpose of debugging. However, as shown below, this algorithm provides a very inefficient implementation of the exhaustive search.

6.2.2 Inefficiency of a State Less Search

Nothing prevents from systematically searching the state space of a concurrent system without storing any intermediate states in memory. Of course, if the global state space contains cycles, a state-less search does not terminate, even if the global state space is finite. That means that if an explicit representation of states is available, the classical state exploration algorithm 6.1.1 may terminate in a reasonable amount of time. Indeed, with the state-less search algorithm 6.2.1, even searches of "small" finite acyclic state spaces (e.g., composed of only a few thousand states) may not terminate in a reasonable amount of time. This phenomenon, is illustrated in [Godefroid97] on the dining-philosophers example, where the problem of the dining-philosophers is defined such as the state space of the system does not contain any cycles. The number of transitions explored by a classical search and by a state-less search are compared, for various numbers of philosophers. The run-time of both algorithms is proportional to the number of explored transitions. From [Godefroid97], it is clear that the state-less search is significantly slower than the classical state-full one. In the case of four philosophers, the state-less search explores 386816 transitions, while they are only 708 transitions explored by the classical search. While every transition is executed exactly once during a classical search, it is executed on average about 546 times during a state-less search ! This tremendous difference is due to the numerous re-explorations of parts of the state space not stored in a cache during the state-less search. The inefficiency of the state-less search lead to the conclusion that it is unusable to perform an exhaustive checking. The objective of this chapter is to try to equip the state-less search with some kind of reduction technique, so that exhaustive search become possible in much more circumstances.

6.3 Operational Semantics and Labeled Transition Systems (LTS)

A convenient way to represent the behavior of a system is to use an operational approach. It describes the transitions between states that the system can execute. Therefore, the behavior can be represented as an oriented graph usually called a *transition system*. Nodes are the set of states the system can pass through. Arcs denote the transitions between states. Transitions can be labeled with a description of the corresponding activity. This defines the notion of *Labeled Transition System* (LTS). Transitions are the atomic units of execution. A LTS can be finite or infinite. A LTS can be deterministic or nondeterministic. In a deterlinistic LTS, from a given state s an enabled transition t can lead to a unique successor state. Formally determinism can be defined as follows :

$$s \xrightarrow{t} s' \text{ and } s \xrightarrow{t} s'' \Rightarrow s'' = s'$$

Note that we are interested only in finite and deterministic LTSs.

Definition 6.3.1 *Labeled Transition System (LTS) : A (finite and deterministic) Labeled Transition System (LTS) is a structure* (Q, l, O, T, s_0) *where :*

- Q *is a finite set of global states.*
- $T \subseteq Q \times Q$ *is a finite set of transitions.*
- O *is a set of operations used to label the transitions.*

- *l is a labeling function, $l : T \longrightarrow O^*$ that associates to each transition the sequence of operations that can be observed when the transition is fired.*

- *s_0 is the initial state.*

It is clear that algorithm 6.2.1 develops a LTS. By construction, the LTS built is deterministic. However, the behavior model does not prevent infinite LTSs to be generated. For instance, a loop resulting as an infinite behavior propagation can be introduced into a system very easily. In such a case no support is provided to try to represent finitely such infinite LTSs using techniques such as the ones mentioned in [Quemener96]. In the presence of infinite loops or too long behavior propagations the classical and pragmatic solution is to stop the behavior propagation at a given fixed depth of the behavior execution tree. Though the resulting LTSs are incomplete the representation obtained is always finite. Note that Q, T, O, and l are not determined a priori, rather they result a posteriori from the development performed by algorithm 6.2.1. In addition, $s_0 = (bet_0, ir_0)$ is the state of the system resulting from the bootstrapping of a behavior propagation as it is described in section 4.8.1. Linking algorithm 6.2.1 with the concept of LTS is rather trivial and does not constitute any significant result as such. LTSs have been recognized for a long time as a common framework to define the operational semantics for modeling languages of concurrent systems. In fact LTSs constitute a low level form of operational semantics onto which any higher level form of operational semantics can be mapped to. That is the reason why a lot of work (theoretical results, tools [Fernandez et al.96a]) has been done based on LTS semantics. So it is at least useful to mention that to the high level behavior model presented in section 3 corresponds a low level LTS semantics. This allows for instance to link existing generic verification tools working at the level of LTSs. This has concretely been done in [Mazziotta97] for the purpose of test generation, with the cæsar / aldebaran development package [Fernandez et al.96a] and the test generation and verification package [Fernandez et al.96c, Fernandez et al.96b].

6.4 Traces and Partial Order Methods

To check for safety properties, it is not actually necessary to perform an exhaustive search on the complete reachability graph of the system under investigation. The intuition behind partial-order methods is that concurrent executions are really partial orders and that expanding such a partial order into the set of all its inter-leavings is an inefficient way of analyzing concurrent executions. Instead, some concurrent transitions can be left unordered (or equivalently it is necessary to execute one single order) since the order of their occurrence may be irrelevant. Precisely, given a property, partial-order methods explore only a part of the global state space that is reduced as much as possible and that is sufficient to check the given property. The difference between the reduced and the global state spaces is that all inter-leavings of concurrent transitions are not systematically represented in the reduced one. Which inter-leavings are required to be preserved may depend on the property to be checked. In this section it is explained how the exploration of some inter-leavings can be avoided by simply following the commutativity principle of consecutive independent transitions. But, first the notion of concurrent alphabet of transitions has to be defined. It is one fundamental structure of Mazurkiewicz's trace theory [Mazurkiewicz86].

Definition 6.4.1 *Concurrent alphabet of transitions : T is a concurrent alphabet of transitions if, T is equipped with a symmetric, irreflexive, binary relation $I \subseteq (T \times T)$ – the independence relation between transitions.*

Elements of T correspond to atomic units of execution also called transitions. During the atomic execution of a transition, operations are performed on the system, e.g. read, write operations on the underlying memory. Transitions are said to be decorated or labelled with these operations. Informally, $(a, b) \in I$ means that in any computation where a and b can be executed concurrently, in the interleaving execution model, they can be serialized in any order. More formally :

Definition 6.4.2 *Independence of transitions : a, b are independent transitions means that : for all states q such that $q \xrightarrow{a} r$, and $q \xrightarrow{b} s$, then $(a, b) \in I$ involves that : for some state p there exist transitions $s \xrightarrow{a} p$, and $r \xrightarrow{b} p$.*

Definition 6.4.2 is illustrated in figure 6.1. Note that I is symmetric. In addition, two non-independent transitions in a state s are dependent. This defines a dependency relation $D : (a, b) \in D$ if $(a, b) \notin I$. Since, I, D are always defined relatively to a system state s, they can be denoted by I_s and D_s.

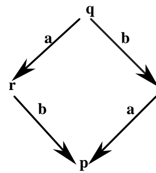


Figure 6.1: Independence of Transitions.

A fundamental result of Mazurkiewicz's trace theory is that I the independence relation induces a congruence $\equiv$ on T^* the free monoid of transition sequences on T. The quotient $T^*/\equiv$ is the free partially commutative monoid generated by $\equiv$.

Definition 6.4.3 *Trace : The elements of $T^*/\equiv$ are called traces.*

ϵ denotes the monoid identity (the empty transition sequence), and for all $x \in T^*, [x]$ denotes the corresponding element of $T^*/\equiv$. Two transition sequences $x, y \in T^*$, are congruent (noted $x \equiv y$), iff one can be transformed into the other by a finite sequence of steps in which only pairs of consecutive (adjacent) and independent transitions are permuted.

In Mazurkiewicz's trace semantics, the behavior of a concurrent system is defined as a set of traces. Mazurkiewicz's trace semantics is often referred to as being a *partial order* semantics because it is possible to define a correspondence between traces and partial orders of occurrences of transitions.

Definition 6.4.4 *Partial Order : A relation $R \subseteq A \times A$ on a set A that is reflexive, antisymmetric, and transitive is called a partial order. A partial order $R \subseteq A \times A$ is also a total order if, for all $a_1, a_2 \in A$, either $(a_1, a_2) \in R$ or $(a_2, a_1) \in R$.*

A partial order $R \subseteq A \times A$ can be represented graphically by a directed graph whose vertices are elements of A and whose edges are elements of $R : (a_1, a_2) \in R$ iff there is an edge from a_1 to a_2.

Definition 6.4.5 *Linearization of a partial order : A linearization of a partial order $R \subseteq A \times A$ is a total order $R' \subseteq A \times A$ such that $R' \supseteq R$.*

Theorem 6.4.1 *Linerizations and partial order : The intersection of all the linearizations of a partial order is that partial order.*

The preceding theorem states that a partial order can be represented by the set of its linearizations (e.g. [Pratt86]). A correspondence between traces and partial orders of transition occurrences can be defined in such a way that the set of transition sequences in a trace is the set of all linearizations of the corresponding partial order of transition occurrences.

Theorem 6.4.2 *Traces and state reachability :* s, s_1, s_2 *are states in* Q, w_1, w_2 *are transition sequences. If* $s \stackrel{w_1}{\Longrightarrow} s_1$ *and* $s \stackrel{w_2}{\Longrightarrow} s_2$*, and if* $[w_1] = [w_2]$*, then* $s_1 = s_2$.

If $w = t_1 t_2 \cdots t_n$ then $s \stackrel{w}{\Longrightarrow} s'$ denotes $s \stackrel{t_1}{\longrightarrow} s_2 \stackrel{t_2}{\longrightarrow} \cdots s_{n-1} \stackrel{t_n}{\longrightarrow} s'$.

Proof of theorem 6.4.2 [Godefroid95] By definition, all $w' \in [w]$ can be obtained from w by successively permuting pairs of adjacent independent transitions. It is thus sufficient to prove that, for any two words w_1 and w_2 that differ only by the order of two adjacent independent transitions, if $s \stackrel{w_1}{\Longrightarrow} s'$ then $s \stackrel{w_2}{\Longrightarrow} s'$. Let's assume that $w_1 = t_1 \cdots ab \cdots t_n - 1$ and $w_2 = t_1 \cdots ba \cdots t_n - 1$. We have
from $w_1 : s \stackrel{t_1}{\longrightarrow} s_2 \stackrel{t_2}{\longrightarrow} \cdots s_i \stackrel{a}{\longrightarrow} s_{i+1} \stackrel{b}{\longrightarrow} s_{i+2} \cdots \stackrel{t_n-1}{\longrightarrow} s_n$, and
from $w_2 : s \stackrel{t_1}{\longrightarrow} s_2 \stackrel{t_2}{\longrightarrow} \cdots s_i \stackrel{b}{\longrightarrow} s'_{i+1} \stackrel{a}{\longrightarrow} s'_{i+2} \cdots \stackrel{t_n-1}{\longrightarrow} s'_n$.

Since $(a, b) \in I$, it follows that $s'_{i+2} = s_{i+2}$, and since the transitions in w_1 from s_{i+2} and in w_2 from s'_{i+2} are identical, we have $s'_n = s_n$. $\diamond$

From Theorem 6.4.2, it follows that, in order to determine if a state is reachable by any sequence of transitions in a trace, it is sufficient to explore only one sequence in that trace. This property is basically what can be used to explore only a reduced part of the global state space Q of a system in order to prove properties of that system.

6.4.1 Principle for an Efficient Detection of Termination States

A termination state or deadlock in a system is a state that is reachable from the initial state s_0 of the system and where there is no enabled transition. If there is a deadlock d in Q, there is a sequence w of transitions from s_0 to d in Q, and hence a trace $[w]$ from s_0 to d in Q. Since all sequences $w' \in [w]$ also lead from s_0 to d, it is sufficient to explore only one of the $w' \in [w]$ to visit d, and thus to detect it. Consequently, it is sufficient to explore only one interleaving for each trace the system can execute from its initial state in order to detect all deadlocks d in this system. Deadlock detection is thus reduced to the problem of exploring (at least) one interleaving per trace the system can execute from its initial state.

6.4.2 Principle for an Efficient Detection of Assertion Violations

The same reasoning can be made for assertion violations. In effect the detection of assertion violations can be reduced to the problem of the detection of states v where transitions representing assertions are enabled, and if executed are violated. To explore a state v where an assertion is violated, it is sufficient to explore only one interleaving of the trace the system can execute from its initial state to v. Note that either the system reaches a termination state, either the system reaches an assertion violation. Consequently, the detection of deadlock and the detection of assertion violations can be reduced to the problem of exploring (at least) one interleaving per trace the system can execute from its initial state.

6.5 Independence Relation on Transitions

In the previous section, it has been described that from an independence relation between transitions :

- following [Mazurkiewicz86] a trace semantics of concurrent systems can be defined.
- from the trace semantics, theorem 6.4.2 is a key result allowing to prove properties about concurrent systems without exploring the global state space Q of the system, i.e. using a selective search that explores one member of each trace is sufficient. Depending on the independence relation and the traces obtained this may reduce significantly the state exploration.

The important question that follows now is : when are transitions considered to be independent ? The intuitive idea is that transitions are independent when the order of their occurrence is irrelevant. This was formally defined in definition 6.4.2. Note that definition 6.4.2 implicitly states that independent transitions can not disable each other. This definition characterizes the properties of possible "valid" dependency relations for the transitions of a given transition system. The next section instantiates the concept of independence between transitions in our execution model.

6.5.1 D Instantiated on (IR, BET) Configurations

The state of the system is defined by the configuration of the system that comprises a control part and a data part. The data part contains the information that is read and written by behavior code specified by the user. The data part is the contents of the information repository (IR). The control part is implicitly defined by the control semantics of the system as it was presented in chapter 4. At each execution step it is precisely defined by the contents of the behavior execution tree (BET). Thus the whole configuration of the system is defined in its turn by a pair (IR, BET). The set of transitions enabled in a state $s = (BET, IR)$ is given by the set of behavior execution nodes in the BET that are in a form of ready state, i.e. either $ready - pre$, $ready$, $ready0$, or $ready - post$. In this context the general independence relationship between transitions can be reformulated as follows[1] :

Let ben_1 and ben_2 two ready BENs representing two enabled transitions in $s = (IR, BET)$, ben_1 and ben_2 are independent iff :

$$s \xrightarrow{ben_1} s_1^{'} = (IR_1^{'}, BET_1^{'}) \xrightarrow{ben_2} s^{'} = (IR^{'}, BET^{'})$$

then we have also :

$$s \xrightarrow{ben_2} s_2^{'} = (IR_2^{'}, BET_2^{'}) \xrightarrow{ben_1} s^{'} = (IR^{'}, BET^{'})$$

In other words, from s executing ben_1 followed by ben_2 leads to the same state as executing ben_2 followed by ben_1. It is important to note that by definition of the execution semantics, even if $(ben_1, ben_2) \in D$, ben_2 would still be enabled in $s_1^{'}$. That means that the execution of a BEN cannot disable another enabled BEN. The only thing that can occur is that ben_2 may execute differently, i.e. the activity observed is different if the transitions are dependent. As a consequence, BEN executions can interfere only on the basis of IR operations. So in order to detect dependencies between transitions (or BEN execution steps) it is only necessary to monitor the corresponding activity exercised on the IR. In the end the dependency relation has to be defined on IR operations sequences that are performed by the execution of each BEN execution step.

6.5.2 D_{op} : the Dependency Relation on Basic IR Operations

In this section, an example of dependency relation is shown based on the current in memory implementation of the IR. In the current in memory implementation, the IR is a repository of record entries. On top of this basic model, higher level information abstractions such as objects containing instance variables and relationships with role members can be implemented without any difficulty. However,

[1] Only conditional dependency relation are considered, i.e. a dependency relation as presented in [Godefroid95] that is defined with respect to each state.

the dependency relation can be described in terms of the basic model of a repository of record entries. In this context, the available IR operations are the following :

1. $(get_record_field\ reccord\ field)$, $record$ is a reference to a record object; field is a field name in this record.
2. $(set_record_field!\ record\ field\ value)$.
3. $(get_repository_entry\ key)$, typically returns a reference to a record object.
4. $(set_repository_entry!\ key\ value)$. If the repository entry does not exist it is created, otherwise the previous entry is replaced by the new one. $value$ is typically a reference to a record object.
5. $(delete_repository_entry!\ key)$.

The dependency relation D_{op} between basic operations can then be defined by these two rules :

1. *set_repository_entry!* and *delete_repository_entry!* operations are dependent with any other operation on the same repository entry, i.e. access is done with the same key.
2. a *set_record_field!* is dependent with a *set_record_field!* or a *get_record_field* operation on the same record reference and field name.

If none of these two rules match the operations are considered to be independent. Since BEN execution steps result in sequences of such basic operations the dependency relation on whole operation sequences is defined as :

$$(ops_1, ops_2) \in D \text{ if } \exists op_1 \in ops_1, op_2 \in ops_2 \text{ such that } (op_1, op_2) \in D_{op}$$

6.5.3 Refining D_{op}

As presented in [Godefroid95] basic operations on IR entries can be refined to obtain more independence between basic operations. For instance, instead of mere *set* operation one can define specific operations on boolean values such as *compl* the complementation operation. Since two complementation operation on the same boolean variable are independent, to obtain more dependency at runtime, the user may be more interested to use each time this is possible the *compl* operation instead of a low level form built from the basic *get* and *set* operations. Many other options are possible for operation refinement to gain more independence :

- used of *add* and *remove* on set valued attributes.
- define independence based on the specific semantics of operations and objects, e.g. for a bounded FIFO channel, the *send* and *receive* operations are independent if there is more than one element in the channel.

In the end, one can go very far in terms of better independence relation that can be obtained by operation refinement. One problem is that independence based on refined operations may result in (i) a more complex instrumentation support required to monitor the occurrence of such operations at runtime, and (ii) naturally in more complex algorithm for the dependency relation itself (typically levels of precedence have to be observed between levels of operations and dependencies checked). In addition fine grain independence are the ones that are the less often used at runtime. However, depending on the application and the executions involved, more independence may result in a significant amount of runtime saved. The independence relation used in the current implementation can be qualified as coarse grain because it is based on the very simple operations presented in section 6.5.2.

6.5.4 Introducing More Dependency to Keep more Behavior

If the dependency relation is based only on IR operations, only termination states that are different in terms of the IR, i.e. of data state, are obtained. In addition all the states where an assertion is violated are also obtained. Though this is still an interesting result, one may be interested to keep more behaviors. Typically for the purpose of testing and in particular for test generation, one important point is the notion of *observable behavior*. Observable behavior is typically based on messages sent that are related to a well identified point of control and observation (PCO). Since tests are typically given as the sequences of messages occurring (either as input or as output) at a given PCO, to obtain all such valid sequence it is possible to strengthen the dependence relationship in order to keep all observable behavior at a PCO. To this end it is merely necessary to monitor the sending of messages on the interesting PCO, to include them in the set of operations observed (in addition to the IR operations) and to state that two such operations sent on a same PCO are dependent. On the other hand operations sent on different PCOs are considered to be independent. In addition a PCO operation and a non-PCO operation (i.e. IR operations) are also considered independent. This approach is presented in details and implemented in [Mazziotta97].

6.6 Selective Search / Exploration

6.6.1 Selective Search / Exploration Principle

The problem of exploring only one path ($w \in [w]$) of the trace of all interleaving leading from the initial state s_0 to a given state s can be solved by performing what is called a selective search in Q [Godefroid95]. A selective search operates as a classical state space search except that, at each state s reached during the search, it computes a subset et' of the set of transitions that are enabled in s (i.e. $et' \subseteq et$), and explores only the transitions in et', the other enabled transitions not being explored. Clearly, a selective search through Q only reaches a subset (not necessarily proper depending on the property to be checked) of the states and transitions in Q. Note well that if, in each visited state s, the first transition of at least one interleaving per trace leading to a deadlock or error state is selected in the set et' of transitions to be explored from s, all deadlocks and error states in Q are eventually visited by such a selective search. Therefore all the problem consists of finding a selection strategy preserving this property. Hereafter, the resulting selective search is called a *property preserving selective search*, and the selection strategy a *property preserving selection strategy*. In the context of algorithm 6.2.1 a selective search consists merely of considering (at line 1) a subset of the BENs that are active in the BET, instead of the entire set of ready-BENs.

6.6.2 Selective Search / Exploration Techniques

Two main families of property preserving selection strategies have been devised in the litterature : the *persistent-set*[2] and the *sleep-set* techniques. The main difference between the two families of techniques is that persistent-set techniques are based on information about the future of the search, in contrast the sleep-set technique is based on information about the past of the exploration.

6.6.2.1 Principle of Persistent-Sets

Persistent-sets are infered from the static structure of a model specification. Persistent-sets techniques differ by the type of information they can use in a given specification language to deduce that certain

[2]The persistent-set techniques is actually a generalization of techniques such as the stubborn-sets [Valmari90]. In [Godefroid95], it is shown that these techniques all compute persistent-sets.

facts (e.g. certain conflicts) will not happen in the future of the search. Deducing facts about the future of the search is only possible for suitable specification frameworks. In particular this has been performed successfully for petri-nets and (extended) finite state machine models. That is specification frameworks where an explicit and static representation of the control flow in the system is available. In such contexts several algorithms have been proposed to compute persistent-sets. Many details about persistent-sets can be found in [Godefroid95].

6.6.2.2 Unsuitability of Persistent-Sets

Since of their static inference nature persistent-set techniques are not suitable in our behavior modeling framework. In effect, the proposed behavior language is supported by a language mapping onto an existing programming language (*Scheme*). Therefore at runtime, the BPE machinery executes *Scheme* code in the *Scheme* runtime environment in a totally blind way. That means that the BPE is not aware of the *Scheme* statements being executed. In fact, when behavior definitions are loaded into the *Scheme* environment, very limited static code analysis of behavior definitions is performed. The behavior parsing consists merely of encapsulating the *Scheme* code given in behavior clauses within *Scheme* closures. As a result no particular static analysis at the level of *Scheme* code in behavior clauses is done. From the implementation perspective a behavior is just a record with fields corresponding to behavior clauses implemented as usual functions in the underlying language implementation. One key advantage is that this makes the internal structure of a behavior reasonably open to any language mapping. In contrast this choice is rather incompatible with any kind of static analysis of the code given in behavior clauses (though this may be done for a particular language mapping). That is the reason why we have not investigated selective search based on persistent-sets techniques. At least it is more reasonable to investigate in a first phase the sleep-sets technique which is in contrast directly applicable.

6.6.3 Principle of Selective Search based on Sleep-sets

A sleep set is a set of transitions associated with each state s visited during the state space exploration. Basically, the sleep set associated with a state s is a set of transitions that are enabled in s but that will not be executed from s. The sleep set associated with the initial state s_0 is the empty set : ($s_0.sleep = \emptyset$). A sleep set can be defined incrementally for a state s' that is the successor of a state s from the sleep set associated to s ($s.sleep$). Let $et' \subseteq et$ be a subset of enabled transitions that have been selected to be explored from s (et represent the entire set of enabled transitions in s). Let us take a first transition t_1 out of et'. The sleep set associated with the state reached after executing t_1 from s is the sleep set associated with s unmodified except for the elimination of the transitions that are dependent on t_1. In other words, only the transitions of the sleep set associated with s that are independent on t_1 in s are passed to the sleep set associated with the state reached after executing t_1 from s. Let t_2 be a second transition taken out of et'. The sleep set associated with the state reached after executing t_2 from s is the sleep set associated with s augmented with t_1, minus all transitions that are dependent with t_2 in s. One proceeds in a similar way with the remaining transitions of et'. The general rule is thus that the sleep set associated with a state s' reached by a transition $t \in et'$ from a state s is the sleep set that was obtained when reaching s augmented with all transitions already taken from et' in previous explorations already performed from s, and purged of all transitions that are dependent with t in s. In figure 6.2 the sleep set selective search principle is illustrated. Let s_m be a state in a path w between s_0 and s_n of the form :

$$w = s_0 \xrightarrow{t_{i_0}} s_1 \cdots s_m \xrightarrow{t_{i_m}} s_{m+1} \cdots s_n$$

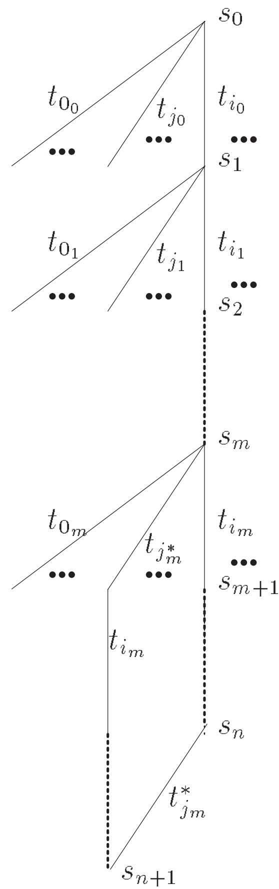


Figure 6.2: Sleep Set Principle.

In a state $s_m, m < n$, t_{i_m} is the transition that is taken at state s_m to follow w, and $s_m.xet = \{t_{j_m}, 0 \leq j_m < i_m\}$ is the set enabled transitions already explored from s_m in past searches. Formally, if we try to characterize the sleep set in the state s_n we have :

$$s_n.sleep = \begin{array}{l} \left\{ \begin{array}{ll} t_{j_0}, j_0 < i_0 : & (t_{j_0}, t_{i_0}) \in I \wedge \\ & (t_{j_0}, t_{i_1}) \in I \wedge \\ & \cdots \\ & (t_{j_0}, t_{i_{n-1}}) \in I \end{array} \right\} \cup \\ \left\{ \begin{array}{ll} t_{j_1}, j_1 < i_1 : & (t_{j_1}, t_{i_1}) \in I \wedge \\ & \cdots \\ & (t_{j_1}, t_{i_{n-1}}) \in I \end{array} \right\} \cup \\ \cdots \\ \cdots \\ \cdots \\ \{\ t_{j_{n-1}}, j_{n-1} < i_{n-1} : \ (t_{j_{n-1}}, t_{i_{n-1}}) \in I\ \} \end{array}$$

or :

$$s_n.sleep = \{t_{j_m} : (t_{j_m}, t_{i_m}) \in I, 0 \leq m < n, 0 \leq j_m < i_m\}$$

Note that in $s_n.sleep$, the sleep set in s_n, the only transitions remaining from $s_m.xet$ are the ones independent to all that has occurred in w after s_m, i.e. all the transitions taken in w from s_m. Since all sleeping transitions coming from $s_m.xet$ noted $t^*_{j_m}$ in the sleep set of s_n are all independent to all transitions in w_2. w_2 is the suffix path of w representing the transitions from s_m in w, so that we have $w = w_1 w_2$. The pruning principle exercised by the sleep set selective search algorithm is based on the property that it is useless to execute any $t^*_{j_m}$ in s_n. This is useless in the sense that this can not lead to a state that has not been explored before. In effect, suppose $t^*_{j_m}$ is executed in s_n, we have :

$$s_0 \xrightarrow{t_{i_0}} s_1 \cdots s_m \xrightarrow{t_{i_m}} s_{m+1} \cdots s_n \xrightarrow{t^*_{j_m}} s_{n+1}$$

And since $(t_{j_m}, t_{j_l}) \in I, m \leq l < n$, by consecutively permuting t_{j_m} with $t_{i_{n-1}}, t_{i_{n-2}} \cdots$ until t_{i_m}, so we obtain :

$$s_0 \xrightarrow{t_{i_0}} s_1 \cdots s_m \xrightarrow{t^*_{j_m}} s' \xrightarrow{t_{i_m}} s'' \cdots s_{n+1}$$

So we can see that s_{n+1} is reachable from the exploration where $t^*_{j_m}$ was selected for execution in s_m. Since $j_m < i_m$ this exploration belongs to a past search from which s_{n+1} is reachable.

6.6.4 Properties of Sleep-sets

The sleep-set technique can be used to extend the state less full walk described in algorithm 6.2.1. In [Godefroid95] it is shown that such a sleep set selective search visits all reachable states. Note that this is not the case if the sleep-set technique is used in combination with the persistent-set technique. In other words, sleep sets used alone cannot reduce the number of states visited in Q. However, they can reduce the number of transitions executed, which is still very useful. An interesting consequence is that with sleep sets all termination states and assertion violations are detected. Note that if we were using persistent-sets, this may not be the case because with persistent-sets some states can be ignored in the search. This is referenced as the *ignoring problem* [Valmari90].

6.7 An Efficient State-less Search Algorithm

Function BPE:SLSSFW described in algorithm 6.7.1 performs a selective search using only the technique of sleep-sets as the selection strategy. It takes two arguments : (i) $sleep - set$ is the sleep set associated with the state of the system currently visited and from which the exploration is considered; and (ii) tr is the trace that was followed to reach s from the initial state s_0. Each element in the sleep-set is noted $sleep - elt$. It is a pair $sleep - elt = (ben, ops)$. ben is a reference to the BEN from which the sleeping transition was executed in a past exploration, and ops is the list of operations that were performed during the execution of the transition. An important point is that a sleeping transition is enabled, and if it was executed it would exercise the same activity as the one that was exercised when it was executed in the past search. Concretely, that means that the same list of IR operations as the one given by $sleep - elt.ops$ would be produced.

Algorithm 6.7.1 *State-less Sleep Set Search / Full Walk Algorithm :*
BPE:SLSSFW$(sleep - set, tr)$
1 $ready - bens \leftarrow \{ben \in BET : ben.state \in \{ready - pre, ready0, ready, ready - post\}\}$
2 **if** $ready - bens = \emptyset$
3 **then** $dump - trace(tr)$
4 **else** $not - sleeping - ready - bens \leftarrow ready - bens \setminus \{ben : (ben, ops) \in sleep - set\}$
5 **for each** ben **in** $not - sleeping - ready - bens$
6 **do** BEN:EXEC(ben)
7 **push** $* undo - info *$ **on** $ben.undo - infos$
8 $new - sleep - elt \leftarrow (ben, *ops*)$
9 BPE:SLSSFW$(sleep - set \setminus \{(ben', ops') \in sleep - set : (ops', *ops*) \in D\},$
10 tr **extended with** $ben)$
11 $undo($**pop** $ben.undo - infos)$
12 $sleep - set \leftarrow sleep - set \cup new - sleep - elt$

In line 4, BENs that are in the current sleep set need not be explored and they are removed from set $ready - bens$. All transitions selected to be explored, i.e. in set $not - sleeping - ready - bens$, are explored (line 5), and the sleep set that is to be associated with each successor state of s is computed (line 9) following the procedure described previously in section 6.6.3.

At line 6, the call to the function BEN:EXEC makes side-effects on two global variables. (i) the undo information required for backtracking is stored in the global variable $*undo - info*$, and (ii) the list of operations representing the activity exercised by the executed transition is stored in the global variable $*ops*$. Variable $*ops*$ includes all the necessary information that is needed to test for dependency between execution steps as described in section 6.5.

6.8 State Caching Combined with Sleep-Sets

As noted in section 6.6.4, an important property of the partial order technique based on sleep-sets is that sleep-sets do not yield any reduction on the number of visited states. This is important because this allows to detect all termination states and states where an assertion is violated. So the benefit of using sleep-sets is to try to limit the number of times the same state is (re)-visited.

One important cause resulting in many visits of a same state is the interleaving execution model. Concurrent or independent executions are emulated by executing all the interleavings of their respective atomic execution steps. By taking advantage of the independence relation existing between these

execution steps, the sleep-sets technique performs very well at avoiding to visit several times a state because of interleavings.

However, another cause resulting in many visits of a same state is that alternative and different execution paths may lead to the same state, just because there may be different ways to do exactly the same thing in a given application. For this cause of several visits, the sleep-sets technique does not help. One way to circumvent this problem is to use state caching. That is a cache representation of visited states has to be defined. Then after each execution step, the cache representation is determined, and if this cache state is already in the cache of visited states this branch of the search can be pruned. Note that, in contrast to sleep-sets, state caching does not exactly avoid to re-visit a given state. More precisely, state caching avoids to re-develop a state already developed in a past search. This is naturally the more important.

Section 6.8.1 introduces how the definition of a cache representation is possible from the existing development representation based of (IR, BET) configurations. Section 6.8.2 extends algorithm 6.7.1 by adding state caching. Finally, in section 6.9 the benefits of combining sleep-sets with state caching are illustrated on a simple examples related to the computation of spanning trees.

6.8.1 Cache Representation

From the development representation that is partitioned in a data part (the information repository or IR) and a control part (the behavior execution tree or BET), a cache representation allowing matching for equality has to be determined.

6.8.1.1 IR Cache Representation

In fact, the IR part do not pose any difficulty, because it is based on user level data structures from which it is straightforward to derive an equality function. However, because of space / memory limitations it is not reasonable to store whole IR snapshots at each step. In particular in the context of models for distributed object frameworks where the data space tend to be very large. A classical solution is to define the cache representation related to the IR as the delta from the initial state of the search. Two visited states are the same if their IR states are the same or equivalently if their IR-deltas are equal.

6.8.1.2 BET Cache Representation

The BET represents the state of the control in the system at a given step in its execution. Two control states are equal if for a same IR, all the ways they can be developed is rigorously the same. This definition implies that it is uninteresting to consider behavior execution nodes (BEN) that have already completed their execution in the BET, i.e. that are in the $done$ state. So the BENs that have to be included in the cache representation of a BET are only the BENs that are currently active or those present in the BET but are waiting for other BENs to complete to be re-activated. In summary, this concerns the BENs in a form of ready state, i.e. $ready - pre$, $ready0$, $ready$ and $ready - post$ states; and the BENs in a form of wait state, i.e. $wait$ and $wait - ccrc = 0$ states. In addition the execution control links between these BENs has to be included in the cache representation, which is defined by the trigger execution control (TEC) structures defined in chapter 4.

Now that the relevant BENs to consider in the BET cache representation have been identified, it is time to define what information has to be additionally included for each such BEN. Naturally, the state and behavior label corresponding to each BEN are mandatory. The behavior execution context (BEC) is also needed. This is sufficient for all states except for a BEN in the $ready$ or $wait - ccrc = 0$ state, that is blocked at an intermediate step in the execution of its body. This particular case is examined just

below (in section 6.8.1.3). In summary, for BENs in a state $wait, ready-pre, ready0, ready-post$, since the associated control state is defined with user level structures, there is no difficulty to find a cache representation supporting equality matching function on these user level structures.

6.8.1.3 Behavior Body Cache Representation

Behavior bodies pose in fact the major problem for a cache representation to be defined. The reason is that the development representation associated to a blocked behavior body at an intermediate step in its execution has to be supported by low level execution structures. That is structures that are not user level structures onto which full control is available. Such low level control structures typically have a very restricted set of operations available. For instance this is limited to an access via a handle that can be stored in user level data structures, and control oriented operations such as *resume* and *terminate* called on this handle. There is little chance (in fact no chance) to have a suitable equality operator on such low level control structures. In the *Scheme* execution environment used in the current implementation of the BPE, *Scheme* continuations provide the suitable representation to be used as a part of the development representation of states. However, two continuations representing the same state of processing in a behavior body are never equal. Note that the problem is by no means related to the use of continuations, but rather on the semantics used that involves intermediate execution steps in behavior bodies. This inevitably leads to the use of low level control structures. As already mentioned, in another programming language one would probably use some form of coroutine or of non-preemptive threads that would cause exactly the same difficulty. As a consequence it is not feasible to define an equality matching function from handles to coroutines, threads or continuations. Therefore another approach allowing equality matching has to be observed.

This approach is based on a trick and a limitation on the form of coding inside behavior bodies. The trick consists to reuse the behavior instrumentation support already available to allow source level debugging. In effect, a static representation of each blocking point in a behavior body is already available by means of the corresponding line number in the behavior source file. If the behavior was instrumented for debugging, at execution this integer is present in the field $beh - src - lno$ in the BEN. Note that this works if only one message sending primitive per line it is used. Note that in any case, this assumption had yet to be observed for the debugging support to work properly, i.e. without ambiguity. Note that ambiguity does not break the system, but this result in the impossibility to determine the cache representation. A way of dealing with that is to enforce that the resulting cache representation are always different from any other one. So in the end potential reduction in the search is lost and that's all.

Then a limitation has to be observed so that no additional information is required to define a cache representation for a behavior at an intermediate step in the execution of its body. The limitation consists of constraining the coding in behavior bodies by prohibiting the use of *Scheme* local variables[3]. Note that this is in fact not a big limitation because it is possible to define local variables in the BEC. With this limitation the whole execution context of a behavior body is delimited by the BEC. This limitation prohibits the use of nested lexical scoping as it is available in *Scheme*, but on the other hand since the BEC is a visible structure in the BEN the monitoring of the execution of behavior bodies is more easy if its local variables are in the BEC. So this limitation may reveal as an advantage.

6.8.2 State Caching + Sleep-Sets Full Walk Algorithm

As shown in algorithm 6.8.1, thanks to the cache representation defined for both the IR and BET parts of the system's configuration, algorithm 6.7.1 based on sleep-sets can be extended to incorporated state catching.

[3] Local variables in *Scheme* are introduced by `let`, `let*`, `letrec`, `do`, and `lambda` statements.

Algorithm 6.8.1 *State Caching combined with Sleep Set Full Walk Algorithm :*

BPE:SCSSFW$(sleep-set, tr)$

1 $ready-bens \leftarrow \{ben \in BET : ben.state \in \{ready-pre, ready0, ready, ready-post\}\}$
2 **if** $ready-bens = \emptyset$
3 **then** $dump-trace(tr)$
4 **else** $not-sleeping-ready-bens \leftarrow ready-bens \setminus \{ben : (ben, ops) \in sleep-set\}$
5 **for each** ben **in** $not-sleeping-ready-bens$
6 **do** BEN:EXEC(ben)
7 $cache-rep \leftarrow$ CACHE-REP$(*IR*, *BET*)$
8 **if** $cache-rep$ **in** $*Cache*$
9 **then** $undo(*undo-infos*)$
10 **else** **insert** $cache-rep$ **in** $*Cache*$
11 **push** $*undo-info*$ **on** $ben.undo-infos$
12 $new-sleep-elt \leftarrow (ben, *ops*)$
13 BPE:SCSSFW$(sleep-set \setminus \{(ben', ops') \in sleep-set :$
14 $(ops', *ops*) \in D\},$
15 tr **extended with** $ben)$
16 $undo($**pop** $ben.undo-infos)$
17 $sleep-set \leftarrow sleep-set \cup new-sleep-elt$

Algorithm 6.8.1 is identical to algorithm 6.7.1, except that at line 7 the cache representation of the visited state is determined from its development representation defined by the $(*IR*, *BET*)$ configuration, that is assumed to be stored in the two global variables : $*IR*$ for the information repository, and $*BET*$ for the behavior execution tree. At line 9, the presence of the visited state in the cache is tested. The cache is assumed to be accessible from the global variable $*Cache*$. If this state was already visited, the previous state is just restored by undoing the executed transition. If it is the first time this state is visited, the sleep-sets algorithm performs as before, except that the visited state is inserted in the cache (line 10) to avoid its re-development in future searches.

6.9 Using eXhaustive Dynamic Behavior Analysis

The be comprehensible, results are presented here based on executions of the simple spanning tree algorithm on different configurations of networks represented as directed graphs. The complete description of this simple case study accompanied with the required behavior is given in appendix C. Note that the spanning tree is used because it is simple in terms of the problem statement, but it is sufficient in terms of the executions and associated state spaces to illustrate our results.

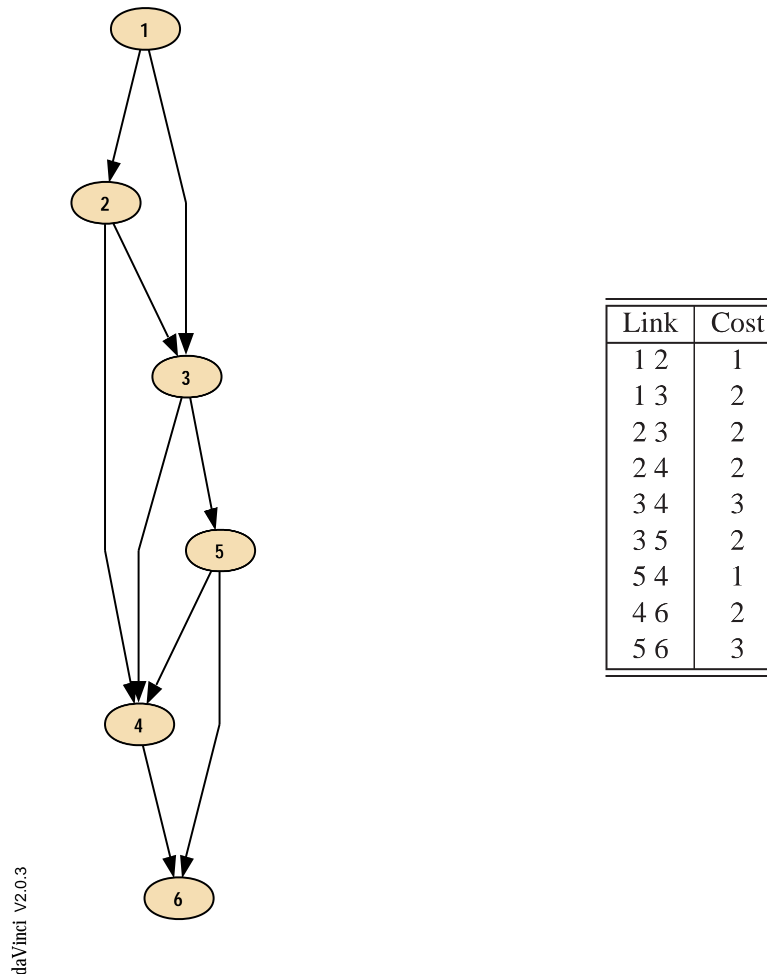
Table 6.1 summarizes the results on the experiments made on full walks of the spanning tree relative to three network configurations. The first network (dg1) configuration in the one used in chapter 5, in figure 5.2. The two other more complex network configurations are given in figure 6.3 (dg2) and 6.4 (dg3) respectively.

The results illustrate the expected benefits of using sleep-sets in combination with state caching. In particular, the use of sleep-sets can substantially improve the performance of the exhaustive search by avoiding a significant number of state matching. Typically this concerns all the state matchings due to the exploration of all possible interleavings of concurrent executions all leading to the same state.

Using these simple simulations one can verify that using sleep-sets does not reduce the number of visited state. But as expected, sleep-sets typically reduce the number of transitions.

Algorithm used in Simulations The algorithms used in simulation are referenced in table 6.1 with the following acronyms :

- SCFW means state caching full walk, this corresponds to algorithm 6.2.1 developed depth first.
- SLSSFW means state-less sleep-set full walk, this corresponds to algorithm 6.7.1.
- SCSSFW means state-caching sleep-set full walk, this corresponds to algorithm 6.8.1.

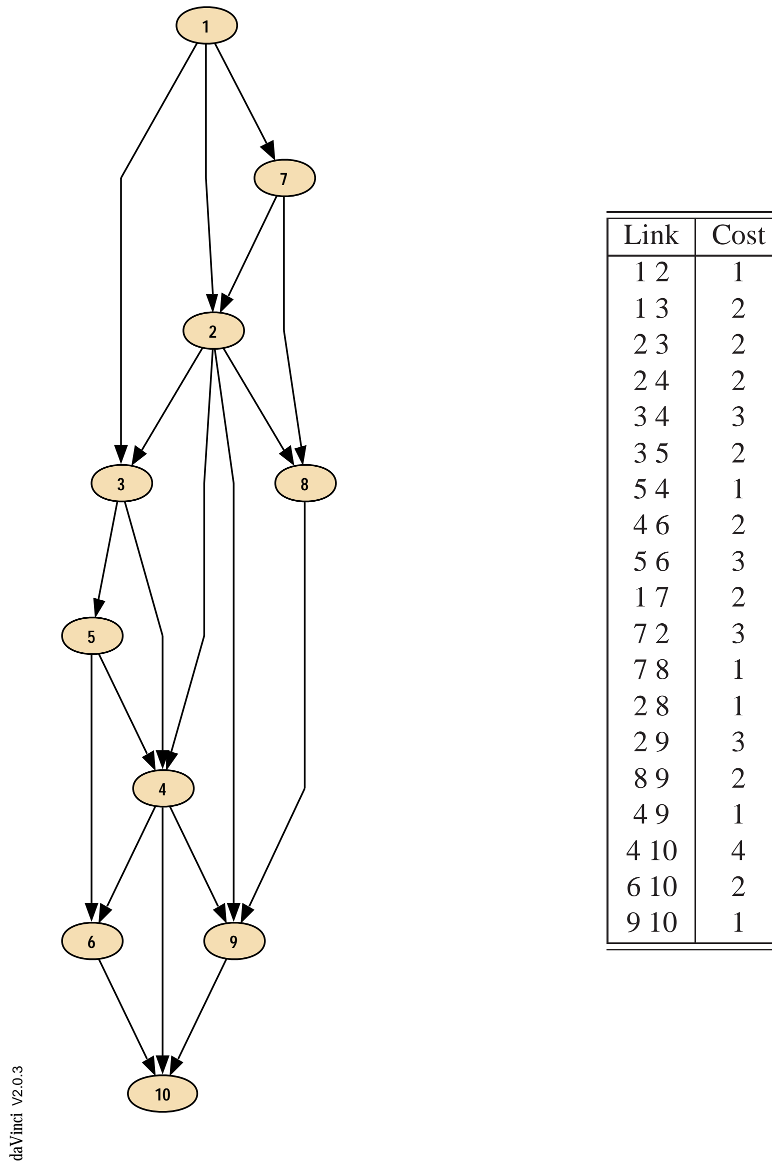


Link	Cost
1 2	1
1 3	2
2 3	2
2 4	2
3 4	3
3 5	2
5 4	1
4 6	2
5 6	3

Figure 6.3: Directed Graph 2 for the Spanning Tree.

Scheme Implementations Used

Bigloo1.9b [Serrano97] is a module compiler for *Scheme*. Bigloo can be used to obtain a compiled version of the BPE *Scheme* files and of the spanning tree case study. Bigloo translates *Scheme* to C. The compiled *Scheme* code is pure R^4RS encapsulated with some module declarations. Therefore a small amount of work is required to obtain a compiled executable. In addition, this work can be automated by preparing the module declaration for a case study and a batch script to compile and link with the modules representing the BPE *Scheme* files.



Link	Cost
1 2	1
1 3	2
2 3	2
2 4	2
3 4	3
3 5	2
5 4	1
4 6	2
5 6	3
1 7	2
7 2	3
7 8	1
2 8	1
2 9	3
8 9	2
4 9	1
4 10	4
6 10	2
9 10	1

Figure 6.4: Directed Graph 3 for the Spanning Tree.

		SCFW	SLSSFW	SCSSFW
dg1	transitions	687	534	361
	terminations	1	10	1
	cache-size	321	-	321
	cache-matches	366	-	40
	sleeps	-	587	77
	time Bigloo1.9b	3s	3s	1s
	time SCM5b1	30s	20s	10s
dg2	transitions	29414	12365	9779
	terminations	20	36	4
	cache-size	8823	-	7861
	cache-matches	20591	-	1918
	sleeps	-	28093	17592
	time Bigloo1.9b	160s	40s	46s
	time SCM5b1	3766s	480s	826s
dg3	transitions	-	-	500000
	terminations	-	-	6
	cache-size	-	-	300668
	cache-matches	-	-	199979
	sleeps	-		1.1 M
	time Bigloo1.9b		-	1.7h (stopped)

Table 6.1: Full Walks on Spanning Trees.

SCM5b1 [Aubrey jaffer97] is a reasonably fast *Scheme* interpreter. With SCM, exhaustive search is immediately available but in a slower interpreter based system.

The experiments were performed on Ultra-Sparcs with 32Mb of RAM. The memory requirements is such that, when used, the cache is never larger than the available RAM space.

6.9.1 Detecting Confluence / Divergence

An interesting property to check is to detect if the system is confluent. That means that at saturation a same termination state is reached. Conversely, if the system is divergent it may be interesting to analyze the different alternatives, i.e. reached states and the corresponding executions. The spanning tree case study illustrates quite well the interest of the detection of confluence and divergence in the system because these properties have a direct meaning in the context of this simple problem. Note that this is most of the time the case for real case studies based on models of distributed object frameworks because confluence and divergence are linked to the termination state of the objects in the system, i.e. the contents of the IR. With the simple spanning tree algorithm, convergence means that only one optimal solution was found in the network for shortest paths from any node to the termination node. On the other hand, divergence naturally exhibits all the solutions to the problem. An example of convergence can be found in the trivial directed graph given in chapter 5, in figure 5.2. Since this trivial directed graph has only one solution, a description of the convergence state can be found in figure 5.3. On the other hand divergence can be illustrates on the directed graph given in figure 6.3, the different solutions are given in figure 6.5. From figure 6.5 one can see that the 4 solutions for the spanning tree are found.

```
1    (((1 next=> 2 dist=> 5)
2       (2 next=> 4 dist=> 4)
3       (3 next=> 5 dist=> 5)
4       (4 next=> 6 dist=> 2)
5       (5 next=> 6 dist=> 3)
6       (6 next=> ??? dist=> 0))
7     ((1 next=> 2 dist=> 5)
8       (2 next=> 4 dist=> 4)
9       (3 next=> 5 dist=> 5)
10      (4 next=> 6 dist=> 2)
11      (5 next=> 4 dist=> 3)
12      (6 next=> ??? dist=> 0))
13    ((1 next=> 2 dist=> 5)
14      (2 next=> 4 dist=> 4)
15      (3 next=> 4 dist=> 5)
16      (4 next=> 6 dist=> 2)
17      (5 next=> 6 dist=> 3)
18      (6 next=> ??? dist=> 0))
19    ((1 next=> 2 dist=> 5)
20      (2 next=> 4 dist=> 4)
21      (3 next=> 4 dist=> 5)
22      (4 next=> 6 dist=> 2)
23      (5 next=> 4 dist=> 3)
24      (6 next=> ??? dist=> 0)))
25
```

Figure 6.5: Divergence Detection in the Spanning Tree Executions (for graph 6.3).

6.10 Discussion

6.10.1 Partial Order vs. Proof Methods

It is worth emphasizing that the practical interest of state-space exploration techniques (and of "verification" in general) is mainly to find errors that would be hard to detect and reproduce otherwise, and not necessarily to prove the absence of errors. As previously mentioned in section 2.7.3, while mathematically proving that a model of a system conforms to a specific set of properties does increase the confidence that the actual system is "correct", it does not provide a proof of this fact. This is still a partial result limited by the amount and relevance of the properties checked and the corresponding proofs performed. Strategies for proving properties of concurrent systems without considering all possible interleavings of their concurrent actions have also been proposed. These proof methods are applied in the context of an inference system, in which the correctness of a system is established by proving assertions about its components. This approach to verification has the advantage of not being restricted to finite-state systems. On the other hand, it requires proofs that are manual. Even with the help of a theorem prover, carrying out proofs with a theorem prover is far from being fully automatic since most steps of the proof require inventive interventions from the user. In contrast, this chapter focuses purely on algorithmic issues, and only fully-automatic state-space exploration techniques are discussed.

6.10.2 Partial Order and Incomplete Checking

Several approximate methods based on simple heuristics have been proposed to restrict the number of inter-leavings that are explored [Holzmann91]. These heuristics carry with them the risk of incomplete verification results, i.e. they can detect errors but cannot prove the absence of errors. In contrast, provided that the search space is finite the partial order methods such as the sleep-sets selective search reduce the number of inter-leavings that must be inspected in a completely reliable manner. And it can be proved that, if the search completes, this is done without the risk of any incompleteness in the verification results.

6.10.3 Partial Order and Too Big / Infinite State Spaces

If the search space is not finite then partial-order methods are theoretically limited because the state space is explored in a rather naive way, i.e. without trying to find regularities, loops or recurring patterns in the model. In addition is is also possible that the state space is very large so that complete checking of properties along the whole state space is not possible due to time and space limitations. However, in practice reduced exhaustive exploration techniques are still useful because even if the state space is too big or infinite, it can still be partially explored using a finite amount of time, and / or using a certain amount of space, and / or up to a certain depth. In fact, the impossibility to check exhaustively large state spaces is only a theoretical limitation. In practice, it turns out that an important amount of errors can be detected even using partial exploration. This is in the end the more important point.

6.10.4 Closed Systems under Scenarios Steps

The state spaces considered are state spaces resulting from the application of a given solicitation to the system in a given state. The solicitation is typically given by the user as a step in a scenario. The objective of performing exhaustive dynamic behavior analysis is to prove that certain properties are observed along all the possible behavior propagations resulting from the application of the scenario step. The scenario-step hypothesis of work is taken for practical reasons, because for distributed object

frameworks it would be infeasible to work on the entire life-cycle of the system. Another important assumption that is made when performing exhaustive dynamic behavior analysis is to consider that, during the execution of a scenario step, the system is closed. That means that the model includes all that is required about the system under investigation and about its environment. As usual, the significance of any result obtained after an exhaustive search is directly and totally dependent on the quality of the model. This is naturally a purely subjective criterion that is entirely under the control of the people involved. Typically, if the relevant behaviors pertaining to the environment model susceptible to interfere with the system are not specified in the model, there is no chance to detect interworking problems related to such interferences. However, as soon as the model is complete relatively to some well identified issues, exhaustive dynamic behavior analysis provides a complete checking.

6.11 Conclusion

The behavior execution environment that is presented in chapter 5 is based on the BPE algorithm that is itself presented in chapter 4. The BPE algorithm defines the semantics of the behavior language according to the behavior model presented in section 3. Thus the BPE is the heart of the behavior execution environment. The BPE and the associated execution environment define a simple dynamic behavior analysis of a system. Though the execution environment is powerful since it features interesting facilities to analyze problems, the detection of such problems (e.g. assertion violations) has to be done on user-driven basis.

The contribution of this chapter is to propose a solution to detect more automatically specification problems. To this end, in this chapter reduction techniques are identified and integrated in order to perform efficiently exhaustive state space explorations on top of the existing BPE algorithm and implementation. In particular, it has been shown how reduced state space exploration can be achieved by using a combination of a partial order method and state caching. Details about the contributions of this chapter can be listed as follows :

Integration of partial order methods (sleep-sets) : The sleep-sets technique (a partial order method) has been integrated into the execution environment. The sleep-sets technique allows to detect all termination states and assertion violations of the system's behavior when solicited through a given scenario step. This provides in the end a very interesting complement with respect to the lack of problem detection support of the execution and debugging environment. Last but not least, this extension has been realized with minimal effort, i.e. by reusing to a maximal extent the facilities already implemented. This concerns mainly the improved debugging support presented in section 5.4.2.

Integration of state caching : State caching has also been considered and has been integrated in the implementation, in spite of the amount of additional work required. State caching allows a very interesting combination with partial order methods. As it is shown in [Godefroid95] and as it was verified in our simulations, state caching is complementary to sleep-sets because state caching fights different causes of the state-explosion problem.

Concluding Remarks Naturally, it is clear that our pretensions in terms of performances obtained in the resulting exhaustive searches with respect to both time and space can not be compared with those that can be obtained by dedicated verification systems such as CAESAR [Fernandez et al.96a], COSPAN [Hardin et al.96], CWB [Cleaveland93], SPIN [Holzmann91], etc. Such verification tools are typically based on optimized C implementations. However on the other hand, the proposed behavior language is a specification language quite different from e.g. PROMELA which is the input

specification language used by SPIN. The behavior language features interesting modeling abstractions and observes a pragmatic approach intended to accommodate the specific requirements of the modeling of distributed object frameworks. In this thesis, a less efficient but rather more abstract and user friendly approach for specification and validation is prefered. Since involvement of the relevant users is considered as the more important aspect the advocated approach should reveal satisfactory.

In this chapter, it was also shown how different *Scheme* implementations may be used to accommodate the different requirements of a modeling activities based on an executable specification approach. It is advantageous to use a *Scheme* implementation with a powerful development environment to proceed in the first phase of a case study. When improved validation can be done it is interesting to use a *Scheme* implementation with less goodies but oriented towards efficiency, e.g. doted with a compiler. In section 6.9 it is shown that using a compiled version of the simple spanning tree case study, a very significant gain in terms of computation time is obtained. In fact, a *Scheme* interpreter, even if reasonably fast can not be used as soon as the exhaustive search becomes important. On the other hand, with a *Scheme* compiler much more computation intensive state space explorations can be envisioned.

The exhaustive state space exploration algorithm implemented has already been experimented on a real case study, pertaining to a distributed object framework intended towards TMN applications. The objective of the METRAN case study is to model X interfaces for the configuration management of SDH transport networks. The departure points are paper specifications available as the deliverable of the EURESCOM project P408 [P40896].

This case study represents a highly distributed TMN application, that revealed very useful to test the behavior specification and validation tool-set. The METRAN case study, is presented in details in [Mazziotta97]. This includes both behaviors and scenarios developed. The exhaustive search implementation has made possible the detection of some specification errors in this case study. In particular the divergence detection facility seems to be very useful. Many stupid problems can be detected by simply observing the divergence in the possible executions the system can go through. For instance, a very simple problem observed was about attribute value change notifications. In some executions an incorrect value was conveyed. This was not detected along the performed user-driven behavior executions. It was trivially identified with the divergence detection facility. The stupid problem was merely that the value inserted in the notifications at sending time was the current value of a changed attribute in the information repository and not the value at the time the change was detected ! Since the emission of the notification was specified in a separate processing, due to other concurrent processing the value of the attribute was allowed to change. Once detected, it has been straightforward to locate the problem using the behavior execution debugging facilities.

The number of such problems in a paper specification of a distributed object framework is typically very important. The improved validation techniques identified in this chapter and implemented allow to detect many problems that would be never observed otherwise. In the end it is the quality of the delivered specifications that is improved to a significant extent.

Chapter 7

Conclusion

7.1 Summary and Contributions

The overall contribution of this thesis is to propose an original approach to cope with functional modeling of distributed object frameworks. This has been done as proposed by ODP, by considering altogether in an integrated way, both object oriented modeling, formalization and distributed processing. The salient features of the behavior modeling frameworks are :

It is generic : The concepts used in behavior modeling are based on existing and well established concepts coming from ODP. ODP covers all the aspects of distributed computing in a consistent way. Thus it is possible to integrate in an ODP based framework all the issues considered in any existing distributed object computing notation, e.g. GDMO / ASN.1, CORBA-IDL, etc. This is a contribution presented in chapter 2 and instantiated in chapter 3.

It is abstract (declarative) : The behavior specification template is based on the principle of a declarative specification of actions. This has been found as the more abstract approach to address functional models. This is a contribution presented in chapter 2. Appendix A describes the behavior language notation in details.

It is abstract (IVP+CVP) : To cover functional modeling aspects in an abstract way it is desirable to model a system without using an engineering model. An interesting approach is to use IVP and CVP models in a complementary way. The information model defines the state (static schema) and transitions (dynamic schema) in the system. Whereas the computational model defines the interfaces of the objects in the system, and thus the interfaces of the system with its environment. With both models a complete functional specification of a system can be envisioned. This can be done by abstracting away unwanted engineering details that are useless in the context of functional modeling. This is a contribution presented in chapter 2 and instantiated in chapter 3.

It is expressive : It integrates powerful OO modeling abstractions such as roles and relationships typically used in object oriented modeling techniques as the more powerful high level dynamic aggregation structures. This has been instantiated using the GRM notation. The integration of powerful OO modeling abstractions is a contribution presented in chapter 2 and instantiated in chapter 3.

It is formal : The semantics of the proposed declarative specification of actions language is operationally defined by an algorithm called the behavior propagation engine algorithm (BPE). The BPE is a forward chaining inference engine inspired from the powerful execution semantics devised for ECA-rules in the context of ADBMS. This is a contribution of chapter 4.

It is usable : The execution environment allows the user to exercise the model through the execution of test cases or scenarios. This can be done in a comfortable environment. Thanks to graphical system visualization facilities the user is able to fully monitor the system at runtime. This includes the visualization of both the state of the data and the state of the control. In addition, execution backtracking is a very powerful dynamic debugging facility in order to find problems in a specification. The execution environment is a contribution presented in chapter 5.

It allows improved validation : The detection of all problems (i.e. assertion violations) and all deadlocks or other termination states in a specification can be done with respect to a given scenario step and system state. This is done through exhaustive state space exploration. Interestingly, this can be easily implemented by reusing the execution backtracking facility provided for the purpose of debugging. However, to fight the classical state explosion problem that occurs in such cases, two reduction techniques have been incorporated in the exhaustive search engine : (i) the sleep-sets technique which comes from partial order methods, and (ii) the state caching technique. Used jointly these two methods are complementary. The resulting reduced state space exploration algorithm is able to detect much more problems, such as assertion violations. In addition, if this reduced search terminates, it is proved that all the problems are detected. The improved validation support is a contribution presented in chapter 6.

It has been experimented on real case studies : A tool-set has been developed in the context of the TIMS project that instantiates the behavior specification and validation approach proposed in this thesis. This tool-set has been used in four significant TIMS case studies. A summary of these case studies is given in appendix F. The more significant case study has been developed in Swisscom and has been used to produce an Ensemble [Etsi97] document that is currently following the standardization process within ETSI. This Ensemble and the case study are presented in appendix G.

The overall conclusion of the thesis is that the behavior modeling framework devised performs well for both specification and validation issues. Concerning specification, it is generic, abstract and expressive and fulfills exactly the requirements of functional modeling in the context of distributed object frameworks. Concerning validation, thanks to an approach based on executable specifications, a powerful simple and exhaustive execution environment, the quality of the models obtained can be improved to a significant extent.

The issues covered in this thesis have been addressed in the following publications where the author has contributed : [Sidou96, Sidou et al.96a, Eberhardt et al.97b, Sidou et al.95b, Sidou95, Sidou et al.96b, Sidou et al.95a].

7.2 Related Work (ITU-SG 4 [G851 0196])

It is worth to situate our behavior model with respect to the modeling approach proposed by ITU-SG 4 [G851 0196]. One similarity is that a rather compatible behavior specification template is used. Another similarity is that GRM is used to define a key part of the static schema in the information model.

A first difference is that [G851 0196] proposes a notation to specify the computational model independently from engineering issues. This effort is theoretically interesting because it allows a better separation of concerns that may allow to map the resulting specification either to the usual OSI-SM (that is a conventional choice in the context of TMN applications) or to CORBA. This is probably an approach to follow in the development of new models for which both an OSI-SM and a CORBA communication infrastructure may be required. However, for TMN object frameworks

based on existing state and interface specifications defined using GDMO / ASN.1, this approach is not required. Since the case studies developed in the context of the TIMS project were based based on existing interface specifications from ITU, ETSI, NMF, etc, it was more useful to pragmatically incorporate directly the existing GDMO / ASN.1 specifications.

Another distinction that can be made from ITU-SG 4 work is that a significant additional effort has been accomplished in this thesis to make the specification framework executable. In particular, execution oriented clauses have been added to the behavior notation template. An important point to note is that a clear separation is observed between pure specification oriented issues and pure execution oriented ones. This is done through the use of distinct and well identified behavior clauses. Last but not least to allow executability a precise operational semantics has been defined and implemented by way of the behavior propagation engine (BPE) algorithm.

7.3 Further Issues

Many further issues are possible as followup activities to the work presented in this thesis and accomplished in the context of the TIMS project :

- Other true case studies are always worth to be performed. In particular true distributed object frameworks involving CORBA interfaces is an important further issue. This would enable to check more intensively the generic behavior modeling approach proposed in this thesis.

  Other case studies or extensions of existing ones may be useful to check the scalability of the approach, in particular with respect to the improved validation approach presented in this thesis.

- The existing exhaustive search implementation can be optimized to allow again larger state spaces to be explored. Without including any other algorithmic technique, optimization of the implementation itself may be worth to be done.

- With respect to modeling issues it may be worth to try to incorporate more elaborated behavior reusing or composition artifacts. The behavior model has been designed in an open way so that experiments in this direction are allowed to a large extent.

- In addition there is no obstacle to the fact that the behavior model could be used in the context of enterprise viewpoint models. As sketched in [Sidou95] the problematic of modeling policies is not very far from the one of functional modeling in computational and information viewpoint models. At least, this is the case if high level information modeling structures such as roles and relationships are accommodated by the behavior model. Indeed, the use of roles models has already been identified as a suitable approach to model management action policies [Lupu et al.96].

Bibliography

[Abrial95] Abrial (J.-R.). – *The B-Book*. – Cambridge University Press, 1995.

[Agrawal et al.90a] Agrawal (Hiralal), DeMillo (Richard A.) et Spafford (Eugene). – *An Execution Backtracking Approach to Program Debugging*. – Technical Report SERC-TR-22-P, Software Engineering Research Center Purdue University, September 1990.

[Agrawal et al.90b] Agrawal (Hiralal) et Horgan (Joseph R.). – Dynamic program slicing. *In : SIGPLAN'90*. ACM.

[Ardis et al.96] Ardis (M.), Chaves (J.), Jagadeesan (L.), Mataga (P.), Puchol (C.) et von Olnhausen (M. Staskauskas J.). – A Framework for Evaluating Specification Methods for Reactive Systems. *Transactions on Software Engineering*, vol. 22 (6), june 1996, pp. 378–389.

[Asn188] ITU-T Recommendation X.208-88: Abstract Syntax Notation Number 1: Part 1: Specification, 1988.

[Aubrey jaffer97] Aubrey Jaffer. – The SCM Scheme System, 1997. Available at http://www-swiss.ai.mit.edu/~jaffer.

[Bartocci et al.93] Bartocci (A.), Larini (G.) et Romellini (C.). – A first attempt to combine GDMO and SDL techniques. *In : SDL'93: Using Objects*, éd. par Faergemand (O.) et Sarma (A.). pp. 333–345. – Elsevier Science Publishers B.V. (North-Holland).

[Berry et al.92] Berry (G.) et Gonthier (G.). – The esterel synchronous programming language : Design, semantics, implementation. *Science of Computer Programming*, vol. 19, 1992, pp. 87–152.

[Biberstein et al.96] Biberstein (Olivier), Buchs (Didier) et Guelfi (Nicolas). – *COOPN/2 : A Specification Language for Distributed Systems Engineering*. – Technical Report 167, Lausanne, Switzerland, Software Engineering Laboratory, Swiss Federal Institute of Technology, 1996. available at ftp://lglftp.epfl.ch/pub/-Papers/biber-TR96-167.ps.

[Booch94] Booch (G.). – *Object-Oriented Analysis and Design with Applications*. – Benjamin Cummings, 1994.

[Brownston et al.85] Brownston (L.), Farrell (R.), Kant (E.) et Martin (N.). – *Programming Expert Systems in OPS5, An Introduction to Rule-Based Programming*. – Addison-Wesley, 1985.

[Chakravarthy et al.89] Chakravarthy (S.) et al. – *A research project in active, time-constrained database management (final report)*. – Technical Report XAIT-89-02, Cambridge, Massachusetts, Xerox Advanced Information Technology, August 1989.

[Chandy et al.88] Chandy (K. M.) et Misra (J.). – *Parallel Program Design*. – Addison-Wesley, 1988.

[Cleaveland93] Cleaveland (Rance). – The concurrency workbench: A semantics-based verification tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, vol. 15 (1), January 1993, pp. 36–72.

[Clinger et al.91] Clinger (W.) et Rees (J.). – Revised-4 Report on the Algorithmic Language Scheme. *ACM Lisp Pointers*, vol. 4 (3), 1991. – Available at http://www.cs.-indiana.edu/scheme-repository/doc/standards/r4rs.ps.gz.

[Cmip] Management Information Protocol Specification - Common Management Information Protocol, ISO/IEC 9596-1, ITU X.711.

[Cmis] Management Information Service Definition - Common Management Information Service Definition, ISO/IEC 9595, ITU X.710.

[davinci] The Interactive Graph Visualization System daVinci. Available at http:-//www.informatik.uni-bremen.de/˜inform/forschung/daVinci/daVinci.html.

[Dijkstra76] Dijkstra (E. W.). – *A Discipline of Programming*. – Prentice-Hall, 1976.

[Dittrich et al.95] Dittrich (Klaus R.), Gatziu (Stella) et Geppert (Andreas). – *The Active Database Management System Manifesto: A Rulebase of ADBMS Features*. – Technical report, University of Zurich, Dept. of Computer Science, 1995. Available at http://www.ifi.unizh.ch/techreports.

[Dowek et al.91] Dowek (Gilles), Felty (Amy), Herbelin (Hugo), Huet (Gérald), Paulin-Mohring (Christine) et Werner (Benjamin). – *The COQ Proof assistant user's guide*. – Technical Report 134, Le Chesnay Cedex, France, Inria-Rocquencourt, December 1991.

[Eberhardt et al.95] Eberhardt (R), Bart (M) et v.d. Burg (M). – TIMS, Evaluation based on Case Study: SDH Protection. – 1995. Swiss Telecom internal report : R&D-RA-95-997.

[Eberhardt et al.97a] Eberhardt (R) et Randini (Marco). – V5-management. – 1997. Swiss Telecom internal report : R&D-RA-97-???

[Eberhardt et al.97b] Eberhardt (Rolf), Mazziotta (Sandro) et Sidou (Dominique). – Design and testing of information models in a virtual environment. *In : The Fifth IFIP/IEEE International Symposium on Integrated Network Management "Integrated Management in a Virtual World"*. – San Diego, CA, USA, may 1997. available at http://www.eurecom.fr/˜tims/papers/im97.ps.gz.

[Eigenschink et al.94] Eigenschink (T.R.), Love (D.) et Jaffer (A.). – SLIB: The Portable Scheme Library, 1994.

[Engberg et al.92] Engberg (Urban), Gronning (Peter) et Lamport (Leslie). – *Mechanical Verification of Concurrent Systems with TLA*. – Technical report, Palo Alto, California, DEC Systems Research Center, 1992.

[Estelle89] Estelle : A Formal Description Technique based on the extended state transition model, ISO / IEC 9074, 1989.

[Etsi97] ETSI. – Transmission and multiplexing, connectivity and service provisioning in access networks, 1997. ES/TM-2235.

[Fernandez et al.96a] Fernandez (J-C.), Garavel (H.), Kerbrat (A.), Mateescu (R.), Mounier (L.) et Sighireanu (M.). – Cadp (cæsar/aldebaran development package): A protocol validation and verification toolbox. *In : CAV*.

[Fernandez et al.96b] Fernandez (J.C.), Jard (C.), Jéron (T.) et Viho (G.). – An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming, Special Issue on Industrial Relevant Applications of Formal Analysis Techniques*, 1996.

[Fernandez et al.96c] Fernandez (J.C.), Jard (C.), Jéron (T.) et Viho (G.). – Using on-the-fly verification techniques for the generation of conformance test suites. *In : Proc. of Conference on Computer-Aided Verification (CAV'96)*, éd. par Alur (Rajeev) et Henzinger (Thomas A.). – New Brunswick, New Jersey, USA, 1996.

[Fuchs92] Fuchs (Norbert E.). – *Specifications are (preferably) executable*. – Technical Report 92, University of Zurich (CS Dept.), 1992. Available at ftp://ftp.ifi.-unizh.ch/pub/techreports/.

[G851 0196] Management of the Transport Network – Application of the ODP Framework, ITU-T G851-01, 1996.

[Gamma et al.94] Gamma (E.), Helm (R.), Johnson (R.) et Vlissides (J.). – *Design Patterns : Elements of Object-Oriented Software*. – Addison-Wesley, 1994.

[Gdmo] Structure of Management Information - Part 4: Guidelines for the Definition of Managed Objects, ISO/IEC 10165-4, ITU X.722.

[Genilloud96] Genilloud (Guy). – *Towards a Distributed Architecture for Systems Management*. – PhD thesis, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, 1996. Thèse N° 1588.

[Geppert et al.95] Geppert (A.), Gatziu (S.), Dittrich (K.), Fritschi (H.) et Vaduva (A.). – *Architecture and Implementation of the Active Object-Oriented Database Management System SAMOS*. – Technical report, University of Zurich, Dept. of Computer Science, 1995. Available at http://www.ifi.unizh.ch/techreports.

[Godefroid95] Godefroid (Patrice). – *Partial-Order Methods for the Verification of Concurrent Systems* – An Approach to the State Exploision Problem. – PhD thesis, Université de Liège, Faculté des Sciences Appliquées, 1995. Available at http://www.montefiore.ulg.ac.be/services/verif/papers/thesis.ps.Z.

[Godefroid97] Godefroid (Patrice). – Model Checking for Programming Languages using VeriSoft. *In : Principles of Programming Languages*. ACM.

[Grm94] ISO/IEC JTC 1/SC 21 – Information Technology – Open System Interconnection – Data Management and Open Distributed Processing – Structure of Management Information – Part 7 : General Relationship Model., mar 1994.

[Hanson et al.92] Hanson (Eric N.) et Widom (Jennifer). – *An Overview of Production Rules in Database Systems*. – Technical report, University of Florida (CIS), 1992. Available at ftp://ftp.cis.ufl.edu/cis/tech-reports/tr92/tr92-031.ps.

[Hardin et al.96] Hardin (R. H.), Har'El (Z.) et Kurshan (R. P.). – COSPAN. *Lecture Notes in Computer Science*, vol. 1102, 1996, pp. 423–??

[Harel et al.96] Harel (D.) et Gery (E.). – Executable object modeling with statecharts. *In : 18th International Conference on Software Engineering*. – Springer-Verlag.

[Hayes et al.90] Hayes (I.J.) et Jones (C.B.). – *Specifications are not (necessarily) executable*. – Technical Report 90-148, University of Manchester (CS Dept.), 1990. Available at ftp://ftp.cs.man.ac.uk/pub/TR/UMCS-89-12-1.ps.Z.

[Helm et al.90] Helm (Richard), Holland (Ian M.) et Gangopadhyay (Dipayan). – Contracts: Specifying behavioral compositions in object-oriented systems. *In : Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications European Conference on Object-Oriented Programming (OOPSLA) (ECOOP)*, éd. par Meyrowitz (Norman). pp. 169–180. – Ottawa, ON CDN, [10] 1990. Published as SIGPLAN Notices, volume 25, number 10.

[Hoare85] Hoare (C. A. R.). – *Communicating Sequential Processes*. – Prentice-Hall, 1985.

[Holland93] Holland (Ian M.). – *The design and representation of object-oriented components*. – PhD thesis, North-Eastern University, 1993.

[Holzmann91] Holzmann (Gerard J.). – *Design and Validation of Computer Protocols*. – Prentice Hall, 1991.

[IT93] ITU-T. – Specifications and Description Language SDL. – Recommendation Z.100, 1993.

[Jagadeesan et al.95] Jagadeesan (L.), Puchol (C.) et Olnhausen (J.). – Safety property verification of esterel programs and applications" to telecommunications software. *In : Proceedings CAV'95 (LNCS 939)*.

[Jarvinen et al.91] Jarvinen (Hannu-Matti) et Kurki-Suonio (Reino). – DisCo Specification Language: Marriage of Action and Objects. *In : Proc. of 11th International Conference on Distributed Computing Systems*. – IEEE Computer Society Press. Available at http://www.cs.tut.fi/laitos/DisCo/DisCo-english.fm.html.

[Java rmi] Java RMI - Remote Method Invocation. available at http://www.javasoft.-com/.

[Kilov et al.96] Kilov (H.), Mogill (H.) et Simmonds (I.). – *Invariants in the Trenches in Object Oriented Behavior Specifications*, pp. 77–100. – Kluwer, 1996.

[Kilov et al.97] Kilov (Haim) et Tyson (Kevin P.). – Semantics Working Group Green Paper, 1997. Object Model SubCommitee, OMG Document ormsc/97-06-02.

[Kilov97] Kilov (Haim). – International Standards define Semantics: RM-ODP and GRM, 1997. Slides from presentation to the Semantics Working Group (part of the Object Model Subcommitee), OMG Document om/97-03-09.

[Lamport94] Lamport (Leslie). – *The Temporal Logic of Actions*. – Technical report, Palo Alto, California, DEC Systems Research Center, December 1994. Revised version of SRC-79 (91), has appeared in ACM transactions on programming languages and systems (Nov. 93).

[Lieberherr96] Lieberherr (Karl J.). – *Adaptive Object-Oriented Software : The Demeter Method with Propagation Paterns*. – Boston, PWS Publishing Company, 1996.

[Lotos87] LOTOS : A Formal Description Technique based on the Temporal Ordering of Observable Behaviour, ISO / IEC 8807, 1987.

[Lupu et al.96] Lupu (Emil C.) et Sloman (Morris). – Towards a role based framework for distributed systems management. *Journal of Networks and Systemss Management, Plenum Press*, 1996.

[Matsuoka et al.93] Matsuoka (Satoshi) et Yonezawa (Akinori). – Analysis of inheritance anomaly in object-oriented concurrent programming languages. *In : Research Directions in Concurrent Object-Oriented Programming*, éd. par Agha (P. Wegner G.) et Yonezawa (A.), pp. 107–150. – MIT Press, 1993. Available at ftp://camille.is.s.u-tokyo.ac.jp/pub/papers/book-inheritance-anomaly-a4.ps.Z.

[Mazaher et al.93] Mazaher (S.) et Moller-Pedersen (B.). – On the use of SDL-92 for the Specification of Behaviour in OSI Network Management Objects. *In : SDL'93: Using Objects*, éd. par Faergemand (O.) et Sarma (A.). pp. 317–331. – Elsevier Science Publishers B.V. (North-Holland).

[Mazurkiewicz86] Mazurkiewicz. – Trace theory. *In : Properties, Advances in Petri Nets*. – Springer-Verlag, 1986.

[Mazziotta97] Mazziotta (Sandro). – *Spécification et Génération de Tests du Comportement Dynamique des Systèmes à Objets Répartis*. – PhD thesis, Université de Nice-Sophia Antipolis, U.F.R. Faculté des Sciences, 1997.

[Milner89] Milner (Robin). – *Communication and Concurrency*. – Prentice-Hall, 1989.

[Milner91] Milner (Robin). – *The Polyadic π-Calculus : a Tutorial*. – Technical report, Computer Science Dpt., University of Edinburgh (UK), Laboratory of Fundations of Computer Science, 1991.

[Najm et al.95] Najm (E.) et Stephani (J.-B.). – A formal Semantics of the ODP Computational Model. *Computer Networks and ISDN Systems*, vol. 27 (8), 1995, pp. 1305–1329.

[Nmf ensco92] Ensembles: Concepts and Format, 1992. Network Mangement Forum.

[Noble et al.95] Noble (R.J.) et Grundy (J.C.). – Explicit Relationships in Object-Oriented Development. *In : TOOLS'95 Pacific*. – Melbourne, nov 1995. Available at http://www.cs.waikato.ac.nz/~jgrundy/publications.html.

[Omg corba96] Common Object Request Broker Architecture, 1996. Available at http:-//www.omg.org.

[Omg oma] Revised OMA Reference Model Chapter. OMG document ab/96-08-01.

[Osi sm] The OSI Systems Management Standards, ITU-T X.7xx Documents.

[P40896] P408 (EURESCOM Project). – Pan-European TMN - Experiments and Field Trial Support, Deliverable 5, Specifications of the Xcoop Interface for SDH Network Management, 1996.

[Partners97] Partners (UML). – Entire UML Submission, OMG documents ad/97-01-01 through ad/97-01-13, 1997.

[Paulson93] Paulson (Lawrence C.). – *Introduction to Isabelle*. – Technical Report 280, University of Cambridge, Computer Laboratory, 1993.

[Pratt86] Pratt (V. R.). – Modeling Concurrency with Partial Orders. *Int. J. of Parallel Programming*, vol. 15 (1), February 1986, pp. 33–71.

[Quemener96] Quemener (Y.-M.). – *Verification de protocoles à espace d'états infini representable par une grammaire de graphes*. – PhD thesis, Université de Rennes 1, 1996.

[Reenskaug96] Reenskaug (Trygve). – *Working with Objects : The OOram Software Engineering Method*. – Manning Publications, 1996.

[Reisig85] Reisig (W.). – *Petri Nets*. – Springer Verlag, 1985.

[Reynolds93] Reynolds (John C.). – *The Discoveries of Continuations*. – Technical report, Carnegie Mellon University, 1993.

[Rm odp1] Basic Reference Model of ODP – Part 1: Overview and Guide to Use of the Reference Model, ISO 10746-1, ITU X.901.

[Rm odp2] Basic Reference Model of ODP – Part 2: Foundations, ISO 10746-2, ITU X.902.

[Rm odp3] Basic Reference Model of ODP – Part 3: Architecture, ISO 10746-3, ITU X.903.

[Rm odp4] Basic Reference Model of ODP – Part 4: Architectural Semantics, ISO 10746-4, ITU X.904.

[Rose et al.90] Rose (M.) et McCloghrie (K.). – RFC1212: Structure of management information, May 1990.

[Rumbaugh et al.91] Rumbaugh (James), Blaha (Michael), Premerlani (William), Eddy (Frederik) et Lorensen (William). – *Object-Oriented Modeling and Design*. – Prentice Hall, 1991.

[Schoffstall et al.90] Schoffstall (M.), Fedor (M.), Davin (J.) et Case (J.). – RFC 1157: A Simple Network Management Protocol (SNMP), May 1990.

[Seiter et al.96] Seiter (L.), Palsberg (Jens) et Lieberherr (K.). – Evolution of object behavior using context relations. *In : ???*

[Serrano97] Serrano (Manuel). – Bigloo user's manual, 1997. Available at http://cuiwww.unige.ch/˜serrano/bigloo.html.

[Sidou et al.95a] Sidou (Dominique), Festor (Olivier) et Mazziotta (Sandro). – Managed Object Behavior Inheritance and Distribution Support in TIMS. *In : ECOOP95 Workshop : Use of Object-Oriented technology for Network Design and Management*. – Aarhus, Denmark, August 7-11 1995. Available at http://www.eurecom.fr/˜tims/papers/ecoop95-paper.ps.gz.

[Sidou et al.95b] Sidou (Dominique), Mazziotta (Sandro) et Eberhardt (Rolf). – TIMS : a TMN-based Information Model Simulator, Principles and Application to a Simple Case Study. *In : Sixth International Workshop on Distributed Systems : Operations & Management*. IFIP / IEEE. – Ottawa - Canada, 1995. Available at http://www.eurecom.fr/˜tims/papers/dsom95-paper.ps.gz.

[Sidou et al.96a] Sidou (D.), Eberhardt (R.), Festor (O.), Mazziotta (S.) et Labetoulle (J.). – Executable TMN Specifications in TIMS. *In : NOMS'96 International Network Operations & Management Symposium*. IFIP / IEEE. – Kyoto - Japan, 1996. Poster Session, available at http://www.eurecom.fr/˜tims/papers.

[Sidou et al.96b] Sidou (Dominique) et Mazziotta (Sandro). – Guidelines for the Specification of Managed Object Behaviors with TIMS. *In : ECOOP96 Workshop : Use of Object-Oriented technology for Network Design and Management*. – Linz, Austria, July 1996.

[Sidou95] Sidou (Dominique). – Behavior Simulation and Analysis Techniques for Management Action Policies. *In : SMDS'95 Workshop : Services for the Management of Distributed Systems*. – Karlsruhe, September 20–21 1995. Available at http://www.eurecom.fr/˜tims/papers/smds95.ps.gz.

[Sidou96] Sidou (Dominique). – A Generic and Executable Model for the Specification and Validation of Distributed Behaviors. *In : TreDS'96 : Trends in Distributed Systems Workshop, published in the Lecture Notes in Computer Science (LNCS 1161)*. – available at http://www.eurecom.fr/˜tims/papers/-treds96.ps.gz.

[Sidou97a] Sidou (Dominique). – Precise semantics for a behavior model in the context of object based distributed systems. *In : Workshop on Precise Semantics for Object-Oriented Modeling Techniques, published as technical report of the Technical University of Munich (TUM-I9725)*, éd. par Kilov (Haim) et Rumpe (Bernhard). – Jyvaskyla, Finland, 1997.

[Sidou97b] Sidou (Dominique). – Towards a good functional and executable behavior model. *In : Workshop on Object Oriented Technology for Telecommunications Services Engineering, published in ECOOP'97 workshop proceedings as LNCS ????*, éd. par Znaty (Simon) et Hubaux (Jean-Pierre). – Jyvaskyla, Finland, 1997.

[Spivey89] Spivey (J.M.). – *The Z Notation*. – Prentice Hall, 1989.

[Stallman87] Stallman (R.). – *GNU Emacs Manual for the extensible, Customisable, Self-Documenting Display Editor*. – Free Software Foundation, 1987.

[Stepney et al.92] Stepney (S.), Barden (R.) et Cooper (D.), editors. – *Object Orientation in Z*. – Springer-Verlag, 1992, *Workshops in Computing*.

[Teitelman et al.81] Teitelman (Warren) et Masinter (Lary). – The Interlisp Programming Environment. *IEEE Computer*, april 1981, pp. 25–33.

[Tinac cmc94] TINA Computational Modelling Concepts, 1994. Available at http://www.-tinac.com.

[Tinac imc94] TINA Information Modelling Concepts, 1994. Available at http://www.tinac.-com.

[Tinac94] Telecommunication and Intelligent Network Architecture Consortium, 1994. Available at http://www.tinac.com.

[Ullman94] Ullman (Jeffry D.). – *Elements of ML Programming*. – Prentice Hall, Englewood Cliffs, 1994.

[Ungar et al.91] Ungar (David) et Smith (Randal B.). – SELF: The Power of Simplicity. *LISP and Symbolic Computation, An International Journal*, vol. 4 (3), 1991. – Available at ftp://self.stanford.edu/pub/papers/selfPower.ps.Z.

[Valmari90] Valmari. – Stubborn sets for reduced state space generation. *In : Advances in Petri Nets 1987, ed. Grzegorz Rozenberg, LNCS 266*. – Springer Verlag, 1990.

[Wilson97] Wilson (Paul). – *Scheme and Its Implementation*. – published by ???, 1997. in preparation, draft versions available at http://www.cs.utexas.edu/users/-wilson.

[Winskel94] Winskel (Glynn). – *The Formal Semantics of Programming Languages*. – MIT Press, 1994, 2nd edition.

[Wood97] Wood (Bryan). – Draft Green Paper on “Object Model for Services”, 1997.

Appendix A

Behavior Language (BL)

A.1 Introduction

Note that this chapter provides more a reference material for the proposed behavior language (BL) rather that an introductory tutorial. BL is based on *Scheme* [Clinger et al.91]. *Scheme* provides all the basic programming structures needed in the specification of behavior clauses. In addition, this makes behavior clauses requiring evaluation to be directly executable using a *Scheme* interpreter. Each BL construct is described and its corresponding syntax is given in BNF-like style.

A.1.1 Behavior Specification Files :

A behavior specification file is a normal *Scheme* file (the extension "`.scm`" is a common convention for *Scheme* filenames). Thus it can contain behavior definitions and any other *Scheme* expressions needed to facilitate the specification of behaviors, e.g. auxiliary functions, data structures, variables, etc.

A.1.2 Some Typographical Conventions :

Within the BNF section for each clause, the following conventions are used :

- Italic face introduces a *non-terminal*.
- *scheming* is a *Scheme* expression.
- Type-writer bold face introduces a **`terminal`** symbol, i.e. a BL keyword.
- brackets "(" and ")" are also terminal symbols.
- *xxx-spec* is a *Scheme* expression used to represent *xxx* entities. Conventions have been defined to represent each relevant *xxx* entity in *Scheme*.

A.2 The Generic Part of the Behavior Language

The BL constructs presented here are generic in the sense that they are not dependent on any underlying notation used to represent the state of objects in the system and the interfaces to the objects in the system. In ODP terms, this would concern notations to represent information or computational models such as GRM, GDMO, ASN.1, CORBA-IDL, etc. So what is defined here is a set of generic syntaxes and semantics for behavior modeling.

A.2.1 `define-behavior` syntax

Each action is specified declaratively using the **`define-behavior`** syntax.

$$
\begin{aligned}
define-behavior \triangleq (\texttt{\textbf{define-behavior}}\ & label-spec \\
& scope \\
& when \\
& exec-rules \\
& pre \\
& body \\
& post)
\end{aligned}
$$

General remark on when, pre, body, post clauses : The contents of these clauses are *Scheme* expressions that can directly be evaluated at behavior simulation time. Therefore, there is no a priori structure imposed on these clauses, and there is no BL specific parsing done. The specification of these clauses is integrated in the system as normal *Scheme* code. Concretely they are represented internally as *Scheme* closures (λ-expression). *Scheme* closures represent functions. These clauses are functions of one single argument : the behavior excution context (BEC)). Executions of these clauses consists to apply the correspesponding function to the fetched behavior execution context. The value returned by when, pre, posts are interpreted as boolean values by the behavior propagation engine (BPE). The value returned by the application of body λ-expressions is meaningless for the BPE.

A.2.2 Behavior Label

The behavior label associates a unique identifier to each behavior. Though in general a string is used, any *Scheme* expression is possible.

A.2.3 Enabling Condition (guard)

The **`scope`** and **`when`** clauses define the enabling condition or guard.

A.2.3.1 `scope` syntax

$scope \triangleq (\texttt{\textbf{scope}}\ scope-spec)$

Behavior Scoping Principle A scope is just a shorter form of a logical condition that could be specified in the **`when`**. This shorter form is intended to capture and highlight the high level modeling abstractions used, e.g. the triggering event message, the roles and relationships of the target object, etc. The logical conditions implicitely represented by scopes expressions are implemented in behavior scoping functions. At behavior fetching time, behavior scoping functions in addtiion to test logical conditions determine a behavior execution context (BEC) that is then used in the evaluation of the other behavior clauses (i.e. when, pre, body and post). Scopes and behavior scoping functions define a bootstrapping step for behavior executions. The syntax of scopes is free. That means that one can specify anything after the keyword **`scope`**, and the BL parser does not complain. This is a feature and not a bug of the system. This way scopes are extensible. The reason for such an extensible syntax is to make possible the introduction of new scoping expressions and new scoping functions in the system (e.g. to deal with other forms of relationships, or information modeling abstractions in general).

Available Scopes In the current implementation, a predefined set of scopes and scoping functions is available. They should accomodate the usual needs. They are mainly oriented towards Role / Relationship Behavior Formalisation (RBF) for a model or relationships compliant with the GRM. RBF provides dynamic delegation / promotion of behavior from objects towards more suitable contexts, i.e. roles / relationships. This can be done using the following expressions as scope expressions in behaviors :

- `(scope (ri "REL-ID") (role "ROLE-ID") (msg "MSG-ID"))`:

  This scope is matched if a message sent to the BPE is a record whose record type descriptor (RTD) is `"MSG-ID"`. This message has to have a target instance (a field `inst`), representing an information object. This information object has to be a member of an existing relationship instance of relationship class `"REL-ID"`. In addition, the target information object has to play the role `"ROLE-ID"` in this relationship instance. When this matching condition is met, the resulting BEC is a `bec` record with the fields `msg`, `inst`, `ri`, `role` initialised to the matched values.

- `(scope (ri "REL-ID") (role "ROLE-ID"))` : Same as before, but the RTD of the message is not considered in the matching condition. This can be used to maintian role invariants. Note that the BEC is still computed in the same way. Note that the `msg` field is still filled as before with the message being sent, even if its RTD is not taken into account in the matching condition.

- `(scope (ri "REL-ID"))` : same as before, but neither the RTD of the message, nor the role played by the target information object instance are considered in the matching condition. Note that the BEC is still computed in the same way.

- `(scope (msg "MSG-ID"))` : This scope is matched if a message sent to the BPE is a record whose RTD is `"MSG-ID"`. e.g. `(scope (msg "ivpmsg-set"))`. When this matching condition is met, the resulting BEC is a `bec` record with the field `msg` initialized. This scope is clearly not RBF oriented. It may be used when no existing target information object instance is available. This may be the case for object creation, relationship establishment or termination. Note that RBF-like scopes are ok for object deletion.

A.2.3.2 `when` syntax

$$
\begin{aligned}
when \triangleq\ & (\textbf{when}\ )\ \ | \\
& (\textbf{when}\ \ scheming)
\end{aligned}
$$

The **`when`** behavior clause defines the enabling condition or the guard for a behavior to be fetched for further execution. The guard is a boolean expression that follows *Scheme* semantics for boolean expressions, it is false if it evaluates to #f, otherwise it is true. The default is #t. In figure A.1 a behavior with a typical **`when`** clause is shown.

A.2.4 `exec-rules` syntax

Given a message entering the system different reaction semantics are possible. Two important issues used to characterize a reaction semantics are the *fetching phase* and the *coupling mode*. These issues are specified using the **`exec-rules`** clause. Because there is no builtin semantics pre-defined for any message[1], a first type of reaction semantics is used to define what has to be actually performed when a

[1] That means that the default reaction to an event message sent to the system is to do nothing.

```
1   (define-behavior "REQUIP:EquipmentRole.<di,ul>-><di,lo>"
      (scope (ri "REQUIP") (role "equipmentRole") (msg "ivpmsg-set"))
      (when (and (equal? (Get (msg-> inst) "administrativeState") 'unlocked)
                 (equal? (msg-> attr) "administrativeState")
5                (equal? (msg-> val) 'locked)))
      (exec-rules (fetch-phase i) (coupled after-trigger))
      (pre
       (every (lambda (x) (equal? (Get x "administrativeState") 'locked))
              (Part (ri) "userPortTtpRole" )))
10    (body
       (display "BEHAVIOUR 2\n\n"))
      (post
       (equal? (Get (msg-> inst)  "administrativeState") 'locked)))
```

Figure A.1: A Behavior with a **when**, **pre** and **post**.

given event message is sent to the system. This reaction semantics is called the **is-trigger** reaction semantics just because it defines the semantics of the execution of the triggering event message. The fetching phase specifies when behaviors are candidate for fetching with respect to the occurrence of the triggering event message. Fetching can be specified to occur immediately (**phase-i**), in that case fetching is done as soon as the trigger is sent to the system. Otherwise, fetching can be deferred (**phase-ii**). A behavior whose fetching phase is deferred is candidate for fetching only when all the behaviors fetched with the **is-trigger** reaction semantics have completed. Note that completion means that the execution of the entire behavior block – i.e. **pre**, **body** and **post** – has terminated. Once a behavior has been fetched, another important issue is to determine how its behavior block has to be executed. Here two types of reaction semantics can be distinguished :

1. The **coupled** reaction semantics specifies that the execution of the behavior block is coupled with the caller, i.e. the behavior that has sent the trigger. An important consequence is that the calling behavior block execution is blocked until all the coupled reactions terminate.

2. The **uncoupled** reaction semantics specifies that the execution of the reaction is to be done in an independent thread of execution of the calling thread. Being able to model such reaction semantics is very useful in a distributed environment, because distribution lends naturally to independent threads of execution, e.g. if a message sent represents a communicatio with a remote object.

To completely define a coupled reaction, one has to specify when the execution of the behavior block is expected to initiate and terminate. For an uncoupled reaction, only the initiation phase is relevant. For both the coupled and uncoupled cases such parameters are specified relatively to the **is-trigger** reaction semantics. The execution phase during which behaviors with the **is-trigger** reaction semantics are executing is called the **during-trigger** phase. Intuitively, it is possible to specify what processing is directly associated to the occurrence of an event. Then, it is possible to specify what behaviors are enabled before, during and after the direct processing. As shown in figure A.2, it follows that the time during which the reactions to a trigger are being executed can be partitioned into three phases : the **before-trigger**, the **during-trigger**, and the **after-trigger** phases.

Finally the complete and concrete syntax for execution rules is defined as follows :

$$exec-rules \triangleq (\textbf{exec-rules}\ \ fetch-phase$$
$$coupled) \ \ |$$

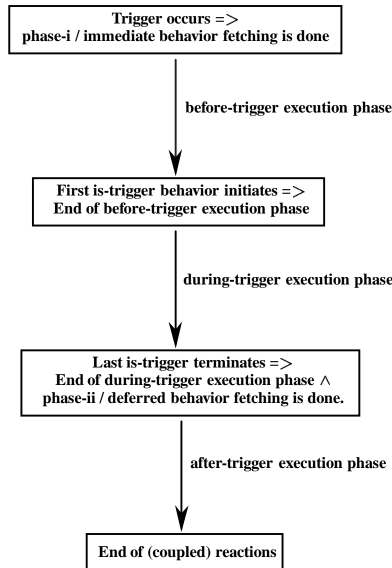


Figure A.2: Trigger Reaction Phases.

$$(\texttt{\textbf{exec-rules}}\ fetch-phase\ uncoupled)$$

$$fetch-phase \triangleq (\texttt{\textbf{fetch-phase i}}) \mid (\texttt{\textbf{fetch-phase ii}})$$

$$coupled \triangleq (\texttt{\textbf{coupled before-trigger}}) \mid (\texttt{\textbf{coupled before-trigger during-trigger}}) \mid (\texttt{\textbf{coupled before-trigger after-trigger}}) \mid (\texttt{\textbf{coupled during-trigger}}) \mid (\texttt{\textbf{coupled is-trigger}}) \mid (\texttt{\textbf{coupled during-trigger after-trigger}}) \mid (\texttt{\textbf{coupled after-trigger}})$$

$$uncoupled \triangleq (\texttt{\textbf{uncoupled before-trigger}}) \mid (\texttt{\textbf{uncoupled during-trigger}}) \mid (\texttt{\textbf{uncoupled after-trigger}})$$

Note that, in the coupled mode, only one execution phase is given, that means that the behavior block is intended to initiate and terminate in that phase. Note also that **`(coupled is-trigger)`** defines a behavior with the **`is-trigger`** execution semantics.

A.2.4.1 exec-rules in Action

In figure A.3 a simple system of two nodes, connected by a link relationship is represented. In figure A.4 a set of behaviors to test miscellaneous forms of **exec-rules** syntax are proposed. These behaviors are all defined with the same scope, which consists of the message `exec-rules-test-msg` being sent to an object expected to be a destination node in the link relationship. The `exec-rules-test-msg` is defined in A.4 (line 5). Each behavior in figure A.4 displays an appropriate output as its body is executed. Figure A.5 shows a possible behavior propagation and the output produced that result from sending the `exec-rules-test-msg` message to node 2, which is a destination node, is shown. Note that because of nondeterminism, many other propagations are possible.

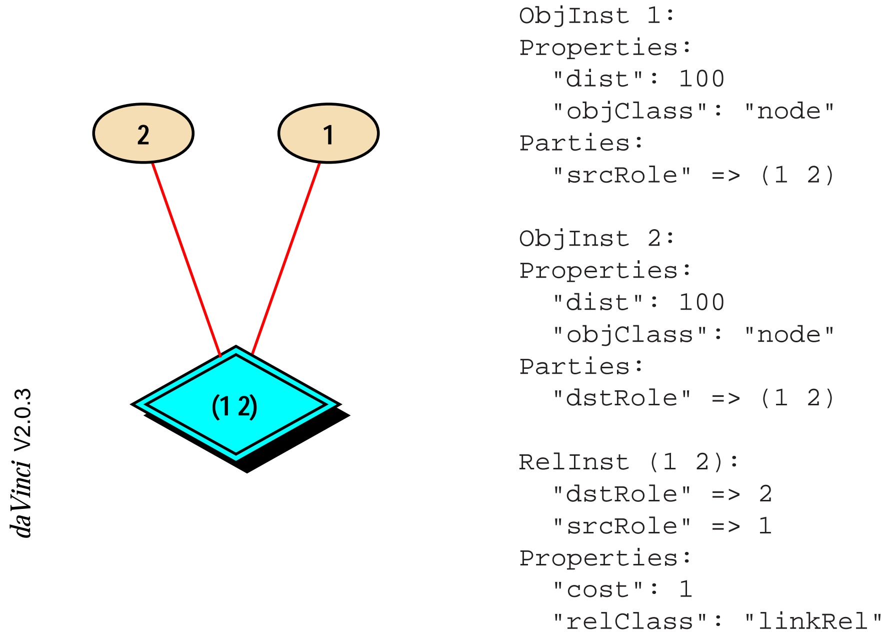


Figure A.3: Simple Instance Repository.

A.2.5 pre

$pre \triangleq$ (**pre**) |
  (**pre** $scheming$)

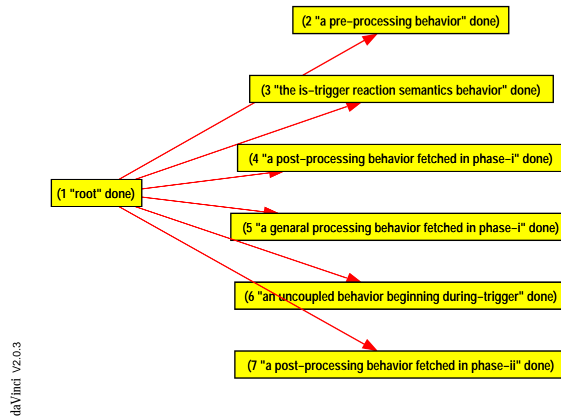
A pre-condition is checked before the execution of a behavior body. Syntactically, it is a *Scheme* expression interpreted as a boolean in the *Scheme* semantics for boolean expressions, i.e. it is false if it evaluates to #f, otherwise it is true. The default is #t. Figure A.1 gives an example of a behavior where a pre-condition is specified.

A.2.6 body

$body \triangleq$ (**body**) |
  (**body** $scheming$)

```
1
    ;;; the message record used to trigger the examples of behaviors about
    ;;; exec-rules.

5   (genrec:define exec-rules-test-msg inst)


    (define-behavior "the is-trigger reaction semantics behavior"
      (scope (ri "linkRel") (role "dstRole") (msg "exec-rules-test-msg"))
10    (when)
      (exec-rules (fetch-phase i) (coupled is-trigger))
      (pre)
      (body
       (printf "exec-rules test:\n   the is-trigger reaction semantics\n"))
15    (post))

    (define-behavior "a pre-processing behavior"
      (scope (ri "linkRel") (role "dstRole") (msg "exec-rules-test-msg"))
      (when)
20    (exec-rules (fetch-phase i) (coupled before-trigger))
      (pre)
      (body
       (printf "exec-rules test:\n  some pre-processing\n"))
      (post))
25
    (define-behavior "a post-processing behavior fetched in phase-i"
      (scope (ri "linkRel") (role "dstRole") (msg "exec-rules-test-msg"))
      (when)
      (exec-rules (fetch-phase i) (coupled after-trigger))
30    (pre)
      (body
       (printf "exec-rules test:\n  some post-processing fetched in phase-i\n"))
      (post))

35  (define-behavior "a post-processing behavior fetched in phase-ii"
      (scope (ri "linkRel") (role "dstRole") (msg "exec-rules-test-msg"))
      (when)
      (exec-rules (fetch-phase ii) (coupled after-trigger))
      (pre)
40    (body
       (printf "exec-rules test:\n  some post-processing fetched in phase-ii\n"))
      (post))

    (define-behavior "a genaral processing behavior fetched in phase-i"
45    (scope (ri "linkRel") (role "dstRole") (msg "exec-rules-test-msg"))
      (when)
      (exec-rules (fetch-phase i) (coupled before-trigger after-trigger))
      (pre)
      (body
50     (printf "exec-rules test:\n  some general processing fetched in phase-i\n"))
      (post))

    (define-behavior "an uncoupled behavior beginning during-trigger"
      (scope (ri "linkRel") (role "dstRole") (msg "exec-rules-test-msg"))
55    (when)
      (exec-rules (fetch-phase i) (uncoupled during-trigger))
      (pre)
      (body
       (printf "exec-rules test:\n  an uncoupled behavior beginning during-trigger\n"))
60    (post))

    ;;; to test:
    ;;; (sif:msgsnd (exec-rules-test-msg:make2 `(inst 2)))
```

Figure A.4: Behavior for Testing **exec-rules**.




```
1   guile> (sif:msgsnd (exec-rules-test-msg:make2 '(inst 2)))

    exec-rules test:
      some pre-processing
5   exec-rules test:
       the is-trigger reaction semantics
    exec-rules test:
      some post-processing fetched in phase-i
    exec-rules test:
10    some post-processing fetched in phase-ii
    exec-rules test:
      an uncoupled behavior beginning during-trigger
    exec-rules test:
      some general processing fetched in phase-i
15
    guile> (pretty-print *bpstate*)
     (%%record
     bpstate
     (trace (1 5 6 7 4 3 2 1))
20   (nowait-set ())
     (nowait-set-stack ())
     (atomic-exec-lev 0)
     (ben-id-ctr 7)
     (test-trace ()))
25
    guile>
```

In line 19, the trace field gives the (reverse) sequence of steps that was followed by this behavior propagation.

Figure A.5: **exec-rules** Test.

The body of a behavior is an imperative / procedural piece of *Scheme* code. The result is not important, i.e. it is not taken into account by the BPE. Note that there is no explicit communication between the BPE and a behavior body. Figure A.6 gives an example of a behavior where a body is specified, with typical examples of useful *Scheme* statements (e.g. `let*`, `cond`, etc).

```
1     (define-behavior "send-unreserve-subpath"
        (scope  (ri "PathMgmt") (role "path") (msg "unreserve-path-msg"))
        (when)
        (exec-rules (fetch-phase i) (coupled is-trigger))
5       (pre)
        (body
         (let* ((path-inst (Part (ri) "path"))
                (dlcid (car (Get path-inst "done")))
                (subpath (cadr (Get path-inst "done")))
10              (msd-mgr (caddr (Get path-inst "done"))))
           (msgsnd (moreq-action:make2
                    `(msd ,msd-mgr)
                    `(invoke ,(msif:new-invoke))
                    `(moc "mLinkConnection")
15                  `(moi ,subpath)
                    `(actype "releaseConnection")
                    `(acinfo ,(asn:seq (asn:field-value
                                        'connectionId
                                        (moi:scm->asn dlcid))))))
20
           (Set path-inst "done" (cdddr (Get path-inst "done")))

           (cond ((null? (Get path-inst "done"))
                  (Delete path-inst)
25                (Terminate (ri)))

                 (else (msgsnd (unreserve-path-msg:make (Part (ri) "path")))))))
        (post))
```

Figure A.6: A Behavior with a **`body`**.

A.2.7 `post`

$post \triangleq$ (**`post`**) |
(**`post`** $scheming$)

A post-condition is checked after the execution of a behavior body. Syntactically, it is a *Scheme* expression interpreted as a boolean in the *Scheme* semantics for boolean expressions, i.e. it is false if it evaluates to #f, otherwise it is true. The default is #t. Figure A.1 gives an example of a behavior where a post-condition is specified.

A.2.8 `bec`

The BL syntax used to access the execution context of a fetched behavior (BEC) is :

$bec \triangleq$ (**`bec`**)

A.2.8.1 Access to BEC Fields

A BEC structure has been defined as a record with well identified fields, typically adapted to RBF. Access to these fields can be done from the value returned by `(bec)` using record access functions[2]. However, to facilitate these accesses, macros have been defined to provide a more convenient way that makes possible to access directly such BEC fields :

- $msg \triangleq$ (**msg**)

  This syntax can be used to access the `msg` field of a BEC. It is equivalent to use the record accessor function as follows : `(bec:msg (bec))`.

- $inst \triangleq$ (**inst**)

  This syntax can be used to access the `inst` field of a BEC. It is equivalent to use the record accessor function as follows : `(bec:inst (bec))`.

- $ri \triangleq$ (**ri**)

  This syntax can be used to access the `ri` field of a BEC. It is equivalent to use the record accessor function as follows : `(bec:ri (bec))`.

- $role \triangleq$ (**role**)

  This syntax can be used to access the `role` field of a BEC. It is equivalent to use the record accessor function as follows : `(bec:role (bec))`.

- $rel \triangleq$ (**rel**)

  This syntax can be used to access the `rel` field of a BEC. It is equivalent to use the record accessor function as follows : `(bec:rel (bec))`.

In figure A.7 a behavior is shown that illustrates the use of the BEC accessors. Figure A.8 shows the output produced by this behavior when executed because of the `bec-fields-access-msg` message is sent to node 2, which is a source node.

A.2.8.2 Access to BEC Message Fields

To facilitate access to the `msg` fields, i.e. the `msg` field in a BEC, a macro has been defined :

$msg\!-> \triangleq$ (**msg->** $field$) |
(**msg->** $field1$
...)

With this BL syntax it is possible to access directly the message fields. Using record accessors, it is equivalent to :

```
(genrec:get (msg) <msg-field>)
```

Example : `(msg-> attr)` can be used to access the `attr` field in e.g. an `ivpmsg-set` message record. In figure A.8 another example is provided.

[2] Though records are not available in standard *Scheme* [Clinger et al.91], the *Scheme* library [Eigenschink et al.94] provides an implementation of records based on vectors.

```
1
    ;;; the message record used to trigger the examples of behaviors about
    ;;; BEC fields accessors.

5   (genrec:define bec-fields-access-msg inst)


    (define-behavior "bec fields accessors test"
      (scope (ri "linkRel") (role "srcRole") (msg "bec-fields-access-msg"))
10    (when)
      (exec-rules (fetch-phase i) (coupled is-trigger))
      (pre)
      (body

15     (newline)
       (newline)
       (display "The value of the entire BEC is:")
       (pretty-print (bec))

20     (newline)
       (display "test of BEC fields accessors:")
       (newline)
       (display "  Access to msg field => ") (display (msg))
       (newline)
25     (display "  Access to inst field => ") (display (inst))
       (newline)
       (display "  Access to ri field => ") (display (ri))
       (newline)
       (display "  Access to role field => ") (display (role))
30     (newline)

       (display "  Access to rel field => ") (display (rel))
       (newline)

35     (newline)
       (display "test of msg fields accessor:")
       (newline)

       (display "  Access to inst field in the msg => ")
40     (display (msg-> inst))
       (newline)

       (local-set! xxx (+ 10 5))

45     (newline)
       (newline)
       (display "The value of the entire BEC is now:")
       (newline)
       (pretty-print (bec))
50
       (newline)
       (newline)
       (display "Getting the loacl BEC variable xxx => ")
       (display (local-get xxx))
55     (newline)
       )
      (post))

    ;;; to test
60  ;;; (sif:msgsnd (bec-fields-access-msg:make2 `(inst 2)))
```

Figure A.7: Behavior for Testing BEC Fields Accessors.

```
1   guile> (sif:msgsnd (bec-fields-access-msg:make2 `(inst 2)))

    The value of the entire BEC is :
    (%%record
5    bec
     (msg (%%record bec-fields-access-msg (inst 2)))
     (inst 2)
     (role "srcRole")
     (ri (2 4))
10   (rel "linkRel")
     (locals ()))

    test of BEC fields accessors:
      Access to msg field => (%%record bec-fields-access-msg (inst 2))
15    Access to inst field => 2
      Access to ri field => (2 4)
      Access to role field => srcRole
      Access to rel field => linkRel

20  test of msg fields accessor:
      Access to inst field in the msg => 2

    The value of te entire BEC is now:
    (%%record
25   bec
     (msg (%%record bec-fields-access-msg (inst 2)))
     (inst 2)
     (role "srcRole")
     (ri (2 4))
30   (rel "linkRel")
     (locals ((%%record pair (key xxx) (value 15)))))

    Getting the loacl BEC variable xxx => 15

35  guile>
```

Figure A.8: BEC Fields Accessors Test.

A.2.8.3 BEC Local Variables

BEC local variables are typically used to pass information between behavior clauses, e.g. from a pre- to a post-condition. Note that an usafe way of doing this is to use global variables because a global variable may be used by several behavior executions. Unlike a BEC that is by definition created at each occurrence of the execution of a behavior.

- $local - get \triangleq ($**`local-get`** $var)$

  This syntax allows one to get the value associated to the local variable `var` of the BEC.

- $local - set! \triangleq ($**`local-set!`** var
  $val)$

  This syntax allows one to set the value associated to the local variable `var` of the BEC.

In figure A.8 an example of the use of **`local-get`** and **`local-set!`** BL syntaxes is shown.

A.2.9 Sending Messages to the BPE

Here are described the syntaxes used to send messages in behavior bodies. In a behavior body, event messages can be built and sent to the BPE for further behavior propagations. Note that these syntaxes are intended to be used only in behavior bodies. A behavior boby is the only place where messages can be sent (other behavior clauses are always evaluated atomically).

- $msgsnd \triangleq ($**`msgsnd`** $msg1$
  $msg2$
  $...)$

  This syntax allows one to send the mesages given as arguments.

- $msgsndl \triangleq ($**`msgsndl`** $msgs)$

  This syntax allows one to send a list of messages that is given as argument.

These two syntaxes achieve in fact the same functionality. The only difference resides in the way the messages to send have to be given, either one by one or in a list. Note that in the end the form :
(**`msgsnd`** msg_1 msg_2) is equivalent to : (**`msgsndl`** (`list` msg_1 msg_2)).

One more important point is the result returned by the use of these syntaxes in a behavior body. In figure A.9 four behaviors are defined to show the result returned by **`msgsnd`**. Three message records are defined to trigger these behaviors. The first behavior `msgsnd-test-msg1` is triggered by the message with the same name. In its body it sends using **`msgsnd`** two messages that trigger the three other behaviors. Finally the result of the **`msgsnd`** is printed. The ouput resulting from this behavior propagation is given in figure A.10. The result is a list, each element is a list of BECs. There is one such list of BECs per message sent. The BECs gathered in the result are the BECs of executed behaviors with the **`is-trigger`** reaction semantics, that is the one that are considered to define the direct semantics of the execution of a message. Therefore in figure A.10 one can check that the resulting list printed contains two lists, because two messages were sent. For the message `msgsnd-test-msg2` one **`is-trigger`** behavior was fetched and executed. For the message `msgsnd-test-msg3` two **`is-trigger`** behaviors were fetched and executed (behavior `msgsnd-test-msg3a` and behavior `msgsnd-test-msg3b`).

```
1
   ;;; the message records used to trigger the examples of behaviors about msgsnd
   ;;; and msgsnd result.

5  (genrec:define msgsnd-test-msg1)
   (genrec:define msgsnd-test-msg2)
   (genrec:define msgsnd-test-msg3)


10 (define-behavior "msgsnd-test-msg1"
     (scope (msg "msgsnd-test-msg1"))
     (when)
     (exec-rules (fetch-phase i) (coupled is-trigger))
     (pre)
15   (body
      (printf "Behavior msgsnd-test-msg1:\n")
      (let ((res (msgsnd (msgsnd-test-msg2:make) (msgsnd-test-msg3:make))))
        (printf "\n\nHere is the result returned by msgsnd:\n")
        (pp res))
20    )
     (post))

   (define-behavior "msgsnd-test-msg2"
     (scope (msg "msgsnd-test-msg2"))
25   (when)
     (exec-rules (fetch-phase i) (coupled is-trigger))
     (pre)
     (body
      (printf "\n\nBehavior msgsnd-test-msg2\n")
30    (bec:res! (bec) `(here is msg2 result))
      )
     (post))

   (define-behavior "msgsnd-test-msg3a"
35   (scope (msg "msgsnd-test-msg3"))
     (when)
     (exec-rules (fetch-phase i) (coupled is-trigger))
     (pre)
     (body
40    (printf "\n\nBehavior msgsnd-test-msg3a\n")
      (bec:res! (bec) `(here is msg3a result))
      )
     (post))

45 (define-behavior "msgsnd-test-msg3b"
     (scope (msg "msgsnd-test-msg3"))
     (when)
     (exec-rules (fetch-phase i) (coupled is-trigger))
     (pre)
50   (body
      (printf "\n\nBehavior msgsnd-test-msg3b\n")
      (bec:res! (bec) `(here is msg3b result))
      )
     (post))
55
   ;;; to test:
   ;;; (sif:msgsnd (msgsnd-test-msg1:make))
```

Figure A.9: Behavior for Testing **msgsnd** Result.

```
1    guile> (sif:msgsnd (msgsnd-test-msg1:make))
     Behavior msgsnd-test-msg1:

     Behavior msgsnd-test-msg3a
5
     Behavior msgsnd-test-msg3b

     Behavior msgsnd-test-msg2

10   Here is the result returned by msgsnd:
     (((%%record bec (msg (%%record msgsnd-test-msg2)) (locals ())
                     (res (here is msg2 result))))
      ((%%record bec (msg (%%record msgsnd-test-msg3)) (locals ())
                     (res (here is msg3b result)))
15     (%%record bec (msg (%%record msgsnd-test-msg3)) (locals ())
                     (res (here is msg3a result)))))

     guile>
```

Figure A.10: **msgsnd** Result Test.

A.2.10 Atomic Execution Syntax

This syntax is used to make part of a behavior body to execute atomically.

- $atomic \triangleq$ (**atomic** $se1$
  ...)

In Behavior A.2.1 an example of use of the atomic syntax is provided. The use of the atomic syntax ensures that the two attributes `next` and `dist` are updated in a same and coherent phase.

Behavior A.2.1 *Step of the Spanning Tree :*

```
(define-behavior "st"
  (scope (ri "linkRel") (role "dstRole") (msg "ivpmsg-set"))
  (when (equal? (msg-> attr) "dist"))
  (exec-rules (fetch-phase ii) (uncoupled after-trigger))
  (pre)
  (body
   (if (not (equal? (Get (Part (ri) "srcRole") "ident")
                    (Get (Part (ri) "dstRole") "ident")))
       (atomic
         (Set (Part (ri) "srcRole") "ident" (Get (Part (ri) "dstRole") "ident"))
         (Set (Part (ri) "srcRole") "next" (Part (ri) "dstRole"))
         (Set (Part (ri) "srcRole") "dist" (+ (Get (Part (ri) "dstRole") "dist")
                                             (Get (ri) "cost"))))
       (if (< (+ (Get (Part (ri) "dstRole") "dist") (Get (ri) "cost"))
              (Get (Part (ri) "srcRole") "dist"))
           (atomic
            (Set (Part (ri) "srcRole") "next" (Part (ri) "dstRole"))
            (Set (Part (ri) "srcRole") "dist" (+ (Get (Part (ri) "dstRole") "dist")
                                                (Get (ri) "cost"))))))
   )
  (post))
```

A.2.11 Message Buffering Syntax

This syntax is used to make part of a behavior body buffering messages instead of sending them one by one to the BPE. The buffered messages are sent later on altogether, when the end of the behavior body marked for message buffering is reached. This syntax is useful because some syntaxes have been defined to ease the construction and sending of some common messages, such as the IVP messages. for instance the syntax **`Set`** can be used as :

`(Set <inst> <prop> <val>)` to ease the construction and sending of an `ivpmsg-set` message as follows :

`(msgsnd (ivpmsg-set:make <inst> <prop> <val>))`

So, a problem occur if one wants to send e.g. two `ivpmsg-set` using the simpler `Set` syntax.

To solve this problem one should use :

```
1   (msgbufsnd
2     (Set <inst1> <prop1> <val1>)
3     (Set <inst2> <prop2> <val2>))
```

- $msgbufsnd \triangleq ($**`msgbufsnd`** $se1$
  $...)$

Appendix B

Open Distributed Processing (ODP)

B.1 Introduction

ODP concepts have been referenced and used in many parts of this thesis. The more important concepts in the context of the thesis concern aspects related to behavior and to ODP information and computational viewpoints. They are considered in section 2.4.1 at the place they are actually used. This chapter aims at giving a more general description of RM-ODP that goes beyond what was strictly needed in this thesis. Basically the materials in this section are inspired from [Genilloud96, Tinac imc94, Tinac cmc94, G851 0196] and the ODP documents themselves. The purpose of RM-ODP is to provide a framework to enable distributed application components to interwork despite heterogeneity in equipments, operating systems, networks, languages, database models and management authorities. The ODP reference model (RM-ODP) is partitioned into four documents :

- RM-ODP1 [Rm odp1] : Overview. Contains a motivated overview of ODP, giving scoping, justification and explanation of key concepts. This part is not normative.
- RM-ODP2 [Rm odp2] : Foundations. Contains the definition of the concepts and analytical framework for normalized description of (arbitrary) distributed processing systems. This part is normative. These concepts are, in fact, so general that they can be used to produce any kind of model. For instance, definitions of generally useful notions such as object, action, behavior, state, type, class, template . . . are provided in RM-ODP2. These concepts are intended to be applicable to arbitrary ODP viewpoints. Though it is advantageous to have a common set of consistent and reusable concepts, the resulting generality and abstractness has a price : it makes the fundamental concepts quite difficult to understand. As noticed in [Genilloud96], it is possible that only experts with broad experience in system modeling and language design can truly appreciate their significance.
- RM-ODP3 [Rm odp3] : Architectural Framework. The RM-ODP is not limited to modeling. It provides an approach and a set of rules that define an architecture to build distributed systems. This architecture allows to integrate existing systems and applications (so-called legacy systems) and to benefit from advances in object, communication, or distribution technologies. This document defines three major concepts of the RM-ODP architecture : *viewpoints*, *distribution transparencies* and *ODP functions*.
- RM-ODP4 [Rm odp4] : Architectural Semantics. Contains the mapping of architectural concepts (that use in their turn concepts of the RM-ODP Foundations) into standardized formal de-

scription techniques such as SDL [IT93], Lotos [Lotos87], Estelle [Estelle89] and **Z** [Spivey89]. This part is not yet standardized.

B.2 Basic Object and Behavior Concepts

ODP defines a model as a set of interacting objects, which means that they all participate in actions. ODP advocates object orientation as a suitable technique for making models, whatever the purpose of the model is and whatever the level of abstraction is. In other words objects are suitable for any kind of application (ranging from simple hardware logic systems to autonomous robots) and can be used at any step in the production of an application (specification, design, and implementation). This can be explained by the fact that objects are based on the more fundamental features required in any kind modeling of modeling activity, that are *encapsulation* and *abstraction*. Then from the basic object modeling concepts, behavior concepts can be elaborated, e.g. *action*, *object state*, *object behavior* . . . . Both object and behavior ODP concepts useful in this thesis are described in section 2.4.1. The purpose of RM-ODP2 [Rm odp2], the ODP Foundations document is to provide a definition of basic concepts to avoid any possible misunderstanding on concepts such as object and behavior that are then extensively used in each viewpoint modeling language. Such a common basis is useful because of all the variants existing about such basic concepts in both object orientation and behavior modeling communities.

B.3 Viewpoints

One important point is the precise specification of distributed application components. For that reason the first fundamental concept of RM-ODP is the notion of viewpoints. This ODP concept implicitly recognizes that the description of complex things such as a distributed application has to be done along several perspectives or viewpoints. So the first problem is complexity that constrains to partition a specification.

A second problem is clarity and separation of concerns. It turns out that very different issues are involved when considering the procurement of a distributed application. The adoption of a particular viewpoint typically allows a perception of a system which emphasizes on one particular concern, while ignoring other characteristics that are temporarily irrelevant to that concern. Being able to build different models of a system according to different levels of abstractions is important, what is again more important is to ensure that these different levels of abstractions (or viewpoints) are wisely chosen so that in the end a model of the complete system can be composed. Composition may be performed in order to ensure that what is specified in one viewpoint is not in contradiction with what is specified in another viewpoint. Composition may also define how specification in one viewpoint are translated or simply referenced into another viewpoint. Today, it seems that there is a widely accepted agreement within the ODP community about the idea of viewpoints and on the specific five viewpoints that have been selected by the ODP architecture. Enterprise models focus on the *role* of the distributed system within the organization. Information models focus on the *semantics* of the information that is processed by the system, irrespective on how the system is built. Computational models focus on how the system can be built, i.e. it defines the basic *components* that constitute the units of distribution of the distributed system. Thus to each component that can be potentially distributed, interfaces are associated, and the interactions that occur at this interfaces are described. Engineering models focus on the complex problem of the actual physical distribution and resource usage. Finally technology models are concerned with the implementation structure in terms of hardware and software involved.

B.3.1 Viewpoint Languages and Notations

It is important to note that RM-ODP distinguishes between languages and notation. A language is defined by a set of terms (vocabulary) and a grammar that defines how the terms of the language can be combined. There is a viewpoint language associated to each viewpoint, to help the specification of models for each viewpoint. However, RM-ODP does not advocate any concrete notation to support the viewpoint languages. In fact, the objective of RM-ODP4 [Rm odp4] is to explain how standardized formal description techniques (LOTOS, SDL, Estelle, Z) can be used to produce models in the ODP viewpoints. An important point is that viewpoint languages are based on a common substrate given by the concepts defined in the ODP Foundations document [Rm odp2].

B.3.2 Enterprise Viewpoint Language

A distributed system is built by an organization for making itself more efficient or productive. The aim of an enterprise model is to explain and justify the role of the system that is considered within the organization. So an *enterprise model* defines the purpose, scope, and policies of an ODP system. It is an object-based model, and as such it consists of a set of interacting objects (in that case enterprise objects) participating in actions. The enterprise objects concern not only the system of interest but also its environment, e.g. its users. One of the key concepts of the enterprise language is that of *contract*. All enterprise objects fulfill one or several roles which are expressed in terms of permissions, obligations, and prohibitions. Enterprise models are design independent. They usually do not reflect the way the system is built and distributed, except when issues related to the construction or distribution are directly relevant to the contracts between the system and its environment.

In summary, the main concepts of the enterprise language are those of *contract*, *community*, *role*, *activity*, *resource*, and *policy*. The main structuring rules of the language is that a system is specified in terms of enterprise objects fulfilling different roles within a community, and that roles are specified in terms of obligations, permissions, and prohibitions.

B.3.3 Information Viewpoint Language

This part is considered in details in section 2.4.2.1. In summary, the main concepts of the information language are those of *information object*, *state*, *invariant*, *schema*, *static schema*, *dynamic schema*. The main structuring rule of the information viewpoint language is that the behavior of a system is specified using dynamic and invariant schemas.

B.3.4 Computational Viewpoint Language

Computational modeling concepts are of direct interest in the context of this thesis, and are thus presented in details in section 2.4.2.2. In summary, the main concepts of the computational language are those of *computational object*, *interface*, *operation*, *signal*, and *stream*. Rules of the computational language are interaction rules, naming rules, binding rules, and type rules.

B.3.5 Engineering Viewpoint Language

Whereas the computational model defines how a system may be distributed, an *engineering model* shows how the system is actually distributed, and specifies the infrastructure that is required to support distribution. In particular, it is explained (i) how communication is actually achieved, (ii) what computing resources are used, i.e. physical storages and processing units. All that being made with respect to the computational objects identified in the computational model. So the relation between a computational model and an engineering model is a refinement relationship. Strong relationships

exist between computational and engineering objects, e.g. one to one correspondence may be used between computational objects and *basic engineering objects* (the engineering objects directly related to the representation of the computational objects are called *basic engineering objects*). Configurations of engineering objects is based on the concept of *channel*. The different engineering objects that make up a channel provide a variety of distribution transparency services, e.g. *Stub* objects for access and concurrency distribution transparencies, *binder* objects for location, resource, and migration distribution transparencies. *Protocol* objects represent protocol machines, and *interceptor* objects represent gateways that may be used to perform protocol conversions. Engineering objects identify also various kind of grouping (i.e. containment relationships) to represent processing resources. The physical grouping that typically represent a computer system is called a *node* object. A node contains one or more *capsules* that owns part of the storage and part of the node processing resources. The usual notion of process can be viewed as a capsule. A capsule is supported by a *nucleus* that is responsible for the intra-capsule communications and for the creation of threads. Finally, basic engineering objects may be grouped into *clusters* within a capsule to allow for group activation, deactivation and migration. *Cluster manager* and *capsule manager* are special objects used to control the management of capsules and clusters. In summary the main concepts of the engineering viewpoint language are : *basic engineering objects*, *capsule*, *channel*, *protocol object*, *interceptor*, *nucleus*. The main rules in the engineering viewpoint language are the channel rules.

B.3.6 Technology Viewpoint Language

An engineering specification gives the details about how the system is physically distributed, what are the communication channels used and the protocols used to convey information on these communication channels. However, the engineering viewpoint remains independent from any technological choices used to implement the system, i.e. the specific particular hardware and software used. This ensures that the engineering viewpoint modeling is portable to a large variety of environments. The choice of the different pieces of technology is specified in the *technology* model, which is closely associated to the engineering model. Because technological issues are required only for the procurement of final and runnable distributed application, we were never concerned with technological issues in this work.

B.4 Other Architectural Concepts

Besides viewpoints, ODP defines the two other important concepts of distribution transparencies and ODP functions.

B.4.1 Distribution Transparencies

Distribution transparency aims at providing solutions to a number of recurring problems resulting directly from distribution. For example, heterogeneity, fault tolerance, location . . . . The goal is to establish underlying standards onto which global solutions for theses difficult problems can be based once and for all. In the end the application designer can concentrate on its pure application issues by working at a layer where the underlying distribution requirements are transparent. The advantages of this approach (compared with a solution where distribution issues are addressed as part of the application process design) are two-fold : (i) application development is simplified, and (ii) portability and inter-working are increased. This approach encourages the re-use and integration at the system level. ODP defines a number of transparencies identified as worth to be available in distributed application standards. Among many others, one can find in this list :

- Access transparency, which is concerned with communication protocols, services, network representation (marshaling, unmarshaling) that should enable low level interoperability between heterogeneous systems.
- Location transparency is intended to provide a logical view of naming, independently of the physical location of objects.
- Failure transparency is intended to mask the failure of a component and its recovery by another one.

A remark than can be made about distribution transparency facilities is that though the goals seem very attractive, in practice such generic facilities work only for the more basic things such as access and location transparency. One problem is that it is very difficult to reach general agreements about complex distribution facilities such as fault tolerance mechanisms. So if needed elaborated distribution transparency facilities are integrated in an add-hoc fashion directly in the application design process. This problem about generic distribution facilities can be compared with the problem of generic object services such as CORBA object services or OSI-SM systems management functions (SMF). Though everybody agrees on their necessity, they are rarely available, or they are implemented in a form that is often quite different from the one stated in the standards. In the end, it is impossible for an architect of distributed applications to rely on the general availability of such general purpose facilities.

B.4.2 ODP Functions

ODP defines a whole set of functions that are expressed to support the needs of the computational and the engineering languages. The following main function groups have been identified among others :

- Management functions are intended to control the life-cycle of the engineering structures and to handle the various grouping of objects identified in the engineering viewpoint.
- Coordination functions are dedicated to ensure dissemination, coordination, and consistency of information between distributed groups of objects.
- Repository functions deal with persistent storage. This includes a general storage function, a general relationship repository, and various specializations of those. Among these specializations one can find the *type repository* function and a *trader* function whose main role is to store information about interface instances in order to allow dynamic configuration and evolution of the system.

Appendix C

Spanning Tree Case Study

This is a simple distributed algorithm that given a finite directed graph and a root constructs a spanning tree, following the classical Bellman-Ford algorithm. For each node n the algorithm computes the distance $dist[n]$ from n to the root and, if n is not the root the next node $next[n]$ in the spanning tree.

Usually in the litterature [Engberg et al.92] the algorithm starts with, $dist[n] = \infty$ for all nodes. A rather different convention is used here using a specific attribute identifying the occurrence of each execution of the spanning tree algorithm at each node. At each improvement step on a node, if the identification at the node is different from the identification of the current execution, then it is the same as $dist[n] = \infty$.

The principle of an improvement step states merely that if $dist[n]$ is set to d, and that if n is a destination node of a link $l = n \to m$ then, if $cost[l] + d[n] < d[m]$ then $d[m]$ can be decreased to $d[m]$ =cost[l]+d[n], and $next[m] = n$. Informally, this states that if a path is discovered to the root taking a given neighboring node, and if the length of this new path is shorter than the old one that was known at that node, then this new path can replace the old one.

Behavior C.0.1 is the only one behavior used to implement improvement steps. This behavior performs two kind of improvement steps. The first one is used for initial improvement step. The second one may be used for further improvement steps. The distinction is simply made by comparing the ”ident” attribute of source and destination nodes. The `"ident"` attribute identifies in fact the current execution of the spanning tree algorithm.

Behavior C.0.1 *The Simple Spanning Tree Improvement Step Behavior :*

```
1   (define-behavior "st"
2     (scope (ri "linkRel") (role "dstRole") (msg "ivpmsg-set"))
3     (when (equal? (msg-> attr) "dist"))
4     (exec-rules (fetch-phase ii) (uncoupled after-trigger))
5     (pre)
6     (body
7      (if (not (equal? (Get (Part (ri) "srcRole") "ident")
8                       (Get (Part (ri) "dstRole") "ident")))
9          (atomic
10           (Set (Part (ri) "srcRole") "ident"
11                (Get (Part (ri) "dstRole") "ident"))
12           (Set (Part (ri) "srcRole") "next"
13                (Part (ri) "dstRole"))
14           (Set (Part (ri) "srcRole") "dist"
15                (+ (Get (Part (ri) "dstRole") "dist")
16                   (Get (ri) "cost"))))
17         (if (< (+ (Get (Part (ri) "dstRole") "dist") (Get (ri) "cost"))
18                (Get (Part (ri) "srcRole") "dist"))
19             (atomic
20              (Set (Part (ri) "srcRole") "next"
21                   (Part (ri) "dstRole"))
22              (Set (Part (ri) "srcRole") "dist"
```

```
23                    (+ (Get (Part (ri) "dstRole") "dist")
24                       (Get (ri) "cost"))))))
25      )
26    (post))
27
```

Appendix D

Continuations in *Scheme*

Since the execution support for the BPE algorithm presented in chapter 4 makes use of continuations to implement intermediate execution steps performed in behavior bodies, this appendix presents more in details this very powerful control structure. In section D.1, to understand this concept the same approach used in [Wilson97] is observed that consists of taking an implementation perspective. Finally, section D.2 give an example of use.

D.1 Implementation Perspective

Scheme has the usual control constructs that most languages have, e.g. conditionals (if statements), loops, and recursion. But in addition, it has also a very special control structure called *call-with-current-continuation*.

Call-with-current-continuation (often abbreviated call/cc) allows to save the state of a computation, package it up as a data structure, go off and do something else. Whenever wanted, the old saved state can be restored, abandoning the current computation and come back where the saved computation left off.

This is far more powerful than normal procedure calling and returning, and allows to implement advanced control structures such as very complex forms of backtracking, cooperative multitasking, and custom exception-handling.

The objective of this section is to describe the concept of continuations with an implementation perspective. It is shown how the mechanisms that support tail recursion also support continuations. This description is fairly inspired from [Wilson97]. The implementation perspective provides a very interesting way to explain the concept of continuation. In [Wilson97] more details and illustrations can also be found.

D.1.1 Tail Recursion

In the implementation of most programming languages, an activation stack is used to implement procedure calling. At a call, the state of the "caller" (calling procedure) is saved on the stack, and then control is transferred to the callee.

Because each procedure call requires saving state on the stack, recursion is limited by the stack depth. In many systems, deep recursions cause stack overflow and program crashes, or use up unnecessary virtual memory swap space. In most systems, recursion is unnecessarily expensive in space and/or time. This limits the usefulness of recursion.

In *Scheme* , things are actually simpler. If the last thing a procedure does is to call another procedure, the caller doesn't save its own state on the stack. When the callee returns, it will return to

its caller's caller directly, rather than to its caller. After all, there's no reason to return to the caller if all the caller is going to do is pass the return value along to its caller. This optimizes away the unnecessary state saving and returning at tail calls.

In fact, elimination of tail calls applies transitively, and can be exercised on whole call chains. As a result, a procedure can return to the last caller that did a non-tail call. Note that this tail call optimization is a feature of the language, not just of some implementations. Any implementation of standard *Scheme* is required to support it, so that it can be counted on and portable programs that rely on it can be written.

The idea of tail call elimination comes from the simple distinction between two things that most languages lump together : saving the caller's state, and actually transferring control to the callee. *Scheme* notices that these things are distinct, and does not bother to do the former when only the latter is necessary. A procedure call is really rather like a (safe) goto that can pass arguments : control is transferred directly to the callee, and the caller has the option of saving its state beforehand. This is safer than unrestricted goto's, because when a procedure does return, it returns to the right ancestor in the dynamic calling pattern, just as if it had done a whole bunch of returns to get there.

D.1.2 The Continuation Chain

The Continuation Chain is a straightforward implementation model of *Scheme* 's state-saving for procedure calling[1]. In this model, the caller's state is saved in a record on the garbage-collected heap. This record is called a *partial continuation*. It is called a *continuation* because it says how to resume the caller when we return into it, i.e., how to continue the computation when control returns. It is qualified as a *partial* because that record, by itself, only tells how to resume the caller, not the caller's caller or the caller's caller's caller.

On the other hand, each partial continuation holds a pointer to the partial continuation for its caller, so a chain of continuations represents how to continue the whole computation : how to resume the caller, and when it returns, how to resume its caller, and so on until the computation is complete. This chain is therefore called a full continuation.

A special register called the continuation register `CONT` is used to hold the pointer to the partial continuation for the caller of the currently-executing procedure. When a procedure is called, the state of the caller is packaged up as a record on the heap (a partial continuation), and that partial continuation is pushed onto the chain of continuations hanging off the continuation register, as shown in figure D.1

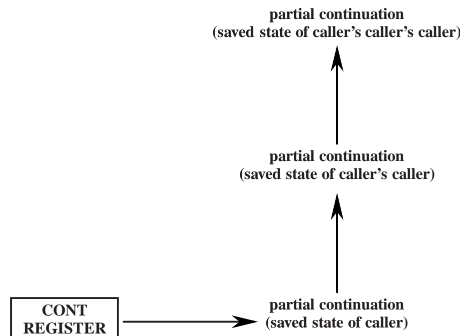


Figure D.1: Continuation Chain.

[1] Actually, some implementations of *Scheme* do use a relatively conventional stack, often so that they can compile *Scheme* straightforwardly to C. However, they must still provide tail-call optimization somehow. Implementation strategies exist to do so, but they are beyond our concerns.

Thus continuations are thus created and used nondestructively, and the continuations on the heap form a graph that reflects the pattern of non-tail procedure calls. Usually, that graph is just a tree, because of the tree-like pattern of call graphs, and the current "stack" of partial continuations is just the rightmost path through that graph, i.e., the path from the newest record all the way back to the root.

Most of the time, the rest of this graph becomes garbage quickly. Each continuation holds pointers back up the tree, but nothing holds pointers down the tree. Partial continuations therefore usually become garbage the first time they are returned through.

D.1.3 Capturing continuations (call/cc)

The fact that this graph is created on the heap allows to implement call-with-current-continuation or call/cc. Call/cc can capture the control state of a program at a particular point in its execution simply by pushing a partial continuation and saving a pointer to it in a live variable or data structure. Then that continuation chain remains live and is not garbage collected. That is, the current state of control have been "captured". Later, control to that point can "magically" return simply by restoring that continuation, instead of the one in the continuation register.

Note that, continuations in *Scheme* can be used multiple times. The essential idea is that rather than using a stack, which dictates a depth-first call graph, *Scheme* allows to view the call graph as an actual graph, which may contain cycles, even directed cycles (which represent bactracking).

When backtracking, the control flows back up and re-executes from some saved continuation. Backtracking (control) is a very direct application of continuations and call/cc.

D.1.4 Syntax of call/cc

The syntax of call/cc is not very self-explanatory. Call/cc is a procedure of exactly one argument, which is another procedure that is called just after the current continuation has been captured. The current continuation will be passed to that procedure, which can use it (and be aborted or not) as it pleases. The procedure given as argument to call/cc is qualified as the abortable procedure.

The captured continuation is itself packaged up as a procedure called the escape procedure, also of one argument. The only thing that can be done with a captured continuation is to resume it, simply by calling it as if it was a normal *Scheme* procedure. If and when the escape procedure is called, it restores the continuation captured at the point of call to call-with-current-continuation. This is called a nonlocal return. From the point of view of the caller of call/cc, it is the same as if call/cc had returned normally. In addition, the call of the escape procedure takes one parameter whose value defines the value returned by the call of call/cc itself. If the escape procedure is not called, the abortable procedure is not aborted and returns normally a value, in that case call/cc returns that value.

D.2 Utilization Perspective

This section presents a very simple illustrating how continuations can be used, i.e. how the control flow can be captured and then resumed. The procedure `set-cont-here` captures and returns a continuation. This is just like setting up a breakpoint.

```
scm> (define (set-cont-here)
                (call-with-current-continuation
                  (lambda (c) c)))
```

This procedure is used to capture three continuations stored in the variables `c1`, `c2` and `c3`. `c1` can be re-called back to redo the three `displays`, `c2` to redo the two last ones and `c3` to redo the last one.

```
scm> (define c1 #f)
scm> (define c2 #f)
scm> (define c3 #f)
scm> (begin ; this executes some display and sets up the continuations.
              (newline)
              (set! c1 (set-cont-here))
              (display "Hi 1!!!") (newline)
              (set! c2 (set-cont-here))
              (display "Hi 2!!!") (newline)
              (set! c3 (set-cont-here))
              (display "Hi 3!!!") (newline))
Hi 1!!!
Hi 2!!!
Hi 3!!!

;;; continuations are Scheme objects, we can build a list of continuations.

scm> (list c1 c2 c3)
(#<cont 316 @ 7d390> #<cont 316 @ 77b38> #<cont 316 @ 78070>)

;;; calling back continuation c1:
;;; It is important to pass itself as argument to the call, because it is this
;;; value that is returned by set-cont-here and assigned to c1 with set!.

scm> (c1 c1)
Hi 1!!!
Hi 2!!!
Hi 3!!!

;;; calling back continuation c2.
;;;
scm> (c2 c2)
Hi 2!!!
Hi 3!!!

;;; calling back the continuation c3.
;;;
scm> (c3 c3)
Hi 3!!!

;;; re-calling back the continuation c3. This shows that a continuation can be
;;; used several times.

scm> (c3 c3)
Hi 3!!!
```

Appendix E

Acronyms

ADBMS active database management system

AN access network

ASN.1 abstract syntax notation one

BEC behavior execution context

BEN behavior execution node

BET behavior execution tree

BL behavior language

BPE behavior propagation engine

BSF behavior scoping function

CMIP common management information protocol

CMIS common management information service

CORBA-DII CORBA dynamic invocation interface

CORBA-DSI CORBA dynamic skeleton interface

CORBA-IDL CORBA interface definition language

CORBA-IIOP CORBA internet inter-ORB protocol

CORBA common object request broker architecture

CVP-TEM CVP triggering event message

CVP computational viewpoint

DFS depth first search

DMI definition of management information, X.721

DOC distributed object computing

DOF distributed object framework

ECA event condition action

ETSI european telecommunication standardization institute

FDT formal description technique

FSM finite state machine

GDIO guidelines for the definition of information objects

GDMO Guidelines for the Definition of Managed Objects

GRM general relationship model

IR information repository

ITU-T international telecommunication union

IVP-TEM IVP triggering event message

IVP information viewpoint

KPN royal PTT Netherland

LOTOS language of temporal ordering specification

LTS labeled transition system

METRAN metropolitan european transport network

MSP Multiplex Section Protection

NMF network management forum

ODP open distributed processing

OF Object Framework

OMA Object Management Architecture

OMG object management group

ORB object request broker

OSI-SM OSI systems management

OSIMIS OSI management information services

OSI open system interconnection

PCO point of control and observation

R4RS revised^4 report *Scheme*

RBF relationships / role-based behavior formalization

RM-ODP reference model of ODP

SCFW state caching full walk

SCSSFW state-caching sleep-set full walk

SDH synchronous data hierarchy

SDL system description language

SLIB *Scheme* Library

SLSSFW state-less sleep-set full walk

SMF systems management function

TEC trigger execution control

TEM triggering event message

TIMS-SCS TIMS simple case study

TIMS TMN information model simulator

TINA Telecommunication Intelligent Network Architecture

TMN telecommunication management network

UML unified modeling language

VE validation environment

XOM-API / XMP-API X/Open object management API, X / Open management protocol API.

168

Appendix F

Summary of Case Studies performed in the TIMS Project

F.1 TIMS Simple Case Study

TIMS-SCS was a generic configuration management problem in the context of the access network (AN). This case study was the first benchmark used to exercise the first prototype of the TIMS tool set. It has been very useful to identify the key design choices of the TIMS tool-set. This concerns the use of roles and relationships, the declarative specification of actions model, the execution semantics, etc. More details about this case study are presented in [Sidou et al.95b].

F.2 SDH Multiplex Section Protection (MSP)

SDH Multiplex Section Protection (MSP) case study was performed by Rolf Eberhardt during its stay at KPN research. The objective was to evaluate TIMS concepts and tool-set on a significant case study. This case study was also the opportunity to get some useful feedback from KPN research experts in modeling of TMN interfaces. The results of the MSP case study have been published in an internal Swisscom report [Eberhardt et al.95]. The main conclusion of the case study is that TIMS is useful for prototyping, although not rapid. The process is not rapid just because modeling is in itself a difficult problem. Before being able to specify behaviors for a given system, it is first necessary to perform a complete analysis of the management information model, and to understand all complexity of the relationships involved. The key advantage of TIMS is that the confrontation with a tool forces people involved to understand fully the information model. The final report of the MSP case study also suggests that TIMS would be most useful and cost-effective if it could be used for multi-purposes, e.g. education and testing. Now that the TIMS tool set has been extended with graphical user interfaces it may be worth to try to use it for education purposes. On the other side testing extensions (link to test generation) has recently been envisioned in [Mazziotta97]. Another problem identified in this case study is related to the proposed behavior language notation. With repent to the learning curve, it is not more important and probably less important that what is required in most of the notations used usual formal description techniques. One problem is that, the behavior language notation is not standardized. This is a problem to publish TIMS behavior language specifications. However, a reasonable solution to this problem would be to build a post processing tool for specification parts of the behavior language (i.e. guards, pre- and post-conditions) to obtain a form suitable for publication, e.g. Z, or the behavior template notation proposed in ITU-T SG4 [G851 0196].

F.3 V5 Interface Management

V5 Interface Management case study was performed in Swisscom by Marco Randini and Rolf Eberhardt. The case study has considered configuration and provisioning for the V5.1 management model. This case study was developed in parallel to the V5 management interface specification which is to be used for procurement purposes. The case study focussed essentially on the overall behavior of the management architecture and less on the mapping between hardware behavior and its TMN representation (e.g. state mapping of protocol engine finite state machines). The developed consisted essentially of the V5.1 interface model for configuration management.

Besides the activity for the V5.1 specification, this case study gave to people in Swisscom the opportunity to better understand the prototyping methodology provided, what it looks like, and how far the specifier can take advantage from the use of the TIMS tool-set. The major effort (40%) was in acquiring the necessary V5 domain knowledge, sometimes down to the protocol level. Implementing the static schema (resource selection, GRM) then could be quickly done. On the other hand, embedding the fragment into an overall MIB architecture turned out to be more difficult than expected (20%), mainly due to functional restrictions in current GDMO libraries (M.3100). Developing behavior itself resulted in a repeated review and refinement of the requirements, sometimes identifying new demands along the way. Behavior-design and debugging made up for another 40% of the effort.

The V5 management interface project running in parallel benefited a lot from the design efforts in the case study, especially when it came to understanding the finer points of the model, its restrictions and pitfalls. The TIMS scenarios could be mapped almost 1:1 into the Ensemble specification that was produced from the case study and that is in the standardization process to become an ETSI standard [Etsi97]. Results on this case study have also been published in an internal Swisscom report [Eberhardt et al.97a]). The V5 case study and the Ensemble document are summarized in appendix G. This chapter gives some credit on the usefulness and effectiveness of the tool-set resulting from the approach, proposed in the thesis, for behavior modeling and validation. This proof of concept is given as an appendix just because the case study itself was not developed by the author of the thesis. It was developed by Marco Randini and Rolf Eberhardt in Swisscom. The fact that the case study is performed by people external to the development of the tool-set gives in some way more credit with respect to the applicability of the tool-set.

F.4 Metropolitan European TRAnsport Network (METRAN)

METRAN (Metropolitan European TRAnsport Network) case study was suggested by Rolf Eberhardt and developed by Sandro Mazziotta at Eurécom, to provide a significant and rather complete benchmark for the TIMS tool set, in particular w.r.t. proposed extensions to the TIMS tool set such as improved validation and test generation. The METRAN case study is a result of the Eurescom project P408. So paper specifications of the case study are already available. The case study itself is concerned with the specification X interface between public network operators (PNO) to solve two specific problems (i) path management and (ii) fault management in the context of an SDH transport network. The main conclusion of the processing of this case study with the TIMS tool-set is that the behavior language revealed enough powerfull to specify the complex algorithm for path reservation and activation that are specified in prose in P408 deliverables. Concerning the test generation aspects, tests cases produced automatically by TIMS are quasi equivalent to those written manually in P408 deliverables. The difference resides in tests including high level semantical verifications. Such tests are incorporated by hand in P408 and can not be easily derived automatically with TIMS. On the other hand, when the complexity grows, test cases are not written at all in P408 while with TIMS, this complexity is reflected by a longer time to produce the test cases with the TIMS tool-set. This

case study was developed in roughly two months. As usual most of the effort had to be devoted to the understanding of the model, then the development with TIMS was made in less than one month.

172

Appendix G

Summary of the V5.1 Case Study and Ensemble

G.1 Introduction

This chapter gives some credit on the usefulness and effectiveness of the tool-set resulting from the approach, proposed in the thesis, for behavior modeling and validation. This proof of concept is given as an appendix just because the case study itself was not developed by the author of the thesis. It was developed by Marco Randini and Rolf Eberhardt in Swisscom. The fact that the case study is performed by people external to the development of the tool-set gives in some way more credit with respect to the applicability of the tool-set.

The case study has considered configuration and provisioning for the V5.1 management model. This case study was developed in parallel to the V5 management interface specification which is to be used for procurement purposes. The case study focussed essentially on the overall behavior of the management architecture and less on the mapping between hardware behavior and its TMN representation (e.g. state mapping of protocol engine finite state machines). The development consisted essentially of the V5.1 interface model for configuration management.

This chapter gives a summary of the V5.1 management Ensemble resulting from this case study. Section G.2 to G.7 reflect faithfully the first sections of the Ensemble document itself, with some details removed (such as managed object conformance statements (MOCS)).

Section G.8 ilustrates the development of a sequence of scenarios with the TIMS tool-set. It is shown how the output, i.e. the *Scheme* trace and system monitoring views available in the TIMS tool-set allows the specifier to check that scenarios work properly. In addition, the scenarion management commands, the *Scheme* trace and pre- / post-conditions can be reused altogether as a solid basis to produce the final text of the Ensemble document.

G.2 Scope of the Ensemble

The ETSI technical standard (ETS) [Etsi97] specifies an Ensemble for the connectivity service provisioning management of Access Networks.

Management Domain : The management domain regarded in the ETS is depicted in Figure G.3 and covers

1. the provisioning of an access network transport bearer services by a Service Provider

2. the resourcing of access network resources required for such a service

3. the coordination between Access Network and Service Node network element configuration

Management Services : The ETS covers management operations necessary for a transmission-technology independent management of narrow-band transport bearer services (pstn, isdn, leased line) across V5 and 2 MBit/s Service Port Functions. This includes the configuration and provisioning of both transport bearer services and their affected resources. Efforts are undertaken to design the management services applicable for narrow- and broadband.

Resources : The management of User Port Functions and Service Port Functions providing User Network Interface and Service Node Interface functionality, respectively, are considered in the ETS. Transmission specific resources lie outside its scope.

Information Models : All management information models used in this Ensemble derive from existing libraries (ETSI or ITU-T). In cases where no satisfactory solution is available, the standard raises this as a work item and provides a temporary, non-normative proposal including solutions developed by industry fora's.

Open Network Provisioning : Information flows are designed as ONP-enabled, i.e. applicable both for management operations within a Telecom Operator and between Telecom Operators (e.g. between Access Network Operators and Service Providers).

G.3 Management Context

G.3.1 Functional architecture

ITU-T Draft G.902 "Framework Recommendation on functional Access Networks" forms the basis for a functional access network architecture. Figure G.1 describes the network boundaries of an Access Network (AN), adapted to the case of separate organizational entities. The figure describes the Service Node (SN) as the entity generating the service transported across the access network to the service user. The Service Node Interface (SNI) is the interface between AN and SN. The User Network Interface (UNI) is located between the Access Network and the Service User. The Q3-interface is the interface between the management function of the Access Network and its Telecommunication Management Network (TMN). Information between different TMN's is exchanged across the X-interface.

The network itself consists of several functions shown in Figure G.2. The User Port Function (UPF) adapts the specific UNI requirements to the core and management functions. The Service Port Function (SPF) adapts the requirements defined for a specific SNI to the common bearers for handling in the core function and selects the relevant information for treatment in the AN system management function. The core, transport and management functions are responsible for AN operations. Table G.1 lists examples of User and Service Port Functions.

The information flow across the x-reference point is concerned with OA&M of the User Port and Service Port Function. The Core and Transport functions are internal to the Access Network management.

UNI's may be shared between Service Nodes. This implies the definition of logical UPF which are multiplexed into a single Transmission Media Layer Termination and demultiplexed in an Adaptation Function outside the AN. Access Network management must take shared-UNI's into account.

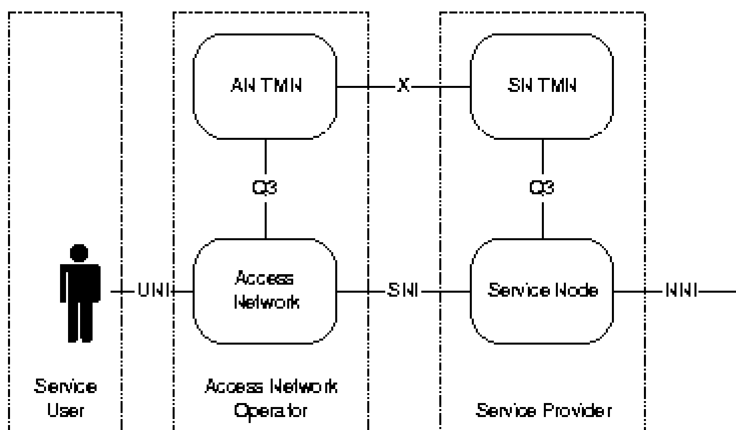


Figure G.1: Access Network boundaries ([G.902 fig 1])

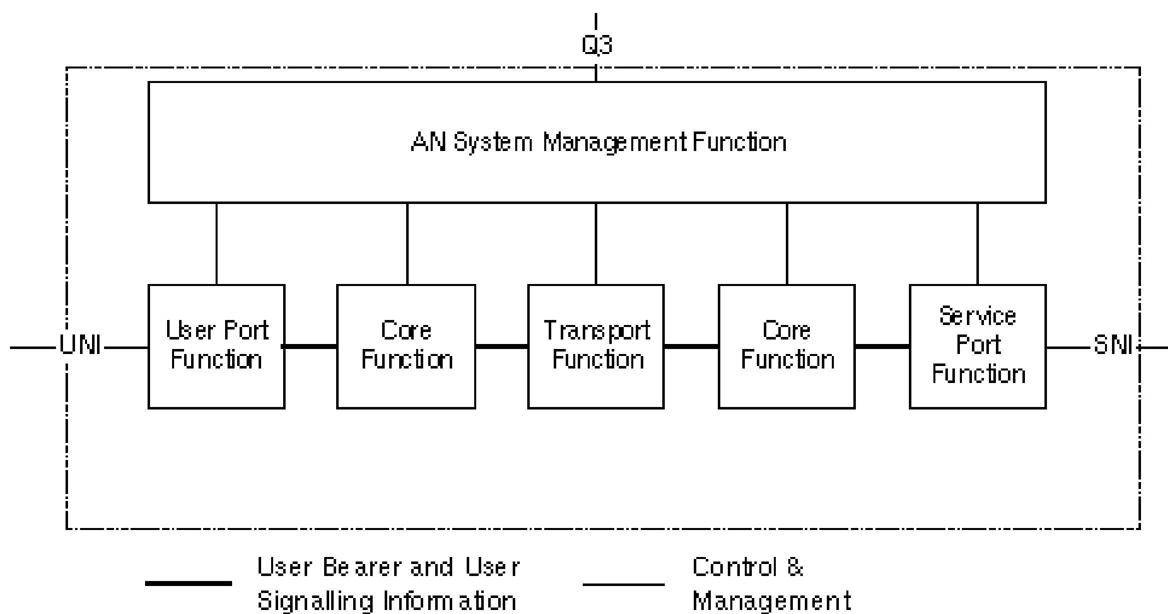


Figure G.2: Example of functional architecture of an AN ([G.902/fig.3])

Service Port Function	User Port Function
Termination of the SNI functions, mapping of the bearer requirements and time critical management and operational requirements into the core function, mapping of protocols if required for particular SNI, testing of SNI, maintenance of SPF, management & control functions	Termination of the UNI functions, A/D conversion, signalling conversion, activation/deactivation of UNI, handling of the UNI bearer channels/capabilities, testing of the UNI, maintenance of the UPF, management & control functions

Table G.1: Examples of SPF and UPF ([G.902/p.11])

A UNI may support several User Port Functions at the same time. This concept is called a ”shared UNI” and is typically found in broadband network applications. The User Network Interface and the User Port Functions therefore constitute two separate resources and must be managed separately.

G.3.2 Transport bearer service, resource and management requirements

A transport bearer service (TBS) provides the facilities for the information transfer of a transport bearer service across the access network.

The access network provides transport bearer services to service providers and network operators. The transport bearer service is the connectivity between the User Port Function and the Service Port Function. The connectivity (the bearer transport) is associated with specific user network and service port interfaces. The connection requires information on the traffic it supports to ensure its quality of service (e.g. jitter, wander, timing delays). Transport bearer services may be augmented by additional capabilities. Capabilities are divided in service-specific and service-independent capabilities.

G.3.3 Functional TMN Architecture and Management Domain

Figure G.3 provides a coarse overview of the management domain relevant for access network management consisting of following functional units:

Access network and service node Network Element Functions (NEF) are connected by the v reference point. For the purpose of this discussion each NEF is subdivided into a ”AN/SN-specific aspect” and a ”common aspect”. The latter provides functionalities required for the co-ordination between the Service Provider’s NEF and the Access Network’s NEF.

EAN-OSF/QAF, NAN-OSF, SAN-OSF and ESN-OSF, NSN-OSF and SSN-OSF provide element, network and service management of the access network and the service node, respectively. Three reference points provide for the information exchange between the TMN of the Access Network and the Service Node. The a3 reference point allows the management of bearer transport services in the AN. The a2 reference point enables the Service Provider to request (in a restricted manner) access network resources. Both a2 and a3, provide for a simple resource co-ordination information exchange. The Service Provider’s OSF’s act as managers. Table G.2 describes all reference points and possible management services in detail.

The management domain necessary for interworking covers all OSF’s listed above as well as those aspects of the Network Element Functions which are common to both the AN and the SN. AN-specific aspects (such as transmission management) lie outside the scope of the management domain. The same applies for SN-specific aspects such as customer administration or SN-fault management.

As described in G.902 (c.f. Figure G.2) the AN may include a transport function. This function may be implemented in several technologies (e.g. SDH or ATM). If transmission handling issues were considered within the scope of this document, Figure G.3 would also have included a technology specific NTransmission-OSF for this purpose. In this case, the EAN-OSF would be connected to the technology-specific transmission NTransmission-OSF via a a1-like reference point, providing management services specific to the management of the transmission part of the AN. The technology specific NTransmission-OSF would also provide to the upper layer OSFs connectivity information about access points in the transmission network that are used on the AN and the SN side, respectively.

G.3.3.1 Management architecture

Figure G.4 provides a complete overview of the interaction between the Access Network Operator’s TMN and the Service Provider’s TMN. It describes the major ”actors” involved in such an environment. On the AN-side the EAN-OSF/QAF manages proprietary network element functions and

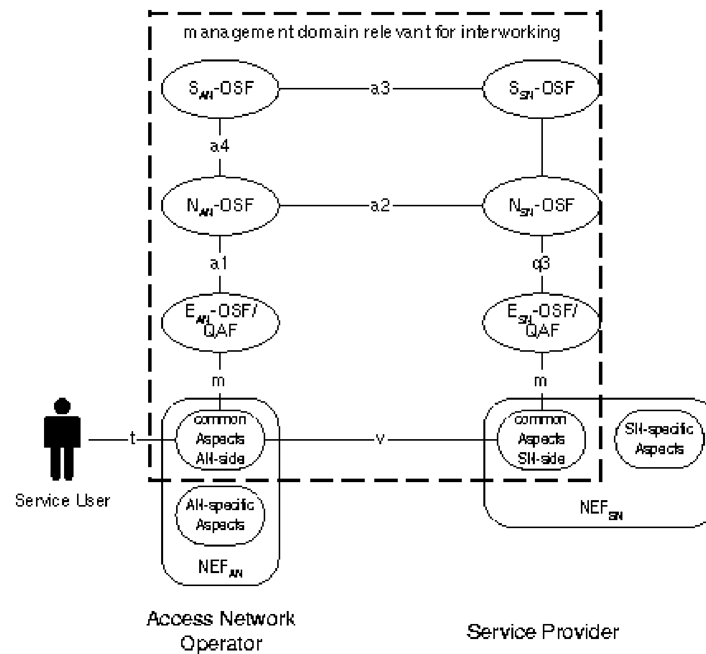


Figure G.3: Interworking management domain and reference points covered in this model

provides standardized reference points where relevant (a1). The NAN-OSF supports transmission-technology transparent network management functions and provides appropriate management services to the SAN-OSF (a2, a4). The a2 reference point provides a restricted network level view of the access network to the Service Provider. The SAN-OSF is responsible for service management of the Access Network and provides reference points to the Service Provider's SSN-OSF (a3). The SSN-OSF and its lower-layer OSF's manage all aspects relevant to the ”service user access” part of its services.

Both the NAN-OSF and SAN-OSF of the Access Network will rely on non-TMN information systems such as legacy cable management, testing, topological databases, work flow management, etc. Access to these actors is provided through non-standardized m1 and m2 reference points. Note, that in a larger Telecom Operator Service Provider OSF's, too, will entertain access to the same support systems. Table G.2 lists possible management services which can be made available through the reference points. Figure G.3 provides an overview of the functional TMN-model and its major reference points.

The lower part of Figure G.4 describes the functional architecture for V5. AN-specific part of the model is the management of the ”Access Network” subnetwork. The second subnetwork visible in the diagram, the ”digital section”, is considered outside the scope of an Access Network Operator. The figure also shows the multiplexing structure of the communication paths as well as the allocation functions mentioned previously.

Three actors are involved in the Access Network, ”the Service User”, the Access Network Operator and the Service Provider. Figure G.4 identifies two boundaries between these actors: a service boundary located vertically in the diagram and a management boundary, found horizontally. Two functions are visible on the service boundary, the user port function and the service port function. On the management boundary the a2 and a3 define the management information flow between Access Network Operator and Service Provider, while the muser provides for the interaction between Service User and Service Provider. The latter is out of the scope of this discussion.

Table G.2 : Management reference points

Ref Point	(incomplete list of) management services available at the reference point	Example
a1	o transport bearer service resource configuration o transport bearer connectivity service provisioning / testing / fault management o transport bearer service performance management o equipment management	configuration of the common aspects, e.g. V5.1 interfaces, instantiation of userPorts slot management
a2	o transport bearer service resource configuration o resource provisioning	V5.1 configuration based on requirements specified by the Service Provider V5 interface request covering a certain area
a3	o transport bearer Service Configuration & Provisioning, ordering o trouble administration o QoS performance reporting o test administration o link management o accounting & billing	order a transport bearer service with certain capabilities configuration of UPF and SPF resources as well as the exchange of link connection information billing information pre-provisioning.
a4	o topology management, o resource provisioning, o subnetwork protection management o network fault management, -performance management o resource management	o domain management o UNI-selection for a specified location o pre-provisioning of resources o network requirements evaluation and planning o alarm collection for the entire AN o resource life-cycle management
m0	o non-TMN reference point providing access to the NEF.	All proprietary NEF management functions
m1	o access to non-TMN management and database services: o cable information o work-flow information o resource information o planning	identify access-point of a cable provision technicians for repair effort issue long-term planning request
m2	o cf. m1	-
muser	o reference point between service user and Service Provider: connectivity service provisioning, trouble administration, billing, Quality of Service, Billing& Accounting	e.g. Customer Trouble Handling
fS	o functions available for the transport bearer service management through AN personnel, possibly regional S/NMC	trouble tickets handling, QoS evaluation of the overall AN network,.
fN	o functions typically available at a regional NMC	network alarms, database update of new resources in the network("churn" handling)
fE	o functions available through the LCT (local craft terminal) or the element manager, typically product specific	-

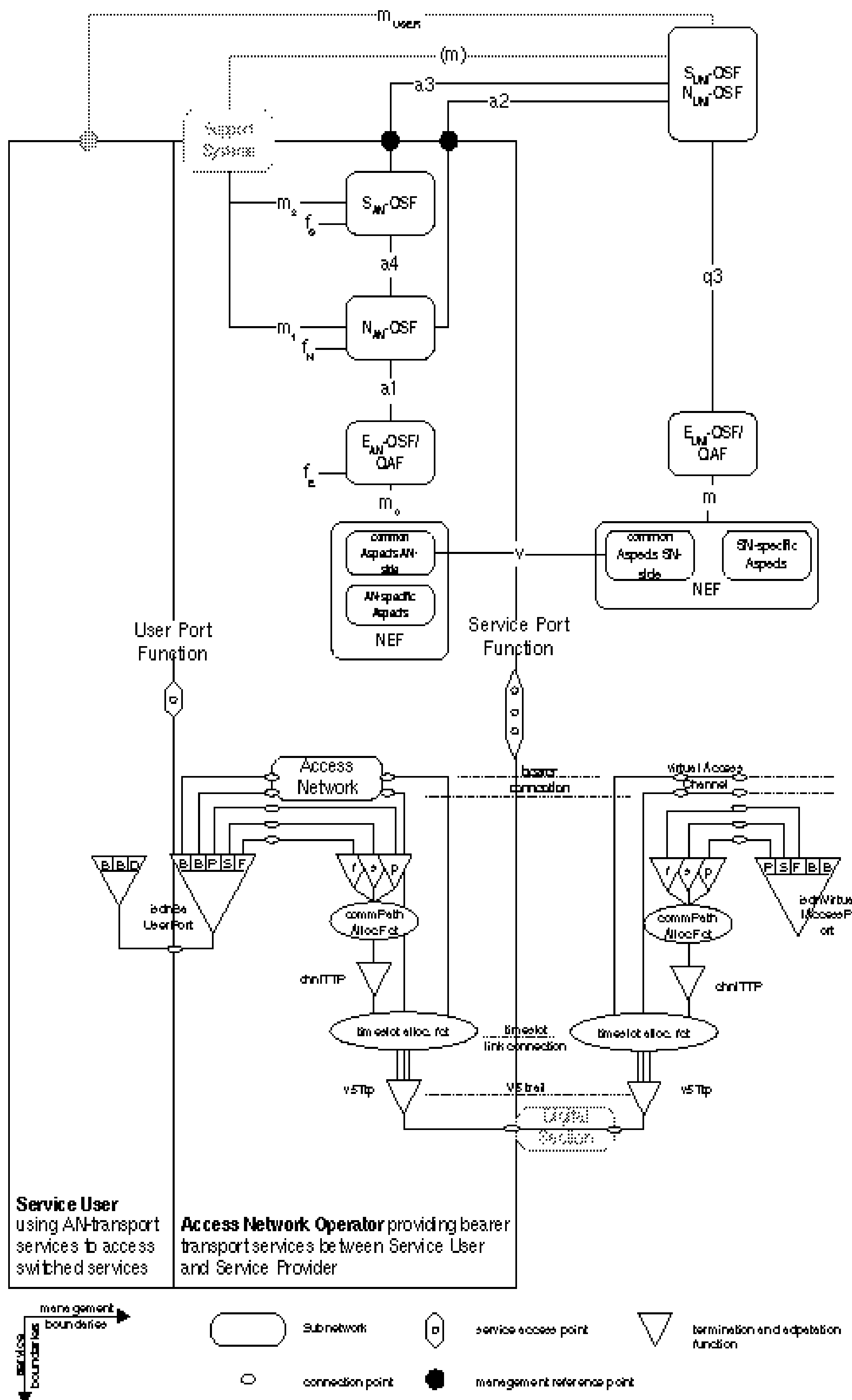


Figure G.4: TMN Architecture for the Access Network with the example of isdnBA

G.3.3.2 Co-ordination

Co-ordination between the Service Provider and the Access Network Operator enables the interworking between the common parts of the Access Network with those of the Service Node. Co-ordination involves both network elements (e.g. configuration management) as well as connectivity service provisioning (e.g. common provisioning time).

Co-ordination information may be exchanged either on a resource basis using the resource-id or on a connectivity service basis (using the service-id). The latter is the preferred method of operation as it provides for a sufficient method of abstraction and keeps the a3 reference point simple and efficient.

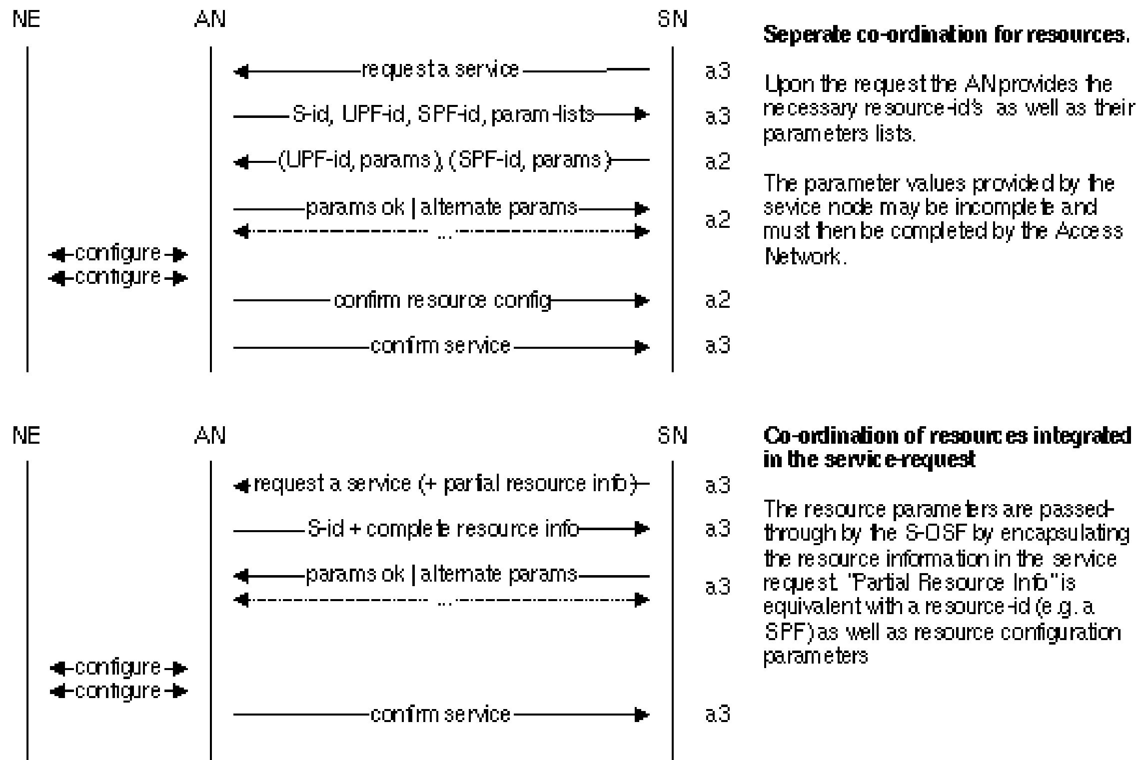


Figure G.5: Interaction Models for co-ordination (both service-id and resource-id based)

G.3.3.3 Management view and level of abstraction

Figure G.6 shows an example of how higher level systems will interact with the Reference points defined in this standard. This example is given to illustrate the use of this ensemble and there is no attempt to standardize the functions at the higher layers. It assumes that the V5 co-ordination function takes place in the NAN-OSF. This needs not always be the case.

The following functions take place in this example :

1. An order for a Service is received from a customer.
2. Interaction takes place with Customer management in order to discover the customers details. This may include his location.
3. The request is passed to a Service Design function that will interact with other systems in order to provide the End-to-End solution.

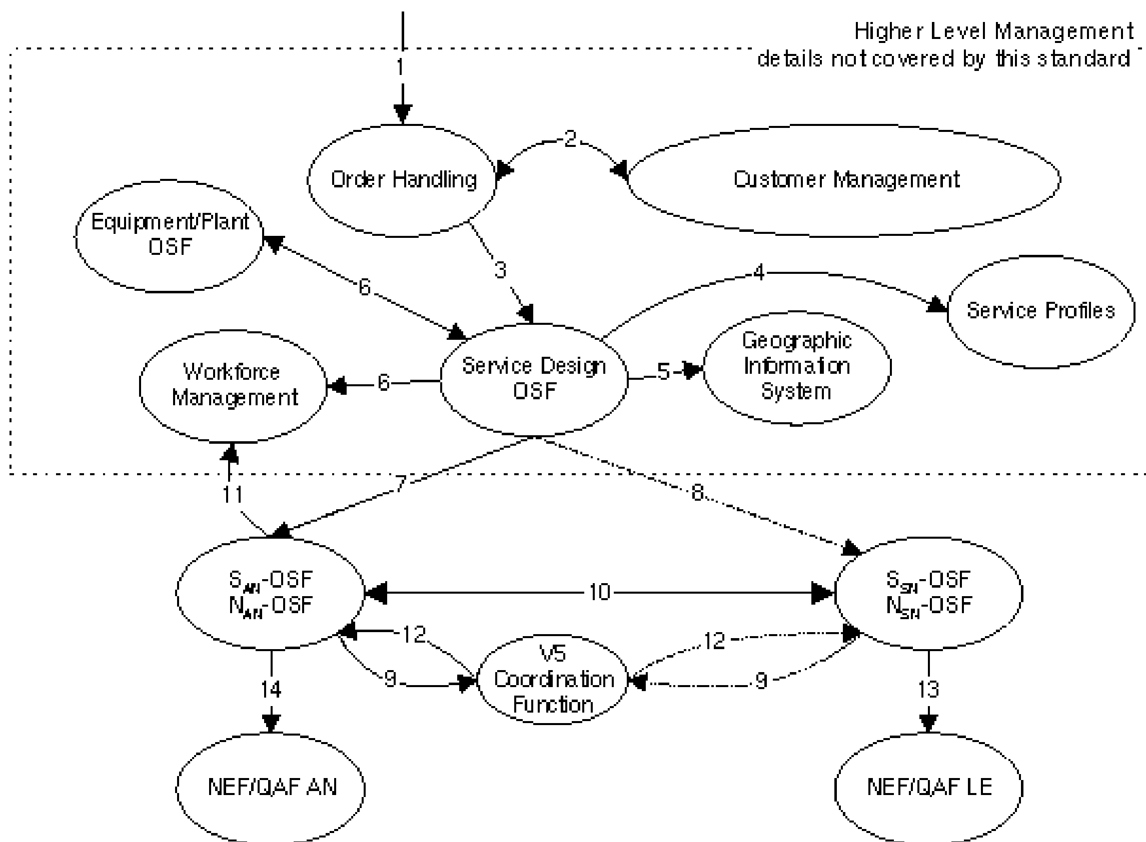


Figure G.6: Example process for connectivity service provisioning

4. Information about Service profiles is accessed in order to determine what is required to deliver this service to the customer.
5. Service Design accesses Geographic Information in order to determine the nearest node to the customer.
6. Copper plant information is accessed to determine if there is an existing drop to the customer and if so, where this is connected.
7. A Service Provider request is made to the Access Network manager.
8. A Service Provider request is made to the Service Node manager.
9. The Access Manager allocates resources for the Service and reports its configuration to the Coordination Function.
10. The co-ordination function (in this case located within the NAN-OSF) requests a configuration of the Service Node manager. The Service Node replies with the configuration parameters.
11. A Work-order is emitted in order to provide details of work required to provide the service.
12. The Coordination function configures any additional parameters on the Access Network, for instance Layer3Addresses, EnvelopeFunctionAddress etc.
13. The Service Node is configured.
14. When the equipment is installed and connected the Access Network is configured.

This Ensemble deals with the following interactions : 7,9, 10, 11,12,14.

G.4 Models

G.4.1 Resource model

The resource model of an Access Networks covers following resources :

- logical resources : describing the logical network and transport bearer service structure
- topological resources : describing the topological layout of the access network, i.e. the location of access points
- physical resources : describing the equipment and their connectability
- the service profiles : describing the transport bearer service, its service profile and the relates to the resources which are used for this service
- digital section resources: describing the transmission network connecting access network to the service node
- transmission network resources : describing the transmission network representing the access network

Figure G.7 illustrates the resources that are of interest to the Connectivity service provisioning Ensemble.

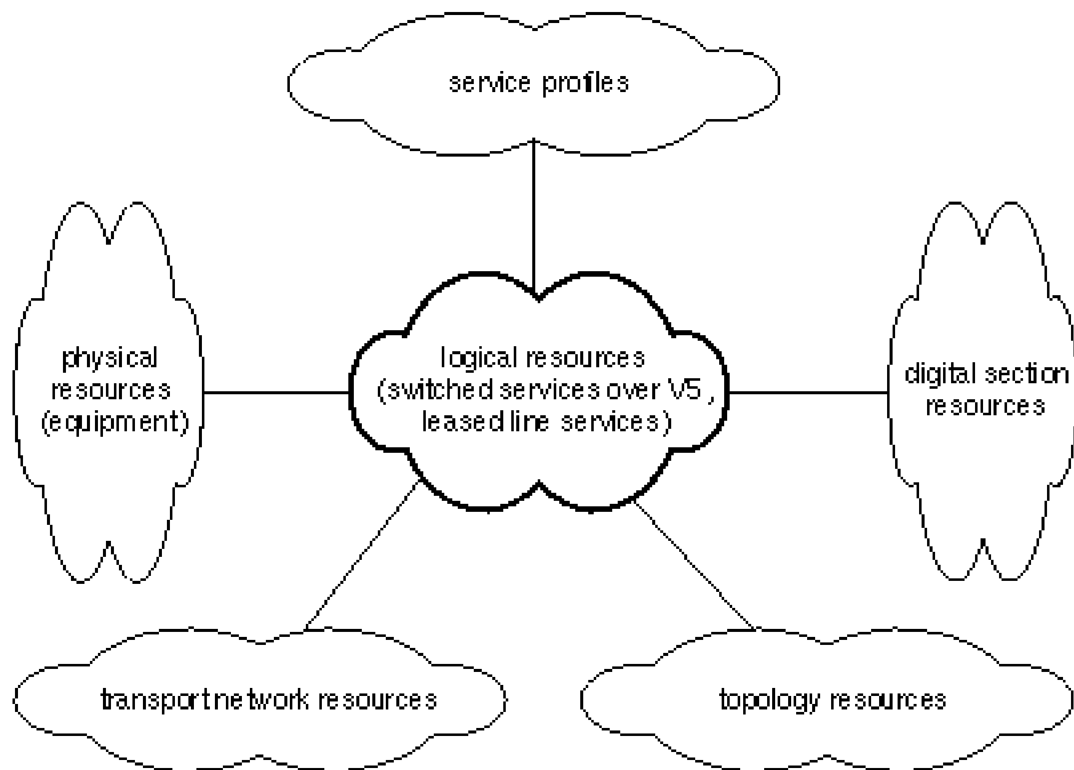


Figure G.7: Resources relevant for the Connectivity service provisioning Ensemble

G.4.2 Topological model

The topological model describes the Access Network as seen from the Service Provider. It identifies the minimal information which must be visible. Based on the functional model described in G.902, naming conventions must exist for the Service User Access Points, Service Node Interface and the Point Of Interconnect.

Access Point The Access Point describes the location to where a service is requested and is necessary for service invocation. A unique naming scheme for the Access Point is therefore necessary. The Access Point is not identical with the UNI. An example would be a request for a bearer transport service to an Access Point which isn't accessible yet (i.e. has no UNI).Access Points are best identified by their geographical location, i.e. their address. Note too, that the Access Point need not necessarily be the location where the Service User receives the service. An access point may be a socket (typical example for a home owner connection) or the connection point to a PABX or in-house network outside the responsibility of the Access Network Operator (situation of a small businesses connection).

Service Node Interface The Service Node Interface is the physical interface to a Service Port which in turn is the physical resource providing Service Port Functions. The Service Node Interface may itself consist of one or more Service Port Functions. The access network connects one or more Service Node Interfaces with a set of Access Points. The set of Access Points is called the bearer domain.

Point of Interconnection Connection between Access Network Operator and Service Providers require a common identifier of this connection point, called the Point of Interconnect (POI). By providing the POI to the Service Provider, the Access Network Operator makes available the location where the physical connection between the Access Network and a Service Node can be achieved. A typical example of the POI is the physical connector on the Main Distribution Frame where the cable to the Service Port is attached. The POI is usually used in international networks to designate cross-border links. Consequently, the POI name must be known by both the Access Network Operator and the Service Provider. POI's are typically named using the ITU-T M.1400 naming scheme (c.f. [M.1400 Table 1]).

Format of designation	SNI AN-side (Town A)	/	Suffix	-	SNI SN-side (Town B)	/	Suffix		Function Code	Serial number
Signs	Characters	Slash	Letters/ digits	Hyphen	Characters	Slash	Letters/ digits	Space	Letters/ digits	Digits
Number of characters	<=12	1	<=3	1	<=12	1	<=3	1	<=6	<=4
										No Space

Table G.3: General format of layer 1 for the designation of international routes

In Figure G.8 a simple FITL-based access network consisting of a single OLT and ONU is drawn. It shows the Access Point, the Service Node Interface and the Point Of Interconnect. The figure displays that several "logical ports" (UPF or SPF) may be made available across the same Service

Node Interface and Access Point, respectively. The Point Of Interconnect is very similar to the Service Node Interface. While the SNI gives port location, the POI defines the location where the interconnect is possible.

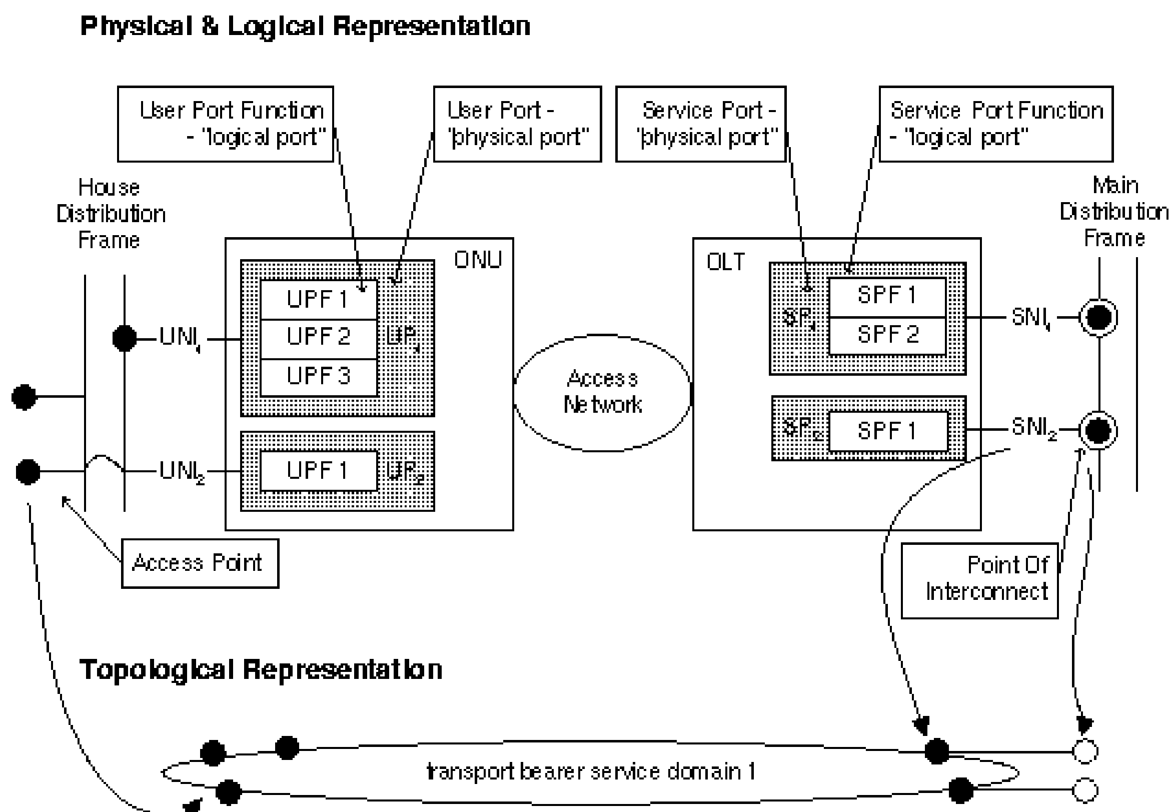


Figure G.8: Sample mapping of the physical/logical representation of a FITL access network onto the topological model

G.5 Functions

The following management services for provisioning are expected:

G.5.1 Resource Configuration & Provisioning Management (a2, a4)

Management Function	Summary
Request a Resource	The Service Provider requests a User Port Function / Service Port Function resource. The resource is identified by the transport bearer service domain it can cover and by its type.
Configure Resource parameters	The management function enables the Service Provider to configure resources-specific parameters. It covers UPF/SPF and management resources.
View Resource Setup	provides a view of the current resource-configuration
Release a Resource	The Service Provider requests the release of resources and become

	re-allocatable to other Service Providers.
Connect Service Port Function to Point Of Interconnect	The Service Provider requests for the connection between SPF and a POI. The result of this service gives the SPF the capability of transferring information from SPF to SNF and visa versa.
Disconnect SPF-resource from Point Of Interconnect	The Service Provider requests for the disconnection of a SPF from a POI, i.e. the service node.
Connect UPF to Access Point	The Service Provider requests that a User Port Function is connected to a Service User location, specified through its Access Point.
Disconnect UPF from Access Point	The Service Provider requests that a User Port Function is disconnected from a Service User location.
Request change of resource capabilities	The Service Provider requests the addition or removal of resource capabilities, e.g. upgrade/downgrade or migration, to conform with new requirements.

G.5.2 Service Configuration & Provisioning Management (a3)

Management Function	Summary
Request a Bearer-Transport Service	The Service Provider requests that a service, specified through the service description, is made available at a certain access point location to a service port function. Connection between UPF and SPF is performed automatically. The Service may be requested with following restrictions: - use of specific UPF instead of one automatically assigned by the Access Network Operator - use of specific service node instead of service port function
Delete a Bearer-Transport Service	The Service Provider requests the termination of a service. The resources become re-allocatable to other services but are assigned to the Service Provider. The resources are released completely if their assignment had occurred automatically through the Access Network Operator.
Identify service capabilities	The service returns the set of capabilities a bearer-transport service has.
Add service capability to a Bearer-Transport Service	A service capability is added to an existing bearer transport service instance.
Remove service capability from a Bearer-Transport Service	A service capability is removed from an existing bearer-transport service instance.
Block/unblock a bearer transport service	During maintenance work such as the upgrade of equipment in the Access Network the Access Network Operator must be able to temporarily block the service in the Access Network. The Access Network Operator may request permission to block from the Service Provider.

G.5.3 Configuration & Provisioning at the resource level (a1)

G.5.3.1 Q3-V5 AN

Management Function	Description
Insert / Delete / Modify / Read a user port	- assign port address; - assign port type; - assign port specific parameters
Insert a V5 interface	Add a V5.1 interface: - V5.1 interface ID; - time slots for communication and for bearer channels; - protocol version; - provisioning variant. Add a V5.2 interface: - V5.2 interface ID; - associated 2 Mbit/s link(s); - time slot(s) for C-channels; - protocol version; - provisioning variant. Augment a V5.2 interface : Add 2 Mbit/s link(s) to the existing V5.2 interface and provide relevant data:- associated 2 Mbit/s link(s); - time slot(s) for C-channels; - provisioning variant.
Delete a V5 interface	V5.1 : Remove a V5.1 interface and delete relevant data mentioned V5.2 : Remove a V5.2 interface ID and delete the relevant data
Modify / Read a V5 interface	Modify one or more information elements
Reducing a V5.2 interface	Remove 2 Mbit/s link(s) from a V5.2 interface and delete the relevant data
Upgrade a V5.1 to a V5.2 interface	The upgrade is performed by deleting the affected V5.1 interface and inserting a V5.2 interface using the relevant data having been assigned to the V5.1 interface
Establish a connection	- assign access port to V5 interface, including V5 port address; - assign port bearer channel to V5 bearer channel; - assign PSTN signaling to V5 communication channel; - assign ISDN Ds-type data to V5 communication channel; - assign ISDN p-type data to V5 communication channel; - assign ISDN f-type data to V5 communication channel
De-establish / Modify / Read a connection	as above

G.5.3.2 Leased Lines

Management Function	Description
Insert / Delete / Modify / Read a user/service port	- assign port address; - assign port type; - assign port specific parameters
Augment / reduce the bandwidth of a connection	Add or remove time-slots from a leased line transport bearer service
Establish a connection	
De-establish / Modify / Read a connection	

G.6 Management Information Model

G.6.1 Overview of the Management Information Model

The management information model describes all information objects needed for the successful operation of the management services listed in this Ensemble. The management information model is independent of the location of the information in OSF's. The information model is structured in fragments. The fragments themselves are derived from various GDMO libraries, such as V5, the AN network level view or the equipment. The overview lists the relationships which describe the cohesion between the fragments and provide an overall information model. Figure G.9 identifies several management information model fragments. Relationships within the fragments are not depicted. The fragments are:

Transport bearer service Fragment(s) provide a functional representation of the transport service offered by the access network. Examples are: Leased Lines, V5-based PSTN/ISDN, non-V5-based PSTN/ISDN, etc. It is the core of the information model.

Topology Fragment takes care of the access network layout, the location of interfaces and user port and service port functions.

Equipment & Cabling Fragment handles the management of equipment structures and their physical connectivity.

Digital Section Fragment provides the service node interface with an association to a digital section.

Service Fragment handles the overall management of a service, i.e. the service identifier, the resources used to provide the service, etc. It also handles the management of the Service Providers making use of the Access Network including which service provider is responsible .

Transport Access Network Fragment manages the overall bandwidth of the access network. It is associated with the topology and the bearer transport subnetworks.

Not depicted is the technology-specific management of access network technologies such as FITL/PON or SDH as it isn't relevant for connectivity service provisioning.

G.6.1.1 Major relationships

Following relationships connecting fragments are identified :

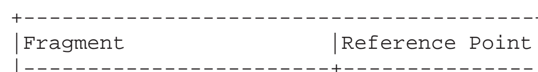
Owner identifies which service provider has "ownership" for which access network resource (user ports and/or service ports). **ServiceResource** associates services with the resources. **Interface** provides the association between a digital section fragment (which provides the "Point Of Interconnection" into the access network) and the transport bearer service fragments. The **resourceLocation** relationship aids the TMN in locating network functionalities in respect to the network topology and the underlying transport access network. Similarly, the **connectivity** associates a subnetwork (in V5 identified as a V5interface, in Leased Lines as a subnetwork) with its topological representation. The **transmission** relationship associates the transport bearer service resources with resources in the transmission network(s). Equipment (port-#, racks, cards, cabling, etc) is mapped to the functional models (transport bearer service fragments) through the resource relationship. The **equipmentLocation** relationship locates equipment within the topology of the access network.

In the following E-R diagrams, a diamond represents a relationship and denoted lines their relationship roles. Boxes correspond to information objects and shaded boxes are groups of objects, called fragments.

Figure G.9: Information Model overview

G.6.1.2 Mapping to reference points

This chapter describes which fragment may be visible at which reference point. It also shows which information would be shared between Access Network Operators and Service Providers.



```
+--------------------------------------+
|Fragment              |Reference Point|
|----------------------+---------------|
```

Service	a3
Transport bearer service	a1, a2 a4
Topology	a1, a4
Transport Access Network	m, (a1)
Equipment & Cabling	a1, a4
Digital Section	a1

With respect to [G.851-01] this mapping corresponds to the initial requirements on computational objects. Further modeling of a computational viewpoint may be carried on from this point.

G.6.2 Service fragment

Due to the general immaturity of work in this field, version 1 of this Ensemble does not propose a service fragment. Possible candidates are the NMF Ordering Model or the X.160 Customer Network Management Model.

G.6.3 Topology fragment

The topology fragment provides the framework in which the other fragments are located. In addition it provides the model for static blocking between ports within a bearer domain. Resources (both logical and physical) are associated to a managed element. A network consists of one or more managed elements(as may be the case in an Active Optical Network - AON). The network object provides the connectivity between the managed elements. The minimal structure for the topology management is depicted in Figure G.10.

The equipment object which is in the physical role of the resourceLocation relationship provides the information on the actual node-location of the resources. Through a fine-grained modeling of the physical resources (see equipment fragment) additional details on the topology may be provided.

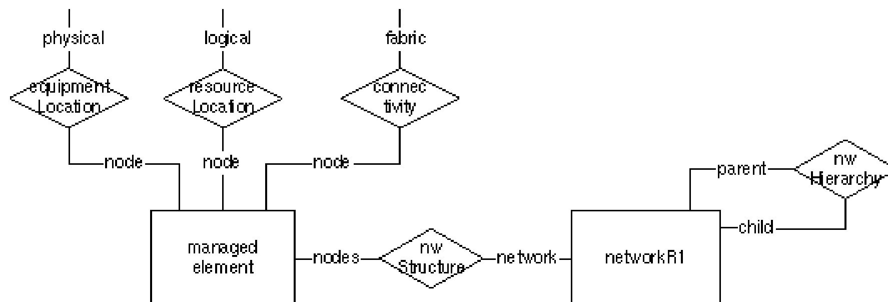


Figure G.10: Topological fragment

The topology shall reflect the topological connectivity (i.e. static blocking) between the UPF and SPF, independent of bandwidth details. The minimal instantiation of a network topology consists of one networkR1 object and one managed element.

G.6.4 Equipment & cabling fragment

Equipment management itself lies outside the scope of this Ensemble. A minimal equipment management model is necessary, however, to provide for the proper configuration and provisioning processes. The equipment model must also take the management of connection points into account should the manager wish to store such information (see topology fragment). Dependencies within the equipment model (e.g. between line cards and their power supply) are not considered here because they are irrelevant for configuration management purposes.

Figure G.11 describes the managed objects and relationships required to cover the minimal requirements.

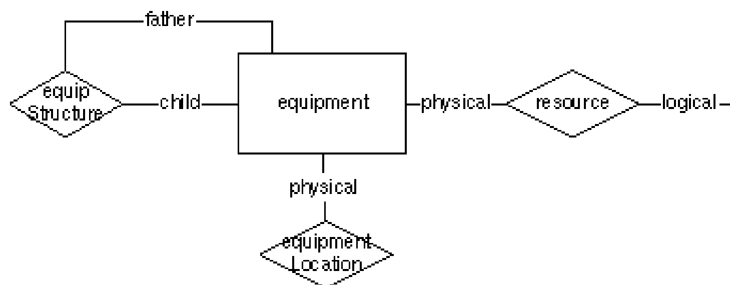


Figure G.11: Equipment fragment

The resource connects logical resources and a physical resource.

The minimal number of instantiations for the equipment fragment is 1, e.g. for the modeling of an multiplexer. In those cases, where the equipment structure isn't modeled, the logical resources may take over this functionality (e.g. in the CMIS/P case by using the RDN as an identifier of the equipment port to which this logical resource is mapped to).

G.6.5 Transport access network fragment

The transport access network fragment describes the transmission aspects of the one or more subnetworks compromising the access network. Its internal structure is outside the scope of this ensemble. Relationships between the transport bearer service fragment and the transport access network fragment describe the mapping of logical resources to transmission resources. The connectivity across the access network (e.g. the relationships R4/R5/R9/R10 in the V5 model) are not explicitly mapped. It is assumed that the implementation of the connection-setup between userport and serviceport handles the provisioning of transmission resources and the connection setup in the transmission network(s).

G.6.6 Digital section fragment

The management domain border between access network and service node may be found in three POI's locations. POI A identifies the port co-located at or within the AN-equipment OLT. POI B identifies a physical port within the digital section, e.g. an STM-1 interface. POI C is equivalent with POI A in terms of functionality but covers parts of the digital section, too.

Upon creating a digital section link connection between the service port and the interface designated at the service node, the following parameter must be provided by the service provider:

for POI A and C: e1TTPId (available through the cable-id)

for POI B: au4CTPId and channel number

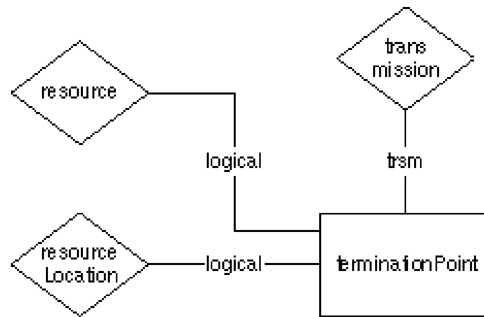


Figure G.12: Transport access network fragment

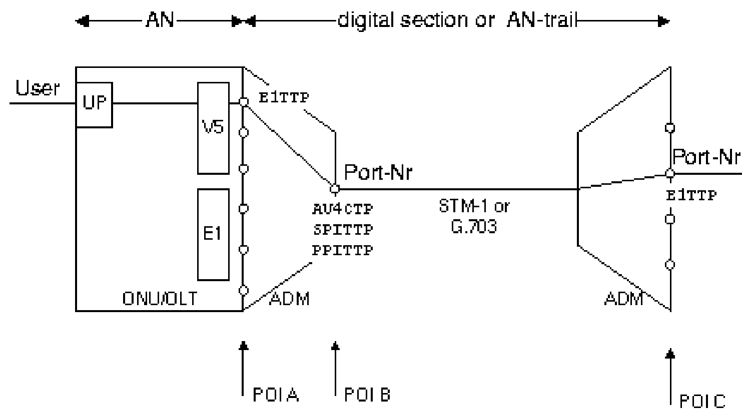


Figure G.13: Connection Points from the Service Node to the Access Network

The digital section is model-led according to its transmission technology. The management of the digital section, including connection set-up, lies outside the scope of this document. Therefore, the only relevant connection point is POI A.

The connection point is of the logical resource type. If the digital section is separated from the access network a physical representation is necessary represented through a pPITtp (for physical E0/E1 interfaces) or a vc12Ttp (if an SDH interface is provided within the access network).

Figure G.14: Digital section relationships

The interface relationship provides for this association between the objects in the portFunction role and those in the networkInterface role. The minimal number of object instantiations is one per interface. Note, that the trail termination point is modeled in any event even if the digital section itself is managed in a different information model.

G.6.7 V5-based transport bearer service fragment

The V5 fragment derives from [300 376] [300 377].

Figure G.15: V5 transport bearer service fragment

The v5Interface fragment consists of the v5Interface object. Note, that the v5Provisioning variant object is not implemented. In V5.1, the commPath Fragment consists of the pstn-, isdn-, control-CommPath objects. V5.2 makes use of the bcc- and protectionCommPaths, too.

The creation of the MIB for the V5 fragment can be split into four parts. This structure can be observed as well in Figure G.15.

1. The initial MIB Configuration: The managed objects which represent the domain for the V5 fragment (container objects) should be present before any other phase starts. (upper part of Figure G.15).

    (a) Insertion of the V5.1 Interface related objects. Besides the V5 Interface MO, the V5Ttp, V5TimeSlot, commPath's and commChannel's too are instantiated is this phase (right part of Figure G.15).

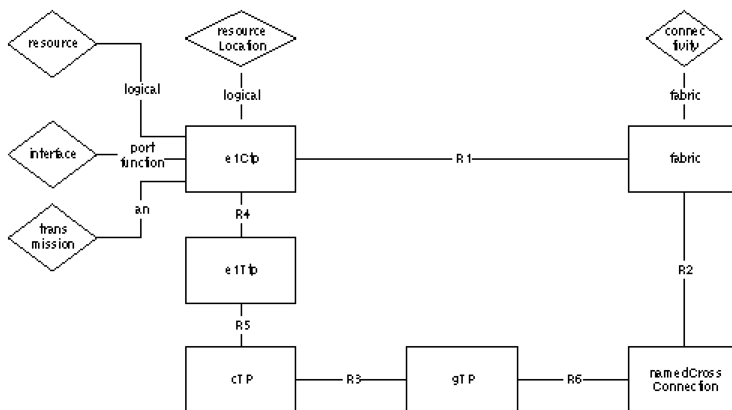
    (b) Insertion of the User Port and related objects. In this phase the UP Ttp and its related Bearer Channel Ctp's are instantiated (left part of Figure G.15). Notice, that the R4 relationship may not exist without an R5 relationship instantiated.

2. Provisioning of a V5.1 User Port. This phase assumes the precedent three phases are already done. No more managed object needs to be created, but the connections (relationship pointers) between the UP side and the V5 side must be set up. After this phase the service is available. In Figure G.15 it is represented by the relationships R4, R5, R9 and R10.

G.6.8 Narrow-band leased line transport bearer service fragment

Next to the E0 and E1 user ports and their associated channel objects (cTP's) the leased line fragment consists of several objects representing channel grouping and crossconnection facilities across the access network.

The gTP (group termination point) managed object is instantiated by the object fabric when the service type of the leased line is n * E0 and a crossconnection is to be created. It will be used to summarize the involved CTPs. The fabric managed object is instantiated a single time during the configuration and is responsible for the handling of connections across the access network. The crossconnection managed object provides the connection between user port and service ports and is instantiated through the fabric, once per connection.



For G703.3 E0 and G.703 E1 unstructured leased lines, the following information model applies.

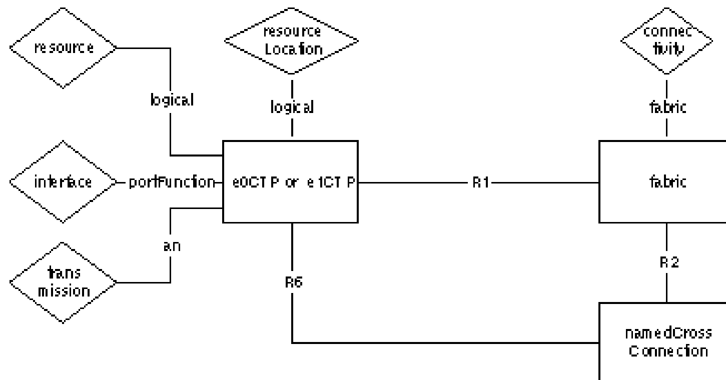


Figure G.16: Leased Line transmission bearer service fragment

G.7 Example of MIBs and Fragments

The details on the management information model (managed objects, relationships, etc) are not given here. The interested reader can refer to the full text of the Ensemble [Etsi97]. The MIM is illustrated on an example. The following figure describes a sample implementation of a V5.1 based access network showing all relevant managed objects in all fragments.

Fragment Overview

1 = equipment & cabling fragment
2 = transport bearer service fragment
3 = transport access network fragment
4 = digital section fragment
5 = topology fragment

G.8 Scenarios

Three types of scenarios are distinguished: general scenario's providing "house-keeping" functions, installation & configuration scenarios for service and user ports and provisioning scenarios. Scenario's with similar algorithms (such as pstn or leased line installation) are considered equivalent and noted as such. Pre and postconditions define predicates which apply to the MIB. They do not list the possible interactions between manager and agent. The scenarios describe the management-interactions between manager and agent.

The content of the scenarios is not included here as it is present in the ETS. It result as a mapping of the scenarios obtained with the TIMS tool-set. The mapping consists mostly of a direct translation

of material in a *Scheme* form into the classical textual form observed in Ensembles. Pre and post-conditions present in scenarios can be found in behaviors involved. They are also transformed from the prefixed *Scheme* notation used in TIMS to the in-fixed notation observed in Ensemble documents. The mangement command is directly obtained from the scenarios in *Scheme*. The trace output in the *Scheme* interpreter, contains the response from the agent, notifications emitted, etc...

Here the work with TIMS to produce scenarios is illustrated based on a particular set of scenarios that can be applied in sequence. The output, in terms of the *Scheme* trace and system monitoring views available in the TIMS tool-set allows the specifier to check the outcome obtained at each scenario. The object view can be used to check that the expected side-effects have occurred or more generally that the MIB is in a correct state. The execution view can be used to check that the proper set of behaviors were executed (e.g. on the agent side).

The complete list of scenarios present in [Etsi97], the Ensemble document (within brackets is the section reference in the ETS [Etsi97]) is :

1. (9.1) General Scenarios
   (a) (9.1.1) Initial MIB structure
   (b) (9.1.2) Locate a Port
   (c) (9.1.3) Ensure connectivity between User Port and Service Port
   (d) (9.1.4) Insert Equipment
      i. (9.1.4.1) Agent driven on-line installation
      ii. (9.1.4.2) Mgr/Agent driven pre-installation
         A. (9.1.4.2.1) Part 1 : pre-installation phase
         B. (9.1.4.2.2) Part 2: installation phase
      iii. (9.1.5) Delete Equipment
         A. (9.1.5.1) Agent-driven deletion
         B. (9.1.5.2) Mgr/Agent-driven Deletion
2. (9.2) User Port Management Scenarios
   (a) (9.2.1) Insert User Port
   (b) (9.2.2) Remove User Port
   (c) (9.2.3) Read User Port
   (d) (9.2.4) Modify User Port Attribute
   (e) (9.2.5) Modify User Port Association
3. (9.3) Service Port Management Scenarios
   (a) (9.3.1) Insert V5.1 Interface
   (b) (9.3.2) Associate v5 link to physical link
   (c) (9.3.3) Startup the V5.1 Interface
   (d) (9.3.4) Delete V5.1 Interface
   (e) (9.3.5) Modify V5.1 Interface
4. (9.4) Provisioning
   (a) (9.4.1) Establish a connection to a V5.1

(b) (9.4.2) De-establish a connection to a V5.1

(c) (9.4.3) Modify/Read a connection to a V5.1

(d) (9.4.4) Block/Unblock User Port

(e) (9.4.5) Find free Leased Line Port

(f) (9.4.6) Establish / Delete Leased Line connection E0 / E1

(g) (9.4.7) Establish / Delete Leased Line connection n * E0

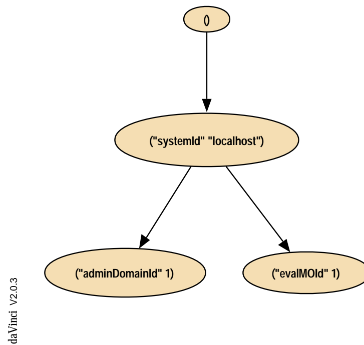
(h) (9.4.8) Augment/Reduce Leased Line n * E0

G.8.1 InitialMIB

This scenario describes the stepwise construction of the initial structure of the MIB as considered relevant for the subsequent scenarios. This scenario is executed once upon the instantiation of the agent.

```
1
2   ;;; the scenario manager command in scheme.
3
4   (define (InitialMIB)
5     (begin
6       (printf "\nInitialMIB\n")
7       (msif:mcreate! (ir-getglobal *msd*) (msif:new-invoke)
8                      "adminDomain"
9                      '(("systemId" "localhost"))
10                     '("adminDomainId" 1)
11                     `("nameBinding" ,*unspecified*)
12                     '("userLabel" "fe217"))
13      )
14    )
15
16
17  ;;; the scheme trace obtained is:
18  ;;; (note that pp is a function used to pretty print scheme data)
19
20
21  > (pp (InitialMIB))
22  InitialMIB
23  (%%record
24   mocnf-create
25   (msd -1)
26   (invoke 1)
27   (moc "adminDomain")
28   (moi (("systemId" "localhost") ("adminDomainId" 1)))
29   (attrvals
30     ((%%record momsg-cmis-param (id "userLabel") (val "fe217"))
31      (%%record momsg-cmis-param (id "adminDomainId") (val 1))
32      (%%record momsg-cmis-param (id "nameBinding"))
33      (%%record
34       momsg-cmis-param
35       (id "objectClass")
36       (val (%%record pair (key globalform) (value (2 37 1 1 100 100 1 4))))))))
37
```

The initial MIB displayed on the object view is :



Note The top level hierarchy (first levels of the management information tree) does not exactly conform to the V5 information model given in section G.6. `adminDomain` is used instead of `managedElement` and `networkR1`. This simplification does not change anything to the part of interest in the information model, i.e. the V5 hierarchy.

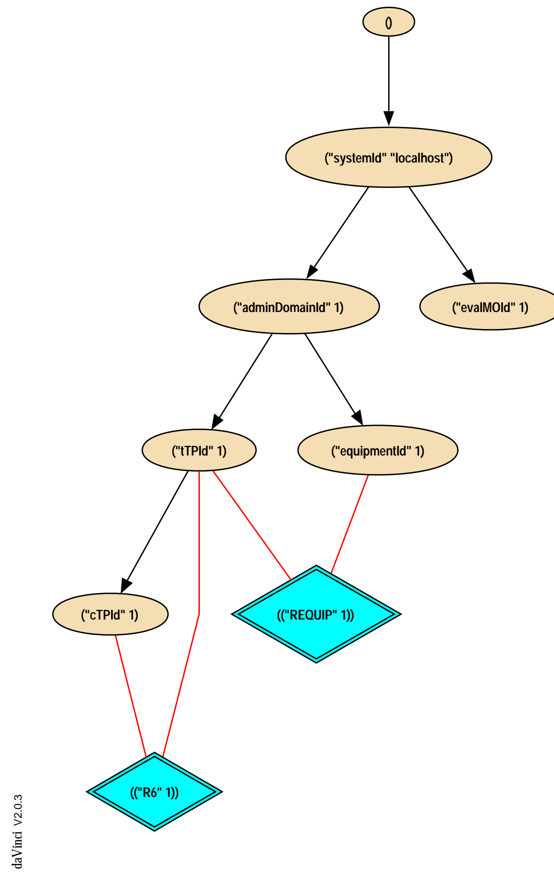
G.8.2 InsertEquipment-Pstn

The insert equipment operation creates the appropriate equipment structure within the MIB. The insert operation creates also the MIB structure of a user port (here PSTN). These operations are performed in conjunction. This operation is applicable to user ports: pstnUserPort, isdnBAUserPort, isdnPRAUserPort, leasedPort, E0cTP and E1Ctp. The precise structure is identified by the equipment inserted.

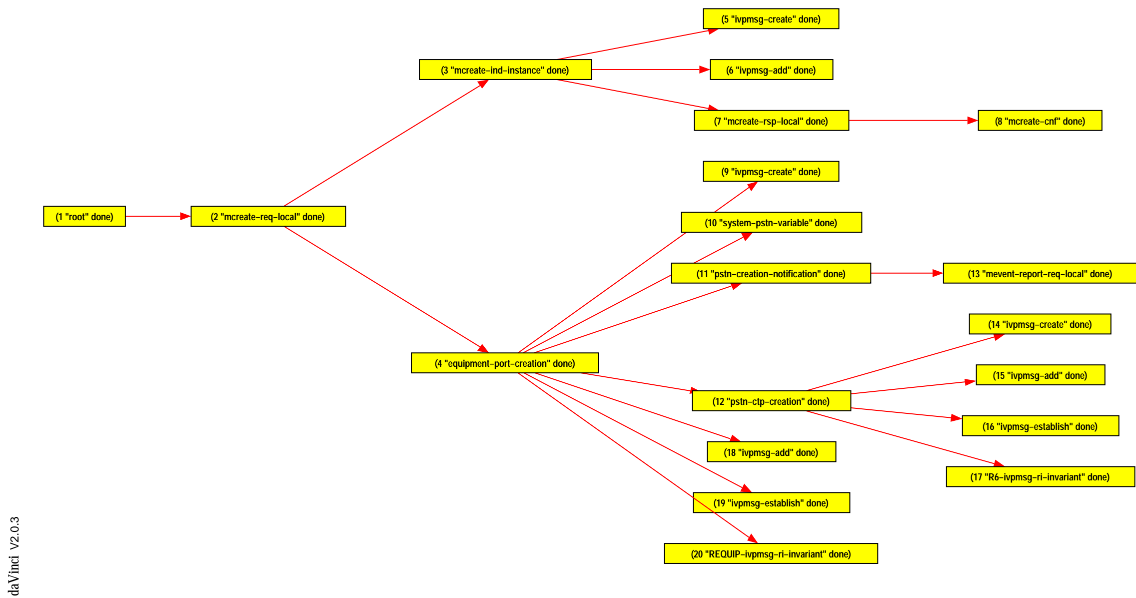
```
1
2   ;;; the scenario manager command in scheme.
3
4   (define (InsertEquipment-Pstn)
5     (let ((equipId 1))
6       (msif:mcreate! (ir-getglobal *msd*) (msif:new-invoke)
7                      "equipment"
8                      '(("systemId" "localhost") ("adminDomainId" 1))
9                      `("equipmentId" ,equipId)
10                     `("nameBinding" ,*unspecified*)
11                     '("operationalState" enabled)
12                     '("administrativeState" locked)
13                     '("userLabel" "fe217")
14                     `("supportedByObjectList" ,*unspecified*)
15                     `("lifeCycleState" ,*unspecified*)
16                     `("assignmentState" ,*unspecified*)
17                     `("availableSignalList" ,*unspecified*)
18                     )
19      )
20    )
21
22  ;;; the scheme trace obtained is:
23
24  > (pp (InsertEquipment-Pstn))
25
26  ;;; this is some output printed by behaviors.
27
28  ###############################
29  ((systemId localhost) (adminDomainId 1) (tTPId 1))
30  ###############################
31  (("systemId" "localhost") ("adminDomainId" 1) ("tTPId" 1))
32  ##############################
33  pstn-creation-notification
34  ((systemId localhost) (adminDomainId 1) (tTPId 1))
35  ##############################
36
```

```
37
38 ;;; this is an event report sent by the agent notifying the objectCreation
39 ;;; there is no behavior fetched to process it on the manager side,
40 ;;; that is the reason why this warning is printed by the BPE.
41 ;;; Note that to obtain a better output (e.g. something more usable for the
42 ;;; insertion into the Ensemble document) a manager behavior could be
43 ;;; inserted in the system.
44
45 bpe:step-body>> nothing to be done:
46 no coupled behavior has been fetched
47 for the execution of
48 ((%%record moind-event-report (msd -101)
49            (moi ((systemId localhost) (adminDomainId 1) (tTPId 1)))
50            (evtype objectCreation)))
51
52 ##############################
53 pstn-ctp-creationpstnUserPort
54 ##############################
55
56 (%%record
57  mocnf-create
58  (msd -1)
59  (invoke 2)
60  (moc "equipment")
61  (moi (("systemId" "localhost") ("adminDomainId" 1) ("equipmentId" 1)))
62  (attrvals
63    ((%%record momsg-cmis-param (id "supportedByObjectList"))
64     (%%record momsg-cmis-param (id "userLabel") (val "fe217"))
65     (%%record momsg-cmis-param (id "lifeCycleState"))
66     (%%record momsg-cmis-param (id "assignmentState"))
67     (%%record momsg-cmis-param (id "administrativeState") (val locked))
68     (%%record momsg-cmis-param (id "operationalState") (val enabled))
69     (%%record momsg-cmis-param (id "availableSignalList"))
70     (%%record momsg-cmis-param (id "equipmentId") (val 1))
71     (%%record momsg-cmis-param (id "nameBinding"))
72     (%%record
73      momsg-cmis-param
74      (id "objectClass")
75      (val (%%record pair (key globalform) (value (2 37 1 1 100 100 1 9))))))))
76
77
```

The MIB obtained after this scenario is :



The behavior execution tree corresponding to this scenario is :



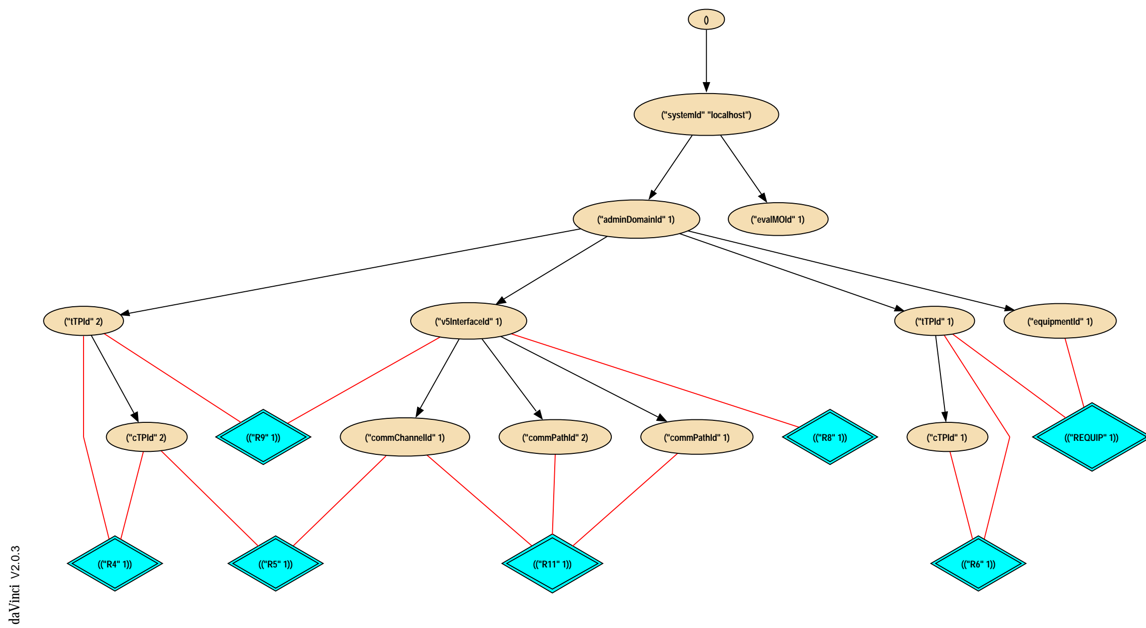
G.8.3 InsertV5Interface

This operation is performed only in conjunction with the ”Insert Equipment” scenario. This scenario builds the entire V5.1 MIB structure.

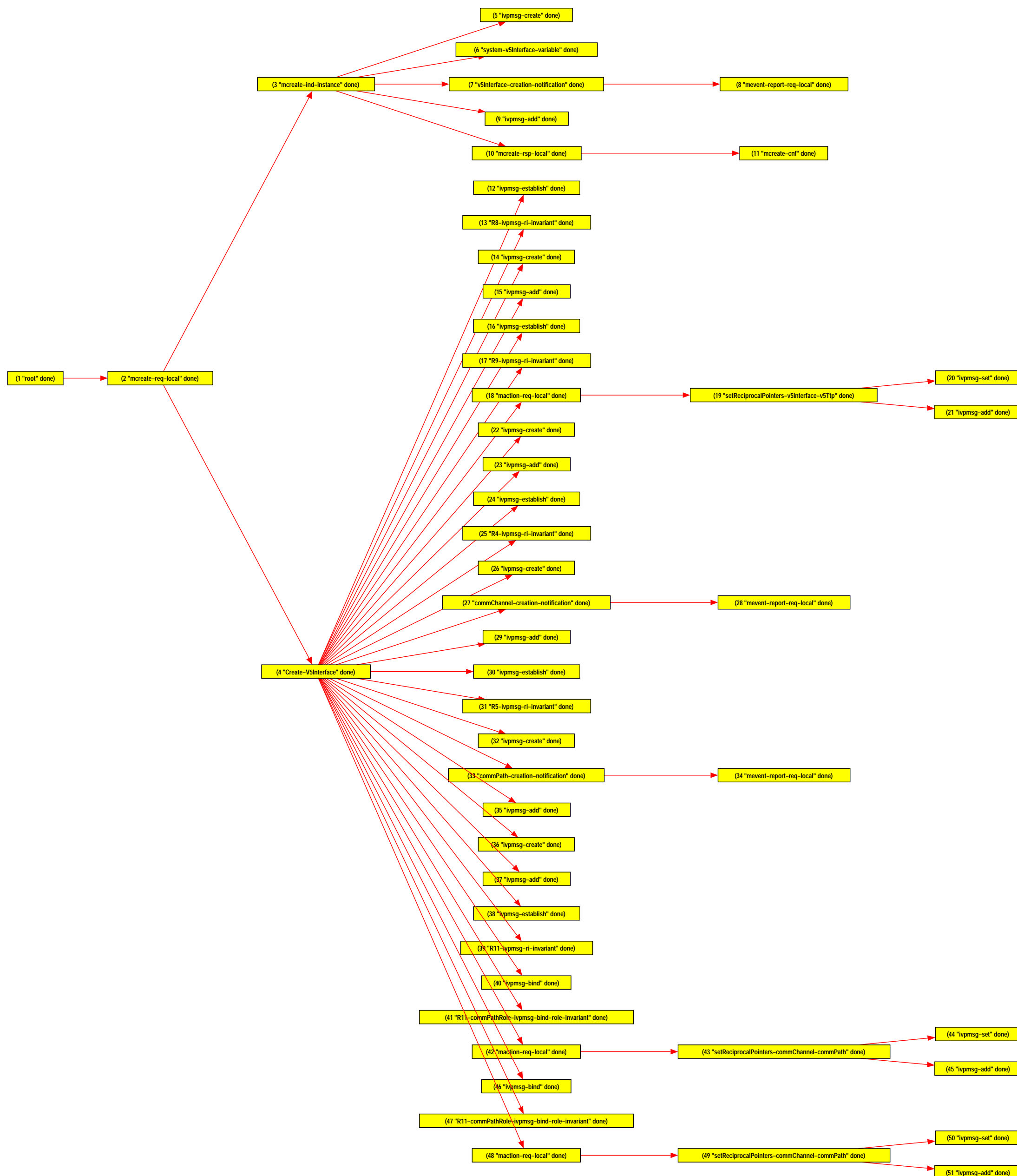
```
1
2   ;;; the scenario manager command in scheme.
3
4   (define (InsertV5Interface)
5     (msif:mcreate! (ir-getglobal *msd*) (msif:new-invoke)
6                     "v5Interface"
7                     '(("systemId" "localhost")("adminDomainId" 1))
8                     '("v5InterfaceId" 1)
9                     `("nameBinding" ,*unspecified*)
10                    '("clientUserPorts" ())
11                    '("serverV5Ttps" ())
12                    '("supportedProtocolVersion" *unspecified*)
13                    )
14    )
15
16  ;;; the scheme trace obtained is:
17
18  > (pp (InsertV5Interface))
19
20  bpe:step-body>> nothing to be done:
21  no coupled behavior has been fetched
22  for the execution of
23  ((%%record moind-event-report (msd -101)
24             (moi ((systemId localhost) (adminDomainId 1) (v5InterfaceId 1)))
25             (evtype objectCreation)))
26
27  Execute Beh: Create-V5Fragment
28  setReciprocalPointers-v5Interface-v5Ttp
29
30  bpe:step-body>> nothing to be done:
31  no coupled behavior has been fetched
32  for the execution of
33  ((%%record moind-event-report (msd -101)
34             (moi ((systemId localhost) (adminDomainId 1)
35                   (v5InterfaceId 1) (commChannelId 1)))
36             (evtype objectCreation)))
37
38  bpe:step-body>> nothing to be done:
39  no coupled behavior has been fetched
40  for the execution of
41  ((%%record moind-event-report (msd -101)
42             (moi ((systemId localhost) (adminDomainId 1)
43                   (v5InterfaceId 1) (commPathId 1)))
44             (evtype objectCreation)))
45
46  setReciprocalPointers-commChannel-commPath
47
48  setReciprocalPointers-commChannel-commPath
49
50  (%%record
51   mocnf-create
52   (msd -1)
53   (invoke 3)
54   (moc "v5Interface")
55   (moi (("systemId" "localhost") ("adminDomainId" 1) ("v5InterfaceId" 1)))
56   (attrvals
57     ((%%record momsg-cmis-param (id "clientUserPorts") (val ()))
58      (%%record momsg-cmis-param (id "serverV5Ttps") (val ()))
59      (%%record momsg-cmis-param (id "supportedProtocolVersion") (val *unspecified*))
60      (%%record momsg-cmis-param (id "v5InterfaceId") (val 1))
61      (%%record momsg-cmis-param (id "nameBinding"))
62      (%%record
63       momsg-cmis-param
64       (id "objectClass")
```

```
65        (val (%%record pair (key globalform) (value (2 37 1 1 100 300 1 1))))))))
66
```

The MIB obtained after this scenario is :



The behavior execution tree corresponding to this scenario is :



G.8.4 Establish-Connection

This scenario provides for the connection between a PSTN user port and a V5.1 interface. The input parameters are the userPort and the v5Interface, identified by their DN.

```
1
2   ;;; the scenario manager commands in scheme.
3
4   (define (Establish-Connection)
5
6     (define Uctp
7       (car (msif:mselect
8             (ir-getglobal *msd*)
9             (msif:new-invoke)
```

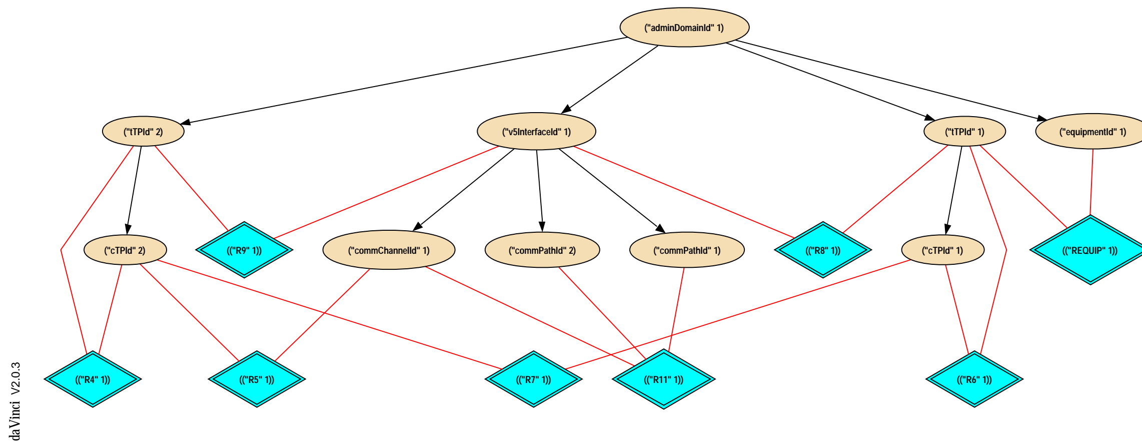
```
10              "pstnUserPort" (ir-getglobal *pstnUserPort*)
11              (asn:choice-value 'namedNumbers 'firstLevelOnly)
12              (asn:choice-value
13               'and (asn:set-of
14                     (asn:choice-value
15                      'item
16                      (asn:choice-value
17                       'equality
18                       (asn:seq
19                        (asn:field-value 'attributeId (asn:choice-value
20                                                       'globalForm
21                                                       (oid:str->oid "objectClass")))
22                        (asn:field-value
23                         'attributeValue
24                         (asn:ber->any
25                          (asn:scm->ber (asn:choice-value
26                                         'globalForm
27                                         (oid:str->oid "userPortBearerChannelCtp"))
28                                        (mor:attribute-syntax "objectClass")))))))
29                     (asn:choice-value
30                      'item
31                      (asn:choice-value
32                       'equality
33                       (asn:seq
34                        (asn:field-value 'attributeId (asn:choice-value
35                                                       'globalForm
36                                                       (oid:str->oid "assocTimeSlot")))
37                        (asn:field-value
38                         'attributeValue
39                         (asn:ber->any
40                          (asn:scm->ber (asn:choice-value 'null 'NULL)
41                                        (mor:attribute-syntax "assocTimeSlot")))))))
42                     )))))
43
44      (define v5Ttp (val:get (msif:mget (ir-getglobal *msd*) (msif:new-invoke)
45                                        "v5Interface" (ir-getglobal *v5Interface*)
46                                        (asn:choice-value 'namedNumbers 'baseObject)
47                                        *unspecified*
48                                        "serverV5Ttps")
49                             '(attrvals 0 attrval 0)))
50
51      (define Vctp
52        (car (msif:mselect
53              (ir-getglobal *msd*) (msif:new-invoke)
54              "v5Ttp" v5Ttp
55              (asn:choice-value 'namedNumbers 'firstLevelOnly)
56              (asn:choice-value
57               'and (asn:set-of
58                     (asn:choice-value
59                      'item
60                      (asn:choice-value
61                       'equality
62                       (asn:seq
63                        (asn:field-value 'attributeId (asn:choice-value
64                                                       'globalForm
65                                                       (oid:str->oid "objectClass")))
66                        (asn:field-value
67                         'attributeValue
68                         (asn:ber->any
69                          (asn:scm->ber (asn:choice-value
70                                         'globalForm
71                                         (oid:str->oid "v5TimeSlot"))
72                                        (mor:attribute-syntax "objectClass")))))))
73                     (asn:choice-value
74                      'item
75                      (asn:choice-value
76                       'equality
77                       (asn:seq
78                        (asn:field-value 'attributeId (asn:choice-value
```

```
79                                                      'globalForm
80                                                      (oid:str->oid "assocResource")))
81                         (asn:field-value
82                          'attributeValue
83                          (asn:ber->any
84                           (asn:scm->ber (asn:choice-value 'null 'NULL)
85                                         (mor:attribute-syntax "assocResource")))))))))
86           )))
87
88    (msif:maction (ir-getglobal *msd*) (msif:new-invoke)
89                  ;; moc
90                  "v5Interface"
91
92                  ;;moi
93                  (ir-getglobal *v5Interface*)
94
95                  ;;mode
96                  'confirmed
97
98                  ;;action
99                  "setReciprocalPointers"
100
101                 ;;scope
102                 *unspecified*
103
104                 ;;filter
105                 *unspecified*
106
107                 ;;argument: ReciprocalPointersInfoArg
108                 `(,(oid:str->oid "userPortBearerChannelCtp")
109                   ,(moi:scm->asn Uctp)
110                   ,(oid:str->oid "assocTimeSlot")
111                   ,(oid:str->oid "v5TimeSlot")
112                   ,(moi:scm->asn Vctp)
113                   ,(oid:str->oid "assocResource"))
114                 )
115
116   (msif:maction (ir-getglobal *msd*) (msif:new-invoke)
117                 ;; moc
118                 "v5Interface"
119
120                 ;;moi
121                 (ir-getglobal *v5Interface*)
122
123                 ;;mode
124                 'confirmed
125
126                 ;;action
127                 "setReciprocalPointers"
128
129                 ;;scope
130                 *unspecified*
131
132                 ;;filter
133                 *unspecified*
134
135                 ;;argument: ReciprocalPointersInfoArg
136                 `(,(oid:str->oid "pstnUserPort")
137                   ,(moi:scm->asn (ir-getglobal *pstnUserPort*))
138                   ,(oid:str->oid "assocV5Interface")
139                   ,(oid:str->oid "v5Interface")
140                   ,(moi:scm->asn (ir-getglobal *v5Interface*))
141                   ,(oid:str->oid "clientUserPorts"))
142                 )
143
144   )
145
146 ;;; the scheme trace obtained is:
147
```

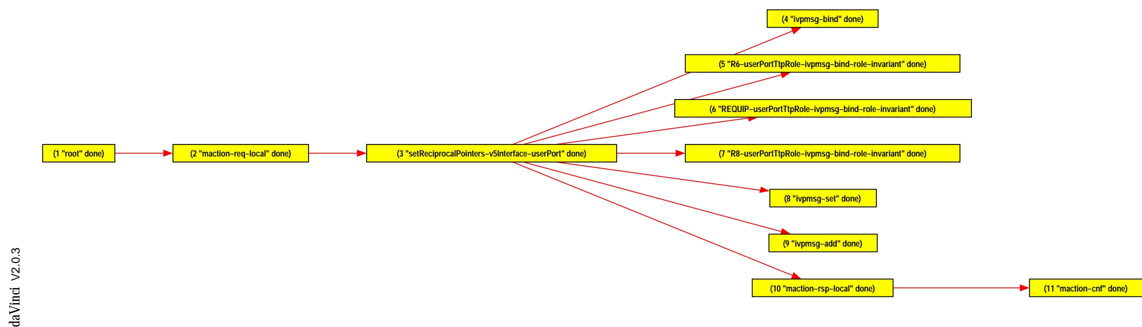
```
148 > (pp (Establish-Connection))
149 *** setReciprocalPointers-bearerChannel-v5TimeSlot
150
151 *** setReciprocalPointers-v5Interface-userPort
152 (%%record
153  mocnf-action
154  (msd -1)
155  (linked 0)
156  (moi (("systemId" "localhost") ("adminDomainId" 1) ("v5InterfaceId" 1)))
157  (actype "setReciprocalPointers")
158  (acreply
159    (((("systemId" "localhost") ("adminDomainId" 1) ("v5InterfaceId" 1))
160      (("systemId" "localhost") ("adminDomainId" 1) ("tTPId" 1)))
161     null)))
162
163
164 ;;; this is obtained by browsing the MIB to check e.g. that the reciprocal
165 ;;; pointers and corresponding relationships have been set properly.
166
167 ;;; the v5Interface MO:
168
169 > ObjInst (("systemId" "localhost") ("adminDomainId" 1) ("v5InterfaceId" 1)):
170 Properties:
171   "supportedProtocolVersion": *unspecified*
172   "serverV5Ttps": ((("systemId" "localhost") ("adminDomainId" 1) ("tTPId" 2)))
173   "clientUserPorts": ((("systemId" "localhost") ("adminDomainId" 1)
174                                                  ("tTPId" 1)))
175   "nameBinding": *unspecified*
176   "v5InterfaceId": 1
177   "subordinates": ((("systemId" "localhost") ("adminDomainId" 1)
178                     ("v5InterfaceId" 1) ("commPathId" 2))
179                    (("systemId" "localhost") ("adminDomainId" 1)
180                     ("v5InterfaceId" 1) ("commPathId" 1))
181                    (("systemId" "localhost") ("adminDomainId" 1)
182                     ("v5InterfaceId" 1) ("commChannelId" 1)))
183   "superior": (("systemId" "localhost") ("adminDomainId" 1))
184   "objectClass": (%%record pair (key globalform)
185                     (value (2 37 1 1 100 300 1 1)))
186   "objClass": "v5Interface"
187 Parties:
188   "v5InterfaceRole" => (("R9" 1))
189   "v5InterfaceRole" => (("R8" 1))
190
191
192 ;;; the user port MO:
193
194 > ObjInst (("systemId" "localhost") ("adminDomainId" 1) ("tTPId" 1)):
195 Properties:
196   "nameBinding": (0 1 2 3)
197   "assocV5Interface": (("systemId" "localhost") ("adminDomainId" 1)
198                                                  ("v5InterfaceId" 1))
199   "downstreamConnectivityPointer": *unspecified*
200   "supportedByObjectList": *unspecified*
201   "upstreamConnectivityPointer": *unspecified*
202   "layer3PortAddress": *unspecified*
203   "specialFeatures": *unspecified*
204   "operationalState": disabled
205   "administrativeState": locked
206   "tTPId": 1
207   "subordinates": ((("systemId" "localhost") ("adminDomainId" 1)
208                     ("tTPId" 1) ("cTPId" 1)))
209   "superior": (("systemId" "localhost") ("adminDomainId" 1))
210   "objectClass": (%%record pair (key globalform)
211                     (value (2 37 1 1 100 300 1 104)))
212   "objClass": "pstnUserPort"
213 Parties:
214   "userPortTtpRole" => (("R8" 1))
215   "userPortTtpRole" => (("REQUIP" 1))
216   "userPortTtpRole" => (("R6" 1))
```

217

The MIB obtained after this scenario is :



The behavior execution tree corresponding to this scenario is :



G.8.5 De-Establish-Connection

This management operation deletes the connection between the V5.1 interface and the user port.

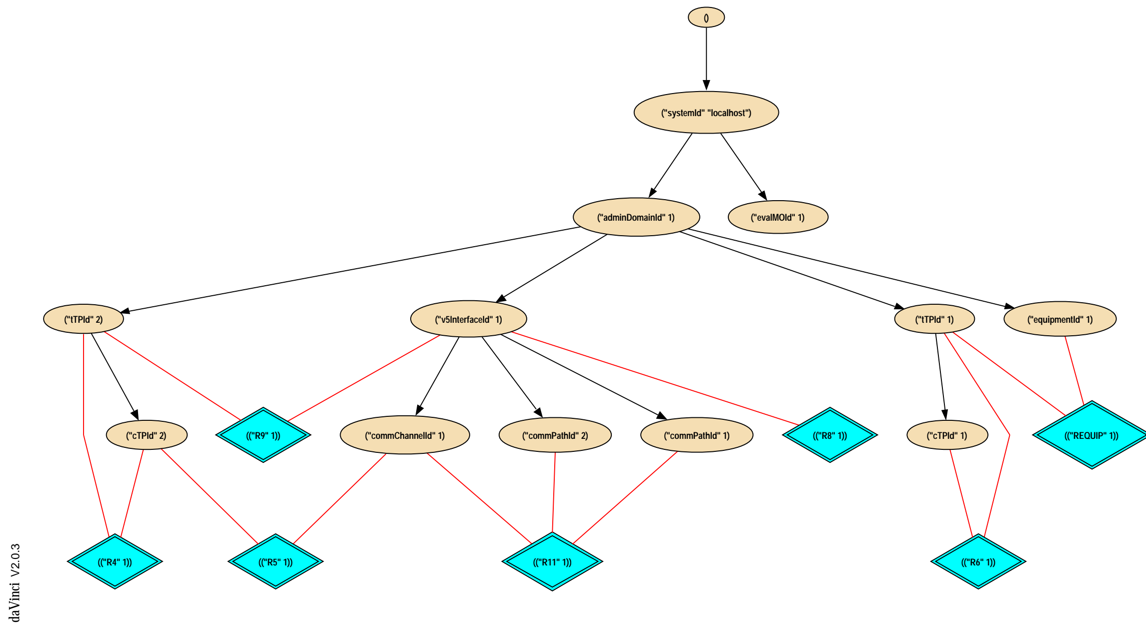
```
1
2   ;;; the scenario manager command in scheme.
3
4   (define (De-Establish-Connection)
5
6     (define Uctp
7       (car (msif:mselect
8              (ir-getglobal *msd*) (msif:new-invoke)
9              "pstnUserPort" (ir-getglobal *pstnUserPort*)
10             (asn:choice-value 'namedNumbers 'firstLevelOnly)
11             (asn:choice-value
12              'item
13              (asn:choice-value
14               'equality
15               (asn:seq
16                (asn:field-value 'attributeId (asn:choice-value
17                                               'globalForm
18                                               (oid:str->oid "objectClass")))
19                (asn:field-value
20                 'attributeValue
21                 (asn:ber->any
```

```
22                 (asn:scm->ber (asn:choice-value
23                               'globalForm
24                               (oid:str->oid "userPortBearerChannelCtp"))
25                              (mor:attribute-syntax "objectClass")))))))))
26
27      (define v5Ttp
28        (val:get (msif:mget (ir-getglobal *msd*) (msif:new-invoke)
29                            "v5Interface" (ir-getglobal *v5Interface*)
30                            (asn:choice-value 'namedNumbers 'baseObject)
31                            *unspecified*
32                            "serverV5Ttps")
33                 '(attrvals 0 attrval 0)))
34
35      (define Vctp
36        (car (msif:mselect
37              (ir-getglobal *msd*) (msif:new-invoke)
38              "v5Ttp" v5Ttp
39              (asn:choice-value 'namedNumbers 'firstLevelOnly)
40              (asn:choice-value
41               'item
42               (asn:choice-value
43                'equality
44                (asn:seq
45                 (asn:field-value 'attributeId (asn:choice-value
46                                               'globalForm
47                                               (oid:str->oid "objectClass")))
48                 (asn:field-value
49                  'attributeValue
50                  (asn:ber->any
51                   (asn:scm->ber (asn:choice-value
52                                 'globalForm
53                                 (oid:str->oid "v5TimeSlot"))
54                                (mor:attribute-syntax "objectClass")))))))))
55
56      (msif:maction (ir-getglobal *msd*) (msif:new-invoke)
57                    ;; moc
58                    "v5Interface"
59
60                    ;;moi
61                    (ir-getglobal *v5Interface*)
62
63                    ;;mode
64                    'confirmed
65
66                    ;;action
67                    "releaseReciprocalPointers"
68
69                    ;;scope
70                    *unspecified*
71
72                    ;;filter
73                    *unspecified*
74
75                    ;;argument: ReciprocalPointersInfoArg
76                    `(,(oid:str->oid "userPortBearerChannelCtp")
77                      ,(moi:scm->asn Uctp)
78                      ,(oid:str->oid "assocTimeSlot")
79                      ,(oid:str->oid "v5TimeSlot")
80                      ,(moi:scm->asn Vctp)
81                      ,(oid:str->oid "assocResource"))
82                    )
83
84      (msif:maction (ir-getglobal *msd*) (msif:new-invoke)
85                    ;; moc
86                    "v5Interface"
87
88                    ;;moi
89                    (ir-getglobal *v5Interface*)
90
```

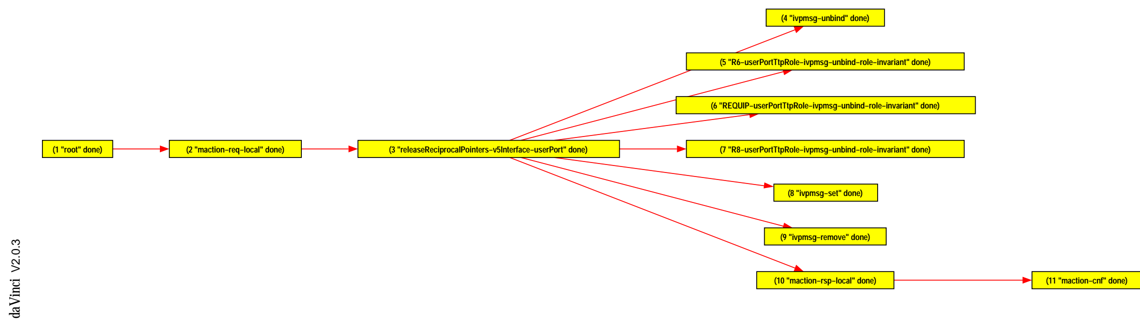
```
91                 ;;mode
92                 'confirmed
93
94                 ;;action
95                 "releaseReciprocalPointers"
96
97                 ;;scope
98                 *unspecified*
99
100                ;;filter
101                *unspecified*
102
103                ;;argument: ReciprocalPointersInfoArg
104                `(,(oid:str->oid "pstnUserPort")
105                  ,(moi:scm->asn (ir-getglobal *pstnUserPort*))
106                  ,(oid:str->oid "assocV5Interface")
107                  ,(oid:str->oid "v5Interface")
108                  ,(moi:scm->asn (ir-getglobal *v5Interface*))
109                  ,(oid:str->oid "clientUserPorts"))
110                )
111
112   )
113
114 ;;; the scheme trace obtained is:
115
116 (pp (De-Establish-Connection))
117
118 *** releaseReciprocalPointers-bearerChannel-v5TimeSlot
119
120 *** releaseReciprocalPointers-v5Interface-userPort
121 (%%record
122  mocnf-action
123  (msd -1)
124  (linked 0)
125  (moi (("systemId" "localhost") ("adminDomainId" 1) ("v5InterfaceId" 1)))
126  (actype "releaseReciprocalPointers")
127  (acreply
128    ((((("systemId" "localhost") ("adminDomainId" 1) ("v5InterfaceId" 1))
129      (("systemId" "localhost") ("adminDomainId" 1) ("tTPId" 1)))
130     null)))
131
132 ;;; this is obtained by browsing the MIB to check e.g. that the reciprocal
133 ;;; pointers and the corresponding relationships have been unset properly.
134
135 ;;; the v5Interface MO:
136
137 > ObjInst (("systemId" "localhost") ("adminDomainId" 1) ("v5InterfaceId" 1)):
138 Properties:
139   "supportedProtocolVersion": *unspecified*
140   "serverV5Ttps": ((("systemId" "localhost") ("adminDomainId" 1)
141                                            ("tTPId" 2)))
142   "clientUserPorts": ()
143   "nameBinding": *unspecified*
144   "v5InterfaceId": 1
145   "subordinates": ((("systemId" "localhost") ("adminDomainId" 1)
146                     ("v5InterfaceId" 1) ("commPathId" 2))
147                    (("systemId" "localhost") ("adminDomainId" 1)
148                     ("v5InterfaceId" 1) ("commPathId" 1))
149                    (("systemId" "localhost") ("adminDomainId" 1)
150                     ("v5InterfaceId" 1) ("commChannelId" 1)))
151   "superior": (("systemId" "localhost") ("adminDomainId" 1))
152   "objectClass": (%%record pair (key globalform)
153                     (value (2 37 1 1 100 300 1 1)))
154   "objClass": "v5Interface"
155 Parties:
156   "v5InterfaceRole" => (("R9" 1))
157   "v5InterfaceRole" => (("R8" 1))
158
159
```

```
160 ;;; the user port MO:
161
162 > ObjInst (("systemId" "localhost") ("adminDomainId" 1) ("tTPId" 1)):
163 Properties:
164   "nameBinding": (0 1 2 3)
165   "assocV5Interface": ()
166   "downstreamConnectivityPointer": *unspecified*
167   "supportedByObjectList": *unspecified*
168   "upstreamConnectivityPointer": *unspecified*
169   "layer3PortAddress": *unspecified*
170   "specialFeatures": *unspecified*
171   "operationalState": disabled
172   "administrativeState": locked
173   "tTPId": 1
174   "subordinates": ((("systemId" "localhost") ("adminDomainId" 1)
175                     ("tTPId" 1) ("cTPId" 1)))
176   "superior": (("systemId" "localhost") ("adminDomainId" 1))
177   "objectClass": (%%record pair (key globalform)
178                     (value (2 37 1 1 100 300 1 104)))
179   "objClass": "pstnUserPort"
180 Parties:
181   "userPortTtpRole" => (("REQUIP" 1))
182   "userPortTtpRole" => (("R6" 1))
183
```

The MIB obtained after this scenario is :



The behavior execution tree corresponding to this scenario is :



G.9 Conclusion

Besides the activity for the V5.1 specification, this case study gave to people in Swisscom the opportunity to better understand the prototyping methodology provided, what it looks like, and how far the specifier can take advantage from the use of the TIMS tool-set. The major effort (40%) was in acquiring the necessary V5 domain knowledge, sometimes down to the protocol level. Implementing the static schema (resource selection, GRM) then could be quickly done. On the other hand, embedding the fragment into an overall MIB architecture turned out to be more difficult than expected (20%), mainly due to functional restrictions in current GDMO libraries (M.3100). Developing behavior itself resulted in a repeated review and refinement of the requirements, sometimes identifying new demands along the way. Behavior-design and debugging made up for another 40% of the effort.

The V5 management interface project running in parallel benefited a lot from the design efforts in the case study, especially when it came to understanding the finer points of the model, its restrictions and pitfalls. The TIMS scenarios could be mapped almost 1:1 into the Ensemble specification that was produced from the case study and that is in the standardization process to becoming an ETSI standard [Etsi97]. Results on this case study have also been published in an internal Swisscom report [Eberhardt et al.97a]).

Finally, what is expected from the TIMS tool-set is that scenarios obtained with the feedback of a machine by executing a model of the system should be much more solid, so that they can actually be used in procurement.

Curriculum Vitæ

Name Sidou

Firstnames Dominique, Jean-Pierre

Status Married, French

Languages French, English, Spanish

URL http://www.eurecom.fr/~sidou

1994–1997 Research Engineer, technical leader in TIMS (TMN-based Information Model Simulator) project with Swisscom, (Institut Eurécom – France)

1993–1994 Research Assistant (Institut Eurécom – France)

1990–1993 Research Assistant Swiss Federal Institute of Technology, Industrial Computing Laboratory (Lausanne, Switzerland)

1989–1990 Diplôme d'Études Approfondies (DEA) Université Paul Sabatier (Toulouse – France)

Publications

Eberhardt et al.96 Eberhardt (R.), Sidou (D.), Festor (O.), Mazziotta (S.) et Labetoulle (J.). – Executable TMN Specifications in TIMS. *In : NOMS'96 International Network Operations & Management Symposium*. IFIP / IEEE. – Kyoto - Japan, 1996. Poster Session, available at http://www.eurecom.fr/~tims/papers/noms96-summary.ps.gz and http://www.eurecom.fr/~tims/papers/noms96-figures.ps.gz.

Eberhardt et al.97 Eberhardt (Rolf), Mazziotta (Sandro) et Sidou (Dominique). – Design and testing of information models in a virtual environment. *In : The Fifth IFIP/IEEE International Symposium on Integrated Network Management "Integrated Management in a Virtual World"*. – San Diego, CA, USA, may 1997. available at http://www.eurecom.fr/~tims/papers/im97.ps.gz.

Festor et al.95 Festor (Olivier), Sidou (Dominique) et Mazziotta (Sandro). – Managed Object Behavior Inheritance and Distribution Support in TIMS. *In : ECOOP95 Workshop : Use of Object-Oriented technology for Network Design and Management*. – Aarhus, Denmark, August 7-11 1995. Available at http://www.eurecom.fr/~tims/papers/ecoop95-paper.ps.gz.

Mazziotta et al.96a Mazziotta (Sandro) et Sidou (Dominique). – A *Scheme*-based Toolkit for the Fast Prototyping of TMN-systems. – 1996. submitted to Seventh International Workshop on Distributed Systems : Operations & Management.

Mazziotta et al.96b Mazziotta (Sandro) et Sidou (Dominique). – Guidelines for the Specification of Managed Object Behaviors with TIMS. *In : ECOOP96 Workshop : Use of Object-Oriented technology for Network Design and Management*. – Linz, Austria, July 1996.

Sidou et al.93 Sidou (Dominique), Vijayananda (Kateel) et Berthet (Guy). – An Architecture for the Implementation of OSI Protocols: Support Packages, Tools and Performance Issues. *In : Proceedings of the IEEE SICON/ICIE'93 : Singapore International Conference on Networks / International Conference on Information Engineering'93, Communication and Networks for the Year 2000*. – IEEE Syngapore. Available at http://www.eurecom.fr/~sidou/papers/litmap93/main.ps.

Sidou et al.95a Sidou (Dominique), Mazziotta (Sandro) et Eberhardt (Rolf). – TIMS : a TMN-based Information Model Simulator, Principles and Application to a Simple Case Study. *In : Sixth International Workshop on Distributed Systems : Operations & Management*. IFIP / IEEE. – Ottawa - Canada, 1995. Available at http://www.eurecom.fr/˜tims/papers/dsom95-paper.ps.gz.

Sidou et al.95b Sidou (Dominique), Vijayananda (Kateel) et Berthet (Guy). – Scheduling Policies for Protocol Stacks for User Application Requirements. *In : International Conference on Networks / International Conference on Information Engineering (IEEE SICON/ICIE'95)*, éd. par ©IEEE, pp. 116–119. – Singapore, July 1995. Available at http://www.cica.fr/˜sidou/papers/-litmap95/paper.ps.

Sidou95 Sidou (Dominique). – Behavior Simulation and Analysis Techniques for Management Action Policies. *In : SMDS'95 Workshop : Services for the Management of Distributed Systems*. – Karlsruhe, September 20–21 1995. Available at http://www.eurecom.fr/˜tims/papers/smds95.-ps.gz.

Sidou96 Sidou (Dominique). – A Generic and Executable Model for the Specification and Validation of Distributed Behaviors. *In : TreDS'96 : Trends in Distributed Systems Workshop, published in the Lecture Notes in Computer Science (LNCS 1161)*. – available at http://www.eurecom.fr/˜tims/papers/treds96.ps.gz.

Sidou97a Sidou (Dominique). – Precise Semantics for a Behavior Model in the Context of Object Based Distributed Systems. *In : Workshop on Precise Semantics for Object-Oriented Modeling Techniques*, éd. par Kilov (Haim) et Rumpe (Bernhard). – Jyvaskyla, Finland, 1997.

Sidou97b Sidou (Dominique). – Towards a good functional and executable behavior model. *In : Workshop on Object Oriented Technology for Telecommunications Services Engineering*, éd. par Znaty (Simon) et Hubaux (Jean-Pierre). – Jyvaskyla, Finland, 1997.