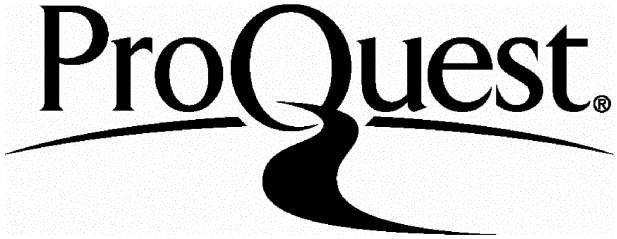
Department of Computer Science
University College London
University of London

Software Agent Architecture for Consistency Checking of Distributed Documents

Danila Stanislavovich Smolko

Submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
at the University of London

November 2001



To Lucie

Abstract

The size and complexity of current software systems often necessitates distributed and cooperative development by numerous participants. This thesis investigates the problem of management of consistency relations between documents in the software engineering domain, which are cooperatively developed in such distributed setting. The thesis builds on related work in expression of consistency relations by means of *consistency rules,* and expands on the framework for centralised *consistency link* generation to transparently provide the developers with inconsistency identification services at distributed locations.

This thesis elaborates a consistency framework, which enables consistency checks to be carried out at locations of the documents, rather than at a centralised server or repository. We focus on advantages that use of software agency brings to implementation of this framework. These advantages are realised in the developed software agent architecture, which capitalises on agent mobility for carrying out checks in the distributed environment.

The thesis describes a method for carrying out consistency checks *incrementally* for the underlying consistency link generation framework. Incremental checks relate *individual* document updates to the current state of the checked document set. Incremental checking facilitates event-orientation of the framework for distributed consistency checks, where individual update events trigger consistency checks of related consistency rules, resulting in generation of consistency links, relevant to the original changes. Inconsistent links indicate problem areas to the developers, where further work may be needed in correction of remaining inconsistencies.

This thesis makes a novel contribution to consistency checking in the software engineering domain by providing the software agent architecture, which enables distributed collaborative development. Initial evaluation of the software agent architecture for distributed consistency checking is carried out on a simulation model.

The software agent architecture is implemented in a prototype, developed to demonstrate and evaluate the architecture. A case study is used throughout the thesis, which illustrates feasibility and applicability of the proposed approach. The case study involves checking consistency between UML diagrams of a scheduling application throughout concurrent distributed development of this application by a team of software engineers.

Acknowledgements

I would like to acknowledge with sincere appreciation a number of people, whom I owe for helping shape this thesis. I would like to express my deep gratitude to Professor Anthony Finkelstein, my supervisor, for reassuring my confidence in achieving of the academic objectives and providing independence in research. This thesis would not have been possible without his attention and unfailing support. I am also grateful to my secondary supervisor, Dr. Wolfgang Emmerich, who has provided constructive feedback for three years of my studies and has given a detailed review to numerous reports and the thesis draft.

I would also like to thank all members of the Software Engineering group and my fellow students, who have provided me with a very supportive environment throughout my stay at UCL. I appreciate the warm welcome into the group from Dr. Andrea Zisman and Dr. Ernst Ellmer. I am particularly grateful to Dr. Cecilia Mascolo for generous encouragement and support through the ups and downs of the PhD, and to Nima Kaveh – for his humour, comradeship and unconditional optimism. Credits are due to Carina Alves, Christian Nentwich, Daniel Dui, Joe Lewis-Bowen, Licia Capra, Luca Zanolin, Rami Bahsoon, Stefanos Zachariadis, my office colleagues, for creation of a friendly and productive atmosphere. Thanks, Christian, for a timely release of the TreeDiff tool as an open source software, and for numerous discussions of the XLinkit framework. Joe deserves special thanks for meticulously reviewing the draft of this thesis at a very short notice. I would also like to thank my friend João Oliveira for his considerate regard and kind attention in the times of pressure.

I owe a special gratitude to my loving partner Lucie, who believes in my abilities and has encouraged me on the way of their realisation. And, last and not least, sincere thanks go out to my parents Liliya and Stanislav for the wisdom of their advice and for the warmth of their hearts. Thank you so much, Mum and Dad!

This work would not have been possible without the financial support of Unipower Ltd. throughout the duration of my Ph.D. studies. I am also thankful to the University College London for awarding me the Graduate Research Scholarship and the International 'Open' Scholarship, which have contributed towards the tuition fees.

Table of Contents

LIST OF FIGURES AND TABLES **10**

CHAPTER 1 INTRODUCTION **14**

1.1 MOTIVATING SCENARIO 15
1.2 CONTRIBUTION 15
1.3 THESIS STRUCTURE 16

CHAPTER 2 RELATED WORK IN CONSISTENCY CHECKING **18**

2.1 THE VIEWPOINTS BACKGROUND 18
2.2 VIEWPOINT-ORIENTED SOFTWARE ENGINEERING 18
2.2.1 ViewPoints 19
2.2.2 Consistency of ViewPoints 19
2.2.3 Consistency Management Activities 21
2.2.4 Policies 22
2.3 SOFTWARE DEVELOPMENT ENVIRONMENTS 23
2.4 META-CASE TOOLS 24
2.5 ATTRIBUTE GRAMMARS 25
2.6 SPECIFICATION LANGUAGES 26
2.6.1 CENTAUR 26
2.6.2 PROGRESS 26
2.6.3 GOODSTEP 26
2.7 XLINKIT 26
2.7.1 Link Generation 28
2.7.2 XLinkit and Distribution Support 28
2.7.3 Exhaustive Consistency Checking 28
2.8 SUMMARY 29

CHAPTER 3 REQUIREMENTS FOR DISTRIBUTED CONSISTENCY CHECKING .. **31**

3.1 STAKEHOLDERS 31
3.2 SPECIFICATION OF INTER-DOCUMENT RELATIONSHIPS 32
3.3 LOCATION OF CONSISTENCY RULES AND THEIR APPLICABILITY 32
3.4 DISTRIBUTED DOCUMENT MONITORING AND CHANGE IDENTIFICATION 33
3.5 TIMING OF CONSISTENCY CHECKS 33
3.6 INCREMENTAL CONSISTENCY CHECKS 33
3.7 RELEVANCE OF CONSISTENCY RULES 34
3.8 DOCUMENT ACCESS LOCALITY 34
3.9 REPRESENTATION OF RESULTS OF CONSISTENCY CHECKS 35
3.10 UPDATE OF RESULTS OF CONSISTENCY CHECKS 35
3.11 EVENT NOTIFICATION AND PROCESSING 35
3.12 SUMMARY 36

CHAPTER 4 SOFTWARE AGENTS AND THE MOBILE AGENT PARADIGM **38**

4.1 WHAT IS A SOFTWARE AGENT? 38
4.2 CHARACTERISTICS OF SOFTWARE AGENTS 40
4.2.1 Weak Agency 40
4.2.2 Strong Agency 41
4.3 AGENT MOBILITY 41
4.4 DISTRIBUTED SYSTEM CONSTRUCTION WITH MOBILE AGENTS 42
4.4.1 Traditional Distributed Systems 42
4.4.2 Mobile Agents 44
4.4.3 Use of Mobile Agents: Pros and Cons 44

4.5 INTER-AGENT COMMUNICATION 48
4.5.1 Communication Primitives 49
4.5.2 Multi-agent Co-operation 49
4.5.3 Location Service for Distributed Agents 50
4.6 TAXONOMY OF MOBILE AGENT FRAMEWORKS 51
4.6.1 Structure of the Taxonomy 51
4.6.2 Comments 52
4.7 THE AGLETS FRAMEWORK 55
4.7.1 Agents 55
4.7.2 Servers 56
4.7.3 Communication 57
4.7.4 Security 57
4.7.5 Aglets Event Model – Agent Cloning Example 57
4.7.6 Aglets Event Model – Agent Mobility 59
4.8 CHOICE OF A MOBILE AGENT FRAMEWORK FOR THE DISTRIBUTED CONSISTENCY CHECKING ARCHITECTURE 60
4.8.1 Requirements For a Mobile Software Agent Framework 61
4.8.2 Selection of a Mobile Software Agent Framework 62
4.9 SUMMARY 63

CHAPTER 5 INCREMENTAL CONSISTENCY CHECKING 64

5.1 CONSISTENCY RULES 64
5.1.1 Rule Example 64
5.1.2 Checking a Consistency Rule 65
5.1.3 Resulting Consistency Links 66
5.1.4 Fragility of XLink Locators 66
5.2 EXHAUSTIVE CONSISTENCY CHECKING 67
5.2.1 Algorithm outline 67
5.2.2 Application Domain for Exhaustive Checks 68
5.3 INCREMENTAL CONSISTENCY CHECKING 70
5.3.1 Initialisation 71
5.3.2 Selection of Relevant Consistency Rules 71
5.3.3 Execution of Selected Rules 72
5.4 DISTRIBUTION OF CONSISTENCY CHECKS 72
5.4.1 Mobile Agents 73
5.4.2 Distributed Incremental Consistency Checking 74
5.5 INCREMENTAL CHECKING CHALLENGES 76
5.6 SUMMARY 77

CHAPTER 6 SOFTWARE AGENT ARCHITECTURE FOR DISTRIBUTED CONSISTENCY CHECKING 79

6.1 INTRODUCTION 79
6.2 ARCHITECTURE DESCRIPTION 80
6.2.1 Resource Interface Agent 82
6.2.2 Domain Agent 83
6.2.3 Gateway Domain Agent 85
6.2.4 Consistency Checking Mobile Agent 86
6.2.5 User Interface Agent 90
6.3 HIERARCHICAL INFORMATION STRUCTURE 90
6.3.1 Document Name Table 91
6.3.2 Distribution of Consistency Rules 92
6.3.3 Agent Lookup Table 94
6.3.4 Event List Table 95
6.3.5 System Policies 96
6.4 RE-CONFIGURATION AT RUNTIME 97
6.5 SATISFACTION OF THE FUNCTIONAL REQUIREMENTS BY ARCHITECTURAL COMPONENTS 97
6.5.1 Resource Interface Agent 97

6.5.2 Domain Agent 98
6.5.3 Gateway Domain Agent 99
6.5.4 Mobile Consistency Checking Agent 99
6.5.5 User Interface Agent 100
6.6 SUMMARY 100

CHAPTER 7 STATE TRANSITION MODEL OF THE SOFTWARE AGENT ARCHITECTURE 102

7.1 INTRODUCTION 102
7.2 THE MODELLING APPROACH 102
7.2.1 Construction of a State Transition Model 103
7.2.2 Modelling Tool - Covers 103
7.3 STATE TRANSITION MODEL OF THE SOFTWARE AGENT ARCHITECTURE 104
7.3.1 Document active object 104
7.3.2 Resource Interface Agent active object 107
7.3.3 Consistency Checking Mobile Agent active object 109
7.3.4 Domain Agent active object 111
7.3.5 Agent Middleware active object 113
7.4 EVALUATION RESULTS 113
7.5 SUMMARY 118

CHAPTER 8 SCENARIOS: DISTRIBUTED DEVELOPMENT OF THE BREAK PLANNER APPLICATION 120

8.1 INTRODUCTION 120
8.2 APPLICATION DOMAIN 121
8.3 AN OVERVIEW OF THE SCENARIOS 121
8.4 APPLICATION DEVELOPMENT TEAM 122
8.5 ANALYSIS AND DESIGN OF THE BREAK SCHEDULER APPLICATION 122
8.6 SCENARIO I: LOCAL CONSISTENCY CHECKING AT A HOST WITHIN A SINGLE DOMAIN 125
8.6.1 Event: Creation of a New Document 125
8.6.2 Response: Consistency Check 125
8.6.3 Result: Generated Consistency Links 126
8.6.4 Processing of the Event 127
8.7 SCENARIO II: DISTRIBUTED CONSISTENCY CHECKING WITHIN A SINGLE DOMAIN 131
8.7.1 Distributed Check: Generation of Consistency Links 131
8.7.2 Event: Document Change 132
8.7.3 Processing of the Event by the Software Agent Architecture 133
8.8 SCENARIO III: DISTRIBUTION OF THE BREAK SCHEDULER APPLICATION 137
8.8.1 Distribution of UML Model Elements 138
8.8.2 Inter-domain Agent Migration Policies 139
8.8.3 Distributed Inter-Domain Consistency Check 141
8.8.4 Inter-Domain Document Location Discovery 142
8.8.5 Multi-Agent Collaboration 145
8.8.6 Firewalls and mobile agent security 153
8.8.7 Disconnected operation 154
8.8.8 Replication 155
8.9 SUMMARY 156

CHAPTER 9 IMPLEMENTATION PROTOTYPE 157

9.1 INTRODUCTION 157
9.2 RESOURCE INTERFACE AGENT 157
9.2.1 Startup 157
9.2.2 Handled messages 158
9.3 DOMAIN AGENT 160
9.3.1 Startup 160
9.3.2 Handled Messages 160
9.4 MOBILE CONSISTENCY CHECKING AGENT 162

9.4.1 Startup ... 162
9.4.2 Handled Messages ... 162
9.4.3 Cloning Procedure ... 164
9.4.4 Migration ... 165
9.4.5 Redundant Consistency Checks ... 165
9.5 GATEWAY DOMAIN AGENT ... 166
9.5.1 Startup ... 166
9.5.2 Handled Messages ... 166
9.6 USER INTERFACE AGENT ... 166
9.6.1 Startup ... 167
9.6.2 Handled Messages ... 167
9.7 SECURITY ... 167
9.7.1 User-level Security ... 168
9.7.2 Execution-level Security of Mobile Agents ... 169
9.7.3 Migration-level Security of Mobile Agents ... 170
9.7.4 Message-level Security ... 171
9.8 SUMMARY ... 172

CHAPTER 10 EVALUATION ... 174

10.1 QUALITATIVE FEATURES ... 174
10.1.1 Elegance ... 174
10.1.2 Manageability ... 175
10.1.3 Rule Applicability Policies ... 176
10.1.4 Flexibility and Dynamic Reconfiguration ... 176
10.1.5 Grouping of Resources Into Domains and Support of Domain Hierarchies ... 177
10.1.6 Disconnected Operation ... 178
10.1.7 Support for Transactions ... 179
10.1.8 Balance of Requirements ... 179
10.2 QUANTITATIVE PERFORMANCE EVALUATION ... 180
10.2.1 Exhaustive Consistency Check vs. Incremental Check ... 181
10.2.2 Distributed Check vs. Centralised Check of Distributed Documents ... 184
10.2.3 Centralised Exhaustive Consistency Check vs. Distributed Incremental Check ... 193
10.3 SUMMARY ... 194

CHAPTER 11 CONCLUSIONS AND FUTURE WORK ... 195

11.1 CONCLUSIONS ... 195
11.1.1 The Software Agent Architecture for Distributed Consistency Checking ... 195
11.1.2 Architecture Model ... 195
11.1.3 Incremental Checking ... 196
11.1.4 Validation of the Architecture ... 196
11.2 OPEN QUESTIONS AND FUTURE WORK ... 197
11.2.1 Consistency Checking Framework ... 197
11.2.2 Integration of the Software Agent Architecture ... 198
11.2.3 Distributed Software Agent Architecture ... 198
11.2.4 Incremental Checking ... 198
11.3 CLOSING REMARKS ... 199

BIBLIOGRAPHY ... 200

APPENDIX A UML WELL-FORMEDNESS CONSISTENCY RULES ... 210

A.1 ASSOCIATIONS ... 210
A.2 ASSOCIATIONCLASS ... 212
A.3 ASSOCIATIONEND ... 213
A.4 BEHAVIORALFEATURE ... 214
A.5 CLASS ... 214
A.6 CLASSIFIER ... 215
A.7 COMPONENT ... 217

A.8 CONSTRAINT 218
A.9 DATATYPE 218
A.10 GENERALIZABLE ELEMENT 218
A.11 GENERALIZATION 219
A.12 INTERFACE 220
A.13 METHOD 221
A.14 NAMESPACE 222
A.15 TYPE 223

APPENDIX B STATE TRANSITION MODEL PSEUDO-CODE 224

B.1 DOCUMENT ACTIVE OBJECT 224
B.2 RESOURCE INTERFACE ACTIVE OBJECT 225
B.3 MOBILE AGENT ACTIVE OBJECT 226
B.4 DOMAIN AGENT ACTIVE OBJECT 229
B.5 AGENT MIDDLEWARE ACTIVE OBJECT 230

APPENDIX C BREAK PLANNER APPLICATION: SELECTED UML DIAGRAMS 231

C.1 ANALYSIS CLASS DIAGRAM 231
C.2 DISTRIBUTED BREAK PLANNER APPLICATION CLASS DIAGRAM 232
C.3 BREAK SCHEDULER APPLICATION - USE CASES 233

APPENDIX D REVIEW OF MOBILE AGENT FRAMEWORKS 234

D.1 D'AGENTS 234
D.2 MOLE 235
D.3 HIVE 236
D.4 CONCORDIA 237
D.5 GRASSHOPPER 238
D.6 MESSENGERS 239
D.7 TACOMA 240
D.8 TELESCRIPT 241
D.9 VOYAGER 242

APPENDIX E QUANTITATIVE PERFORMANCE MODEL OF THE ARCHITECTURE 244

E.1 INTRODUCTION 244
E.2 ASSUMPTIONS 244
E.3 INITIAL FINDINGS 245
E.4 LOCAL CONSISTENCY CHECK 245
E.5 DISTRIBUTED IN-DOMAIN CONSISTENCY CHECK 246
E.6 SINGLE AGENT DISTRIBUTED CHECK ACROSS MULTIPLE DOMAINS 246
E.7 MULTI-AGENT DISTRIBUTED CHECK ACROSS MULTIPLE DOMAINS 247
E.8 THEOREM ON MULTIPLE-AGENT DISTRIBUTED CONSISTENCY CHECKING 248
E.9 THEOREM ON MULTI-AGENT DISTRIBUTED AND LOCAL CENTRALISED CONSISTENCY CHECKING 250

APPENDIX F INCREMENTAL CHECKING: TECHNICAL CHALLENGES 253

F.1 SELECTION OF RELEVANT CONSISTENCY RULES 253
F.2 LIGHTWEIGHT RULE SELECTION APPROACH 254
F.3 REFINED APPROACH: SELECTIVE EXECUTION OF XPATH EXPRESSIONS 256
F.4 MERGING SETS OF CONSISTENCY LINKS 257
F.5 DISTRIBUTION OF UML MODELS 262

List of Figures and Tables

TABLE 4.1. TAXONOMY OF MOBILE AGENT SYSTEMS, PART 1. 53
TABLE 4.2. TAXONOMY OF MOBILE AGENT SYSTEMS, PART 2. 54
FIG. 4.1. SEQUENCE DIAGRAM FOR AGLET CLONING. 58
FIG. 4.2. EXAMPLE OF THE CLONING CODE, CREATING LINKED MASTER AND CLONE AGENTS. 59
FIG. 4.3. EXECUTION LOG OF THE EXAMPLE CLONING CODE. 59
FIG. 4.4. SEQUENCE DIAGRAM FOR AGLET MIGRATION. 60
FIG. 4.5. REVIEW OF REQUIREMENTS FOR A SUCCESSFUL MOBILE AGENT IMPLEMENTATION WITHIN THE CONSISTENCY MANAGEMENT DOMAIN. 61
FIG. 5.1. CONSISTENCY RULE: WELL-FORMEDNESS CONSTRAINT FOR "GENERALIZATION" ELEMENTS. 65
FIG. 5.2. THIS GENERALIZATION FULFILS THE CONSISTENCY CONSTRAINT. 65
FIG. 5.3. THIS GENERALIZATION DOES NOT FULFIL THE CONSISTENCY CONSTRAINT. 66
FIG. 5.4. INCONSISTENT LINK GENERATED AS A RESULT OF A CONSISTENCY CHECK. 66
FIG. 5.5. PSEUDO CODE OF THE EXHAUSTIVE CONSISTENCY CHECKING ALGORITHM. 67
FIG. 5.6. PSEUDO CODE OF THE INCREMENTAL CHECKING ALGORITHM. 70
FIG. 5.7. PSEUDO CODE FOR DISTRIBUTED INCREMENTAL CONSISTENCY CHECKING. 75
FIG. 6.1. A SINGLE DOMAIN OF THE SOFTWARE AGENT ARCHITECTURE FOR DISTRIBUTED CONSISTENCY CHECKING. 81
FIG. 6.2. CONSISTENCY CHECKING AGENT'S GOAL "SELECT" AND THE RESULT – THE AGENT'S ITINERARY 88
FIG. 6.3. EXTRACT FROM THE DOCUMENT NAME AND TYPE TABLE. 91
FIG. 6.4. STRUCTURE OF AN ELEMENT OF THE AGENT LOOKUP TABLE. 94
FIG. 6.5. AN ELEMENT OF AN EVENT LIST TABLE, REGISTERING THE EVENT "ONCHANGE". 95
FIG. 6.6. A CONSISTENCY RULE EXECUTION POLICY EXAMPLE IN XML. 96
FIG. 6.7. THE STRUCTURE OF A RULE EXECUTION POLICY. 96
FIG. 7.1. STRUCTURE OF THE SOFTWARE ARCHITECTURE MODEL. 105
FIG. 7.2. DOCUMENT ACTIVE OBJECT; ITS STRUCTURE AND BEHAVIOUR. 106
TABLE 7.1. PARTIAL CODE: DOCUMENT ACTIVE OBJECT 106
FIG. 7.3A. STRUCTURE OF THE RESOURCE INTERFACE AGENT ACTIVE OBJECT. 107
FIG. 7.3B. BEHAVIOUR OF THE RESOURCE INTERFACE ACTIVE OBJECT. 108
TABLE 7.2. PARTIAL CODE: RESOURCE INTERFACE AGENT ACTIVE OBJECT. 108
FIG. 7.4A. STRUCTURE OF THE CONSISTENCY CHECKING MOBILE AGENT ACTIVE OBJECT. 109
FIG. 7.4B. BEHAVIOUR OF THE CONSISTENCY CHECKING MOBILE AGENT ACTIVE OBJECT. 110
FIG. 7.5B. BEHAVIOUR OF THE DOMAIN AGENT ACTIVE OBJECT. 112
FIG. 7.6. AGENT MIDDLEWARE ACTIVE OBJECT: ITS STRUCTURE AND BEHAVIOUR. 113
FIG. 7.7. VARIATION OF MODEL PARAMETERS. 114

FIG. 7.8A. PERFORMANCE EVALUATION OF THE ARCHITECTURE MODEL. 116
FIG. 7.8B. PERFORMANCE EVALUATION OF THE ARCHITECTURE MODEL (CONTINUED). 116
FIG. 7.8C. PERFORMANCE EVALUATION OF THE ARCHITECTURE MODEL (CONTINUED). 117
FIG. 7.9. PERFORMANCE OF THE MODEL AND A CENTRALISED CONSISTENCY CHECKER. 118
FIG. 8.1. INITIAL STRUCTURE OF THE CDS DOMAIN. 122
FIG. 8.2. DISTRIBUTION OF XMI DOCUMENTS FOR SCENARIOS I AND II (SUMMARY). 122
FIG. 8.3A. DOCUMENT DISTRIBUTION (HOST A) 123
FIG. 8.3B. DOCUMENT DISTRIBUTION (HOST B). 123
FIG. 8.4. THE ANALYSIS CLASS DIAGRAM CONSIDERED IN THE SCENARIOS I AND II. 124
FIG. 8.5. RELEVANCE OF UML WELL-FORMEDNESS RULES TO MODEL ELEMENTS. 125
FIG. 8.6A. INITIAL VERSION OF CLASS TEACHER. 126
FIG. 8.6B. INCONSISTENT LINK BETWEEN AN OPERATION IN THE CLASS AND THE CLASS DESCRIPTOR. 127
FIG.8.7. DOCUMENT CHANGE IS INDICATED IN THE TREEDIFF. 128
FIG. 8.8. ACTIVITY LOG OF THE PROTOTYPE: SELECTION OF RELEVANT CONSISTENCY RULES. 128
FIG. 8.9 MOBILE CONSISTENCY AGENTS RECEIVE ITINERARIES FROM THE DOMAIN AGENT. 129
FIG. 8.10. ACTIVITY LOG OF THE PROTOTYPE. LOCAL LINK GENERATION FOR RULE "C1" 130
FIG. 8.11. CONSISTENT LINK BETWEEN A NAMESPACE AND THE ENDS OF AN ASSOCIATION. 131
FIG. 8.12. TREEDIFF: AN ASSOCIATION END ELEMENT IS ADDED TO THE UML MODEL. 132
FIG. 8.13. ACTIVITY LOG OF THE PROTOTYPE AT HOST A: IDENTIFICATION OF CHANGES, RELEVANT RULES, PROCESSING OF LOCAL RELEVANT DOCUMENTS, MIGRATION TO HOST B 133
FIG. 8.14. SEQUENCE OF CLONING AND MIGRATION ACTIONS IN A DISTRIBUTED CHECK 135
FIG. 8.15. ACTIVITY LOG OF THE PROTOTYPE AT HOST B: PROCESSING OF LOCAL RELEVANT DOCUMENTS, LINK GENERATION, PROPAGATION OF LINKS TO HOST A. 135
FIG. 8.16. ACTIVITY LOG OF THE PROTOTYPE AT HOST A: STORING OF CONSISTENCY LINKS, DISPOSAL OF THE PARENT MOBILE AGENT. 136
FIG. 8.17. INDIVIDUAL STATISTICS FOR THE AGENT FAMILY (PARENT AND ALL CLONES) 137
FIG. 8.18. DOCUMENT DISTRIBUTION FOR SCENARIO III. 139
FIG. 8.19. MULTI-DOMAIN HIERARCHY OF NETWORK HOSTS. 139
FIG. 8.20. SUMMARY OF THE POLICIES FOR INTER-DOMAIN CONSISTENCY CHECKS. 141
FIG. 8.21. ROUTING POLICIES FOR MOBILE AGENT INTER-DOMAIN CONSISTENCY CHECKS 141
FIG. 8.22. INTER-DOMAIN LOCATION DISCOVERY PROCESS EXAMPLE FOR CONSISTENCY RULE "GENERALIZATIONS" 143
FIG. 8.23. RESULT: INTER-DOMAIN ITINERARY FOR RULE "GEN1", CHECKED FROM THE CDS DOMAIN. 143
FIG. 8.24. XMI SOURCE OF GENERALIZATION "ORGANISERIMPL IMPLEMENTS ORGANISER" AND THE RESULTING INCONSISTENT LINK. 144
FIG. 8.25. MULTI-AGENT COLLABORATION SCENARIO FOR CONCURRENT CHECKING OF A RULE AT EACH PARTICIPATING HOST. 146

FIG. 8.26 INCONSISTENT LINK: TWO COPIES OF THE SAME ASSOCIATION EXIST IN THE UML MODEL. . 147
FIG. 8.27. GENERATION OF SUB-ITINERARIES. 147
FIG. 8.28. STAGE 1 OF MULTI-AGENT COLLABORATION: EXECUTION OF SUB-ITINERARIES. 148
FIG. 8.29. STAGE 2 OF MULTI-AGENT COLLABORATION: COLLECTION INTO THE REPOSITORY. 148
FIG. 8.30. STAGE 3 OF MULTI-AGENT COLLABORATION: GENERATION OF LINKS AND THEIR PROPAGATION. 149
FIG. 8.31. COMPLETE ITINERARY OF DOCUMENTS, RELATED TO CONSISTENCY RULE "N1", FROM ALL PARTICIPATING DOMAINS 150
FIG. 8.32. CONCATENATION OF ITINERARIES AND NODESETS FROM REDUNDANT CONSISTENCY CHECKING AGENTS..... 151
FIG. 8.33. PARTIAL ITINERARY FOR CHECKING RULE "GEN1" AT THE DISCONNECTED DOMAIN BCL... 155
FIG. 10.1. TIMINGS OF EXHAUSTIVE CHECKS OF UML WELL-FORMEDNESS CONSISTENCY RULES. 182
FIG. 10.2. CHECK TIMES FOR INCREMENTAL AND EXHAUSTIVE CHECKS..... 183
FIG. 10.3A. CHECK TIMINGS OF DISTRIBUTED CONSISTENCY CHECKS BY NUMBER OF HOSTS..... 186
FIG. 10.3B. CHECK TIMINGS - PLAIN GRAPH WITH TREND LINES. 186
FIG. 10.4. PERFORMANCE COMPARISON OF DISTRIBUTED AND CENTRALISED CHECKS OF RULE A1. 188
FIG. 10.4B. PERFORMANCE COMPARISON: PLAIN GRAPH WITH TREND LINES..... 188
FIG. 10.5. EXECUTION OF CONSISTENCY RULE: WELL-FORMEDNESS OF NAMESPACE. 190
FIG. 10.6. EXECUTION OF CONSISTENCY RULES: WELL-FORMEDNESS OF ASSOCIATIONS..... 190
FIG. 10.7. EXECUTION OF CONSISTENCY RULES: CLASSES AND BEHAVIOURAL FEATURES..... 191
FIG. 10.8. PERFORMANCE OF CONCURRENT MULTI-AGENT CHECKS..... 192
FIG. 10.8B. PERFORMANCE OF CONCURRENT MULTI-AGENT CHECKS. 192
FIG. 10.9. PERFORMANCE OF BEST AND WORST DISTRIBUTED CHECKING APPROACHES COMPARED WITH CENTRALISED CHECKING OF DISTRIBUTED DOCUMENTS..... 193
FIG. F.1. EXAMPLES OF COMPLEX XPATH EXPRESSIONS FROM THE UML RULE BASE. 254
FIG. F.2. ALGORITHM FOR FINDING INTERSECTION BETWEEN TWO XPATH EXPRESSIONS..... 255
FIG. F.3. CONSISTENCY RULE SELECTION SCENARIOS. 256
FIG. F.4. RESULTS OF PROCESSING XPATH EXPRESSIONS..... 256
FIG. F.5. ALGORITHM FOR MERGING LINKSETS..... 258
FIG. F.6A. EXAMPLE DOCUMENTS FOR DEMONSTRATION OF THE XPATH FRAGILITY PROBLEM..... 258
FIG. F.6B. LINKS BETWEEN THE TWO DOCUMENTS..... 259
FIG. F.7A. CHANGED DOCUMENTS. 259
FIG. F.7B. INCREMENTAL LINK BASE. 260
FIG. F.8. XPATH UPDATE ALGORITHM. 261
FIG. F.9. XPATH EXPRESSION MAPPING HASHTABLE. 261
FIG. F.10. UPDATED LINK IN THE OLDER LINKBASE..... 261
FIG. F.11. A CHANGED LINK IS DETECTED AFTER LINKBASES ARE UPDATED AND MERGED. 262
FIG. F.12. MODEL ELEMENTS AND XPATH EXPRESSIONS TO THEM..... 262

FIG. F.13. CLASS ELEMENT, EXTRACTED FROM A UML MODEL 263
FIG. F.14. DISTRIBUTION OF UML MODELS INTO SEPARATE XMI DOCUMENTS. 264

Chapter 1 Introduction

The size and complexity of current software-intensive systems necessitates their distributed and collaborative development. In this setting, a number of participants in a joint effort produce documents, which are often required to follow certain restrictions on their format and content. These restrictions define relations between the documents that are required to hold by a standard, a development process, a company policy or another factor. For example, software system descriptions, expressed in the Unified Modelling Language (UML), are required to follow the well-formedness constraints, defined by the UML standard [OMG 2000b].

In many cases, produced documents have "overlapping" content, as they refer to common objects or phenomena. The overlap gives rise to *consistency relations* between documents and their parts referred to as document elements. In a variety of cases, the developers intend to constrain these relations, due to the software development process used, the document structure or semantics demanded, or project requirements. In this thesis, we use the term *consistency management* to describe the process, by which existing consistency relations between overlapping documents are checked for conformance to specified constraints.

The task of consistency management is complicated by the distribution of documents and their concurrent collaborative development by numerous distributed participants. With the expansion of the Internet, company intranets and distributed collaboration technologies, company-wide distribution of developers has already become widespread, and there exists a growing trend in adoption of the geographically distributed, Internet-scale development. Existing centralised approaches to consistency checking cannot cope with a scenario of checking the mutual consistency of documents in the distributed setting of this scale.

This thesis addresses the problem of checking consistency relations between distributed documents. We build a distributed architecture, consisting of autonomous collaborating components, which automatically provide the consistency checking service at the distributed document locations. This architecture is proposed as an alternative to the traditional approach, where checks are inherently centralised. The developing software agent technologies and code mobility are making feasible the construction of the new architecture.

As a motivating scenario in this thesis, we consider the problem of consistency management in the software engineering domain.

1.1 Motivating Scenario

As an application domain and a benchmark for the proposed software agent architecture for distributed consistency management, we consider consistency checks between distributed software engineering documents. A running scenario of throughout the thesis deals with the design of a break scheduler application, discussed in [Bergner, et al. 1997]. In the scenario, a team of distributed participants is concurrently developing on a UML model [Booch, et al. 1999] of the software application. The scope of consistency relations, demonstrated in the scenario, is that of checking well-formedness constraints [OMG 2000b] of the UML model throughout the development process.

Checking of well-formedness constraints of UML models has been chosen as the application domain due to variety and complexity of the UML constraints. The approach to distribution of UML model elements for collaborative development and distributed checks allowed us to generate evaluation scenarios of significant scale for larger UML models.

1.2 Contribution

This thesis builds on the approach for expression of consistency relations between documents, the concept of consistency checks of these relationships, and the method for representation of the state of relationships between individual documents [Zisman, et al. 1999]. We draw on the recent development of this framework – a novel XLinkit rule language [Nentwich, et al. 2001a] and the framework for carrying out consistency checks of well-formedness consistency relations in UML models [Nentwich, et al. 2000b].

The XLinkit framework follows the approach, which originates in software development environments [Donzeau-Gouge, et al. 1984, GOODSTEPTeam 1994, Habermann and Notkin 1986, Reps and Teitelbaum 1981] and provides the facilities for carrying out centralised consistency checks, where documents need to be stored at, or moved to, the location of the consistency checker.

The growth in distributed software development in industry and in the software engineering community has initiated re-consideration of the centralised architectures. With respect to the existing tools, such as XLinkit, most significant concerns arise in the scalability of the centralised approach, and in its usability for incremental development, since consistency checks have to be initiated by a user and upon every check, all distributed documents have to be copied to the central location and check results need to be returned to the locations, where development takes place.

This thesis proposes a distributed architecture, where consistency checking is carried out locally at the network hosts where documents are located. We introduce a distributed checking algorithm, which separates checks of documents in space and time and executes at numerous locations. The approach enables concurrent checking at numerous locations and improves efficiency and security of checks by eliminating the need for transfer of complete documents across the network during consistency checks.

The architecture facilitates distribution by scaling out, through introduction of self-contained autonomous domains and forming of a domain hierarchy, and by scaling up with the increase in a number of hosts and documents in each domain. The concept of domains – groups of related documents allows us to tackle large scale distribution and offer a de-centralised solution, capable of delivering an efficient consistency checking service. Disconnected operation of autonomous domains enables us to improve fault tolerance and achieve stability by use of persistent components. In order to fulfil its task, the distributed architecture builds on software agency and mobility in construction the architectural components.

In a brief summary, this thesis makes the following contributions:

- Development of an architecture for distributed consistency checking, which capitalises on locality of access to documents.
- An incremental checking algorithm, which extends the capabilities of existing exhaustive checker and provides better support for the incremental development process. We have addressed technical challenges of extending the current non-incremental checker XLinkit to provide support for incremental checks. Having achieved interoperability between the checkers, we consider them as complementary and have suggested deployment scenarios for both checkers
- Development of a model of the architecture, where components' operations are specified via state charts. This model allows us to better explain component roles, to provide initial verification of the architecture, and to give an initial estimation of its performance characteristics.
- Construction of an implementation prototype of the architecture, which we have evaluated on a number of scenarios, demonstrating its scalability and explaining in detail the features provided by the architecture. The chosen scenarios follow distributed cooperative development of a software application by a number of participants.
- We have suggested a number of approaches to multi-agent collaboration and concurrent execution of checks, which aim at improving performance of the distributed consistency checks, and gave recommendations on optimal configurations of the distributed system.
- We have carried out a performance evaluation of the architecture and compared it with the centralised checking approach. This evaluation has allowed us to highlight the distribution configurations, in which the distributed checks are most effective.

1.3 Thesis structure

Related work to consistency checking in the areas of ViewPoint software engineering, software engineering environments, and the XLinkit consistency framework follows in Chapter 2. Chapter 3 sets out the primary considerations - functional requirements for construction of a distributed consistency management architecture. Chapter 4 gives an introduction to software agency and mobility concepts, on which the software agent architecture draws. We also provide a taxonomy of mobile software agent

frameworks, specify our requirements and select a framework, on which the architecture and its prototype are constructed.

Chapter 5 introduces incremental checking development for the XLinkit framework, and considers in detail a centralised algorithm and a distributed, incremental consistency checking algorithm. By describing the checking process, this chapter sets a context for the following description of the architecture.

Chapter 6 gives a high level overview of the proposed software agent architecture for distributed consistency checking. It describes the roles of the architectural components and elaborates on fulfilment of the functional requirements, demanded in Chapter 3, by each of the components.

A model of the architecture, outlining internal operations of each component in state transition diagrams, is presented in Chapter 7. The model allows us to give a detailed demonstration of internal operations of the components and their interaction, and gives an initial performance estimation of distributed consistency checks.

A number of scenarios, occurring during the development of a UML model of a break scheduler application, are introduced in Chapter 8. The scenarios are extensively developed for distributed checks in different distribution configurations, where events, triggering consistency checks, actions and interactions of components during the checks, and results of the checks are described in detail. These scenarios are a result of deployment of an implementation prototype of the software agent architecture, which serve as validation of the architecture.

Chapter 9 lays out the internal structure of the implementation prototype that was informally introduced on scenarios in Chapter 8. In Chapter 9, each component of the event-driven architecture is described in terms of event generation and handling.

The software agent architecture is evaluated in Chapter 10. Qualitative evaluation outlines advantages and disadvantages of the architecture, and refers to the functional requirements, demanded of the architecture in Chapter 4. Quantitative performance evaluation of the implementation prototype uses a UML design example and contrasts the traditional centralised checker with distributed incremental checks in the software agent architecture.

Quantitative and qualitative evaluations in Chapter 10 complement the evaluation and validation of correctness of the implementation prototype, carried out on examples from the running scenario in Chapter 8. Chapter 9 also plays a part in evaluation, and serves to demonstrate the event-orientation feature of the proposed architecture.

Conclusions in Chapter 11 outline the strengths and weaknesses of the proposed software agent architecture approach. The conclusions confirm the primary argument of this thesis, that distributed consistency checking is effectively carried out by the proposed software agent architecture. Recommendations for future work are also given in the chapter.

Chapter 2 Related Work in Consistency Checking

Techniques for expressing and checking consistency constraints are found in many areas of computer science. This chapter gives a summary of major work in consistency management: the Viewpoints framework and programming and software development environments. Here we also describe XLinkit - the framework for consistency checking, on which the software agent architecture of this thesis builds. The review of existing methods accentuates comparisons between proposed approaches, and aims to establish the position of the architecture, proposed in this thesis.

2.1 The ViewPoints Background

A number of researchers have worked on formal techniques for viewpoint specification [Ainsworth, et al. 1994, Boiten, et al. 1996, Finkelstein, et al. 1992, Frappier, et al. 1995, Larsen, et al. 1995, Zave and Jackson 1996]. Use of different abstractions when reasoning about complex systems has been recognised as an effective way of separating concerns. Not surprisingly, many other disciplines involved in information systems development have come up with similar approaches. View-integration in conceptual database design has been a widely researched topic in the 1980s [Navathe, et al. 1986]. The use of separation of views in requirements engineering even dates back to the late 1970s [Mullery 1979]. More recently, viewpoints have been proposed and researched for program development environments [Meyers 1991] and information systems design [Baldwin 1993].

Within the software and requirements engineering communities, numerous researchers have been working on the "multiple perspectives problem" [Finkelstein, et al. 1992, Kotonya and Sommerville 1992, Nuseibeh 1994, Reeves, et al. 1995, Spanoudakis, et al. 1997]. By this term they refer to the problem of how to organise and guide software development in a setting with multiple actors, using diverse representation schemes, having diverse domain knowledge and different development strategies. A framework has been developed [Finkelstein, et al. 1992, Nuseibeh 1994] in order to address the diverse issues related to this problem. Multi-perspective software development within this framework is referred to as Viewpoint Oriented Software Engineering (VOSE).

2.2 Viewpoint-Oriented Software Engineering

There is a growing awareness in distributed software engineering that the development of complex software systems can no longer be seen as a linear, top-down activity [Steen 1998]. The ViewPoint Oriented Software Engineering framework (VOSE) [Finkelstein, et al. 1992] advocates structuring of the specification and development of such systems according to, so called, ViewPoints. The VOSE

framework [Finkelstein, et al. 1992, Nuseibeh 1994] considers consistency relations within the context of ViewPoints.

ViewPoint models allow developers to split up the complete specification of a complex system into a number of viewpoint specifications, each concentrating on a particular concern or aspect of the system. The specifications are potentially distributed, as they belong to different participants. Each viewpoint then represents a particular abstraction, either from a stakeholder's, or a modelling perspective.

2.2.1 ViewPoints

A ViewPoint combines the notion of 'actor', 'role' or 'agent' in the development process with the idea of a 'perspective' or 'view', which an actor maintains. ViewPoints are *defined* as loosely coupled, locally managed, distributable objects; thus containing identity, state and behaviour. A ViewPoint is more than a 'partial specification'; in addition, it contains partial knowledge of how to develop that partial specification.

Each ViewPoint is composed of the following components, called slots:

- The *representation style* defines the notation or language to be used in that ViewPoint.
- The *work plan* defines the actions that can be performed on specifications in the given style, and their recommended ordering. Different kinds of actions are identified within the framework. Most obvious are the assembly, or "editing", actions. Of particular interest to us in this thesis are consistency checking actions, which we consider below.
- The *domain* is a label, identifying the area of concern of that ViewPoint.
- The *specification* slot contains the actual description of the identified domain, in the notation defined in the style slot.
- The *work record* describes the actions that were performed on the ViewPoint specification. It thus defines the development state of the ViewPoint.

Typically, different ViewPoints will share the same notation and development process. Creation of a multiple-viewpoint system often occurs by instantiation of 'ViewPoint templates'. A template is a ViewPoint, in which only the style and work plan slots have been filled in. Domain, specification and work record slots of a template are then filled in during instantiation, thus allowing for dynamic ViewPoint creation from templates. Creation of new ViewPoints may be the result of certain viewpoint actions, which are a part of their work plans. Such as, a translation action would transform a ViewPoint from one notation into another by creating a new ViewPoint-translation.

2.2.2 Consistency of ViewPoints

One of the main problems in any multiple ViewPoint approach to specification is defining and establishing consistency of various ViewPoint specifications. This problem becomes particularly

challenging when we consider that different specification techniques may be applied to different ViewPoints.

Consistency in this setting can be seen as a relationship between viewpoint entities. In the viewpoint model for specification and development of software, consistency is defined as a set of syntactic constraints. For example, the Unified Modelling Language (UML) standard [Booch, et al. 1999], based on the Booch method [Booch 1991] for object oriented design, requires UML models to comply with a significant number of static constraints [Appendix A] in order to be well-formed. In addition to static constraints, techniques for tackling behavioural, or semantic consistency, which would provide consistency management foundations for State Charts and Sequence Diagrams by means of process algebraic specification techniques, have been studied in [Steen, et al. 1999].

The ViewPoints framework distinguishes two kinds of inconsistencies: in-ViewPoint semantic inconsistency and inter-ViewPoint inconsistency. Consistency constraints are specified by consistency rules [Easterbrook, et al. 1994]. In-ViewPoint consistency rules define the constraints that should hold in order for a ViewPoint specification to be 'well-formed'; these define the static semantics for the ViewPoint's representation style. Inter-Viewpoint rules define relationships between the representation styles of different ViewPoints, thus identifying areas of "overlap" between the ViewPoints.

Consistency rules define partial consistency relationships between the different specification notations. Specifications are consistent if all the rules that should hold between particular viewpoints actually do hold. The general form of a consistency rule in [Easterbrook, et al. 1994] is the following:

$\forall VP_S, \exists VP_D : VP_S \Re VP_D$, where VP_S – is a "source ViewPoint", VP_D – a "destination ViewPoint", and $\Re$ is a relationship, which must hold in order for the consistency constraint to be satisfied. It is envisaged that these consistency rules will be specified using an appropriate logic [Finkelstein, et al. 1994], where representation in first order logic is a possibility.

In-Viewpoint and inter-Viewpoint checks can uncover inconsistencies in and between specifications that prevent a set of specifications from being globally consistent. However, self-consistency and pair-wise mutual consistency are in general not sufficient to obtain global consistency. It is suggested, that deficiencies of such pair-wise specification of constraints can be remedied by construction of a macro-ViewPoint, containing a graph of all other ViewPoints [Easterbrook, et al. 1994]. Constraints are then composed on the graph, specifying relationships between its nodes.

The consistency checking actions and inconsistency handling actions are contained in the work plan of each ViewPoint. The handling actions are resolution actions that may be performed in the presence of inconsistency.

Although the method defining consistency in ViewPoints is very expressive, it has a number of drawbacks. Firstly, the approach requires the designer to define all possible consistency relations between all possible combinations of viewpoint templates to be able to identify the global consistency status. Secondly, a case may arise that all specification notations may not be suitably expressed in the same logic. Thirdly, the advocated development model is highly fluid and new types of viewpoint may be created at any time.

The consistency checking techniques, defined for the VOSE approach, are focusing mainly on the *structure* of specifications. Typical inter-Viewpoint rules require that for each viewpoint of a particular type there exists a specification component in a viewpoint of some other type.

Consistency checking techniques have been proposed, which focus more on the specification *content*, in particular the specified behaviour. In the PROST report [Cowen, et al. 1993] it is suggested that specifications are consistent if there exists a common refinement. An analysis of these variations of consistency has resulted in a generalised definition of 'semantic consistency' based on the notion of common refinement [Bowman, et al. 1996]. It has also been shown that this general definition encompasses the notion of logical consistency [Bowman, et al. 1995].

An important similarity between the two approaches is that both allow inconsistencies between viewpoint specifications to exist. This is in contrast with, for example, program development environments [Meyers 1991], where consistency is maintained at all times. VOSE provides for development processes to be considered explicitly. As such, in VOSE, ViewPoint templates can be defined, which prescribe a specification notation and a work plan, where the latter includes policies for carrying out consistency checks and inconsistency handling actions, allowing toleration of inconsistency.

2.2.3 Consistency Management Activities

In application of the VOSE framework to management of interference relations between the viewpoints, [Finkelstein, et al. 1996] introduce a number of interference management steps or activities. An *interference* consistency relation between viewpoints arises when goals of actors or viewpoint owners are mutually interdependent (for example, at the requirements engineering stage). [Finkelstein, et al. 1994] admit that interference is inevitable and acceptable in system development: inevitable as a consequence of multiple perspectives, and acceptable in support of innovation in development, fulfilment of commitments and consideration of alternatives. As a notion, embodying a broad class of consistency relations, interference management is subject to complexities due to difference of viewpoints held by their owners, use of different languages or heterogeneous tools, existence of numerous levels of development and elaboration, as well as degrees of formality and granularity.

The process of resolution of interference relationships [Finkelstein, et al. 1994] comprises the following significant activities:

Overlap identification for recognition and classification of system components, which refer to common objects or phenomena in the domain of discourse. This essential step sets the stage for future definition of consistency relations between the viewpoints, and is concerned with comparison, analysis and formalisation of the views of participants. In general, overlap identification would require some form of user input and cannot be fully automated, since certain types of references are usually implied or not evidently found within the components themselves. While being one of the starting steps of the interference management process, overlap identification is based on existence of somewhat elaborated

viewpoint templates, which determine the essence, types of components envisaged to be deployed in the system.

Construction of the instances from the templates, *ViewPoint instantiation,* is the activity of transformation of classified system component templates into viewpoints. The development of viewpoint templates reflects similarities in component structure, content or function. Instantiation of templates with specific information taken from system components enables to trace overlaps between particular viewpoint instances.

Consistency relation construction involves the creation of consistency relationships between different viewpoints to reflect consistency overlaps. Construction of relationships can be automatic and dynamic (on-the-fly), on demand (for example, as a result of a performed consistency management check), or embedded (baseline in-system relations). The relations can then be specified in terms of particular elements of the structures of the viewpoints, and expressed in first-order logic, as overlap between sets or in another way, which is convenient and efficient given a language that information in the viewpoints is expressed in and the structure of this information.

The three consistency management activities described – viewpoint instantiation, overlap identification and consistency relation construction constitute the basic activities for consistency management, set out in the ViewPoints framework. The scope of this thesis considers overlap identification and consistency relation construction. Leaving the viewpoint instantiation and inconsistency resolution to the developers, we concentrate on the consistency checking activity.

2.2.4 Policies

Depending on the required level of consistency enforcement in the system, it is advantageous to set the policies, governing execution of checks of consistency relations between the viewpoints and use of the results of these checks for reconciliation. *Policy specification* is an identification and composition of appropriate policies for execution of checks of consistency relations. Originating from the standards compliance domain, policies are considered by us as a useful abstraction for instrumentation of Viewpoint consistency checks. One successful realisation of the notion of policy has been developed within the Standards Compliance Manager project [Armitage, et al. 1998, Emmerich, et al. 1999], where each consistency relation is associated with at least one policy, which defines:

- An indication of the consistency *relation*, the management of which the policy is concerned with.
- A type of *event* that will trigger the checking of a consistency relation. In an example, where viewpoints are represented by documents that are being modified by developers, creation or removal of a document and modifications to a document would constitute those events, which should trigger checks or some consistency relations.
- A *mode* that the policy is applied in. In our example, in case if inconsistencies are identified, the user – owner of an inconsistent viewpoint - will be notified ("guideline" mode), or interrupted

and allowed to continue his present activity ("warning" mode), or prevented from continuing until viewpoints have been changed and are consistent again ("error" mode).

- A type of *diagnosis* that is displayed. Depending on the particulars of a system under consideration and the type of the user, a user can be either presented with a listing of inconsistencies found (links between related documents and document elements), or a statistical output of the number of inconsistent elements.

A policy can be specified in terms of high-level strategies or goals. Preventive policies (such as an "error" mode policies) involve the immediate rejection of an action, whose completion would cause the occurrence of an interference inconsistency. Remedial policies are those in which interference triggers resolution actions. These are diversified by toleration policies, which define the scope and extent of toleration of interference (choice of "guidance" or "warning" mode affects the extent of toleration).

Preventive policies parallel the traditional perspective on conflict [Meyers 1991] in studies of organisational behaviour, where conflict is a malfunction within an organisation or a group and as such it should be avoided. By contrast, remedial policies reflect the behavioural perspective in which conflicts are the natural result of individuals and groups each pursuing their own interests and objectives within an organisation, and compromises need to be achieved in order to resolve them.

Policy is a useful configuration concept for consistency checks. Policies have triggered our special interest, because they relate system events, which trigger checking of a consistency relation to the consistency relation, to a check of this relation in an appropriate application mode and to a specific diagnosis – result of a consistency check. Self-containability of checks results from coupling of the results of a policy execution, which explicitly expresses the connection between the ViewPoint slots.

2.3 Software Development Environments

The consistency management technology for incremental development is provided in the syntax-directed tools [Donzeau-Gouge, et al. 1984, Reps and Teitelbaum 1984] of the software development environments. These environments offer document production support in such a way, that document accesses and updates are issued in terms of commands, and each command corresponds to an increment. Since commands in syntax-directed tools are directed towards the syntax of the underlying language, it is possible to immediately establish the "propagation" of a consistent relation through all opened documents (or an inconsistent relation when inconsistencies are tolerated). As a result, such tools can be effectively used in the incremental document development process.

The commands, that syntax-directed tools perform, internally correspond to operations on a particular abstract syntax tree [Habermann and Notkin 1986, Reps and Teitelbaum 1981]. Additional attributes of this tree often contain semantic information. Therefore, static semantic checking of a document, represented as an abstract syntax tree, can be carried out by evaluating the mentioned attributes along the paths in the tree [Knuth 1968], which are computed at tool construction time based on required attribute dependencies. For inter-document consistency checks, based on attribute

evaluations, an artificial root is appended to the forest of abstract trees of individual documents. In this case, consistency constraints are checked starting from the root node. The framework for consistency management, XLinkit [Nentwich, et al. 2000b] uses a similar approach, in which constraints are checked on a macro-graph of the trees of document elements.

Abstract syntax graphs generalise the concept of the abstract syntax trees [Johnson and Fisher 1982, Nagl 1985] by using of non-syntactic paths for implementation of semantic relationships between syntactically unconnected parts of different documents, which may even correspond to different types. Representing a set of documents in a project by an abstract syntax graph allows one to carry out static semantic analysis, checking of consistency relations and navigation between separate documents along the paths in the project-wide graph.

Among the first syntax-directed tools developed were the Cornell Program Synthesizer [Reps and Teitelbaum 1981], tools developed in the Mentor project [Donzeau-Gouge, et al. 1984], and the Gandalf project [Habermann and Notkin 1986]. In terms of architectures, all of these tools maintained the document abstract syntax trees in main memory and did not provide sufficient support for persistent document storage. As a result, inter-document constraints could not be checked between different document types. Additionally, none of the tools supported version and configuration management.

Next generation syntax-directed tools from the IPSEN environment [Engels, et al. 1987, Nagl 1985] provided persistent document representation based on the proprietary GRAS database model [Lewerentz and Schurr 1988], where all documents were stored in a single graph. This graph effectively constituted a document "universe", where each document was represented by a subgraph with reference edges leading to other documents' subgraphs. The architecture enabled checks of static semantics and inter-document consistency constraints. Additionally, transaction support was provided via the GRAS transaction mechanism: any information loss was limited to the last completed command-increment. An enhanced version of the IPSEN prototype also supported document revision control [Westfechtel 1989], based on the functionality offered by the GRAS database system. The GRAS database system used a proprietary data model called the *graph pool*, that has been explicitly defined for storing abstract syntax graphs. While the discussed syntax-directed tools clearly benefited from the proprietary GRAS database system providing persistent document storage, today's challenge is to be able to deploy an open standard for document representation.

2.4 Meta-CASE Tools

Computer Aided Software Engineering (CASE) tools provide automated tool support for a range of software engineering activities. Software engineering methods, which outline these activities, provide techniques for specification, implementation, quality assurance, coordination, management and other stages of the development process [Nuseibeh 1995]. Meta-CASE technology [Alderson 1991] is designed to provide automated support for a part of the development process, by taking a formal (or a

precise) description of the software development method as an input and producing CASE tools as its output.

Meta-CASE tools rely on rule checking systems and techniques [Welland, et al. 1990] for processing of, typically graphical, method notations and their relationships [Nuseibeh 1995]. One example is a commercial meta-CASE tool – the IPSYS Tool Builder's Kit (TBK) [Alderson 1991, Alderson and Elliott 1989], which allows a tool developer to specify the development methods and to define the syntax and semantics of method notations.

The Methods Workbench Tool, formerly known as Virtual Software Factory (VSF) [Metasoft 2001] allows construction of a semantic model of the method in a special-purpose language, a syntactic definition of textual and graphical views, and a kernel environment for execution of the formal method definition for production of a CASE tool for that method.

The capability of meta-CASE tools to provide the functionality for integration of methods, notations and tools is traditionally supported by a common centralised repository. Both the TBK and VSF are also based on a shared data access model.

2.5 Attribute Grammars

Attribute grammars have been suggested for the specification of semantics of context-free languages [Knuth 1968] and were successfully used for the definition of static semantics. A subclass - ordered attribute grammars [Kastens 1980], allows one to determine at compile-time whether the static semantic evaluations terminate. Ordered attribute grammars form the basis for the specification language of the Cornell Synthesizer Generator [Reps and Teitelbaum 1988].

Equations in the attribute grammars, however, can be defined using attributes based on the syntactic structure of *one* language. Consistency constraints between documents of different types can only be simulated by considering one common type and representing all participating documents as children of an artificial root. In this case, concurrent editing of different documents by multiple users could not be supported by existing syntax-directed tools: all inter-document consistency checks would modify attributes of the root, thus necessarily causing concurrency conflicts.

At the same time, attribute grammars give developers such flexibility, that efficient and incremental evaluation of static semantics equations is automatically derived from dependencies between equations by the Synthesizer Generator [Reps and Teitelbaum 1984]. The evaluation sequence can achieve efficiency of one assignment per attribute [Kastens 1980]. Furthermore, the Synthesizer Generator is capable of *incremental evaluation* of consistency rules, where only attributes, changed since the last evaluation, and the transitive closure of attributes, depending on those changed ones, are evaluated.

2.6 Specification Languages

2.6.1 CENTAUR

Mentor-Project [Donzeau-Gouge, et al. 1984] and its successor, the Centaur system, support the prototyping of languages by providing a language framework, which can be used to define a language in terms of syntax, static and even dynamic semantics. Rule-based language TYPOL [Despeyroux 1988] is used for expression of semantics. Analogous to attribute grammars, TYPOL is not capable of expressing inter-document consistency constraints, as the universe of defined expressions is limited by the abstract syntax specification of one language.

2.6.2 PROGRESS

PROGRESS, a general purpose specification language for defining graph grammars, evolved from the IPSEN [Lewerentz and Schurr 1988, Nagl 1985] project, where graph grammars were used for the definition of abstract syntax graph structures [Engels, et al. 1987]. Use of PROGRESS enables expression of structure and operations on syntax graphs. As a general purpose language, it can obviously not provide the specific support for tool specification that the Centaur language provides.

2.6.3 GOODSTEP

A serious weakness of the early programming environments like the Cornell Synthesizer Generator, CENTAUR and Mentor is in use of attribute grammar languages or TYPOL to check semantic validity of statements. The languages constrained the consistency checking power of these environments and did not support inter-document consistency checks. The proposed approach of maintaining the abstract tree structure of a "super-document" leads to problems with distribution of documents and with scalability, since any modifications trigger consistency checks on the whole central structure.

GOODSTEP [GOODSTEPTeam 1994] succeeds the discussed systems by enabling expression of constraints across different types of documents and allowing wider integration of heterogeneous development tools. Semantic rules, specified in the imperative GoodStep Tool Specification Language (GTSL), allow use of general purpose control statements in the action block. While giving power to the language, this makes consistency rules verbose.

2.7 XLinkit

XLinkit [Nentwich, et al. 2000b] is a language for expression of consistency relations between document types, based on simplified first order logic. The declarative form that the language is based on removes the need for general-purpose control statements (that are used, i.e., in GOODSTEP), and allows developers to concentrate on constraints, rather than having to deal with the language overhead.

XLinkit provides a consistency relation specification language and a tool, succeeding the consistency link generator [Ellmer, et al. 1999]. The latter provided a foundation for a framework, which specifies a way of evaluating consistency language expressions against a set of documents and generating hyperlinks between related document elements, which reflect the consistent or inconsistent status of a consistency relation, specified in the expression. As a result, when the language expression is satisfied, a consistent link is created between corresponding elements of the documents, participating in the relation. Otherwise, an inconsistent link is created, which alerts developers that the consistency relation is not currently being satisfied.

This approach in identification of inconsistencies [Ellmer, et al. 1999] [Nentwich, et al. 2000b] follows the trend of inconsistency toleration rather than necessary resolution, which originates in [Balzer 1991]. Balzer proposed a tolerant approach to inconsistency in databases, which uses pollution markers to identify inconsistent data and exclude them from integrity checks. Furthermore, the inconsistency is assumed and allowed, and the focus shifts towards identification of the inconsistencies.

In the scope of this thesis, we consider a problem of consistency management from the viewpoint of sole identification of inconsistencies, which is in itself a significant problem in software engineering. Our concentration on this scope is complemented by our determination to construct a consistency checking tool, which will "manage" distributed documents by identifying inconsistencies and notifying the developers accordingly. Correction or toleration of inconsistencies is left to the developers themselves, and depends on a number of factors, including the stage of the project in its lifecycle, degree of collaboration between developers and an individual development style.

The consistency link generator and its successor XLinkit are built to check inter-document relations between heterogeneous documents of different types. The framework is based on an assumption that documents, participating in consistency checks, can be represented in the eXtensible Markup Language (XML) [Bray, et al. 1998]. XML has gained acceptance in the software development world as an open standard mechanism for bridging data heterogeneity problems, and in many application areas this assumption is already being supported by developing standards and recommendations. For instance, the examples in this thesis consider software engineering documents, described in the Unified Modeling Language (UML) [OMG 2000b], which is supported by the eXchange of Model Information (XMI) [OMG 2000c] standard. This standard provides the storage of MOF-compliant models in the XML format and includes a document type definition (DTD) for the UML as an example application.

The XLinkit framework includes support for consistency checking in the software engineering domain: a complete set of well-formedness rules for UML models with respect to the UML standard [OMG 2000b] has been specified in the XLinkit language [Nentwich, et al. 2001a] and appears in [Appendix A]. Demonstrative UML examples in this thesis were chosen, because they contain a variety of complex consistency relations and explore the expressiveness of the rule specification language.

2.7.1 Link Generation

Consistency rule language of XLink uses XPath [Clark and DeRose 1999] expressions to specify the paths to document elements. When these expressions are executed on document DOM trees, values of elements and their attributes are used by the XLinkit checker to evaluate rule language operators and compute a consistent or an inconsistent status of a consistency relation among a set of documents.

Results of consistency checks consist of n-ary hyperlinks, called *consistency links*. A set of elements that violates a rule is linked using inconsistent links. A link to the consistency rule is included to specify what relation is violated and to aid the developer in reconciliation of an inconsistency.

Consistency links are built using XLink [DeRose, et al. 2000], which allows to link document elements without inserting the actual links into the documents. XLinkit stores links "out-of-line" in separate files called link sets or *linkbases* [Ellmer, et al. 1999, Nentwich, et al. 2000b].

2.7.2 XLinkit and Distribution Support

The consistency link generator and its successor XLinkit provide functionality for centralised checking of distributed documents by means of transferring complete documents across the network for each consistency check. The XLinkit architecture performs link generation at a centralised location [Nentwich, et al. 2000b]. A number of potentially distributed configuration files are used to specify the *document universe* [Ellmer, et al. 1999] - a set of documents, participating in the consistency check. Each document in the universe is identifiable via a URL, and all distributed documents are downloaded by the XLinkit tool each time the consistency check is carried out. A web interface to XLinkit [Nentwich, et al. 2000a] allows users to upload the documents to be checked, and once again results in the transfer of complete documents across the network.

Centralisation of consistency checks and the necessity to transfer complete documents to the checker and resulting consistency links back to the user for each check constitute major weaknesses of the XLinkit architecture. XLinkit evaluations [Nentwich, et al. 2000b, Nentwich, et al. 2001a, Nentwich, et al. 2001b] concentrate on performance of local consistency checks, where documents are located at the same host as the checker. Distribution scalability concerns of the architecture, posed by real-world concurrent project development in a distributed team, have not yet been addressed. It is clear, however, that the centralised checker constitutes a central point of failure, and for a distributed project of a sufficient size, repeated document transfers will use up the network resources.

2.7.3 Exhaustive Consistency Checking

At the moment of writing, the current XLinkit tool prototype does not include support for incremental checks and supports exhaustive checking only. This thesis proposes a principal algorithm for incremental consistency checks on top of the XLinkit consistency checking framework. We have provided and evaluated an initial implementation of this algorithm, which is integrated in the

implementation prototype of the software agent architecture for distributed consistency checking. Work is underway to integrate incremental checking into the XLinkit prototype as well.

Although incremental checks could improve XLinkit scalability issues, document distribution issues would still remain unanswered. This thesis proposes a software agent architecture for distribution of consistency checks, which avoids centralisation of checks and capitalises on local access to distributed documents in order to improve efficiency. This component-based architecture benefits from software agency and relies on agent mobility to fulfil its task. Related work in software agents and mobility is discussed in Chapter 4.

2.7.5 Current Work in the Thesis and the Related Work

This thesis inherits much of the history of research in ViewPoints [Finkelstein et al., 1992][Easterbrook et al., 1994]. The software agent architecture for distributed consistency checking is based on and extends functionality of the XLinkit consistency checking engine. The exhaustive checker functionality of XLinkit [Nentwich et al., 2000b] and the consistency rules for UML well-formedness constraints [Appendix A] form the work, on which the author has based the research.

Incremental checking algorithm [Chapter 5], which extends functionality of the exhaustive checker, the software architecture for distributed consistency checking of documents in the software engineering domain [Chapter 6], its model [Chapter 7] and the implementation prototype of the architecture [Chapter 9] are original contributions of the author. A number of scenarios, on which the architecture is evaluated [Chapter 8] originate from the study of the distributed development project [Bergner et al., 1997], but were re-engineered for the distributed setting.

2.8 Summary

The domain of this thesis is application of the consistency identification methodology, described in this chapter to consistency checking in the distributed setting. ViewPoint-oriented software engineering approach provides us with a clear representation of loosely coupled, locally managed and distributable objects. The model of ViewPoints – potentially distributed partial specifications, is a coherent abstraction of distributed software engineering objects, undergoing collaborative development. We have described in- and inter-ViewPoint consistency checking techniques, and referred to the approach of toleration of identified inconsistencies, rather than mandatory resolution, which we adopt in this thesis.

In this chapter, we have set the scope for the concept of consistency management activity in the context of this thesis. Drawing on related research in management of inference relations, we have set the focus of our dissertation on overlap identification and consistency relation construction between distributed ViewPoints. We also referred to policy specification activities, developed in the standards compliance domain, which outline a consistency checking process by explicitly relating an occurrence

of an event to the subsequent check of a relevant consistency relation and to the resulting feedback diagnosis that establishes a consistency status of the original event.

In the remainder of the chapter we have described existing consistency checking tools in the software development environment domain and the XLinkit framework for consistency checking. XLinkit provides support for heterogeneity of checked documents and platforms on which checks are executed, expresses consistency relations in a rule language, where general rules relate to document types rather than instances, and generates out-of-line consistency links as a diagnosis. The framework is concerned with overlap identification and consistency relation construction, and is based on toleration of inconsistencies.

However, XLinkit only supports exhaustive consistency checking functionality, and cannot execute consistency checks incrementally, which is a standard feature in a number of software development environments. The XLinkit architecture necessitates the transfer of complete documents from distributed document locations to the location of the checker for every consistency check, and the upload of resulting consistency links to the location of every participant.

In this thesis, we have developed a distributed checking architecture, which capitalises on locality of access to distributed documents, and aims to eliminate disadvantages of a centralised approach. We build on the consistency checking functionality, provided by XLinkit, as a foundation and extend it by constructing a distributed consistency checking architecture, which would eliminate inherent centralisation and provide support for incremental consistency checks. The following chapter outlines the functional requirements for such architecture, which are inspired by the review of the existing consistency checking technologies.

Chapter 3 Requirements for Distributed Consistency Checking

This chapter defines the functional requirements for consistency checking between distributed heterogeneous documents in the software engineering domain. These requirements are being put forward as a foundation for development of the incremental consistency checking technology (Chapter 5) and for the design of the software agent architecture for distributed consistency management (Chapter 6).

The functional requirements take their origin in the analysis of problems in existing consistency checking frameworks (Chapter 2). The main rationale behind the requirements is to elaborate on the principle of locality of access to documents. This rationale establishes that distributed, intra-document consistency checks can be carried out at the locations of the documents, without a need for networked transport of each document as a whole to a remote location for centralised processing. The principal contribution of this thesis - a distributed architecture for consistency checking, builds on this rationale.

3.1 Stakeholders

Stakeholders, participating in a distributed, collaborative software development project, are the target users of the system for consistency checking. This brief introduction of the stakeholders precedes a discussion of functional requirements for the distributed consistency checking architecture.

Each stakeholder authors a number of documents, which are located at the stakeholder's workstation that is connected to the network, or at a remote network host. Local area network (LAN) connectivity is the most common configuration, although support for other connectivity configurations, involving a number of connected LANs and intermittent connectivity is also provided (Chapter 8, Scenario III).

Documents that the stakeholders produce often constitute different points of view on the system under development. As a running example throughout this thesis, we use a scenario, where a UML model of a software application is being collaboratively developed by a number of software engineers. In this example, we check well-formedness of the model, where the consistency relations of interest to us are the required relations between elements of the model. These relations stem from static semantic constraints, demanded by the UML standard.

Stakeholders concurrently undertake development of groups of model elements, which are represented as documents at stakeholders' individual workstations. Throughout development, stakeholders introduce modifications to the existing documents, add and delete documents from the project. Movement of documents between network hosts is considered as a composite of the addition and removal actions.

3.2 Specification of Inter-Document Relationships

Consistency relationships between documents should be expressed in a declarative form. The specification of relationships should not refer to particular document instances, but to document types.

Both requirements formed a construction base for the consistency checking framework XLinkit [Nentwich, et al. 2000b]. This thesis builds on the infrastructure, provided by XLinkit, and therefore we use these requirements as a common "denominator" for the distributed consistency checking architecture as well. Because this requirement is satisfied by the underlying XLinkit framework, below we give some brief detail as to how this requirement is implemented.

Specifying consistency relationships between document types is a natural way to provide a generalization of the specified relationships. Within an application domain, where a set of document types and their structure have been established, satisfaction of the requirements provides flexibility and capability of reuse of the declarative consistency rules between individual projects within the domain.

In the XLinkit consistency framework, a consistency rule language is used for expression of consistency relations [Nentwich, et al. 2000b]. The language uses first-order logic, which gives it sufficient expressiveness to specify a variety of constraints in the software engineering domain [Nentwich, et al. 2001a]. The XLinkit rule language fulfils both requirements, because constraints are specified between particular elements of document types. Representation of documents in XML requires a document type definition DTD, restricting otherwise arbitrary element sets and specifying allowed element trees. For navigation of those trees, XPath [Clark and DeRose 1999] expressions are used in the rule language constraints.

In a running scenario throughout this thesis, we use a set of consistency rules, specified in XLinkit language. The rule set [Appendix A] describes UML well-formedness constraints [OMG 2000b].

3.3 Location of Consistency Rules and Their Applicability

Consistency rules should be stored and accessed locally to the document, which they are applicable to. In other words, when a certain consistency rule is applicable to types of documents stored at a certain location, it should be made available at that location.

This requirement for consistency rules follows the principle of local access to documents. At any network host, containing documents and relevant consistency rules, all intra- and some inter-document checks can be carried out within the limits of this host. Resulting autonomous operation and increased fault-tolerance with respect to network failures serve as a basis for support of "disconnected operation".

The requirement is conceptually similar to the principles of the ViewPoints framework, where in-ViewPoint and inter-ViewPoint consistency constraints form a part of each ViewPoint.

A policy defines applicability of consistency rules. Applicability should not be confused with *relevance* of a consistency rule to a particular document, where the latter is dependent on the document type and the consistency constraint.

Policies [Chapter 2, 2.2.4] enable or disallow execution of a consistency rule for a particular document and can define inter-dependency of rules by specifying a sequence of rules to be executed. Policies would be used in a situation, when in addition to execution of *relevant* consistency rules, an additional level of abstraction in specifying rules for execution becomes necessary.

3.4 Distributed Document Monitoring and Change Identification

Each document, participating in the managed project, should be monitored for occurrence of events, such as document changes. Removal and addition of new documents from the managed project should also be monitored.

When a document change event is identified, it must be determined, which elements of the document's structure have been changed, and which have been added or removed.

These requirements stem from the need to provide continuous, real-time monitoring of documents and identify the relevant changes once they occur. Fulfilment of this requirement sets a basis for construction of a "reactive" consistency checking system, where consistency checks can be automatically executed, following document changes. Reactivity would enable a software development environment to provide continuous feedback to the developers by identifying inconsistencies and issuing appropriate notifications. XLinkit does not provide an infrastructure for reactive checking: developers must submit documents for checking to a central server.

3.5 Timing of Consistency Checks

Consistency checks occur at appropriate points in the document production process, as a result of events occurring on documents.

This requirement expands on the requirement of "reactivity". In addition to events – document changes, addition and deletion of documents, which can be used to trigger consistency checks, developers must be able to request execution of checks at certain stages of the development. In a common example, comprehensive checks of all consistency rules would be executed at the end of each project iteration. Request for consistency rule execution is thus another type of event, which can be used to control timings of execution of consistency checks.

3.6 Incremental Consistency Checks

Consistency checks should be carried out incrementally.

Non-incremental checkers execute all consistency rules on all documents, regardless of the nature and extent of modifications made since the last consistency check. Current implementation of XLinkit operates in this manner.

At the same time, a small change in a single document should not cause a need for execution of all rules, as we expect a large number of consistency rules and documents in any software engineering

application. An incremental check in this case would execute only those consistency rules, which are relevant to the particular change.

3.7 Relevance of Consistency Rules

A mechanism should be in place, which would identify relevance of a consistency rule to a given document, or to an element of the document's structure.

A first step in incremental checking, identification of relevant rules must find out whether a consistency relationship, represented by a consistency rule, has anything to do with a given document or particular elements of its structure. A particular implementation of the incremental checking algorithm would be dependent on the language used for expression of consistency rules and on the structure of documents. Chapter 5 is dedicated to description of the proposed incremental consistency checking approach, which serves as the underlying consistency checking framework of the distributed software agent architecture.

The basis of an incremental checking concept is expressed in the requirement for identification of relevance of consistency rules and the requirement for identification of document changes.

Among numerous concurrent changes occurring to the documents in a document set, the incremental checker tracks each individual change and identifies particular consistency rules, which are relevant to that change. Execution of relevant rules across the complete set of participating documents determines the status of consistency relations, in which this particular document is involved. Among numerous relations established as a result of an incremental check, all relations, involving changed elements, will be present. It is precisely these relations we consider of importance to our stakeholders.

3.8 Document Access Locality

Access to documents and retrieval of document elements during a consistency check should occur locally with respect to the document, rather than from a remote host across the network.

Architectural de-centralisation and a number of other advantages (i.e., improved security and scalability) justify this requirement [Finkelstein and Smolko 2000, Smolko 2001]. In this section, we briefly point out some significant advantages of access locality, while a performance evaluation of an implementation of the proposed approach is given in Chapter 10.

Large software engineering projects contain documents of significant size; therefore, it is most often impractical to transport these documents to a checking server for every consistency check. In addition, intellectual property is better protected when access locality is deployed, since, unlike centralised checking of distributed documents, local checking at distributed locations ensures that contents of documents involved is never transmitted across the network in the complete form.

3.9 Representation of Results of Consistency Checks

> Results of consistency checks must provide diagnostic information on the found consistencies and inconsistencies. This may take the form of references to related documents and the relevant elements of these documents.

The results of reactive incremental checks are a step in enabling pro-active response to detected inconsistencies. In a fully pro-active scenario, automatic (or user assisted semi-automatic) inconsistency resolution process would follow identification of inconsistencies.

In this thesis, we take a view of *toleration* of identified inconsistencies, originating from [Balzer 1991], and concentrate on providing developers with diagnostic information, which is an outcome of a consistency check. Based on this information, developers may choose to resolve inconsistencies by introducing further changes to the documents when appropriate.

This requirement underlies the XLinkit consistency checking framework, where consistency *links* are constructed automatically and are a result of consistency checks [Ellmer, et al. 1999, Nentwich, et al. 2000b].

3.10 Update of Results of Consistency Checks

> Results of consistency checks must be kept up to date. Timings of updates depend on the development process.

Updates of consistency links are carried out by re-checking consistency rules on a participating set of documents. This mechanism keeps established consistency links up to date with any document changes. Proposed incremental checks are triggered by individual document changes, but we would not wish to constraint developers to a reactive link update mechanism only. In practice, links should be updated at least after completion of every stage in an iterative development process.

In the distributed software agent architecture, requests for consistency checks are handled through system events. Document changes are represented as events, and the distributed watchdog monitor [Finkelstein and Smolko 2000] serves the basic purpose of identification of changes occurring on the documents. In a similar manner, scheduled consistency link updates are triggered by scheduled events. The following requirement is concerned with notification and processing of events.

3.11 Event Notification and Processing

> An event-oriented consistency checking system should provide facilities for distributed event notification and processing.

Below we outline events of most relevance to the status of consistency constraints, which occur in a distributed consistency checking architecture. This event list is not as exhaustive as the list of system events of the implementation prototype (Chapter 9), but gives a practical highlight of the concept of an event within the scope of this thesis.

- A change has occurred on a resource and the document has been saved.
- A new resource has been added to the system, deleted or moved in the distributed system.
- A resource has been found consistent or inconsistent with respect to a constraint.

A number of other events are characteristic to a particular implementation of the software agent architecture.

By defining roles of different architectural components, the distributed consistency checking architecture delegates to different components the monitoring of conditions, which trigger the events, notification of the events to relevant components, and processing of these events. In our implementation of the architecture, events are mapped to messages, which are exchanged between components.

Event-orientation enables loose coupling between components. Message processing interfaces, standardised across all components, ensure interoperability at the event notification level. Internal structure of each component is then not relevant with respect to interoperability, as long as notified events are processed and responses are generated according to an established message sequence (protocol).

In contrast to loosely coupled event-oriented components, the consistency checking infrastructure of XLinkit is based on a tightly integrated set of classes. A public interface allows a developer to specify a set of consistency rules and a set of documents. Finer granularity access is not supported; modification of source code was required in order to enable use of inner XLinkit objects by externally built components. However, tight integration has enabled XLinkit developers to achieve performance benefits, since class interfaces are optimised to data formats exchanged and no conversion is required between classes.

The scope of this thesis is consistency checking in the distributed setting, and our goal was to build a system with a degree of interoperability between heterogeneous components. At design time, distribution overhead was considered an inevitable performance impediment, and conversion between raw data and serializable events was thus tolerated. Performance evaluation of the implementation in Chapter 10 has confirmed this hypothesis.

In addition to interoperability advantages, event-oriented architectures benefit from flexibility of re-configuration. Due to the asynchronous nature of event notification, components can be added or removed at system runtime, without a need to re-start the system as a whole. Deployment of persistent event queues provides support for disconnected operation, which is a beneficial advantage for intermittently connected or mobile stakeholders. Disconnected operation ability and re-configuration flexibility improve overall fault-tolerance of a distributed system.

3.12 Summary

In this chapter, we have set out the main functional requirements for the distributed consistency checking architecture that we construct in this thesis. In the next chapter, we define software and mobile agents and consider their characteristics. We survey existing software agent systems, and outline

advantages that agent systems possess in contrast with traditional distributed architectures. In fulfilment of the functional requirements for a distributed consistency checking architecture, we draw on advantages of software agency and mobility.

Chapter 4 Software Agents and the Mobile Agent Paradigm

Design and deployment of agents is a rapidly expanding field of interest, research and development within mainstream computer science. In this chapter, we introduce agents as both a technical concept and a term and show that although there is no recognised and agreed upon definition for the term *agency*, this concept can be defined through its common characteristics, or features.

There are different views on mobile agent technologies. The first view looks at the motivations behind the agent paradigm, highlighting some of the fundamental technologies that are emerging. The second view presents sets of characteristics, which agents could and should possess. These are requirements for software agents and mobile agents, which are going to be deployed in the distributed consistency management architecture presented in Chapter 6. The final view on mobile agents details a comparison of agents according to the types of actions and different behaviours that are expected of them. We follow this view by carrying out a review of the mobile agent frameworks, presented in this chapter, and by mapping of the features of existing mobile agent engines onto the specified requirements for a mobile agent framework, suitable in the context of this thesis.

This chapter is constructed to the following outline. Before we give an overview of existing mobile agent systems and outline the requirements for a mobile agent, we set the scene by expanding on the definition of software agency through classification of agents' features. Then, we consider advantages and disadvantages of mobile agents' use, and provide our point of view on why use of mobile agents in the distributed consistency management architecture is beneficial. We present a taxonomy of mobile agent frameworks, where a number of agent systems are evaluated. We outline a brief set of requirements for a mobile and software agent framework to be used in an implementation of the software agent architecture. Finally, based on the requirements, we make a provision for selection of IBM Aglets as the agent framework of choice for the development of the distributed consistency management system.

4.1 What is a software agent?

The task of giving a general description to a *software agent* is difficult in the absence of a common definition. It is not always clear what comprises an agent and what does not, which often gives grounds for loss of coherence

The most basic understanding of an agent is "one who takes action" [Laurel 1990]. [Kay 1984] expands the notion with goal-orientation of agents:

... when given a goal, [an agent] could carry out the details of appropriate computer operations and could ask for and receive advice, offered in human terms, when it was stuck. An agent would be a "soft robot", living and doing its business within the computer's world.

Further, agency has become pictured as central to the concept of artificial intelligence (AI) [Wooldridge and Jennings 1995]:

... One way of defining AI is by saying that it is the sub-field of computer science, which aims to construct agents that exhibit aspects of intelligent behaviour.

The problem of producing a definition of agency is in part a result of a wide variety of subject areas in which agents are being applied. Agents and agent technologies are deployed in such diverse fields as telecommunications to assist creating of electronic market places [Magedanz, et al. 1996], personalised shop assistants [Geddis, et al. 1995b], tailored information retrieval [Lieberman 1995] and human-computer interaction [Chin 1991].

Requirement for a software agent's characteristic of intelligence typically appears in modern agent definitions. In practice, however, software agent applications achieve varying levels of complexity of the agent's decision making engine. Most notably, agent intelligence is exhibited in the agent's characteristic – autonomy. Autonomous software agents execute in order to achieve a certain *goal* on *behalf* of the *user*. Autonomy is a unifying characteristic of existing agent applications, and thus, we believe, constitutes an important requirement for identification of a software agent.

In distributed computing environments, achievement of goals often requires a software agent to access distributed information resources. Traditional distributed computing architectures rely on *mobile data* being retrieved from distributed resources and transferred for processing elsewhere. By contrast, the concept of *mobile code* advocates migration of code across the network and its execution at the locations, where required data is available.

Software agents, which use code mobility to effectively achieve their goals in distributed computing environments, are referred to as *mobile agents.* This concept of having migrating software agents to carry out tasks on behalf of their owners is a relatively novel paradigm for network-enabled distributed computing.

Despite different application areas and implementation details, mobile agent applications share common features. In terms of purpose, these applications seek to utilise a network of distributed resources (e.g., knowledge and data bases, documents and other information resources) in an opportunistic way in order to solve a problem for an end-user. In terms of architecture, a mobile agent follows a static or a dynamically generated *itinerary,* which specifies a list of locations to be visited and actions to be carried out at each location in order to achieve the agent's goal.

An already broad range of application fields and implementations of software agents continues to expand, the question of definition of agency shifts from 'what are agents', to questions 'what typifies agents' and 'what are their common features', which are addressed in the following section.

4.2 Characteristics of software agents

Characteristics of software agents in a way define the term 'software agent' due to lack of a formal common definition. The characteristics are classified into various notions of agency [Franklin and Graesser 1997, Nwana and Wooldridge 1996]. In this section we describe *weak* and *strong* agency notions, by setting out the characteristics that typify them. These notions set a basis for distinguishing feature sets of software agent frameworks, contrasted later in this chapter, and are referred to in the remainder of the thesis.

4.2.1 Weak Agency

An agent is considered to adhere to a *weak* notion of *agency* if it supports all of the following characteristics.

Autonomy. The life span of an agent typically includes a "launch" or instantiation, and a period of activity. Upon instantiation, an agent is assigned a goal, describing the bounds and limitations of its task. In the period of activity, the agent should be able to operate independently from the user, taking steps to achieve the goal autonomously, "in the background" [Castelfranchi 1995]. In this regard, an agent needs to have control over its actions, so that it can develop a contingency plan, should any planned action fail. In such situations, an agent must be able to change (to a certain limit) or extend its goal, and make rational decisions based upon the information it has gathered.

Communication ability. In order to query the agent's environment or make changes to it, an agent must possess the ability to communicate with the outside world [Genesereth and Ketchpel 1994, Mayfield, et al. 1995]. This interaction can exist at a number of levels depending upon the specifics of the agent's function, but typically an agent would need to communicate with other agents and the local environment (to discover or manage information), as well as with the users.

Reactivity. Agents need to be able to perceive their environment and respond to changes to it in a timely manner, where response would depend on an agent's goal. As an example of a reactive agent, consider a watchdog agent with a task of monitoring the file system on a certain network host and informing a user when changes occur to a particular set of files. This goal implies, that the agent has knowledge about the access mechanism to the respective file system and is able to track the state of and changes in the environment.

Pro-activity. When trying to achieve a goal in a changing environment, a software agent must be able to take the initiative. Pro-activity it related to autonomy; in an example when a planned action fails, an autonomous agent will generate contingency plans, and a pro-active agent would then execute these plans, and provide the results of execution as a feedback to the planning algorithm. Thus, pro-activity is demonstrated through appreciation of the state of agent's environment, and by exploiting the current state effectively in achievement of the agent's goal.

Supporting the weak agency in a software agent application is not a trivial task. However, a number of software agent frameworks that we review in this chapter allow us to construct weak software agents.

In development of a distributed software agent architecture, we are building architectural components on weak agency features. We have introduced those in this section, and will refer to them in the discussion of the software agent architecture and throughout the thesis.

4.2.2 Strong Agency

Strong agency expands the notion of weak agency and includes agent characteristics, which are usually attributed to humans. Strong agency refers to ontological concepts of knowledge, intention and goal-oriented obligation and extends beyond the requirements for weak agents with the following characteristics.

Intelligence. Intelligence (and thus, reasoning and understanding) may be attributed to both weak and strong agents, to a different extent. It determines how agents behave in different situations and react to certain events. In practice, intelligence is associated with an ability to learn, and development of an artificial personality through learning.

Learning in the notion of agency is achieved through storage of facts and experiences, acquired by an agent during its lifespan. Strong agency implies that such knowledge can be shared and improved overtime by communities of software agents.

Situatedness. Software agents execute in the real world, rather than in a model of the environment. Any model or description, no matter how extensive, can rarely exhaustively cover the span of real life situations. Situated agents have an ability to adjust the model of the environment, which they follow in their execution, for the model to be consistent with the real world.

From the described strong agency features, the distributed agent architecture we develop in this thesis makes use of learning, where multiple agents co-operate and exchange previously acquired knowledge. Construction of an intelligent agent personality and full support for situatedness is outside of the scope of this thesis. In general, support for strong agency in practical applications typically requires substantial a-priori knowledge of the application domain before the underlying agent framework can be constructed. Use of generic AI engines in software agent applications is not common, because the decision making system needs to be "trained" to the particular application domain.

4.3 Agent Mobility

A stationary software agent executes only on the system, where it begins execution, and interacts with other systems by means of messaging or other transport [OMG 2000a]. By contrast, a *mobile* agent can be described as having the unique ability to transport itself from one system in a network to another. This ability permits an agent to move to a destination system that contains an object, with which the agent wants to interact [OMG 2000a].

Agent mobility is noted as an ability of a software agent to move between different network hosts in fulfilment of its goals [Gray 1996b]. Research in mobile agents, agent architectures and applications continues to be a flourishing field, dedicated to investigating the potential of this new paradigm

[Mobility 2001]. Despite the popularity, the research field is only beginning to gain maturity [Picco 1998].

Similarly to stationary software agents, there is no commonly agreed definition for a mobile agent. In this thesis, in consistency with a software agent's definition, which requires autonomy, we follow the definition of a mobile agent from [Papaioannou and Edwards 1999] that similarly requires *migration autonomy*:

> A *mobile agent* is "a software agent that is able to autonomously migrate from one host to another in a computer network."

With respect to mobility itself, mobile agent frameworks provide support for the two following characteristics of mobile agents: strong and weak mobility.

Strong mobility is characteristic of mobile agent systems, which allows an agent to transfer its code and the state of execution of this code with all relevant data and the program counter. Execution of an agent is suspended during migration and continues at the destination host at exactly the point, where it was suspended.

Mobile agent systems supporting *weak mobility* automatically transfer the agent's code only. The execution state and any data used by an agent have to be programmatically packaged up (*serialized*) before migration and de-serialized after arrival at the destination host. It is the programmer's responsibility to provide data serialization and structure the code in a way, which would enable code execution to continue where it was suspended by migration. The majority of mobile agent frameworks are able to provide weak mobility.

4.4 Distributed System Construction With Mobile Agents

In the previous sections, we have given a definition to software agency and mobile agent paradigm through their behavioural characteristics. This section builds an architectural contrast of mobile agents with traditional distributed middleware. We discuss location transparency supported by distributed middleware, location awareness and resource access locality inherent in the mobile agent concept. We argue, that mobility inherits a number of advantages from locality that are significant for construction of distributed architectures. Further, we outline how we can capitalise on these advantages in development of a distributed architecture for consistency checking.

4.4.1 Traditional Distributed Systems

In this thesis we consider a distributed system as a collection of autonomous hosts, which are connected through the network as defined in [Emmerich 2000]. Hosts and components operate a distribution middleware, which enables co-ordination and resource sharing within the system, such that users and components *perceive* the distributed system as a single, integrated computing facility.

The central abstraction of modern distributed systems is *location transparency*, with inter-component communication being achieved via an intermediary request broker. The broker creates a

notion of presence of distributed components executing on the same machine. Remote calls to distributed components and exchange of data between these components are handled by the broker. The broker establishes a client/server relationship between the interacting parties, making it unnecessary for the parties to be aware of each other's location.

The underlying communication mechanism supporting contemporary distributed systems is *mobile data*. Remote Procedure Calls (RPC) [Colouris, et al. 2000] was a historic step forward from inter-process communication methods in that RPC facilitates a request-reply interaction between two *distributed* processes [Simon 1996]. The calling procedure executes in one computing machine, and the called procedure executes in another [Cerutti and Pierson 1993], whilst *data is exchanged* between the two communicating parties. The RPC infrastructure attempts to create an illusion that the processes exist within the same virtual machine.

The RM-ODP model [Linington 1995] was an attempt to unify proprietary RPC systems. Its specification extends the concepts of transparency first visited by RPC, and identifies eight separate forms of transparency [Colouris, et al. 2000]. Among those are *location transparency* (enables distributed objects to be accessed without knowledge of their location) and *access transparency* (enables distributed objects to be accessed with the same operations).

Since RM-ODP, a number of distributed object frameworks have been developed on the RM-ODP principles: Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) [OMG 1995], Remote Method Invocation (RMI) [RMI 1998] and Distributed Component Object Model (DCOM) [Box 1998, Redmond 1997]. All of these frameworks base on and the concepts of location and access transparencies, and make use of mobile data.

Despite apparent success of traditional distributed systems, [Waldo, et al. 1994] argues that objects in a distributed system are intrinsically different to those in a local system and therefore must be treated very differently. Waldo et al. identified four major problem areas outlined below, which are inherent in the architecture of traditional distributed systems.

- *Partial failure* is a significant problem in distributed computing, where if a programmer is to take advantage of location transparency, the behaviour must be the same for local or remote systems. [Sloman and Kramer 1987] argue, that this can be costly and difficult to achieve, especially in the face of failures.
- As a program cannot control the invocation order in a sequence of calls, *concurrency control* measures have to be introduced, which add overhead to the programming model.
- Call types for local and remote *memory access* have to differ. Processes cannot share the same memory space or use pointers.
- Latency of remote calls can degrade performance in orders of magnitude in certain applications.

4.4.2 Mobile Agents

One significant advantages of the mobile agent paradigm is enabling of the *access locality.* In contrast with RPC-style middleware, mobile agent architectures do not mask location information from the communicating components, but instead make it apparent. There is no request broker to mediate communication; one component requires explicit knowledge of location of another object in order to engage in interaction. In addition, instead of accessing another component *across* the network, a mobile component migrates to the relevant host and interaction occurs *locally* at that host.

The mobile paradigm allows components to decide how to make most effective use of access locality, and thus to partially alleviate some of the problems of traditional distributed systems outlined by [Waldo, et al. 1994]. A comprehensive architectural comparison between mobile agents, remote computation, code on demand and client-server models [Papaioannou 2000] argues that location and access transparencies are not an abstraction of a single virtual machine, distributed across numerous hosts. Mobile agent paradigm is considered a successful adaptation of inter-process communication to distribution, since local communication, a natural consequence of mobility, allows programmers to take advantage of traditional programming concepts in inter-process communication.

Major differences between the mobile agent paradigm and traditional distributed systems have been highlighted by [Picco 1998]. Mobility is a choice of an application developer or of the agent itself during runtime, instead of being unilaterally decided upon and triggered by the middleware framework. Programming is location-aware at application design time, and location information is available to the components at runtime. In this respect, mobile agents are better prepared for internet-scale execution, which necessitates coping with low bandwidth, network failures, latency, questions of trust and inherent heterogeneity. In addition to load balancing, mobile agent systems can sustain disconnected operation, demonstrate autonomy, fault-tolerance and provide certain flexibility.

The following section elaborates on the mentioned advantages of the mobile agent paradigm and considers them in a context of construction of a distributed consistency checking architecture.

4.4.3 Use of Mobile Agents: Pros and Cons

It would be way beyond the scope of this thesis to draw a final conclusion to an argument on the best approach to construction of a distributed system. As outlined in introductory Chapters 1 and 2, we focus instead on building support for distributed consistency checks.

The previous sections of this chapter defined software agency and carried out a comparison of traditional distributed systems and mobile agent paradigm. This section follows suit and considers significant advantages and disadvantages of mobile agents with respect to construction of a distributed architecture for consistency checking.

Simpler architectural design

Mobile agent architectures are built on the principle that a service is executed at the location where it is required. Successful construction of a mobile agent necessitates its construction in such a way, which enables the agent to take into account the information about its location at runtime. Having designed an agent, which is able to perform its goal at a given location, and adding itinerary planning facilities to this agent enables the agent to function in a distributed environment. A single agent design can then be used to provide the required service at numerous locations.

The application domain of distributed consistency checking maps well onto the mobile agent paradigm. As we discussed in Chapter 2, distributed consistency checking involves multiple potentially distributed stakeholders and their views on the developed system. In practice, stakeholders store and edit representations of their views at numerous hosts, distributed across the network. Once a view changes, a stakeholder would delegate the consistency checking task to an agent. This autonomous agent would undertake migration across the network, visit the hosts, where related views are located, and come back to the stakeholder with a report on any found inconsistencies.

In addition to the advantage of architectural design, mobile agent approach to distributed consistency checking is able to capitalise on numerous additional advantages, most importantly from the scalability point of view due to access locality and local information exchange efficiency. We elaborate on these advantages below.

Self-containability of mobile agents

Mobile agents transport the data, which they collect in achievement of their goal, with them during migration. When all necessary data has been collected by an agent and access to resources at the current location is no longer required, data processing can occur at any host in the network, which provides the agent with an execution environment. This advantage accounts for certain flexibility in distribution of agents at runtime.

Self-containability gives rise to the capability of mobile agents to exchange collected information, which facilitates achievement of their goals. Multi-agent co-operation and information exchange bridges the gap between the mobile agent paradigm and the mobile data approach in traditional distributed systems.

Multiplicity of related documents in a distributed consistency check suggests existence of numerous mobile agents, working towards the same goal of checking a consistency relationship between the documents. In such a case, co-operation between the agents is essential for carrying out the distributed consistency check in an efficient manner.

Stability

During its operation, the mobile code is less dependent on the network than the traditional code that relies on some form of remote calls. In this respect, mobile code is potentially more stable, because

issues of unsatisfactory network quality of service are not relevant while an agent operates on local resources at a network host.

Mobile agents can also be replicated and execute concurrently at different hosts to improve fault tolerance. Flexibility of choice for the information processing location in a mobile agent application allows us to construct an architecture with multiple points of control and avoid the problem of a single point of failure, a common characteristic of centralised systems [Emmerich 2000].

We argue, that the increased stability of the distributed system, gained by processing locality, and presence of multiple points of control in a mobile agent system are essential in the application domain of a distributed consistency checker. The described existing architecture for centralised consistency checking [Nentwich, et al. 2000b] (Chapter 2) is not capable of offering this required level of stability.

Disconnected operation

As an additional benefit of local access to resources, mobile agents are capable of disconnected operation. While all required data is available at a network host, a mobile agent will continue to carry on its uninterrupted execution at this host without requiring any network connectivity.

Disconnected operation is particularly suitable for mobile users, and those working from home with intermittent network connections. Its advantage is in enabling a user to specify a goal for a new agent while disconnected, instantiate an agent and launch it during a brief connection session and then immediately disconnect. Agent's results or feedback are then collected upon a subsequent re-connection.

We foresee a number of users of a distributed consistency checking application operating from mobile and thin clients with intermittent network connectivity. It is also important for the users of "thinner" clients to be able to execute consistency checks on a different workstation for computationally intensive consistency checks, and the disconnected operation feature gives them such flexibility.

Disconnected operation is also beneficial for local area network (LAN) and larger, internet-wide applications. Should a network link between several groups of users fail, near-normal operation can nonetheless continue within each group, while any transactions between groups are queued for resolution after the connection has been restored. While management of transactions and smooth operation of re-connected parts of the system is a complex issue in itself, mobile agent's support for disconnected operation is a useful step in the direction of enabling this desired functionality.

Added simplicity of upgrade for installed code base

When evolution of code base and expansion of application functionality requires upgrade of existing code, the concept of mobile code is an efficient solution. A number of issues have to be addressed, primarily interoperability between code versions, and an ability of mobile agents to share the same communication protocols and co-operation assumptions. Apparent security concerns need to be addressed with respect to upgrade; we discuss security in more detail below.

Bandwidth savings

Instead of transferring unprocessed data over the network, mobile agents move logic with essential data and benefit from local access to a data source. The distributed consistency checking framework has to be able to process large volumes of data, distributed across the network. In this framework, consistency rules specify selection of sets of document elements from the input data, which need to be collected before consistency status of a relationship can be computed. These element sets are relocated between distributed hosts during a consistency check, and they are normally in orders of magnitude smaller than the original documents (although this depends on the nature of consistency relations being checked).

As a contrast, in a centralised architecture, all input data needs to be downloaded onto a central location before a consistency check can commence [Nentwich, et al. 2000b]. Naturally, significant bandwidth and latency savings can be realised in certain configurations with a mobile agent architecture in comparison to a centralised architecture.

In addition to the advantages of mobile agents stated above, which can be readily related to the domain of distributed consistency checking, core *domain-independent* advantages of mobile code usage have been identified in [Chess, et al. 1997]: "individual advantages of mobile agents ... rest on relative technical and commercial factors compared to alternative methods". Discussion of these domain-independent technical advantages of mobile agents follows.

Scalability

"As a method of supporting simple queries and transactions, mobile agents benefit from the scalability inherent in concurrent execution and messaging. The asynchronous nature of mobile agents appears likely to enable higher transaction rates between servers. However, a need to execute agents and to support rigorous security around the agent execution environment could become significant computational loads in themselves. Thus, a question whether agent-based computing itself is efficiently scalable will depend on the extent to which service providers permit resident agents to work on their servers, thus providing computational capacity to the users. This kind of service is expected to become a business of the future [Chess, et al. 1997]."

The software agent architecture that we develop in this thesis deploys mobile agents for execution of distributed consistency checks. Scalability of the approach is determined on an evaluation of the model and the implementation prototype of the architecture.

Scripting

Scripting, used in programming of mobile agents provides better support for heterogeneous environments. The use of script language for program and data exchange enables the program and data representation to be independent of the platform, once the script environment has been ported to all necessary platforms. With a similar effect, the Java programming language provides a convenient

abstraction of a virtual machine, and is therefore most commonly used for cross-platform execution of mobile components.

Security

Mobile agent security adds an extra dimension to security issues, raised by inter-process calls on a local machine and the issues, relevant to remote calls and mobile data in traditional distributed systems. As mobile code is a relatively novel and developing field, existing levels of security are most often considered as a disadvantage of mobile agents and a reason for slow adoption of the technology in real world applications [Milojicic 1999].

The majority of mobile agent frameworks implement mobile code security features, although providing varying levels of functionality and granularity for security policies. Review of prominent mobile agent frameworks further in this chapter and in Appendix D contains a brief description of security mechanisms deployed in the frameworks.

The mobile agent paradigm is able to offer better data protection than the mobile data approach, deployed in centralised and traditional distributed applications. When migrating between network hosts, mobile agents transport only the essential, previously collected data. For the distributed consistency checking application domain, as argued in the section on bandwidth savings above, the transported data forms a relatively small portion of all data, locally available to mobile agents at distributed network hosts.

In a mobile agent system, as a result, *complete documents are never transmitted across the network* in a single transfer. Only smaller sets of document elements are transmitted with the agent in a serialized form. Therefore, if a security is breached and transmitted data has been decoded, it is still not possible to reconstruct source documents from the information transmitted.

Interoperability

Very limited interoperability between mobile agent frameworks from different vendors is provided at the time of writing. There is no agreed standard yet on a common API for mobility, although an OMG standard, Mobile Agent Facility (MAF) [OMG 2000a] has recently emerged. We further discuss in the review of mobile agent systems below the IBM Aglets [Lange and Oshima 1998] mobile agent framework, which is to a large extent compliant with the principles of MAF. As IBM has actively participated in the standardisation process, Aglets are MAF-interoperable at the level of features and available agent management and class transfer functionalities.

4.5 Inter-agent communication

This section briefly describes the technologies, which allow distributed software agents to carry out of their tasks effectively by coordinating activities through communication.

4.5.1 Communication Primitives

The basic building blocks for inter-agent communication are constituted by communication primitives [Lux, et al. 1993], drawn from speech act theory. These primitives are targeted towards expression of types of communication and co-operation objects. [Haugeneder and Steiner 1998] came up with generic primitives for expression of goals, plans, and agents' tasks. We briefly describe these primitives below, as they form a general basis for agent-to-agent messaging.

- Propose – a proposal starts or continues communication among agents about an object of co-operation. The knowledge transferred by a proposal to other agents is hypothetical, as agents sharing this knowledge have not yet committed to it.
- Accept – indication of commitment to the co-operation object.
- Refine – an agent proposes a further instantiation of the co-operation object.
- Reject – indication of failure to commit to the co-operation object.
- Modify – a counter-proposal of an altered co-operation object.
- Query – a query for arbitrary knowledge.
- Inform – the answer to a previous information request or broadcast of a new knowledge.

The software agent architecture for distributed consistency management constructed in this thesis is oriented on processing and notification of events. The described co-operation primitives provide a relevant communication abstraction between architectural components. For instance, a large part of communication between agents can be abstracted by *inform* and *query* primitives. Negotiation between software agents involves task assignment, thus *propose/accept/reject, refine* and *modify* primitives are used. Although message names (or "types") in the implementation of the architecture differ from the names of primitives, message goals correspond to the discussed classification [Haugeneder and Steiner 1998].

4.5.2 Multi-agent Co-operation

So far in this chapter we have considered agents working independently, each of them having certain features, roles and goals. In the distributed environment, however, agent-to-agent communication is essential.

Almost from the very creation of the concept of an agent, the agent community has been looking at the issues of inter-agent co-operation. From the perspective of Distributed Artificial Intelligence (DAI), the emphasis of research is on collections of agents and their interaction rather than single agents. The central problem underlying the design of agent-based systems is characterised by the following question [Haugeneder and Steiner 1998]:

> "How can individually motivated, independently designed computational artefacts, i.e., agents, act together to achieve some (at least partial) common goal in a genuinely distributed problem space?"

Co-operating agents need to know *what* they are co-operating about and *how* to co-operate efficiently. Depending on an agent's *goal*, and based upon the set of *actions* an agent can perform, it can develop a number of *plans* for achieving this goal. In DAI, co-operation occurs not only by carrying out individual actions together, but also by executing shared plans as well as actually planning together [Haugeneder and Steiner 1998].

Carrying out a distributed consistency checking task by multiple mobile agents will necessarily require inter-agent co-operation. As one example, due to symmetry of consistency relations between numerous documents, multiple agents may be pursuing the same goal of checking a relation, but surveying distributed documents in a different sequence. In this case, allowing all such "redundant" agents to proceed uninterrupted would waste resources, as each document would be surveyed several times by different agents.

As a solution to this problem, we have developed a multi-agent co-operation mechanism, where agent redundancy is detected through agent registration, goals of redundant agents are shared and a co-operative plan is generated, which takes into account all previously surveyed documents. The co-operation mechanism facilitates multiple agents planning together, and ensures that the shared plan deals effectively with interdependencies between agent's actions, as required by [Haugeneder and Steiner 1998].

4.5.3 Location Service for Distributed Agents

Another aspect that needs to be considered is how agents can reference other agents during communication. Referencing other agents through network or machine address seems to be a poor solution in the distributed setting, where agents move frequently. Additionally, some form of registration needs to take place when an agent moves, so that its current address can be resolved and located.

The heterogeneous communication model [Goose 1995] provides an addressing mechanism based on agent registration. As agents migrate into a domain, they register their presence with a central "router". The agent can also register the types of messages it wishes to receive from others. For example, an agent could register to indicate that it had information on a particular subject and that it required information on another subject. Due to the modular nature of the heterogeneous communication model, the scope of registration extends over domains through a Domain Name Service-like lookup mechanisms. Agent address resolution via a unique agent identifier is then carried out by a central "router", or a number of interconnected agent name servers.

A convenient programming model of a persistent agent *proxy* is provided in some mobile agent frameworks [IBM 1998]. The proxy acts as a "pointer" to an agent instance, regardless of the agent's location. A distributed agent location mechanism underlying agent proxy implementation makes use of agent post-migration registration, DNS-like lookup mechanism and cached agent location information. Agent proxies hide the underlying implementation from a programmer, and enable the programmer to take advantage of agent location transparency when sending messages to distributed agents.

4.6 Taxonomy of Mobile Agent Frameworks

Currently available mobile agent frameworks differ in the mechanisms they provide to support code mobility and communication. This section and its extension in Appendix D describe and analyze interesting mobile agent systems with respect to agents, server, mobility, communication and security. The systems were chosen because they represent interesting solutions in this field or for historical reasons, as in the case of Telescript, which in fact coined the basic mobile agent concepts.

Current comparisons and surveys focus mainly on mobile agent systems implemented with Java programming language [Karnik and Tripathi 1998, Kiniry and Zimmerman 1997, ObjectSpace 1997] and principles of mobile code systems [Cugola, et al. 1996, Fuggetta, et al. 1998, Vigna 1997].

4.6.1 Structure of the Taxonomy

The taxonomy of mobile agent systems is provided in Tables 4.1 and 4.2, which cover relevant issues in the scope of this thesis. The taxonomy tables are structured as follows. The general section lists all supported execution platforms, programming languages for mobile agents and supported standards in the area. The agent section considers whether an agent system supports unique agent identity and allows agent cloning. The server section focuses on the execution environments for agents in terms of multiple places where agents can execute, and resource control at the server to prevent excessive resource consumption by visiting agents (e.g., memory, CPU resources, etc.).

The mobility section compares agent migration mechanisms (weak or strong mobility) and whether a migration decision is taken by an agent or by a server. Further, the taxonomy table compares migration mechanisms used to transfer an agent and the corresponding executable code. The code can be transferred with agent's data or loaded from a given code base. The communication section focuses on the local and distributed communication mechanisms implemented. The communication can be carried out via messages, local method invocation (LMI), remote method invocation (RMI) and its variation remote procedure calls (RPC), and via shared files.

The last section of the taxonomy lists important concepts for large-scale mobile agents systems. These systems must support agent tracking to find and contact roaming agents; a service directory, where agents can find interesting places to migrate to; fault-tolerance to prevent loss of agents in a network; and support for disconnected operation, where agents can be dispatched from a partly connected host and wait somewhere to be retrieved by the user.

Individual mobile agent systems, classified in the provided taxonomy, are described in [Appendix D]. Where particulars of a discussed system are of interest, features of the systems are expanded in relation to agents, servers, communication and security, within the scope of the taxonomy.

The features of the Aglets mobile agent framework are described "inline" in this chapter. The Aglets framework is selected in this thesis for construction of an implementation prototype of the software architecture. Selection of the framework follows in the Section 4.8 and is based on the features, described in the taxonomy.

4.6.2 Comments

All mobile agent platforms are built on interpreted programming languages, except MESSENGERS (using a subset of C) and TACOMA (C, C++), which compile the agent into native machine code before executing it. Java is the language of choice for newly developed agent systems.

In respect to interpretation of programming languages, agent systems can be classified into three categories. First are the systems, supporting one interpreted programming language, where agents execute on top of the language interpreter (Telescript and Java-based systems such as Aglets, Concordia, Grasshopper, Mole and Voyager). Second are the systems, integrating multiple programming languages on top of a common system core like D'Agents, and finally systems, which run agents individually, each on top of their own language interpreter, such as MESSENGERS and TACOMA.

System interoperability, such as provided in the industry standard OMG MASIF [Milojicic, et al. 1998] or OMG MAF [OMG 2000a], is only supported in Grasshopper and Aglets, respectively, while other agent systems (for instance, D'Agents) plan to support it in the near future. Since Voyager architecture is essentially an object request broker architecture (ORB), Voyager supports CORBA and DCOM.

All agents in those agent systems, where a directory service is supported, can be set a limitation on their lifetime for visiting agent *places*, where mobile agents execute. In some of the systems examined here, agents must obtain a fixed list of hosts to visit before they are launched. Systems like Concordia, MESSENGERS, and Voyager can clone agents without any danger that the network may become overpopulated by "immortal" agents.

All systems except MESSENGERS, which was developed for use in intranets, extend the security management features of the runtime interpreter through predefined and configurable policies, protecting agent places at the servers from malicious agents. The protection of an agent from actions of other agents is provided in Aglets only. It has not been clearly stated for the other agent systems if their security policies can also be used for this problem.

Table 4.1. Taxonomy of mobile agent systems, part 1.

Mobile Agent System	Aglets	Concordia	D'Agents	Hive	Jini	Grass-hopper
Developed by	IBM Corp.	Mitsubishi Corp.	Dart-mouth College	MIT Media Lab	Sun	IKV++ GmbH
Supported Platforms	JDK 1.1	JDK 1.1	Unix	JDK 1.1, 1.2	JDK 1.1, 1.2	JDK 1.2
Languages	Java	Java	Tcl, Java, Scheme, Python	Java	Java	Java
Distributed App Standards Support	OMG MAF	-	-	-	-	MASIF
Agent Identity	+	UID	+	+	+	+
Cloning	+	+	-	+	+	-
Places per Server	many	one	many	many	many	many
Resource Control	-	+	+	+	+	+
Migration Type	weak	weak	strong	weak	weak	weak
Migration Decision	agent	agent	agent	agent	agent	agent
Migration Mechanism	ATP, RMI	RMI, SSL	sockets, email	RMI	RMI	IIOP, RMI, SSL
Code Migration	code base	with data	with data	code base	code base	code base
Agent Tracking	-	-	-	-	-	-
Service Directory	local	local	-	local	-	MASIF
Fault-Tolerance	+	+	+	-	-	+
Disconnected Operation	+	+	+	+	+	-

Legend: (+) – feature is supported, (-) – elementary or insufficient support for the feature is provided, improvement can be achieved.

Table 4.2. Taxonomy of mobile agent systems, part 2.

Mobile Agent System	MESSEN-GERS	Mole	Tacoma	Telescript	Voyager
Developed by	Univ. California	Univ. of Stuttgart	Univ. of Tromso/ Cornell	General Magic, Inc.	Object-space, Inc.
Supported Platforms	SunOS	JDK 1.1	Unix/Windows NT	Unix	JDK 1.1, 1.2
Languages	C	Java	C, C++, Java, Tcl, Scheme, Perl, Python	Telescript	Java
Distributed App Standards Support	-	-	-	-	CORBA, DCOM
Agent Identity	-	UID, badges	-	telename	UID
Cloning	+	-	-	-	+
Places per Server	many	one	one	many	many
Resource Control	-	+	-	+	-
Migration Type	strong	weak	weak	strong	weak
Migration Decision	agent	agent	agent	agent	agent
Migration Mechanism	sockets	RMI	sockets	sockets	SSL, IIOP, RMI
Code Migration	with data	code server	with data	with data	code server
Agent Tracking	-	-	-	-	+
Service Directory	-	local	-	-	local
Fault-Tolerance	-	-	+	-	+
Disconnected Operation	-	+	-	-	-

Legend: (+) – feature is supported, (-) – elementary or insufficient support for the feature is provided, improvement can be achieved.

4.7 The Aglets Framework

The Aglets Software Development Kit (ASDK) [IBM 1998] was developed by the IBM Tokyo Research Laboratory, Japan [Karjoth, et al. 1997, Lange and Oshima 1998, Oshima and Karjoth 1997]. An aglet is a mobile agile agent written in Java and named after the base class in the framework. ASDK is one of the most commonly used mobile agent systems in industry [Milojicic 1999] and for electronic commerce [Dasgupta, et al. 1999, IBM 1997].

The Aglets Framework consists of the following components. The ASDK library provides basic migration and messaging functionality. Agent Transfer Protocol (ATP) is an application-level protocol that extends HTTP and is used to transfer agents over the network. A visual Agent Manager (Tahiti) acts as local agent server, and the Agent Web Launcher (Fiji) allows creation of applets, which execute aglet agents and allow one to create and retract aglets from within a client's Web browser.

The Aglets framework implements the following basic concepts: agents (aglets), execution context, migration, messaging, and policies.

4.7.1 Agents

An aglet is a mobile autonomous agent, which is based on the concepts of the aglet core and the aglet proxy. The core is the essence of an agent and contains all of the aglet's internal data and methods. The core also provides messaging interfaces, through which the aglet may communicate with its environment.

The Aglet class defines the core of the agent, contains default properties and methods for a mobile agent to control its mobility and its lifecycle. Upon instantiation, each agent obtains a unique agent identifier. Additional agent properties are available through AgletInfo object. AgletInfo contains an aglet's inherent attributes, such as its creation time, URL of the host-origin, URL of the code base, and security attributes ("authority"). Dynamic attributes are also made available, such as agent's migration time and the address of its current execution context.

Access to a number of public methods that an aglet provides may be obtained via a proxy object called *AgletProxy*, through which interaction between aglets is carried out. This interface acts as a handle of an aglet and provides a common way of accessing the agent. The interface enables the location transparency feature of Aglets, though the agents are aware of their location and therefore can make adjustments to efficiently communicate with remote resources. The AgletProxy interface is implemented in the Aglets runtime library, and hides the underlying implementation of agent location discovery process from the programmer.

Agent classes, extending the base Aglet class, can implement their own listener methods for events and received messages. As examples of aglet system events – storage of an agent in a persistent storage, agent migration and cloning – can be considered. An aglet can trigger its own termination, and can also be terminated from outside.

With respect to cloned agents, throughout this thesis we will be referring to the term "family of agents", by which we mean a group of mobile agents – clones, which originate from a particular parent-agent. During cloning, an exact copy of an original agent is created, and values of all agent class properties are replicated. Consequently, newly created clones share the same goal with the parent agent, hence the term "family of agents".

4.7.2 Servers

The Aglets server runs one or more execution environments where agents operate, called aglet *contexts*. The AgletContext interface is used by an aglet to access core facilities of the framework, to get information about its environment and to gain references to other agents' proxies.

The context also manages the lifecycle of an aglet. Agent lifecycle allows the programmer to describe behavior an aglet should perform in reaction to system events. Cloning, migration and storage of an agent have already been referred to above as example events, and Sections 4.7.5 and 4.7.6 consider the event model in more detail.

Of all the frameworks reviewed, Aglets enforces the mobile agent paradigm in the most complete fashion. Event handler methods can be replaced at runtime, which provides greater flexibility and autonomy for an agent. For example, an agent can reconfigure message handler methods on receipt of a re-configuration message or when network conditions change. By exercising this capability, an aglet is able to decide which actions to take in respect to a particular message type or group of messages at runtime.

The ASDK runs on top of unmodified Java Virtual Machine, and therefore is not capable of fully supporting strong agent mobility. However, the agent lifecycle and the event model are flexible enough and enable the execution of agent's code to resume from a pre-defined method after migration has occurred. It is the AgletContext, which maintains the infrastructure that supports the agent lifecycle.

The Agent Transfer and Communication Interface (ATCI) is responsible for migration of agents between contexts and establishment of communication between agents. ACTI is built on Agent Transfer Protocol (ATP) [Lange and Aridor 1997], and supports migration transfers in a fault-tolerant way. When an aglet is dispatched to a remote aglet server, which appears to be unavailable at the time, an agent can be temporarily stored in persistent storage until the server becomes available and the agent can finally be transferred to the destination.

The Aglets framework provides a class loader equipped with a class cache. When an aglet migrates, the receiving server checks whether class definitions of the incoming agent are compatible with any existing classes in the cache. A new instance of a class loader is created if there is at least one "conflicting" class, and this class is then transmitted from a remote location and is stored in the cache.

4.7.3 Communication

An application program or an agent can communicate with the aglet object by local or remote messaging. ASDK defines a synchronous message passing method ("now"-type), an asynchronous "future"-type method with acknowledgement and an asynchronous "one-way" type method without acknowledgement. Non-blocking "future"-type method returns a FutureReply object, which can be used as a handler to receive the response to the message at a later time, asynchronously.

All aglets objects maintain a message queue. All incoming messages are stored in the message queue, and then handled sequentially. Messages in ASDK arrive in the order in which they were sent, but can have priorities associated with different types of messages. Priority affects placement of messages in a message queue.

4.7.4 Security

The security model of the ASDK focuses on protection of hosts and aglet agents against other, possibly malicious agents. The model makes use of security *policies*, defined by administrator as a set of rules, which specify agents' access permissions and authentication mechanisms. Policies can be defined hierarchically and stored in a policy database. In relation to policies, aglets may have different levels of trust: "trusted" and "untrusted". By default, trusted agents are those with local code bases, and any other agents are untrusted. Access rights can be inherited: a trusted aglet may create another trusted aglet on the same aglet server.

The AgletProxy acts as a shield object that protects an aglet from invocations of its own methods by other, possibly malicious agents. When a method is invoked, the proxy checks with the Aglets security manager to determine whether the current execution context or aglet has a permission to execute a method.

4.7.5 Aglets Event Model – Agent Cloning Example

A delegation-based event model provides uniform handling of aglet cloning, mobility and persistence. When an aglet is cloned, moved or saved to persistent storage, this will result in a number of events being sent and processed by the aglet.

Cloning is a way of creating new agents, which is an alternative to instantiation. Calling a *clone* method of an aglet context creates an identical clone to the current agent and returns a proxy to the clone. The proxy is a placeholder for the aglet, serving to shield it from direct access to its public methods. Throughout the cloning procedure, a pre-determined sequence of methods is executed in the master and the clone agents, as shown in Fig. 4.1, which appears in [Lange and Oshima 1998].

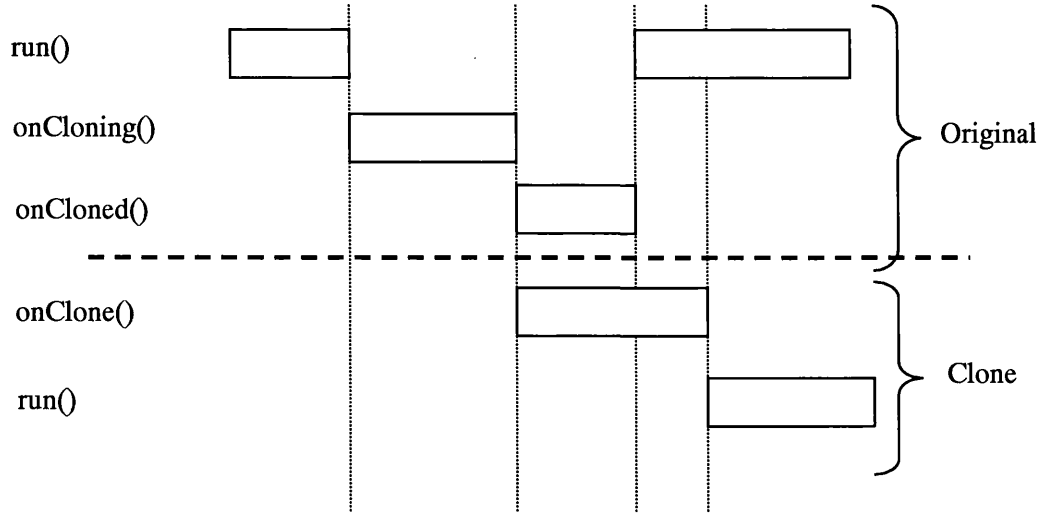


Fig. 4.1. Sequence Diagram for Aglet Cloning.

When Aglets agents are used for distributed data processing, it would be logical to use the cloning methods *onCloning* and *onCloned* in order to prepare the necessary data for the clone and notify other collaborating agents about the successful cloning result. Method *onClone* can be used for any required initialisation within the agent clone before the execution of the main method *run* can commence.

Below, we demonstrate a particular example of the cloning procedure, where the master and clone are linked after creation. Cloning is used extensively in the software agent architecture for distributed consistency management. This example explains the Aglets event model, and cloning procedure in particular, in essential detail.

```
import com.ibm.aglet.*;
public class exampleClone  extends Aglet {

AgletProxy prev = null;  /* proxy of the previous, parent agent in the
linked agent list */
AgletProxy next = null;  /* proxy of the next, clone agent in the list */

public void onCreation(Object o) {
            addCloneListener(
                  new CloneAdapter() {
public void onCloning(CloneEvent e) {
      prev = getProxy();
// the clone will inherit a proxy of  the master agent, contained
// in "prev". This is how a clone will "know" its parent agent.
}
public void onCloned(CloneEvent e) {
      System.out.println(getAgletID().toString()+
" Cloning completed.");
prev=null; /* returning the value to null at parent agent */
}
public void onClone(CloneEvent e) {
      next = prev; /* marking next to prevent further cloning */
}
} ); } /* end onCreation() */
public void run() { try {
      if (next==null)  /* I am a parent, have to make a clone */
            next = (AgletProxy) clone();
      else next=null; /* else I am a clone */
      if (prev==null) System.out.println(getAgletID().toString()+
      " I am a parent. My clone is" +next.getAgletID().toString());
```

```
      if (next==null) System.out.println(getAgletID().toString()+
      " I am a clone. My parent is "+prev.getAgletID().toString());
      } catch (Exception e) { }
} /* end run() */
} /* end ExampleClone class */
```

Fig. 4.2. Example of the cloning code, creating linked master and clone agents.

```
aadd0546a46d278b Cloning completed.
aadd0546a46d278b I am a parent. My clone is b3da340474b04e51
b3da340474b04e51 I am a clone. My parent is aadd0546a46d278b
```

Fig. 4.3. Execution log of the example cloning code.

The exampleClone class contains two aglet proxy variables, which after cloning will point to the "parent" and the "child" agent. OnCreation method of the parent, called only once in the parent instance (Fig. 4.1), adds custom implementations of clone listener methods, which overload empty defaults. Run method of the parent triggers agent cloning if the proxy variable *next* is null and does not point to a valid agent. After cloning is completed, a proxy to the clone is assigned to the parent's *next* proxy variable.

The OnCloning method, executed in the parent agent, assigns the value of the current (parent) agent to the *prev* proxy pointer; this value is copied by the child and correctly links the child with the parent. The OnCloned method in the parent notifies the user of the completion of cloning.

The OnClone method in the child agent is the first method to be called. *Next* is marked in order to prevent further cloning of the child in the method *run.*

After cloning, the parent agent returns to the method *run.* The clone also enters *run* for the first time. Agent identifiers are printed on the console (Fig. 4.3), and each agent identifies itself as a parent or a clone, depending on the values of the *prev* and *next* variables.

The Aglets event model is flexible enough to allow an agent to re-define event handlers at runtime. In the given example, the clone agent can substitute its own action listener with a different (but of course, pre-defined) set of method implementations for OnCloning, OnCloned and OnClone. In an example listener substitution that we could implement, all future clones of a cloned agent ("grand-children" of the original parent) inherit a value of the property *prev,* which would point to the original *parent,* rather than to the *son* that the current event handler implements.

The dynamic behaviour described above is essential for co-ordinating agents within the same agent family, and between different families. In the implementation of the software agent architecture, agent cloning allows us to combine the benefit of concurrent agent execution at multiple distributed hosts and to avoid unnecessary multiplication of consistency checks with "redundant agents".

4.7.6 Aglets Event Model – Agent Mobility

Aglet migration is a choice of an agent; migration is controlled by the agent at runtime. Migration is triggered by a *dispatch* method of the Aglets framework, the destination host is specified in the parameter to the method. Agent instances can overload onDispatching method, which is executed in the

agent before migration, and onArrival method, which is executed straight after migration. The sequence diagram for Aglet migration [Lange and Oshima 1998] is shown in Fig. 4.4.

The Aglets framework cannot provide support for strong agent mobility, because the underlying Java virtual machine does not allow persistent saving of the state of code execution. At the same time, event model provides a programmer with the facility to "mimic" strong mobility. In an implementation of the software agent architecture for consistency checking, mobile agent code is divided into loosely coupled fragments, which execute at different hosts. Agent message handlers refer to one of these fragments, depending on the message received (Chapter 9). OnDispatching method is then used by an agent to send an appropriate message to itself. Upon arrival, this message is processed by the message handler and consequently, an appropriate code fragment is executed.

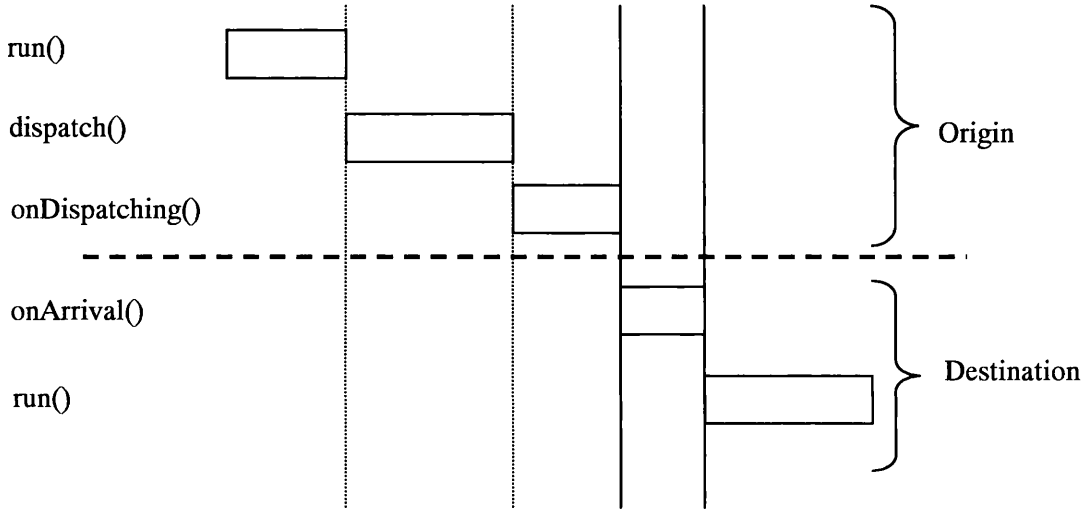


Fig. 4.4. Sequence diagram for Aglet migration.

4.8 Choice of a Mobile Agent Framework for the Distributed Consistency Checking Architecture

In this chapter, we have defined software agency and mobile agents through their generalized characteristics. We have also surveyed a number of popular mobile agent frameworks [Appendix D], and looked in detail at the IBM Aglets (Section 4.7). This survey has enabled us to compile the taxonomy of mobile agent frameworks, where the frameworks are contrasted.

Chapter 3 specified the requirements, which are demanded of the distributed consistency checking architecture that we construct in this thesis. At this point, we can elaborate on the discussed characteristics of software and mobile agents with respect to these functional requirements. As a result, we specify the requirements for a mobile agent framework that would fulfill the ambitions of the architecture, and carry out the selection of an agent framework to be used as a basis throughout the remainder of this thesis.

4.8.1 Requirements For a Mobile Software Agent Framework

In Fig. 4.5 we highlight important requirements, which we use for the selection of a mobile agent framework.

Autonomy	Decision making must be done by an agent without recourse to user's attention. This applies to consistency checks run based on events, decisions in respect to results of consistency checks and policies.
Mobility	Agents must be able to seamlessly transfer themselves from one node to the other without violating the state of any of these systems. All data relevant to the current goal of an agent must be carried with the agent.
Arbitrary migration	From the high level of abstraction, migration between nodes must be achieved by invocation of a single operation.
Planning	Agent must be able to derive a work plan from the goal, which has been stated. In case of a mobile agent checking a consistency rule, it must be able to determine which parameters still need to be filled within the consistency rule template currently undergoing a check, and where on the network can these parameters be found.
Route Determination	Route determination is an ability of a mobile agent to construct a path of nodes, which it needs to visit in order to fill in all required parameters of a consistency rule template. This sequence will indeed depend on the logic of consistency rule itself (an AND rule would turn out false if any of composite parts are false).
Failure handling and intelligent re-planning	Includes route re-planning, choosing another network path to the destination, as well as finding alternative locations for a requested document and such.
Persistence	An important quality of a mobile agent, by which it actively takes all possible action to achieve its goal without any further input from its owner or user.
Inter-agent communication	Language must be structured to provide necessary functionality to describe all types of events or messages, which need to be conveyed.
Event notification functionality	Agents must be able to exchange meaningful messages, and react upon those.
Heterogeneity of accessible resources	Agents must share a common resource access mechanism for documents of all possible types. This can be achieved by use of resource agents (drivers) and use of an intermediary meta-language document representation (such as XML).
Security	Domains must agents grant different access rights, depending on agent's host domain and goal (analogous to agent's trustworthiness)

Fig. 4.5. Review of requirements for a successful mobile agent implementation within the consistency management domain.

4.8.2 Selection of a Mobile Software Agent Framework

Of all mobile agent frameworks described in the review (sections 4.6, 4.7, Appendix D), we have chosen to base our implementation of the distributed consistency management architecture on the IBM Aglets. In this section, we highlight several of the important factors that affected our choice of ASDK. At the time when the research, described in this thesis, had started, some of the mobile frameworks discussed above were in development (i.e., Hive), and others were undergoing a transformation (Telescript and Voyager). Aglets has proven as a stable platform, well documented [Lange and Oshima 1998], and used in industry [Dasgupta, et al. 1999].

ASDK provides a comprehensive and well developed Java API. A large number of successful projects have been completed based on ASDK, and therefore extensive documentation is available.

The Aglets framework has been an industry standard in the past for some time, and is proposed for submission to the Object Management Group (OMG) Mobile Agent Facility RFP [OMG 2000a].

Aglets support the notion of an agent goal called "itinerary". Historic agent systems like Telescript do not provide support for this concept. The scheduling functionality, provided by the itinerary, facilitates implementation of agent planning, route determination, and intelligent re-planning in case of failure.

Aglets fully support weak mobility, and are able to provide near-strong mobility through use of the event model (see cloning and migration, Sections 4.7.5 and 4.7.6). Support of arbitrary migration is instrumented in the ASDK libraries. The Aglets event model achieves a high standard of inter-agent communication and event notification by providing messaging functionality via persistent agent proxies. Aglet proxies conveniently and effectively emulate distribution transparency, while all agents are explicitly location aware.

ASDK draws on the Java security model and extends it with agent framework specific functionality (Aglet security policies). Overall, ASDK provides sufficient security features for the considered application domain.

Mobility in ASDK is based on the Agent Transfer Protocol (ATP) and Java Agent Transfer and Communication Interface (J-ATCI). ATP uses Universal Resource Locators (URLs) for locations of hosts and places; the protocol works well with heterogeneous platforms.

Heterogeneity is enhanced by deployment of Java serialization mechanisms, and access to heterogeneous resources in our architecture is provided via a Java XML parser, operating in a common Java environment with the agent framework.

4.9 Summary

In this chapter, we have given a definition to software agency by describing and classifying the characteristics of agents. We considered the approach taken by traditional distributed systems, contrasted it with a mobile software agent solution and highlighted some major advantages and disadvantages of the mobile agent approach. As mobility is not a universal cure in the distributed systems domain, we introduce inter-agent communication by giving a brief account of agent co-operation and communication languages.

In this chapter and in Appendix D we have provided the taxonomy of mobile software agent frameworks. One of these frameworks, the Aglets from IBM, was considered in more detail. Then, we outlined our functional requirements for a software agent framework, on which later construction of the software agent architecture prototype will be based. We have provided our considerations on the characteristics of Algets, which satisfy these requirements, and result in selection of Aglets as a base software agent framework in this thesis.

One of the significant motivations for this thesis is in application of novel technologies –software agency and code mobility to the software engineering domain, and to the problem of distributed consistency checking in particular. We have chosen to base our architecture "a-priori". There are relatively few systematic applications of this technology in software engineering at present, and such application constitutes a research problem in its own right. In this chapter, we have contrasted software agent systems with more traditional distributed systems, but of course, our conclusions are not intended to draw the final line in the ongoing discussion. The novel contribution of this thesis is in construction of the software agent architecture, its evaluation on a model, and on the implementation prototype, and in application of principles of the novel software agent technology to the software engineering field. As a result, we present some of the lessons learned in the evaluation and conclusion chapters of the thesis.

Chapter 5 Incremental Consistency Checking

This chapter describes the proposed incremental consistency checking approach, which extends the exhaustive checking capabilities, provided by the existing consistency framework XLinkit. The incremental approach is used in this thesis as a basis for construction of a software agent architecture for distributed consistency checking. The software architecture is described in Chapter 6, and the incremental checking approach is referred to in the following chapters.

5.1 Consistency Rules

The incremental and exhaustive checkers process consistency rules, written in the XLinkit consistency rule language [Nentwich, et al. 2001a]. In this section, we consider the structure of a consistency rule and give a brief example of how the rule is checked. The example derives from the scenario for checking well-formedness of a UML model of a break scheduler application, which we introduced in Chapter 1 and use as a running example throughout this thesis.

5.1.1 Rule Example

In order to relate document elements, consistency rules reference document elements, involved in a consistency relationship. XLinkit uses XPath expressions to specify paths to the elements of XML documents. An XPath processor [Apache 2000a] allows us to "execute" XPath expressions and retrieve groups of elements - nodesets. In a consistency check, these nodesets are checked for existence of elements of a certain type, equality and non-equality of elements, and other constraints that a consistency rule prescribes.

Consider the UML model well-formedness rule, relating to Generalizations, expressed in the XLinkit rule language [Appendix A, A.11]. This rule (rule identifier "gen1") is also shown in Fig. 5.1. A consistency constraint, underlying this rule, concerns UML model elements of type "GeneralizableElement". A well-formedness relation, underlying the constraint, demands that a child and a parent GeneralizableElement are of the same type.

The representation of this consistency rule in XLinkit language (Fig. 5.1) contains a textual description and a "forall" operator with a nested "equal" operator. The nesting of operators requires equality between names of GeneralizableElement sub-elements of "supertype" and "subtype", for existing generalizations in the UML model.

```
<consistencyruleset>
<globalset id="generalizations" xpath="//Foundation.Core.Generalization"/>
<consistencyrule id="gen1">
  <description>
A GeneralizableElement may only be a child of a GeneralizableElement of the
same kind
  </description>
  <forall var="g" in="$generalizations">
      <equal
op1="name($g/Foundation.Core.Generalization.supertype/*[1])"
op2="name($g/Foundation.Core.Generalization.subtype/*[1])"/>
  </forall>
</consistencyrule>
</consistencyruleset>
```

Fig. 5.1. Consistency rule: well-formedness constraint for "Generalization" elements.

XLinkit consistency rule language supports global properties; the consistency rule in Fig. 5.1 contains the property "generalizations". Properties serve as points of reference in a structure of the documents being checked; XPath expressions in the rule operators include global properties and specify sub-elements relative to the properties.

5.1.2 Checking a Consistency Rule

An example generalization in Fig. 5.2 fulfils the consistency constraint, specified by the consistency rule in Fig. 5.1. The generalization is between two *classes*, BreakPlanImpl and UnicastRemoteObject. Another generalization in Fig. 5.3 does not fulfil the constraint, since the generalization is between a *class* "Teacher" and an *interface* "TeacherImpl". As a result of a consistency check, an inconsistent link is created (Fig. 5.4). The link refers to the generalization in Fig. 5.3 and to consistency rule in Fig. 5.1.

```
<!--
XPath expression to the Generalization in the UML model:
/XMI/XMI.content[1]/Model_Management.Model[1]/Foundation.Core.
Namespace.ownedElement[1]/Foundation.Core.Generalization[2]
-->
<Foundation.Core.Generalization xmi.id="G.307">
   <Foundation.Core.ModelElement.name/>
   <Foundation.Core.ModelElement.visibility xmi.value="public"/>
   <Foundation.Core.Generalization.discriminator/>
   <Foundation.Core.Generalization.subtype>
     <Foundation.Core.Class xmi.idref="S.10023"/><!--BreakPlanImpl-->
   </Foundation.Core.Generalization.subtype>
   <Foundation.Core.Generalization.supertype>
     <Foundation.Core.Class xmi.idref="S.10078"/><!--UnicastRemoteObject-->
   </Foundation.Core.Generalization.supertype>
</Foundation.Core.Generalization>
```

Fig. 5.2. This generalization fulfils the consistency constraint.

```
<!-
XPath expression to the Generalization in the UML Model:
/XMI/XMI.content[1]/Model_Management.Model[1]/Foundation.Core.
Namespace.ownedElement[1]/Foundation.Core.Generalization[4]
-->
<Foundation.Core.Generalization xmi.id="G.305">
   <Foundation.Core.ModelElement.name/>
   <Foundation.Core.ModelElement.visibility xmi.value="public"/>
   <Foundation.Core.Generalization.discriminator/>
   <Foundation.Core.Generalization.subtype>
     <Foundation.Core.Class xmi.idref="S.10001"/>
<!--TeacherImpl-->
   </Foundation.Core.Generalization.subtype>
   <Foundation.Core.Generalization.supertype>
     <Foundation.Core.Interface xmi.idref="S.10087"/>
<!--Teacher-->
   </Foundation.Core.Generalization.supertype>
</Foundation.Core.Generalization>
```

Fig. 5.3. This generalization does not fulfil the consistency constraint.

5.1.3 Resulting Consistency Links

```
<xlinkit:ConsistencyLink ruleid="generalization.xml#/consistencyrule[1]">
        <xlinkit:State>inconsistent</xlinkit:State>
        <xlinkit:Locator xlink:href="UMLexample.xml#/
XMI.content[1]/Model_Management.Model[1]/Foundation.Core.Namespace.
ownedElement[1]/Foundation.Core.Generalization[4]"
            xlink:label="" xlink:title=""/>
</xlinkit:ConsistencyLink>
```

Fig. 5.4. Inconsistent link generated as a result of a consistency check.

Each consistency link contains a reference to the consistency rule, which created this link, and XPaths to document elements, which were detected consistent or inconsistent during execution of the specified rule. The inconsistent link in Fig. 5.4 refers to generalizations well-formedness rule (Fig. 5.1) and "Generalization[4]", which is a generalization element, shown in Fig. 5.3.

5.1.4 Fragility of XLink Locators

Consistency links include link locators, which identify an element in the XML document, which is involved in a consistency relationship. The link locator contains an XLink [DeRose, et al. 2000], which consists of a URL of a document (UMLexample.xml, Fig. 5.4), followed by a separator '#', and a simple XPath to an element in that document. For each element, participating in a consistency relation, the consistency link contains a separate link locator.

Simple XPath expressions, used by XLink in link locators, refer to consecutive *numbers* of elements (i.e., '[4]'), which correspond to the order in which elements appear at each "level" of the XML document tree. Use of element numbers in the XLinkit locators makes consistency links fragile: every time an element is inserted or deleted at a certain level in the document tree, the numbering of all adjacent elements changes. As a result, the links generated previously refer to incorrect elements.

XLinkit avoids tackling the XLink fragility problem by re-generating all consistency links in one go in each exhaustive check. In this case a problem of partial correctness of older link bases is not relevant. An incremental consistency checker does not discard old link bases, and therefore, has to address this problem.

In the domain of consistency checking of UML models, a solution is based on uniqueness of model elements' identifiers. The XMI representation of any UML model contains unique identifiers for each element of the model. The identifiers appear as "xmi.idref" attributes of elements in the XML file (Fig. 5.2 and 5.3).

Use of unique element identifiers instead of element numbers in link locators would completely eliminate link fragility of the XLinkit approach. However, in order to maintain compatibility with existing XLinkit framework, we propose a different, locator mapping solution for keeping previously generated links referring to correct document elements. Our approach maps element numbers, used by XLink, to the unique element identifiers, and corrects the numbering within existing linksets after each document modification. We give a detailed explanation of the mapping approach in Appendix F, where an appropriate illustrative example is also provided.

5.2 Exhaustive Consistency Checking

Our work on incremental consistency checks takes its beginning from analysis of disadvantages [Smolko 1999] of techniques for "batch" processing of consistency checks introduced in [Ellmer, et al. 1999] - *exhaustive* consistency checking. The exhaustive approach was subsequently refined in [Nentwich, et al. 2000b], and the current iteration of the XLinkit framework [Nentwich, et al. 2000a] provides only exhaustive checking functionality.

5.2.1 Algorithm outline

In order to demonstrate the difference between exhaustive and incremental consistency checking, we contrast both algorithms in the pseudo code. Exhaustive checking algorithm is depicted in Fig. 5.5; incremental checking follows in Fig. 5.6.

```
0: REM Exhaustive consistency check,
   all documents are available to the checker locally
1:  FOR each consistency_rule DO
2:    FOR all XPath_expressions in the rule DO
3:      FOR each document DO
4:        Find elements, corresponding to the expression,
          in the current document and store them in the NodeSet;
5:      NEXT document;
6:    NEXT XPath_expression;
7:    Retrieve values of the elements from the NodeSet,
      and execute rule language operators on these values.
8:    Create consistency link depending on return value of
      the topmost operator in the rule;
9:  NEXT consistency_rule;
```

Fig. 5.5. Pseudo code of the exhaustive consistency checking algorithm.

The pseudo code in Fig. 5.5 is based on the assumption that the numerous documents are locally accessible and the consistency check can be executed completely and autonomously on a single host. This assumption is satisfied for XLinkit - a centralised checker of distributed documents, where all documents, participating in the check, are transmitted in their complete form across the network, where they are checked locally at a central server.

In an exhaustive check for *each* document, *every* rule is checked to find if there exist any document elements in this document that correspond to each XPath expression, specified in the consistency rule. If there are such elements, they are retrieved and added to the nodeset.

When all documents have been surveyed and a nodeset for all expressions in a given rule has been constructed, the link generator applies rule operators to values of all elements in the nodeset. The return value of the top-most operator of the consistency rule corresponds to a consistent or an inconsistent state of the relationship between document elements with respect to a constraint, specified in the consistency rule.

The algorithm in Fig. 5.5 contains three nested loops, and therefore its algorithmic complexity is a multiple of the complexity of a single consistency check by the number of documents and by the total number of rules deployed in the system. Significant computation, performed by each exhaustive check may be justified if all consistency relations need to be re-established. In this thesis, we argue that while such approach suits project milestone checking, it is unsuitable for repeated consistency checks during incremental development.

The exhaustive centralised checking approach (Fig. 5.2) is further disadvantaged in a distributed setting by the necessity to make all documents available at a single location, where the checking takes place. In the XLinkit implementation, complete documents are transmitted from their distributed locations for each centralised check. In addition to this architectural disadvantage, exhaustive checking presents further difficulties in identifying the differences between linkbases, generated for different project iterations.

Expanding on the mentioned disadvantages, below we outline the domain, where the exhaustive checking approach can be applied with success.

5.2.2 Application Domain for Exhaustive Checks

Exhaustive consistency checks perform cross-checking of all documents in relation to all consistency rules in *one go.* The resulting link base is a snapshot of consistency relations between the current versions of all documents. The exhaustive approach is best used to identify any remaining inconsistencies at the completion stage of an iteration of project development, or at the stage of its final release. At the same time, this approach proves hardly suitable for frequent consistency checks, which accompany incremental development.

The effect of individual changes is difficult to track within the results of an exhaustive check. Each time a change is made, a complete set of consistency rules has to be re-checked on all documents in the

project in order to find out if the change has resolved any previously existing inconsistencies or has introduced new ones. As we expect a significant number of potentially large documents and consistency rules to be used in a software engineering project, it is likely that a lot of processing time and resources would have to be dedicated to the repeated exhaustive checks.

Even if appropriate resources are allocated and the developers are able to regularly execute exhaustive checks, centralisation of the XLinkit architecture would require all distributed documents to be transferred to a central location in their complete form on every check, and would also require the resulting consistency links to be downloaded back to the developers' workstations. Scalability limitations of this approach for frequent consistency checks are apparent.

Let us also consider an exhaustive checker from the usability point of view in a typical scenario, where a developer is interested in tracing inconsistencies, introduced by individual document changes. All useful information about a *change* in the *state* of relationships between document elements is found by comparing a linkbase from a previous XLinkit check to the current linkbase. As we have pointed out above, fragility of XLinks in the link locators makes comparison of linkbases difficult, because correctness of links in older linkbases is violated by subsequent document changes. Even if reconciliation of linkbases is carried out, inconsistencies introduced by distinct changes in different documents would be indistinguishable within a global linkbase.

Exhaustive consistency checks are not suitable for tracing impact of individual changes on the status of existing consistency relationships. Such functionality is, however, required during incremental development.

5.3 Incremental Consistency Checking

Pseudo code for an incremental checking algorithm that we propose is given in Fig. 5.6.

```
1: REM Stage 1. Identification of initial relevance of
   consistency rules to documents
2: Execute exhaustive check of all documents, remember which
   rules applied to which documents
3: REM This initialisation stage 1 executes only once for each
   participating document.
4: REM Stage 2. On document change
5: WHILE there is no change on a document DO ;
6:  REM Stage 2.1. Identification of rules, relevant to the change
7:  Execute TreeDiff between a backup copy of the document and a
    current copy. Result: list of modified document elements
8:  Compare XPaths to changed elements (in TreeDiff) with XPaths
    in each consistency rule
9:  If XPaths are comparable, then add the rule to Relevant rule
    set, remember relevance of the document to this rule
10: REM Result of Stage 2.1: Selected consistency rules, relevant
    to a particular document modification.
11: REM Stage 2.2: Checking of relevant consistency rules on all
       documents, previously found relevant to these rules.
12: FOR each consistency_rule in Relevant rule set DO
13:   FOR all XPath expressions in the consistency rule DO
14:    FOR each document, relevant to this rule DO
15:      Find elements in the current document, which correspond
         to the XPath expression, and store them in
         a NodeSet for this consistency rule.
15:    NEXT document;
16:    IF no elements are selected for this document THEN
       remember that the document is no longer relevant to the rule
17:   NEXT expression;
18: NEXT consistency_rule;
19: REM Result of Stage 2.2: Nodesets for each consistency rule,
       containing values of document elements.
20: REM Stage 3. Generation of consistency links
21: FOR each consistency rule DO
22:   Retrieve values of the elements from the NodeSet, and
      execute rule operators on these values.
23:   Create consistent or inconsistent link depending on the value
      returned by topmost operator in the rule.
24: NEXT consistency_rule;
```

Fig. 5.6. Pseudo code of the incremental checking algorithm.

The incremental checking algorithm aims to remedy the main disadvantage of an exhaustive checker – a mandatory execution of all consistency rules on all documents at every document change.

Our incremental checking approach refines the exhaustive checker in two significant areas. Stage 2.1 of the algorithm limits the number of consistency rules needed to be executed based on relevance of rules to a particular incremental change. Stage 2.2 executes all selected rules on a sub-set of documents that have been identified relevant to the rules. Initial relevance of documents to all consistency rules is identified in Stage 1, and subsequently refined as changes are introduced (line 9, Fig. 5.6), and as consistency checks occur (line 16).

5.3.1 Initialisation

The initialisation code (Stage 1) is executed only once when the consistency checking framework is started, and once for every new document added to the system. For each document, an exhaustive check determines which rules are applicable to this document, and which relationships between this document and all other documents hold. The generated link base serves as a basis, and subsequent incremental link bases are merged in when required (the merging algorithm is discussed in detail in Appendix F).

Stage 2 of the incremental algorithm executes when a change on a document has been identified. This reactivity characteristic of an incremental checker allows it to operate autonomously. The stakeholders can also trigger execution of Stage 2 manually, if necessary, and can disable the automatic check execution.

5.3.2 Selection of Relevant Consistency Rules

Stage 2.1 (Lines 6-10, Fig. 5.6) of the pseudo code selects consistency rules, relevant to a particular document change. If necessary, rule applicability policies can be set up [Chapter 2, 2.2.4, Chapter 3, 3.3], which may disallow execution of certain selected rules, or demand execution of additional rules if certain rules are selected.

Each document change is reflected in a result of a tree-differencing algorithm (TreeDiff), which compares two versions of the same document and returns a set of XPath expressions to all modified elements. The process is similar to the forward differencing technique [Alderson 1988], where in our case the deltas are necessarily merged into the latest version by the user, and are determined in order to detect the consistency rules, relevant to the changes. Line 8 compares the XPath expression in the consistency rule with an XPath expression from the TreeDiff. The expressions are matched to establish whether one XPath is a sub-path of another.

Expressive power of the XPath language, inherent in availability of functions, wildcards and relative sub-paths complicates the process of comparing XPath expressions. The TreeDiff returns a simple XPath expression, but the UML well-formedness consistency rules [Appendix A] use extended features of the XPath language.

A lightweight algorithm for string-wise comparison of XPaths is proposed in [Appendix F, F.2]. For performance reasons, this algorithm does not *execute* XPath functions, and its accuracy in matching paths depends on the complexity of the paths.

In comparing XPath expressions, simple string-wise comparison bears a danger of overlooking that two XPath expressions match, because they include function calls or wildcards, and any statically defined elements of the paths have little in common. Complex XPath expressions that cannot be matched by this lightweight algorithm are executed on a document instance [Appendix F, F.3]. The discussion of the XPath matching algorithm as a technical challenge of incremental checking continues in this chapter further [Section 5.5].

5.3.3 Execution of Selected Rules

The Stage 2.2 of the incremental checking algorithm (Fig. 5.6) is identical in its operation to the exhaustive algorithm, except it performs checking of selected rules on selected documents. Due to a smaller number of documents and rules in an incremental check, performance advantages are realized in this part of the algorithm. An amendment of the exhaustive checker code is found on line 16 (Fig. 5.6), where information about relevance of consistency rules to a particular document is updated, depending on whether any elements of the document were selected by the rule. Updated information allows the checker to select relevant documents for the rule in Stage 2.1, and to effectively limit the span of incremental checks.

The final code fragment (Stage 3) computes the result of an incremental distributed consistency check using a similar nodeset collection and link generation mechanism as the exhaustive check. For each relevant consistency rule, values of document elements that are referred to from the rule operators are retrieved into a nodeset. The operators are then executed on the nodeset, and for each rule, the root operator returns a consistent or an inconsistent status of the relationship. Consistency links are then created between the document elements in the related documents, which are referenced by the rule.

When describing an incremental checking algorithm, we have not yet considered an access mechanism to distributed documents and, in order to simplify description of the basic algorithm, so far assumed that all documents are accessible locally. In the following section, we propose an approach for distribution of consistency checks.

5.4 Distribution of Consistency Checks

During consistency checks, extensive access to documents occurs only at the stage of selection of document elements. At this stage, XPath expressions in a consistency rule are executed on the XML document's DOM tree, and extracted nodes are stored in a nodeset for future processing by rule operators. Both the exhaustive (Fig. 5.5, Line 4) and the incremental (Fig. 5.6, Line 15) algorithms make use of the described mechanism when compiling nodesets. Results of consistency checks – consistency links, are generated using the nodesets (Fig. 5.5, Lines 7, 8 and Fig. 5.6, Lines 22, 23). The resulting links are stored externally to the documents. Consequently, access to documents is no longer required after the nodesets have been collected from the distributed documents.

In the distributed setting, the traditional distributed document access approach used in XLinkit involves transportation of documents in their complete form to a centralised location for consistency checking, where nodesets are extracted and links are generated. In the software agent architecture we propose in this thesis, we advocate extraction of nodesets at the documents' locations and the following transport of the nodesets where necessary.

In consistency checking practice, the size of extracted nodesets is smaller than the original document itself. Size difference is due to omission of redundant XML structure in the nodeset and use of

internal representations of element values (integers, strings, etc.) instead of their string representations. We expect a large number of documents of significant size in any software engineering project, and therefore envisage a performance advantage from transport of nodesets instead of complete documents. In a given scenario, efficiency gained would ultimately depend on the nature of document elements and the structure of consistency rules being checked.

Our approach to transport of nodesets has a significant security advantage. Without question, intellectual property is better protected when only parts of documents are transmitted across the network, rather than the documents as a whole. We do not claim that nodesets constitute a complete solution for all security concerns, and suggest that a security-critical implementation should deploy appropriate cryptography mechanisms for all transmitted information. In such case, use of smaller nodesets instead of complete documents is an advantage, as reduction in message size results in an improvement of performance of the security infrastructure.

5.4.1 Mobile Agents

In this thesis, we propose to deploy mobile agents for collecting nodesets from all distributed documents, participating in a consistency relationship that is specified in a consistency rule. After the locations of all documents have been visited by a mobile agent and nodesets from all documents have been extracted, the agent executes rule operators at any convenient location, where processing resources are available. This mobile agent approach eliminates a single point of failure problem, which exists in the centralised checking service XLinkit.

Development of autonomous agents allows us to delegate itinerary planning [Chapter 4; 4.1, 4.4.3] to the agent, which would determine a list of locations and documents to visit and carry out a consistency check on its own, without a need for user intervention. The incremental checking algorithm provides an agent with the necessary information on relevance of consistency rules to participating documents (Stage 2.2 in Fig. 5.6), which is used to compile the itinerary. As a result, mobile agents allow us to combine reactivity of the incremental algorithm in response to document changes with autonomy of a distributed consistency check. When the resulting consistency links are delivered to the user, they constitute a pro-active response of the software agent architecture to a particular user modification of a document. In contrast with the "passive" nature of the existing consistency checking service, the ability to provide pro-active responses to user activity is a significant advantage of the proposed approach.

The main contribution of this thesis – a software agent architecture for distributed consistency checking, provides an infrastructure and enables mobile agents to carry out distributed consistency checks as we have outlined in this section. The architecture is considered in more detail in the following chapter (Chapter 6).

5.4.2 Distributed Incremental Consistency Checking

Below, we continue to describe our approach to tackling document distribution and introduce a distributed version of an incremental checking algorithm. The algorithm capitalises on a number of characteristics of mobile agents and relies on the infrastructure provided by the software agent architecture. Embodied in the algorithm are the locality of access to documents during extraction of nodesets, the use of agent itinerary and autonomy of execution of a mobile agent at numerous locations.

The distributed incremental algorithm (Fig. 5.7) follows the 3-stage structure of the incremental algorithm (Fig. 5.6), but these stages are executed at different hosts on the network. The initialisation code is executed at each host, where one or more documents are located. An initial exhaustive consistency check of local documents occurs independently at each location (Fig. 5.7, line 2), and, similarly to the original incremental algorithm, aims to establish initial applicability of consistency rules to all participating documents.

Stage 2.1 (Fig. 5.7, lines 3-18) of the distributed algorithm is executed at the location of a modified document. Consistency rules, relevant to the modification, are selected here. Although this fragment is executed for every document change, rule applicability policies, mentioned in Chapter 3, may disallow subsequent execution of consistency rules for certain changes, or require execution of additional rules. To establish relevance of a consistency rule, the algorithm carries out the comparison of XPath expressions in a consistency rule with XPaths to changed document elements.

Stage 2.2 (Fig. 5.7, lines 18-31) is executed at numerous locations of one or more documents, relevant to the executed consistency rule. Initial relevance of documents to rules is determined from the results of exhaustive consistency checks, executed at initialisation, and is updated for every document with respect to every document change (Fig. 5.7, lines 12, 13).

```
0: REM Initialisation part: RUNS ONCE
1: REM Stage 1: identification of initial relevance of documents to
consistency rules
2: Execute exhaustive check on all local documents.
3: REM Incremental check: RUNS ON EVERY CHANGE
4: REM Stage 2: on document change
5: IF document has changed THEN BEGIN
6: REM Stage 2.1: identification of relevant rules
7:   Compute a TreeDiff between the current document version and its
     backup copy;
8:    FOR each consistency rule DO
9:      FOR each XPath expression in the rule DO
10:       IF an XPath to the change is comparable
             to the XPath in the rule operator
11:     THEN BEGIN
12:       REM: Remember relevance of the document to the consistency rule
13:       Set the document as relevant to the consistency rule
14:       ADD consistency rule to 'Relevant' rule set
15:     END;
16:   NEXT expression;
17: NEXT consistency rule;
18: FOR each location of relevant documents DO
19:   REM Stage 2.2. distributed checking of relevant rules
          at each location of relevant documents
20:   FOR each consistency rule in Relevant rule set DO
21:   REM Stage 2.2.1: processing of relevant documents
          at current location
22:     FOR all expressions in the consistency rule DO
23:       FOR each local document DO
24:          IF document is relevant to the rule THEN
25:          Find elements, corresponding to the expression,
             in the current document, and store them in the NodeSet
             for this consistency rule;
26:       NEXT document;
27:     NEXT expression;
28:   NEXT consistency rule;
29:   REM Stage 2.2.2. migration to the next location
30:   Move to the next location
31: NEXT location;
32: REM Stage 3. Generation of links
33: FOR each consistency rule DO
34:    Retrieve values of the elements from the NodeSet,
       and execute rule language operators on these values.
35:    Create consistency link depending on return value
       of the topmost operator in the rule;
36:  NEXT consistency rule;
37: END;
```

Fig. 5.7. Pseudo code for distributed incremental consistency checking.

At each location, where Stage 2.2 is executed, document nodes, selected by consistency rule XPath expressions, are stored in a nodeset (Stage 2.2.1, Fig. 5.7, lines 21-28). Line 30 triggers relocation of the checker agent, implementing the incremental checking algorithm, together with the collected nodesets to the next location in the agent's itinerary. At the next location the nodeset collection (Stage 2.2.1) and the following migration (Stage 2.2.2) are repeated.

The remaining part of the algorithm (Stage 3, Fig. 5.7, lines 32-37) computes a result of the distributed check. The operators in each relevant consistency rule are executed on the values of document elements, previously collected in the nodeset. For each rule, the root operator computes a

consistent or an inconsistent status of a consistency relationship. Consistency links are created between elements of the related documents.

The Stage 3 can be executed at the same location as the last nodeset collection, or at any other location, where sufficient computational resources can be provided for execution of the rule operators. Additional flexibility of the distributed incremental checking algorithm in separating the processing and the information retrieval stages enables its deployment on "thinner" and mobile clients, supports intermittent connectivity for participating hosts and allows a user to off-load consistency checking processes from her primary workstation.

The discussed algorithm intentionally does not go into detail about propagation of generated consistency links (Stage 3) to document locations. It also omits compilation of the agent itinerary and the details of the infrastructure, which makes document relevance information, collected at initialisation, available to the checker agents at runtime (at Stage 2.1). This functionality and the required infrastructure are provided by the software agent architecture for distributed consistency checking, described in the following chapters. The distributed incremental checking algorithm forms a basis of a mobile consistency checking agent, one of the components of the architecture and specifies the process of a distributed consistency check.

The proposed distributed incremental approach provides the means for concurrent checking of several documents, related to the same or different consistency rules. The nodeset collection stage can be carried out concurrently by several instances of a mobile checker agent at different locations. Concurrently retrieved nodesets are subsequently concatenated before link generation takes place. Multi-agent consistency checks, supported in an implementation of the software architecture [Chapter 8, Scenario III], are based on this feature of the distributed consistency checking algorithm.

5.5 Incremental Checking Challenges

Implementation of the incremental checking algorithm is by our choice an extension of the codebase of the exhaustive consistency checker XLinkit. We have chosen to reuse existing code unchanged in order to support interoperability between the incremental and exhaustive checkers, so that both could be used by the same user, although for different purposes.

Some of significant challenges, which we were able to overcome during implementation of the incremental checker, are described in Appendix F, where we consider the task of identifying relevance between rules and documents, and a problem of comparing XPath expressions for identification of relevant consistency rules. We describe the proposed lightweight string-matching approach for XPath comparison [Appendix F, F.2], and its more accurate amendment, where XPaths are selectively executed to improve comparison accuracy [Appendix F, F.3].

The lightweight rule selection approach and its amendment for selective XPath execution constitute initial attempts to tackle the XPath comparison problem. The author is a part of further work, aiming at generalizing the proposed approach and implementing a comprehensive algorithm for comparing XPath

expressions of any complexity. In comparison of the applicability of the lightweight approach and its somewhat more heavyweight extension where complex expressions are executed, the actual deployment of one or the other approach is dependent on the proportion of consistency rules with complex XPath expressions in the rule set. Performance overhead, introduced by rule selection in the incremental checking algorithm is considered in Chapter 10, where performance measurements of an incremental and exhaustive algorithms are provided.

We also consider a task of merging results of incremental checks into a common linkset [Appendix F, F.4], and a merger of results of an incremental and an exhaustive check. The merger problem is relevant to interoperability between the incremental and the exhaustive checkers.

Finally, we introduce a UML model distribution tool, which we constructed in order to enable a number of software engineers to concurrently develop parts of a single UML model at several distributed locations. The UMLXMI tool [Appendix F, F.5] separates an XMI representation of a UML model into a number of XMI files, each containing representations for one or more model elements. These files can be distributed and authored independently at distributed locations. The tool also provides functionality for merging separated model elements into a single UML model, and checks internal consistency of the model by establishing uniqueness of element identifiers.

We used UMLXMI in construction of the scenarios (Chapter 8), on which an implementation prototype was tested. We found, that the tool facilitates performance of a differencing algorithm (TreeDiff) by reducing document size for representations of model elements in comparison with the size of a complete model. The use of UMLXMI thus has a positive impact on the performance of the incremental checker. This characteristic of UMLXMI is a welcome addition to fulfilment of the initial objective of enabling distribution and distributed development of UML models.

5.6 Summary

Exhaustive consistency checking of a substantial number of consistency rules on large documents, typically required in the software engineering domain, could take up significant computational resources away from execution of the development environment that a developer utilizes. Exhaustive checks create consistency links between *all* related document elements in the project. The exhaustive linksets are snapshots of all consistency relations, taken at the time of a check. Exhaustive checks are best used for a project milestone or a final release, when all documents are checked to ensure that consistent state of all relations has been achieved. An existing consistency checker, XLinkit, provides exhaustive checking functionality.

In a commonly used incremental development process, developers sequentially introduce modifications into the documents, and are interested in effects of these modifications on consistency relations between the modified elements and other documents in the project. In this chapter, we argue that exhaustive checks are ineffective in meeting the needs of incremental development. We outline the

XLink fragility problem, which makes comparison of exhaustive linksets difficult and prevents traceability of an impact of individual document modifications on the state of consistency relations.

We propose an incremental checking approach, which extends the functionality of the XLinkit consistency framework. In the incremental check only those consistency rules are executed, which are relevant to particular document changes. The resulting linkset is related to the changed elements and other elements of the same type, thereby indicating an impact of a document change to the developer. The incremental approach is not intended to replace an exhaustive checker; we have outlined the application domains for both technologies and consider them complementary.

The validity of the incremental checking algorithm in correctness of computation of consistency links is equivalent to that of the exhaustive algorithm. The incremental algorithm extends the exhaustive algorithm with selection of relevant consistency rules with respect to a certain document modification. All consistency rules, which correspond to the modified elements, are selected for an incremental check, and therefore no consistency relations that may be affected by the modification are omitted in the check. The incremental algorithm operates on the documents, which have been found relevant to the selected consistency rules by an exhaustive check at the initialization stage of the incremental algorithm. Relevance of *modified* documents is updated as a result of the incremental check, where the same procedure is followed for nodeset extraction and link generation as in the exhaustive algorithm. The validity of the incremental checking algorithm is therefore the same as that of the exhaustive algorithm. The latter question is considered elsewhere [Nentwich et. al., 2000b].

In this chapter, we have described the incremental checking algorithm and its distributed version, which takes advantage of local access to distributed documents. The distributed incremental algorithm separates consistency checks of individual rules on different documents in space and time, and avoids the centralisation, imposed by the existing consistency checking approach. We have adopted the distributed incremental approach for the consistency checking software agent architecture, which we describe in the following chapter.

We have referred to some of the technical challenges that we had to address during an evolutional implementation of an incremental checker on the basis of XLinkit. Among these challenges, we have considered comparison of XPath expressions for selection of relevant consistency rules (Appendix F, F.2-F.3) and an approach to distribution of a UML model (Appendix F, F.5), which gives developers extra flexibility and improves performance of incremental checks. Both solutions are implemented in the architecture prototype, which we demonstrate on scenarios in Chapter 8.

The incremental checking approach for XLinkit and a distributed incremental checking algorithm, based on local document access, are novel contributions of this thesis. Our primary contribution, the software agent architecture for distributed consistency checking, builds on the mentioned contributions.

Chapter 6 Software Agent Architecture for Distributed Consistency Checking

6.1 Introduction

The software agent architecture for distributed consistency checking [Smolko 1999, Finkelstein and Smolko 2000, Smolko 2001] emphasises deployment of mobile agents, and also contains stationary components. *Stationary* agents within a distributed networking environment are deployed to provide supporting services to *mobile* agents, which move between network hosts, take advantage of supporting services and provide services of their own, capitalizing on locality of access to distributed resources.

The design of the architecture is primarily driven by the functional requirements for a distributed consistency management system (Chapter 3). Among the key principles, targeted by this architecture are openness, event-orientation, distribution transparency and processing locality.

Openness – The architecture should allow re-configuration, integration and removal of information resources and instances of architectural components. Openness to new types of components is to be achieved by building on defined interfaces of loosely coupled components.

Event orientation – The operations, carried out by components, should be triggered by, and their timing should be synchronized via use of a distributed event notification mechanism. Loose coupling of components is achieved through propagation of events between the components and processing of these events by the components.

Distribution transparency is a characteristic of a location service, which is made available to the components that are seeking access to remote distributed components. In the context of this software agent architecture, distribution transparency is used by mobile and stationary components for location-transparent communication [Chapter 4, 4.4.1] via message exchange.

Processing locality – Mobile components benefit from *processing locality,* when services are provided at the location, where such services are required [Chapter 4, 4.4.3].

To help achieve these considerations, all functional components within the architecture are embodied and abstracted through agents. In this chapter, we break down the desired distributed consistency checking functionality, outlined in a distributed consistency checking algorithm [Chapter 5, 5.4.2], into clearly defined roles. These roles are embodied in the tasks that the agents can perform; subsequently, a larger task of distributed checking includes the management of agents and resources and is expressed in terms of agent interaction. The concept of roles allows the architecture to be modular and extensible in its design and also facilitates the development of common communication mechanisms and data formats to help deliver interoperability.

The following subsections describe the design of the architecture in terms of the roles of each agent, its place within the architecture, the functionality that agents provide and interactions that can be established with other agents.

6.2 Architecture Description

The development of a distributed agent-oriented architecture for consistency management started from an idea that all distributed consistency services *can* be provided by means of deploying a mobile agent of only one type. It was envisaged that this mobile agent, equipped with all necessary functionality, would be able to monitor, check and maintain consistency relations in the distributed setting. Then, open problems in the field of distributed consistency management were identified and explored [Smolko 1999, Smolko 2001], and it has become apparent, that *decomposition* of the consistency management task into sub-tasks is required in order to provide an effective solution. As one example, in order to address the problem of heterogeneity of document formats, a mobile agent would have to be fluent in all file formats and would be requiring frequent updates as new formats are being introduced. Being decomposed into a sub-problem, the issue of heterogeneity has been addressed by use of static parser components and proposed deployment of XML as an intermediary document format and a consistency rule language [Ellmer, et al. 1999, Nentwich, et al. 2000b].

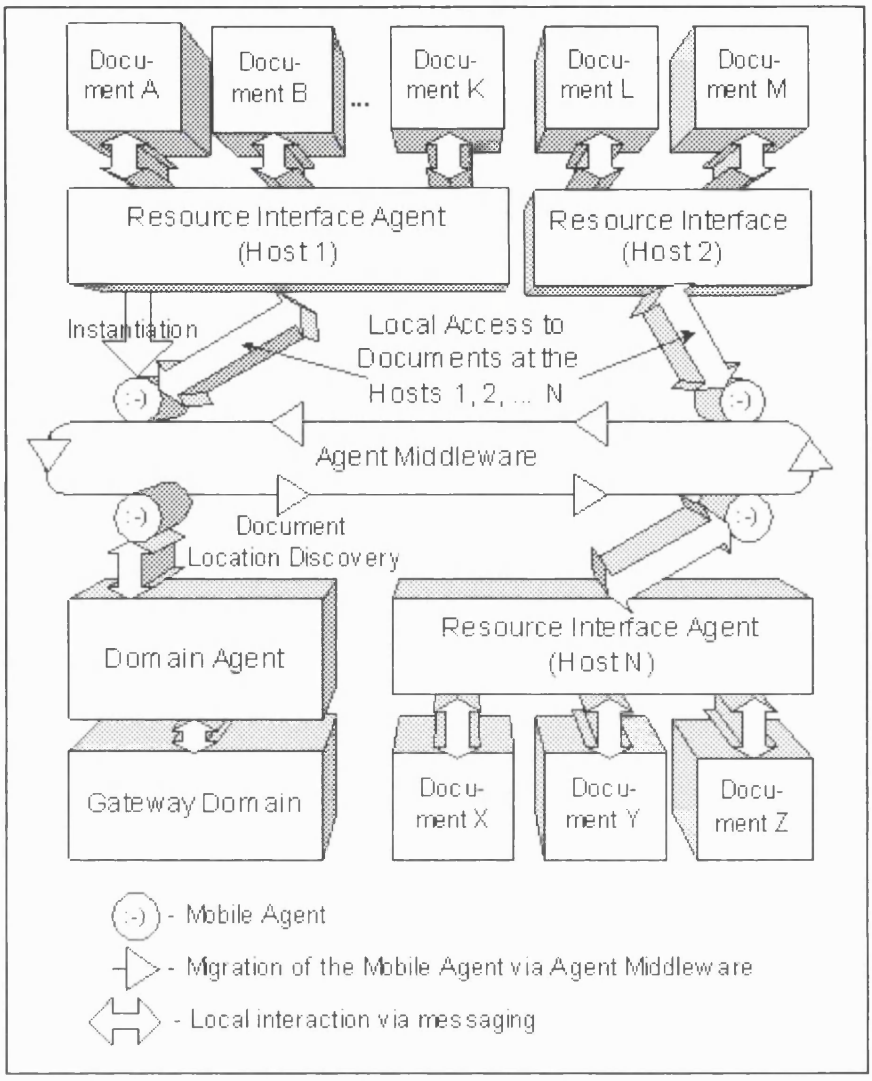


Fig. 6.1. A single domain of the software agent architecture for distributed consistency checking.

An overview of the structure of a domain of the agent architecture is provided in Fig. 6.1. The purpose of the domain is to group agents, resources, and hosts into logical collectives and to provide a standard environment for the agents to function in. The domain can include one host machine, or may span across a number of host machines (host1, ... hostN). One or more heterogeneous documents are located at each of the hosts in the domain. The documents, participating in the consistency management system, are monitored by stationary *Resource Interface Agents*. On numerous occasions during a life cycle of the documents, *Consistency Checking Agents* are instantiated by resource agents for the purpose of verification of consistency relations, in which the documents participate. The mobile consistency checking agent draws information on location of documents, configuration of domains and capabilities of its own runtime environment from the *Domain Agent*. The architecture also provides for a *User Interface Agent*, which would enable user involvement and could provide the representation function for results of consistency checks as necessary.

A minimal configuration of the consistency management system involves a single domain. With the growth in the size of the system, the architecture scales up with addition of hosts into domains, and

scales out with addition of domains, connected between each other via *Gateway Domain Agents,* which perform routing functions and serve as firewalls.

6.2.1 Resource Interface Agent

Resource Interface Agents are stationary software agents, existing within a domain to provide a level of mediation between resources and mobile agents. The resource agent runs on each host, where one or more resources are located. In the context of this thesis, resources are document, participating in consistency checks.

The main objective of the Resource Interface Agent is monitoring a set of resources; the agent is constructed in order to implement the functional requirements on distributed document monitoring and change identification. The agent also "understands" the mechanisms for access to heterogeneous resources, and ensures that mapping of numerous heterogeneous document formats to the common XML format is carried out. XML is deployed across the consistency management system for uniform representation of information.

The functions of the Resource Interface Agent are described below.

1. It possesses complete **knowledge** on the **structure** of the resource and **access protocols** to the resource. In particular, this ensures that the XML representation of a resource, used for consistency checking, is updated and contains the most current version of the relevant original document (for instance, a UML model). Implementation of this function addresses a problem of heterogeneity of representation formats for resources.
2. It **advertises** the **presence of the resource** itself, and the presence of a particular type of resource by registering the resource with the domain agent. This information is used by mobile Consistency Checking agents to find out which resources are present within a domain.
3. It **mediates access to the resource** at resource level. The Resource Interface Agent is able to resolve conflicts between numerous mobile Consistency Checking agents' access requests to the resource. To any given mobile agent, the resource agent permits access to the resource in accord with the checking agent's domain-wide access permission (as allocated by the domain agent). Further access restrictions can be imposed on a particular mobile agent by the resource agent, in addition to domain-wide permissions.
4. The resource agent **carries out monitoring** of the resource **for modifications.** When a resource is updated, the resource agent ensures that an updated copy is translated into XML, and a backup copy of the XML representation of the previous version is saved. If a history of document changes and versioning support is required, the resource agent is the component of the software agent architecture, most suitable to provide such functionality. Incremental difference between document versions is computed, so that specific consistency relations, relevant to the changes, can be checked.
5. The resource agent **executes** the **rule selection algorithm** to identify the consistency rules, relevant to a particular document modification.

6. The resource agent **initiates consistency checks** of the relevant rules, facilitates their execution, and takes action on the result of the check. Storage of consistency links and notification of the user through a User Interface Agent are the actions usually performed upon termination of a consistency check.

The functions performed by the resource interface agents and interactions between the resource agents and the consistency checking mobile agents form the crux of the distributed software agent architecture. It should be noted, that resource agents enable mobile agents to access heterogeneous resources in a generic and flexible fashion. At the same time, it is not the access mechanism, but rather the way, in which mobile agents *interpret* document structure and content, that will determine usefulness of the consistency management system for an end user.

The nature of resources is not envisaged to be ultimately limited to distributed documents or document sets. A resource can be any system that exposes an external interface, through which it communicates or can be accessed by a resource interface agent. In this way, the architecture can be extended to integrate numerous additional types of heterogeneous resources through development of functionality of the resource interface agents.

6.2.2 Domain Agent

The Domain agent is a stationary agent, which supervises activities of other mobile and stationary agents in the domain. It is responsible for a number of functions, outlined below.

1. It **provides lookup service** for names, locations and types of all resources, located at the current domain and may cache information about resources, located outside of the domain. By providing lookup services to consistency checking mobile agents, the domain agent **controls access to resources** at the domain level. All architectural components rely on the provided location service, because no assumption is made about existence of a-priori knowledge by any of the components of locations of distributed resources.
2. The domain agent is an **initial point of contact for mobile agents** within the domain. It authenticates mobile agents and performs a verification check on mobile agents wishing to execute within the domain. Every agent is assigned access rights within the domain. These rights serve as guidance for the resource agents of the domain on access permissions to particular resources. Depending on the security policies of the domain, agents that cannot be authenticated are rejected or are given limited rights of an anonymous agent.
3. It **provides a migration service** to mobile agents entering and leaving the domain. Unless the underlying mobile agent middleware ensures atomicity of the migration operation and supports strong mobility, the domain agent saves the migrating agent's state and transfers the agent to a new location. It must ensure that the mobile agent is transferred successfully, and is able to continue execution at the destination domain. In case of failure, the domain agent allows the mobile agent to

resume execution at the present location or to choose another migration destination. These domain agent functions assist active agent migration and improve fault tolerance.

4. The domain agent **launches** migrating mobile **agents in a** suitable **runtime environment**. For Java-based agents, the provided environment is a Java virtual machine, for binary executable agents – dedicated secure memory space and a sandboxed virtual machine. Functionality of the provided environment in a particular case will depend on the access permission granted to the mobile agent (its "trustworthiness") and the functions, which the agent declares it wishes to perform ("goals", or "intentions").
5. The domain agent ensures that the domain does not become overwhelmed by agents, whether mobile or stationary. Assigning allowances to an agent's execution and restricting the number of agents that can exist within the system at a given time are possible strategies for resource management. For one example, if a mobile agent attempts to monopolise a resource within a domain, it could only do so for as long as its allowance lasted. It is envisaged, that particular methods of **resource management** used for the implementation prototype will depend on the choice of runtime environment for software agents.
6. The domain agent **provides a point of registration** for all agents, resources and users within the domain. In this area, both stationary agents (resource and user interface agents) and mobile agents can register and advertise their knowledge, functionality, interests and intentions. In this way, the domain agent may be considered as a meeting point for agents of different types, where communication and sharing of information between agents becomes possible.
7. The domain agent **advertises a list of agents**, which are resident in the domain, and their functionality for the benefit of other agents. Distributed agents are initially unaware of availability and presence of users, resources or other agents on different hosts and must query the domain agent to discover available services. For example, a list of operating mobile agents is collected by the domain agent upon registration of each migrating agent. This list is used by the domain agent to eliminate redundant consistency checks and to facilitate co-operation and information exchange between mobile agents, which are checking related consistency rules.
8. In-domain **resources** are **advertised outside the domain** through *gateway* domain agents. In this way, mobile agents of other domains can learn of the location of a specific resource in advance of their migration to the destination domain. By making in-domain resource lookup table accessible by the gateway domain, the distributed architecture will ensure that mobile agents become aware of all resources, which need to be processed during distributed consistency checks.

Domains group stationary agents that are running on several network hosts, resources that are stored at these hosts and users, authenticated by the hosts. It is envisaged that domains are configured by administrators in such a way, that they would include resources, grouped by *relevance* to each other, although other approaches can be used.

Grouping resources into domains enables the distributed architecture to scale up with the growth in number of resources and network hosts. Domains are self-contained entities: most of the services required by software agents within the domain are provided by the domain agent locally in this domain. In this way, it is possible to optimise the performance of an implementation of the distributed agent architecture by grouping highly related resources into a domain, or ungrouping them into different domains to balance the processing load. In the first case, mobile agents will be able to obtain efficient access to resource lookup information and to multi-agent coordination functions, provided by the domain agent. However, if a large number of mobile agents execute within the same domain at any given time, the lack of resources may justify distribution of the load across several hosts or even different domains. In such load-balancing scenario, more operating resources for execution become available, but check efficiency may suffer as agent migrations become necessarily more frequent. Replication of the domain agent across a number of network hosts also has a capacity to improve load balancing.

By introducing a notion of *domain*, where services are available locally from the domain agent, the distributed agent architecture aims to "marry" the advantages of distributed and centralised processing models. Making a decision on a particular configuration of domains for a specific application of the architecture is outside the scope of the development of the architecture itself, and is rather a configurational choice of the system user. However, we attempt to provide guidelines for this important issue. In Chapter 10, where a software architecture prototype is evaluated, we give recommendations for an optimal distribution configuration. These recommendations are based on conclusions, drawn from performance benchmarks of the software agent architecture implementation prototype.

6.2.3 Gateway Domain Agent

In order to provide a hierarchical information structure and to distribute document name and location information, a *gateway* domain is used as a grouping of domains and other inter-domain gateways. A Gateway Domain Agent is a stationary agent that provides the logical connectivity between individual domains. It allows a number of domains, which perhaps contain related resources and host co-operating teams of users to be treated as a single virtual domain. The Gateway Domain Agent offers the following services to the domains within its locale:

1. It **provides a hierarchy of domains** where queries from sub-domains can be dealt with and resolved by domains at a different level in the domain hierarchy. The queries may relate to lookup of resources by name and/or type, and it is envisaged that they are handled similarly to the name resolution DNS (Domain Naming Service). The gateway domain agent would forward the query to other connected domains and aggregate the responses it receives into a single set of results. For example, loosely resembling the DNS analogy, gateway agents will provide access to "countries" at the root level, regions at the next level and so on down to the constituent domains at the "leaves".

2. Existence of the gateway domain agent allows **firewalls** at the architectural level, where access to information within the sub-domains of the gateway domain can be restricted to requests from mobile agents, originating from certain pre-defined domains.
3. As the architecture aims to support **heterogeneous networks** and hosts, gateway domain agent can equip the mobile agents entering the gateway domain with the necessary software libraries to enable them to execute and communicate within the virtual domain. Such libraries can implement service descriptions, communication protocols and the like, which are specific to and are required within the sub-domains. Efficiency of mobile agents may thus be improved, as they would not need to transfer such libraries with their code between gateway domains; all the necessary code can be linked at runtime.

The Gateway Domain Agent is used as a mechanism for supporting distribution and hierarchical organisation of resources, located at multiple network hosts and groups of such hosts. It attempts to combine the best of *location transparency* services, which are implemented via document name lookup services, with *location awareness*, which benefits the mobile components with localised access to in-domain resources. Building a *hierarchy* of domains, connected via gateway agents allows us to scale out the architecture, as most of services from a given domain are requested and provided within the same domain. The burden of dealing with document location discovery requests can be spread throughout the hierarchy, thus reducing the demands on individual gateway and domain agents.

6.2.4 Consistency Checking Mobile Agent

The Consistency Checking Agent is a mobile software agent, which can migrate between hosts in a domain and across the domain boundary between the domains. Mobile agents constitute the mechanism, which implements the consistency checks between the distributed documents.

One of the advantages that the proposed distributed agent architecture provides is a goal orientation of its mobile agents. Specifications of goals are assigned to mobile agents on instantiation; specification of different goals allows us to differentiate agents by their roles. Specifications are specified in terms of communication primitives, such as "check" for checking of consistency rules, "notify" for messaging and notification of events or results of the check, or "select" in preparation of a consistency check by locating relevant distributed documents.

Use of the structured goals, analogous to inline communication primitives, allows reuse of the mobile agent class for creation of agent instances with different goals, thus eliminating a need for multiple heterogeneous mobile components. The distributed software architecture is based on reuse and cooperation between instances of simpler software agent templates, instantiated with the goal statements, rather than on a large family of heterogeneous agents, implementing specific system functions, as proposed in [Prestegard, et al. 1999].

The essential functions of mobile agents are defined by specific tasks that agents are to carry out, as described below.

1. A mobile consistency checking agent can be **authenticated by** the **goal statement** that it carries, and a unique identifier or a signature, which can be verified at the originating ("source") domain. Additionally, an implementation of the distributed agent architecture can make use of mechanisms to ensure correct transmission of mobile agents during migration. A mobile agent without a defined goal or without a valid signature can be rejected by the receiving domain.
2. It **determines the route of migration** between hosts based on the *itinerary*. This itinerary contains locations of distributed documents, related to the consistency rule being checked by the agent. The itinerary is a result of a request to document location service or type lookup service of the local domain agent. In addition to the information available locally, the itinerary is complemented by an inter-domain document discovery request, where the query is forwarded to the gateway domain and the search for relevant documents continues through the domain hierarchy.
3. The mobile agent possesses the characteristic of **persistence**, and its ability to migrate enhances durability. The hierarchical domain structure allows the agent *not* to solely rely on services of any given single domain, even if a part of the agent's itinerary is bound to remain uncompleted. If failures occur in a domain, the mobile agent can continue its execution at another domain. Such flexibility is particularly useful for distributed computing, and persistence is essential for disconnected operation, where the user is only connected to the network for short time periods.
4. It is capable of **communicating** with stationary resource interface agents, and with other mobile consistency checking agents. The communication function allows mobile agents to **co-operate** and **exchange information** about their goals, and the information they have previously acquired from distributed resources.
5. The mobile agent "marries" the advantages of location transparent messaging and network awareness, which allows the agent to **access resources locally** by migrating between network hosts and between the domains.

Consistency checking mobile agents perform a number of roles within the distributed architecture. These are location of documents, relevant to a consistency rule, checking of rules and messaging of events results of the consistency checks between network hosts. In development of the software agent architecture, we sought to minimise the number of different types of architectural components. Therefore, solutions for the agent roles are provided based on the same mobile agent code.

All three roles have a lot in common. Migration and serialization of data structures is applicable both to consistency checking and messaging. Location of relevant documents is deployed prior to and forms a part of a consistency check. In addition to notification of events, messaging is also deployed throughout location discovery of relevant documents. Logically, differentiation between particular instances of mobile agent code is carried out by means of different goals, assigned to the agents according to their role.

6.2.4.1 Locating relevant documents

Upon instantiation, a consistency checking agent is given a reference to the consistency rule, which needs to be checked. Usually, this consistency rule is selected by a resource interface agent, because it is relevant to a particular document modification. Alternatively, a user may have requested a check of this rule, or the execution of this check has been scheduled.

The checking agent queries a domain agent for an itinerary, containing names and locations of documents, related to the rule. Relevance of individual document instances to particular consistency rules is reported to the domain by resource interface agents, and is one of the activities demanded by the incremental checking approach (Chapter 5). The resulting mobile agent's itinerary contains URLs of the document instances. An example itinerary for the consistency rule, checking well-formedness of generalizations in a distributed UML model, relating to a number of generalizations, located at different hosts, is shown in Fig. 6.2.

```
<Goal action="Select">
<RuleFile href="generalizations.xml"
xpath="/consistencyruleset/consistencyrule[1]"/>
<!—Itinerary – result of discovery of documents, relevant to gen1
consistency rule -->
<DocumentSet name="Relevant to Generalization rule">
  <DocFile href="A/XMI/Generalization0.xmi" type="gen"/>
  <DocFile href="A/XMI/Generalization1.xmi" type="gen"/>
  <DocFile href="B/XMI/Generalization5.xmi" type="gen"/>
</DocumentSet>
</Goal>
```

Fig. 6.2. Consistency checking agent's goal "select" and the result – the agent's itinerary.

The consistency checker agent's itinerary contains references to the rule "generalizations" of the UML well-formedness rule set [Appendix A], and to three relevant documents with their type – generalization – indicated. The documents are located at network hosts A and B; subsequently, checking of this rule will require migration between the respective hosts.

6.2.4.2 Checking consistency rule

In order to carry out a distributed consistency check, the mobile agent migrates between network hosts and gains local access to the documents, specified in its itinerary. Following the distributed checking algorithm [Chapter 5, 5.4.2] at each host specified in the itinerary, the mobile agent extracts values of document elements, which participate in a consistency relation, expressed by the rule.

A case of redundant consistency checks may arise, when the same consistency rule is checked *independently* and non-cooperatively by numerous mobile agents. A redundant check occurs when changes are concurrently made to a number of distributed related documents. Consequently, independent consistency checks of the same rule are triggered at all distributed locations of such documents.

A procedure is in place, which facilitates co-operation between redundant consistency checking agents in exchange of already collected nodesets in order to speed up the distributed check. The domain

agent plays a key role in identification of redundancy, since redundancy is identified through comparison of mobile agents' goals, which are registered with the domain agent by each mobile agent, migrating into a domain.

Once the itinerary has been processed, the mobile agent computes a consistent or an inconsistent result of the check and links the affected document elements accordingly (Chapter 5, 5.4.2, Stage 3). Propagation of the resulting consistency links through all hosts in the itinerary, where the links are saved as local consistency link files, is one of the tasks that the *messenger* agents carry out in the architecture.

6.2.4.3 Messaging

Messaging service in the architecture is used by all architectural components for information exchange between distributed locations. The software agent architecture uses messenger agents to deliver notification of events and other communication between resource interface agents, domain agents and mobile checking agents. Conceptually, the messenger agent contains a small portion of code, responsible for persistence and failure handling, an itinerary of hosts to visit and agents to deliver a message to, and the message itself.

There are several occasions when messaging is deployed in the architecture. First of all, consistency links are propagated to distributed locations by means of messaging. The consistency checking agent sends a messaging agent to the resource interface agents of processed documents and thus notifies them of the consistency links that need to be registered. Secondly, co-operation and information exchange between redundant checking agents is based on messaging services. Event notifications of document changes, of document relevance to a certain consistency rule, and of the beginning and ending of a consistency check are registered by resource interface agents at the domain agent by means of messaging.

Messaging implements the required functionality for the event orientation of the software agent architecture, where components operate by raising, processing events and communicating these events as messages. The architecture includes a notion of a *messenger agent* for the purpose of abstraction, as no assumption is made about availability of messaging services in the mobile agent middleware, which is to be used for the implementation. The implementation prototype of the architecture, described in Chapters 8 and 9, is built on the mobile agent middleware IBM Aglets, which supports location transparent messaging between agents. Messaging agents are thus replaced by use of messaging services in the prototype, and identification of messenger agents by their goal is implemented through typed messages.

6.2.5 User Interface Agent

The User Interface Agent resides on the hosts, where users of the distributed consistency management facility are working. This stationary agent provides a user interface to the functionality offered by the architecture and the underlying distributed consistency checking framework. The User Interface Agent is capable of performing the following tasks:

1. It allows a user to **log in** to the consistency checking facility, by which the user is authenticated and acquires certain access permissions for access to the resources, consistency rules and policies.
2. If an ability to **change consistency rules** is required by the application domain, the User Interface Agent allows authenticated users to edit consistency rules, and provides a graphical interface [Zisman, et al. 2000] to make this process easier. Likewise, it can facilitate access to system policies, though in both cases a user can access these directly, without the user interface agent.
3. The User Interface Agent allows a user to **launch consistency checking agents** at any point in time of the document production process in addition to the automatic checks, triggered when particular events, occurring on the documents, are identified by a resource interface agent. The user interface agent can provide functionality and enable the user to specify which document elements need to be checked, allow the user to select consistency rules from rule database and schedule or immediately launch the consistency check of these rules.
4. It creates a user-friendly **representation of consistency links**, which are generated by mobile checking agents and sent to resource interface agents. The representation may involve prioritising inconsistencies in one way or another, i.e., by possibility of toleration, urgency or estimated difficulty of correction.
5. The User Interface Agent provides a user with a **view** on the goals, state and locations of **agents**, history of document changes and results of consistency checks.
6. The User Interface Agent may act on behalf of the user when the user is absent from the workstation. This functionality may be most beneficial in world-wide distributed development projects, where daytime hours of distributed developers differ. Prioritising requests and messages and forwarding those with highest priority to the user in a convenient form (i.e., by electronic mail) would be a simple example of such functionality.

6.3 Hierarchical Information Structure

In order to provide the required services at the domain level of the architecture, data structures have been developed to facilitate execution of such services. In order to support the organisation of the information space and to assist with data location, we use a metadata structure that records information about the contents and locations of the various resources and events in the system.

6.3.1 Document Name Table

The Document Name Table stores document names, their types and URL addresses of document locations. A table of this kind is maintained by a domain agent in each domain, and contains information about documents, located in that domain. Maintaining correctness of information in this table for each participating resource is a task of resource interface agents, which overlook the resources. The resource interface agent initially sends and subsequently updates document name and location information at the domain agent level.

The gateway domain agent maintains a similar document lookup table, where it caches document location discovery results from numerous domains. The information is updated by successive location discoveries of documents of each particular type. Despite the best effort, cached information becomes outdated, thus the domain's document table contains an additional field: expiration time, after which cached information is re-requested (Fig. 6.3).

Field Description	Comments	Example
Document Name	Full name, may include type in it, this field will be searched	ClassDiagram.xml
Document Type	Document type, or type of the UML element, represented by the document in the UML scenarios.	Class Diagram
Location	Document URL: host(domain), path, file name	host.C/ClassDiagram.xml
Relevant Consistency Rules	Identifiers of consistency rules, relevant to the document	c1, c2, c3, ...
Additional Fields at the Gateway Domain		
Last Requested	Time stamp of last request	25/06/2001, 14:19:53
Location information expires	When to re-verify existence	30/06/2001, 12:00:00

Fig. 6.3. Extract from the Document Name and Type Table.

Fig. 6.3 presents an example of a record of the document name table, stored at domain C. The domain agent of domain C is responsible for receiving the information in this table from resource interface agents.

In order to speed up distributed document location discovery between domains, our approach advocates use of partially redundant document location tables. Initially, within each domain, the tables contain complete location information on the documents of that domain. Additionally, the tables at gateway domains often possess information on most frequently accessed resources of the sub-domains. The latter information is replicated across numerous gateway domains, is redundant, and is not guaranteed to be always correct. This redundant information is acquired through inter-domain location discovery mechanism, where a gateway domain caches results of a location query, collected from the connected domains. Inter-domain location discovery example [Chapter 8, 8.8.4] explains the mechanism in detail.

The proposed caching mechanism draws on techniques often deployed in artificial intelligence applications, and is similar to the "ant path" planning algorithm [Steels 1990]. In search for a resource (food or building material), ants climb "resource pheromone" gradient, left behind by other successful ants, which have already found the way to the resource. As a result of general randomness of ant behaviour and decaying of the pheromones with time, ants form minimum spanning trees of paths.

The document location caching mechanism is based on somewhat random pattern of inter-domain location discovery requests. Correct paths to locations to sought resources, established at the gateway domain table, expire with time, alike the pheromones. The deployed exploration technique of the changing environment has proven successful in numerous industrial applications of agent-based systems [Panurak 1998].

6.3.2 Distribution of Consistency Rules

Consistency rule databases of individual domains and gateway domains are organised in a hierarchical structure, and information contained in the hierarchy is intentionally partially repetitive. When the approach was being developed, a decision had to be made whether to distribute consistency rules and store them locally to the documents, which these rules are relevant to, or to store the rules centrally and retrieve them as necessary when distributed checks take place. Each of these approaches was found to be too radical, and a solution aiming to "marry" the advantages of both has been proposed.

6.3.2.1 Approaches to Distribution of Consistency Rules

Localised storage of consistency rules at the document locations is an attractive solution. In this approach, consistency rules can be treated as resources, which makes the architecture simpler. Similar to documents, rules are then monitored for change by resource interface agents and are locally accessible by mobile checker agents. Access locality to consistency rules allows us to capitalize on certain advantages (i.e., distribution flexibility, disconnected operation, some performance advantage), in the same way as access locality to distributed documents, which we have adopted for the software agent architecture.

In the software engineering domain, we expect consistency rules to change much less frequently than the documents. In a similar fashion to UML well-formedness constraints, which are used for a running example in this thesis, consistency rules would normally specify static semantic constraints of a language, in which the documents are represented. As a consequence, there is minimal evolution of consistency rules throughout the duration of a software engineering project.

When a consistency rule does change, however, changes would almost always require update of all distributed instances of this rule in the system. As a result, complete replication of all changed rules across numerous domains must then be carried out. A distributed rule update mechanism must be in place to ensure that once a change in a consistency rule has been made, the change is *atomically* propagated through the domain hierarchy and all corresponding instances of this rule are updated.

Centralised storage of consistency rules avoids the necessity to replicate rule updates. A consistency rule would need to be updated at a single location, and all following consistency checks will take into account the new instance of the rule.

Despite the relative ease of deployment that the centralised storage of consistency rules provides, this solution exposes the rule database as a single point of failure. The software agent architecture aims to provide a non-centralised solution for the task of distributed consistency management of heterogeneous documents. It advocates carrying out consistency checks *locally* at locations where they are required, and thus deployment of a centralised storage facility for consistency rules would undermine the principles, on which the architecture is based.

6.3.2.2 Proposed Solution for Distribution of Consistency Rules

In our approach, consistency rules, relevant to the document types that are found in a domain are stored locally within that domain. All consistency checks, originating in this domain, access consistency rules locally. This approach makes a domain self-contained and ensures, that in-domain consistency checks can be carried out during disconnected operation.

Gateway domains contain a comprehensive database of all consistency rules, applicable within the sub-domains, and the system administrator or users with sufficient privileges make changes to the rules at a gateway domain in the first instance. Mobile messenger agents are then automatically invoked by the gateway domain agent, which propagate all changes to domain agents and other gateway domains through the hierarchy of domains. Each domain agent replicates rule changes to the locations of the documents within the domain. Location information from the Document Name Table is used in the replication process.

Resource interface agents monitor rule sets at document locations, detect changes in a particular rule and, as a response, call back any mobile checking agents, which have started checks on the previous version of that rule. Checking is restarted with the new version of the rule, thus "atomicity" of rule update is achieved.

The proposed approach assumes that consistency rules change much less frequently than the documents. We propose a centralised retrieval mechanism to be deployed for rules that change often and for which replication of updates to distributed domains proves ineffective. When a consistency check is started, consistency rules marked as "frequently modified" are retrieved directly from the central database at a gateway domain and stored as a local copy. Since this approach ensures that only current version of the rule is used in every check, such checks are never recalled and a corresponding performance penalty is not incurred. However, during busy times requests to a centralised rule repository increase the latency of consistency checks. Frequent rule requests can potentially overload the gateway domain and will not return any result if network connectivity to the central rule database fails. In such cases, the latest local copy of a rule can be used instead.

Note that both for locally accessed and frequently modified rules, local copies of all rules, relevant to documents in the domain, exist and can be checked within the domain. Each domain can thus operate independently of other domains regardless of the state of other domains.

To summarise, the proposed approach attempts to "marry" the advantages of centralisation and distribution of consistency rules across the domains. We suggest distribution of consistency rules to locations of documents, relevant to these rules, for all rules, which are not changing as frequently as the documents. A mechanism for propagation of rule updates has been proposed for this context. Use of a centralised repository is justified for frequently modified rules. Our approach enables flexibility in configuration of an implementation of the architecture.

6.3.3 Agent Lookup Table

An agent lookup table is maintained by the domain agent and is updated when mobile agents register or unregister with the domain upon migration into the domain. The purpose of the table is for the agents to declare their goals and to advertise the information they have previously retrieved from the distributed documents.

Field Description	Comments
Agent ID	Unique mobile agent identifier. This is generated by the interface agent when a mobile agent is instantiated.
Agent Type	Complementary field, which specifies the purpose of the agent: messenger, rule selector or consistency checker. Agent type is inferred from the agent's goal.
Agent Source	Domain of origin and the host within that domain.
Agent Goal	Goal of the agent: "notify change", “select rules” or "check rules".
Consistency Rules	Unique consistency rule identifiers for consistency rules being checked by the agent.
Itinerary	List of documents, relevant to the rule, and their locations. The completed part of an itinerary specifies which documents have already been processed by this agent.

Fig. 6.4. Structure of an element of the Agent Lookup Table.

Agents, operating within the same domain, can identify whether their goals relate and if exchange of information between the agents could be beneficial to achieving the goal of one or the other. For example, relating goals of consistency checking agents would require checking of the consistency rule of the same type (i.e., type "classes" or "generalizations"). If goals of different agents coincide, then it may be sensible to establish communication between the agents in order to exchange information, which different agents have already collected. Exchanging information relating to a common goal increases efficiency of the agents' performance.

6.3.4 Event List Table

An event list is used to store events, occurring on the documents within the domain. The events of interest include document modifications, carried out by the user and the history of established consistency links, generated following individual modifications.

Maintaining the events list allows all mobile agents registered within the domain to screen through the list and to determine whether any recent events are related to the agents' consistency checking goals. In a simple example, when the event list indicates removal of a document, which is specified in an agent's itinerary, the agent updates its itinerary accordingly. In another example, an agent determines that the consistency rule it is checking had already been checked across the current domain, and none of the related documents have changed since that check. As a result, the agent can reuse the existing consistency links and refrain from re-checking the documents at this domain. In this case, the agent's goal is amended and a number of documents in the agent's itinerary are marked as processed.

Consistency links between related documents, created as a result of a consistency check, are being referenced to from the events list (Fig. 6.5). After each modification, consistency links relating to that modification can be retrieved and navigated by the user. If a document has been made inconsistent by a certain modification, the user can undo the latest changes and roll back to a previous "consistent" state of the document.

Field Description	Comment	Example
Timestamp	Time stamp when the event has occurred	25/06/2001, 14:25:20
Event Type	Type of event: OnChange, OnDelete, OnNew, ...	OnChange
Document ID	Document identifier if used, or file name	Class0.xmi
Document Name	Name of the document	Class "Teacher"
Document Type	Type of the document, or type of UML element, represented by the document in the UML checking scenarios.	Class
Node Path	XPath expression, containing a path to the changed node.	
Former Content	An extract of the sub-tree starting from the "Node Path" and engulfing the part of the document, which has changed	
Current Content	Current content of the affected sub-tree of the "Node XPath". If multiple changes have occurred, they may be specified here.	
Document Type	Type of the document	LinksClass0_01.xml

Fig. 6.5. An element of an Event List Table, registering the event "OnChange".

The event list table is maintained by a domain agent at each domain. All stationary and mobile agents, registered in the domain, report their actions on resources of the domain and results of these actions to the domain agent. These reports are stored in the event list table, and become accessible by all in-domain agents for analysis. Each agent can update its goal (itinerary) as a result of recent events, occurring on resources specified in that goal. The event list, therefore, serves as a coordination structure for all agents, distributed across the domain.

6.3.5 System Policies

The architecture uses the notion of *policy* to specify configuration parameters and execution rules for its components, where necessary. Most of the "policies" will be coded into the components upon their implementation and form the essence of component logic, but some are left to be defined by a user or system administrator, because they can change at runtime.

As an example of specifiable policies, consider consistency *rule execution policies* within a domain (Fig. 6.6, 6.7). These policies allow a user to specify a sequence of execution of consistency rules, where if one rule in the sequence is executed, all other rules have to be executed as well. Although related in the particular application domain, these consistency rules may affect different document elements or even different documents, and therefore would not be selected by the incremental checker as relevant rules to a document modification. At the same time, sequential execution of these rules may be important within the application domain that these rules represent, and the rule execution policy allows us to provide such functionality.

When rule execution policies are used, in addition to the consistency rules table, each domain has a database, where rule policies are defined. In this database, an explicit relationship between events and consistency rules is established (Fig. 6.7). This relationship binds an event, occurring on a resource, to the execution of a set of consistency rules. The policy database is thus ideologically analogous to the ViewPoint workplan.

```
<Policy type="Rule">
<Event documentType="Class">OnChange</Event>
<ConsistencyRules>
  <rule_id>gen1<rule_id/>
  <rule_id>gen2<rule_id/>
</ConsistencyRules>
</Policy>
```

Fig. 6.6. A consistency rule execution policy example in XML.

Event:	
Event On	On document of a certain type AND/OR with a certain Document ID AND/OR with a certain Document Name.
Event Type	Type of event occurred: OnChange, OnNew, OnDelete, ...
Check action:	
Rule IDs	Identifiers of the consistency rules to be executed when the specified events have occurred.

Fig. 6.7. The structure of a rule execution policy.

The example in Fig. 6.6 defines a rule execution policy, that on any change event ("OnChange") occurring on classes within the domain, executes checks of consistency rules generalizations (rule IDs "gen1" and "gen2"). A system administrator has designed this rule execution policy in order to "batch" process consistency checks of generalizations each time the classes change. Policies also allow to

"batch" together all relevant consistency rules for a pre-release documentation check, as well as to carry out an exhaustive consistency check when required.

6.4 Re-configuration At Runtime

When dealing with a distributed document management system, it is crucial to have a mechanism for dynamic update of configuration information. New documents should be able to join the system, and registered documents should be able to unregister or relocate. In approaches proposed by Giladi [Giladi and Shoval 1994] and Millimer [Milliner and Papazoglou 1994], all participating resources are registered during system initialisation. Our architecture aims to provide capabilities of dynamic re-configuration at runtime.

It is worth mentioning once again that the architecture is distributed and therefore does not have a single origin: domains are self-contained and can work independently of each other (within the context of in-domain consistency checks). Therefore, when reconfiguration is required, document name and type table within the domain, affected by a change, is updated to make account of the change. For example, when a new document is added on a host within the domain, the resource interface agent, monitoring the document set at that host notifies the domain agent of the new document's name and type, and computes the initial relevance of consistency rules to the document, which is computed at initialisation of the incremental checking algorithm. Relocation of resources between the hosts, or hosts together with a set of resources to a different domain triggers updates to the records of each document. Configuration changes can therefore be registered at runtime.

6.5 Satisfaction of the Functional Requirements by Architectural Components

In this section, we match the components of the mobile agent architecture with the functional requirements demanded from the distributed consistency checking system (Chapter 3) that these components fulfil.

6.5.1 Resource Interface Agent

The agent implements the following requirements:

Each document, participating in the managed project, should be monitored for occurrence of events, such as document changes [Chapter 3, 3.4]. The resource interface agent serves as a "watchdog", which monitors documents for modification.

When a document change event is identified, it must be determined, which elements of the document's structure have been changed, and which have been added or removed [Chapter 3, 3.4]. The agent runs the TreeDiff algorithm between a backup copy of the resource and its current version. As a result, a set of elements is identified that have been changed, added or removed.

Consistency checks occur at appropriate points in the document production process, as a result of events occurring on documents [Chapter 3, 3.5]. *Results of consistency checks must be kept up to date* [Chapter 3, 3.10]. In addition to reactive consistency checking of document modifications, the resource interface agent provides support for rule execution policies, which give a user flexibility in scheduling checks throughout the project life cycle.

An event-oriented consistency checking system should provide facilities for distributed event notification and processing [Chapter 3, 3.11]. The agent operates as a multithreaded watchdog, which raises events when documents are changed. It notifies the domain agent of the occurring events, which are registered in the events table. The resource agent processes document changes by instantiating mobile checker agents. The agent processes rule updates by re-launching consistency checks of the modified rules. In all its activities, components communicate via events – typed messages, which control the components' operation.

The Resource Interface Agent facilitates implementation of the following requirements by acting jointly with other components:

A mechanism should be in place, which would identify relevance of a consistency rule to a given document, or to an element of the document's structure [Chapter 3, 3.7]. The Resource Interface Agent follows the incremental checker algorithm and selects consistency rules, relevant to the document changes.

Consistency checks should be carried out incrementally [Chapter 3, 3.6]. The agent facilitates incremental checking by identifying incremental document changes and selecting relevant consistency rules. It instantiates the mobile consistency checking agent, which carries out the incremental checks.

Access to documents and retrieval of document elements during a consistency check should occur locally with respect to the document, rather than from a remote host across the network [Chapter 3, 3.8]. The Resource Interface Agent implements the access protocol to a local resource, and is able to provide an appropriate concurrency control mechanism.

6.5.2 Domain Agent

The agent implements the following requirements:

Consistency rules should be specified between types of documents, rather than between particular document instances [Chapter 3, 3.2]. Consistency rules do not refer to related document instances, but to document types. The domain agent participates in the resolution process: it maintains the document name lookup table, which allows it to compose itineraries for mobile checking agents. These itineraries contain references to document instances, which have been identified as relevant to a consistency rule being checked.

A policy defines applicability of consistency rules [Chapter 3, 3.3]. The domain agent maintains a table of rule execution policies. Execution of consistency rules is also regulated by security policies, which are enforced on incoming mobile checking agents from the other domains.

The domain agent and the event table it maintains also serve as a coordination point for stationary and mobile agents in the domain. The agent table, also maintained by the domain agent, allows agents to identify commonality between their goals and engage in information exchange. The coordination function of the domain agent is specific to the software architecture and is not demanded by the general functional requirements.

The Domain Agent facilitates implementation of the following requirements, by acting jointly with other components:

Consistency rules should be stored and accessed locally to the document, which they are applicable to [Chapter 3, 3.3]. The domain agent receives rule updates from the central rule database at a gateway or other domain agents in the hierarchy of domains and forwards the rules to the hosts where documents are located.

Consistency checks should be carried out incrementally [Chapter 3, 3.6]. *A mechanism should be in place, which would identify relevance of a consistency rule to a given document, or to an element of the document's structure* [Chapter 3, 3.7]. The domain agent maintains the document name and type table, which registers the relevance of consistency rules to the documents within the domain. The relevance information and the information on documents' locations at the current and other domains are used by the mobile checker agents in carrying out incremental consistency checks.

6.5.3 Gateway Domain Agent

This agent provides a number of important functions in the software agent architecture, which are specific to the architecture. Consequently, it does not directly relate to the general functional requirements.

6.5.4 Mobile Consistency Checking Agent

The agent implements the following requirements:

Consistency rules should be specified between types of documents, rather than between particular document instances [Chapter 3, 3.2]. The mobile agent composes an itinerary for checking a consistency rule. The rule itself does not indicate related document instances, but document types (or UML element types in the UML well-formedness scenario). The agent itinerary is composed from the information, available in the document name table at the domain agent, in order to specify the document instances to be checked for consistency.

Access to documents and retrieval of document elements during a consistency check should occur locally with respect to the document, rather than from a remote host across the network [Chapter 3, 3.8].

Mobile checking agents migrate between network hosts and capitalize on advantages of local access to documents.

Consistency checks should be carried out incrementally [Chapter 3, 3.6]. The mobile checking agent builds on the implementation of the incremental checking algorithm.

Results of consistency checks must provide diagnostic information on the found consistencies and inconsistencies [Chapter 3, 3.9]. Resulting consistency links are delivered by the mobile checking agent to all locations of relevant documents.

6.5.5 User Interface Agent

The agent implements the following requirements:

Results of consistency checks must provide diagnostic information on the found consistencies and inconsistencies [Chapter 3, 3.9]. The agent provides a user interface to facilitate accessibility of diagnostic information, contained in consistency links.

Consistency checks occur at appropriate points in the document production process, as a result of events occurring on documents [Chapter 3, 3.5]. The user interface agent allows users to launch consistency checks, call back ongoing checks and otherwise manipulate mobile agents (i.e., send control messages, etc.). Administrators manage the configuration of a distributed system via the user interface agent: they can launch and shut down individual mobile and stationary agents.

6.6 Summary

This chapter presented the architecture for distributed consistency checking, where each component of the architecture abstracts a set of clearly defined roles – subtasks of a distributed consistency check. The components are instrumented as autonomous agents, co-operation between which is essential in completion of the consistency checking task. The architecture is decomposed into a number of stationary agents, which provide services to mobile consistency checking agents. Mobile agents are created lightweight, as use of stationary resource interface and domain agents allows us to off-load some significant functionality from the mobile checker agent. The following services were discussed in the chapter and are used by a mobile checker agent in a distributed check: access to heterogeneous documents through resource interface agents, location of distributed documents during a consistency check through the domain agent, and user interaction with the agents through the user interface agent. Decomposition of the architecture into autonomous components enables flexibility in evolution of individual components and upgrade of their functionality, and extensibility of the services provided by the architecture by addition of components with new agent roles.

In this chapter, we described the domains – autonomous units of distribution, which form a hierarchical structure of logical connections via gateway domains. The hierarchy of information, contained at the domains, allows us to avoid the necessity of using centralised repositories for document location information, consistency rules, event histories and agent contact lists. Existence of a

continuously updated document name table and a rule table within each domain gives the domain autonomy with respect to in-domain checks, allows disconnected operation and improves fault tolerance.

In the remainder of this chapter, we provided the mapping of roles of each of software agent in the software agent architecture onto the functional requirements, demanded of a distributed consistency checking framework in Chapter 3. The mapping demonstrates how the agent tasks directly implement the demanded functional requirements or facilitate them jointly with other agents.

The following Chapter 7 is dedicated to the detailed design of the architectural components, and to the construction of a model, which enables initial evaluation and validation of the architecture. In Chapter 8 we introduce an implementation prototype of the architecture, explain and verify its operation on a number of consistency checking scenarios, arising from the distributed collaborative development of a UML model of the break scheduler application. The prototype allows us to further evaluate the performance and qualitative features of the architecture in Chapter 10 and to support the proposed software agent architecture.

Chapter 7 State Transition Model of the Software Agent Architecture

7.1 Introduction

In this chapter we describe the state transition model of the agent architecture. Prior to implementation, all software systems necessarily go through preliminary stages of general and more detailed analysis and design. We have followed this standard route in design of all architectural components. The state transition model is one of important results, which we present in this chapter. The state transition diagrams were created using the standardized representations, offered by the Unified Modelling Language.

In addition to a role that the model plays during the analysis and design stage of the architecture, the model also serves as an initial validation and evaluation of architecture. Subsequent development of an implementation prototype, its validation in deployment scenarios (Chapter 8) and the following scalability evaluation (Chapter 10) complement the material presented here.

The state transition model allows us to better clarify the particulars of the architecture, and to provide more detail on functioning of components and interactions between them. In addition, as the model estimates timings of inter-state transitions, initial performance estimation of the architecture is provided in the model evaluation section. The evaluation allows us to give an initial recommendation on a configuration of the distributed system, in which the distribution penalty can be lessened.

7.2 The Modelling Approach

We use state charts [Harel 1987] for architecture design and construction of the architecture model. The state chart is an advanced form of a state transition diagram, which defines the state space of a certain object, events that cause a transition from one state to another and actions that result from the transitions between the states. The state charts attempt to encompass in their notation a diagrammatic representation scheme, a mathematical model, describing conditional relations and transitions between the states, a linguistic model, devising names and descriptions of simple and complex states, and a pictorial model, providing an overview of the system and its decomposition that is close in the level of detail to that of an algorithm.

Mobile agent systems have been modelled in different ways; the approaches include process algebras (including π calculus), Petri nets, coordination languages (i.e., based on Linda), temporal logics, category theory and others. A survey can be found in [Seregeundo, et al. 1996]. For application

of Petri nets to mobility, Mobile Petri nets [Asperti and Busi 1996] and Communicative and Cooperative nets [Sibertin-Blanc 1994] have been proposed.

For iterative development of an implementation prototype of the software agent architecture, we have attempted to follow the Rational Unified Process, and to use a de-facto standard UML for the design of the prototype. The UML state transition diagram modelling tool (Covers), which we describe in more detail below, has also allowed us to "simulate" runtime behaviour of the statecharts, which we have designed for each architectural component. We have thus chosen to use the statecharts as a methodology for modelling the software agent architecture and to include the model in this thesis.

The notion of a state chart is actively linked with the *active object* [Booch 1994], the state of which the chart reflects. Booch has defined an active object as one that encompasses its own threads of control and is autonomous, meaning that it can exhibit some behaviour without being operated upon by another object. During its lifetime, the object performs operations in response to the external or internal events and conditions. The existence of state within an active object means that the order, in which operations are invoked, is important. In the software agent architecture, where event- and time-ordering of operations is pervasive, the best description of the behaviour of the involved active objects can be achieved in a state transition diagram.

7.2.1 Construction of a State Transition Model

The choice of state based modelling for design and initial evaluation of the mobile agent architecture is determined by characteristics of the architecture itself, which consists of event-driven, independent collaborating components, each being in a particular operational state at any given moment. Each transition between the states can be mathematically expressed in terms of the complexity of the algorithm, implementing this transition.

A significant contribution of the state based model is in its ability to validate the architecture. The model describes component behaviour and inter-component collaboration at different configurations of the system (i.e., different number of documents, concurrently checked rules, network hosts, etc.). A derived benefit of the model construction is in our ability to engage in detailed design of the architecture during the construction process.

The state based model has allowed us to simulate the complete process of a distributed consistency check, where all architectural components are involved. Prior to implementation of the architecture in a prototype, the model has allowed us to identify and correct a number of problems – configurational bottlenecks, de-synchronisation between components and alike – at design time. The model is also used to visualise component behaviour.

7.2.2 Modelling Tool - Covers

The modelling tool was chosen based on its availability and the functionality of its development environment in the design of object oriented distributed systems. The choice of the author was the

modelling language framework and design environment "COVERS" [Covers 1998], which enables the user to build and simulate the state models of such systems.

The main construction unit of a COVERS model is based on the notion of an active object. Similarly to Booch's definition, active objects in COVERS are independent, concurrently active, event-driven logical machines. The modelling framework supports diagrams of object structure and interconnection, statecharts for description of behaviour, and C++ language for implementation of transition actions in data objects. COVERS' statecharts are based on David Harel's Statecharts [Harel 1987] - a simple yet highly expressive approach, superior to conventional flat state machines [Covers 1998].

In COVERS, the structure and behaviour of active objects is specified graphically in a development environment. Then, this specification and behaviour is translated into C++ code. Active objects are compiled as independent classes, and the model is linked with COVERS runtime C++ libraries to produce the executable. Resulting models execute on the Windows platform, with ports to Unix becoming available as well.

7.3 State Transition Model of the Software Agent Architecture

Construction of the architecture model is based on a top-level active object *Domain*. The structure of Domain is shown in Fig. 7.1: COVERS alias "TSystem" applies to the Domain as a top-level system object. The domain model embodies all active objects of other architectural components, which execute within the domain. Fig. 7.1 depicts the domain structure as it appears within the COVERS Project Editor, where design takes place and executable models are compiled.

Active objects, represented as boxes in Fig. 7.1, communicate with each other via ports (shown as small circles on borders of the boxes). As such, each of the Document active objects, representing the document component of the architecture, possesses one port connected to the active object of the Resource Interface (Fig. 7.1).

7.3.1 Document active object

A Document active object is replicated in the model (which is indicated by presence of a "shade" behind the object, Fig. 7.1). Replication allows us to reuse an already developed object with its structure and behaviour, create collections of similar objects and enable objects to concurrently co-exist and co-function during execution of the model.

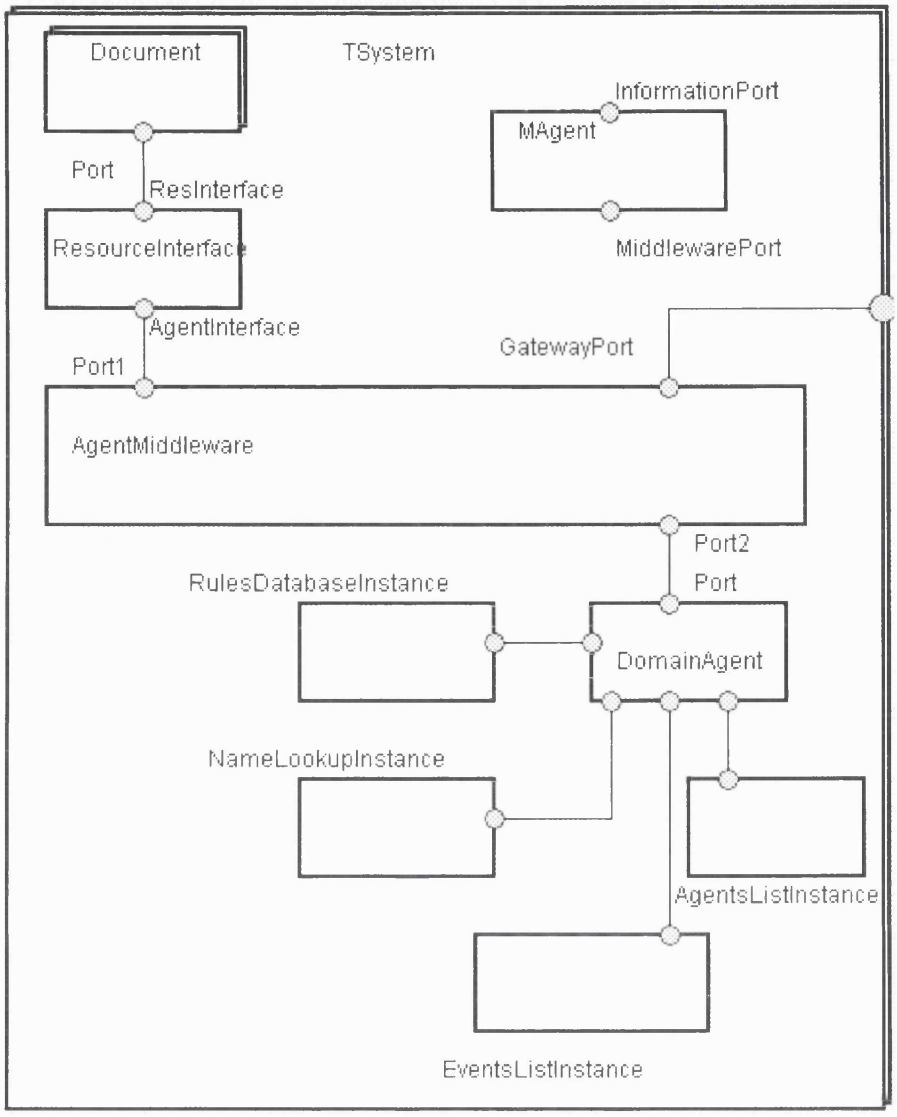


Fig. 7.1. Structure of the software architecture model.

All active objects in the simulation model have their own structure and behaviour. Behavioural aspect is represented as a state chart. Fig. 7.2 shows the state chart and behaviour for the Document active object.

The model of the Document active object contains one class TDocument and a port reference. The state chart diagram of this class contains four states and transitions between them (Fig. 7.2); each transition executes a number of operations (Table 7.1). Each state has entrance and exit conditions. These conditions also contain sequences of operations, which are executed when an active object is entering and leaving a state.

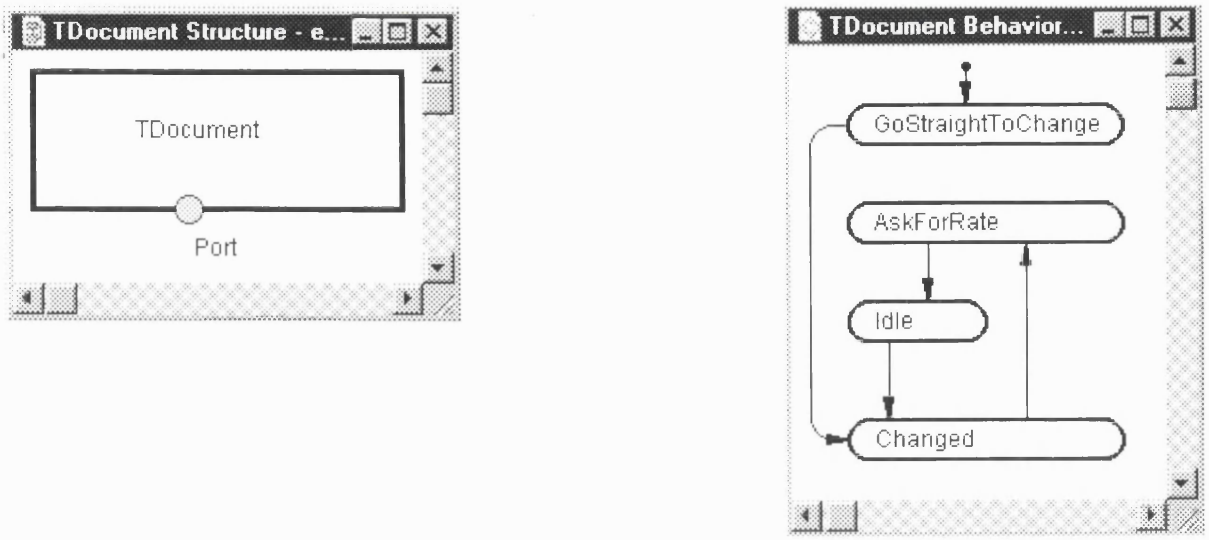


Fig. 7.2. Document active object; its structure and behaviour.

Table 7.1. Partial code: Document active object

```
// This is a partial code: FULL CODE of the model is provided in
// Appendix B.
Changed – document has been changed. Event must be raised.
Entry:
// Event has occurred. Send new event (notification message)
// "Check" through Port
  Port->Send(new TNotificationMessage("Check"));
Exit:
Transition: -> AskForRate
Idle – document is awaiting next event.
Transition: -> Changed:
Transition delay: 1000*(Rate+Rate*TExponentDistr( Rate ))
// Calculation of a delay until the next change - randomised with
// an exponential distribution model parameter Rate in milliseconds.
```

The task of the Document active object is to generate "changes" – events, which will be reacted upon by other architectural components. The model does not intend to describe particulars of individual changes (i.e., the changed elements, and which document they belong to). Unlike a working implementation prototype of the software agent architecture, discussed in the later chapters, the model does not process documents. It "simulates" processing, as if the processing is carried out by the components.

The state chart for the Document object contains a loop, where an event "Changed" is raised at random intervals of time (Fig. 7.2). A delay between the generated events is controlled by a parameter Rate (Table 7.1), which is set for each of the instances of the Document active object when the model executes. This approach allows simulating the documents that are often modified (i.e., at the initial stage of development), and the documents, modified rarely. Replication of the Document active object allows us to include a specifiable number of documents with different modification rates into the simulation

model. All "Change" events, generated by documents, are forwarded through the Port to the Resource Interface Agent, where these events are processed and trigger actions in response.

7.3.2 Resource Interface Agent active object

Fig. 7.3 shows the structure and behaviour of the Resource Interface Agent active object. The port ResInterface connects to the output ports of a collection of replicated Document objects, through which the Resource Interface Agent object receives "Change" event notifications from the documents. The AgentInterface port connects the agent to the Agent Middleware active object.

Alike the Document active object, the Resource Interface is implemented by one class TResourceInterface; the state chart defining the behaviour of this class is shown in Fig. 7.3. The state chart is composed of two macro-states: ResInterfaceOperation and LinksChange. In the former state, changed document elements are identified, critical consistency rules are selected and mobile agents are instantiated for checking of each selected rule. Mobile agents' requests for values of specific document elements are handled in the interrupt mode. Upon receipt of such request, any active sub-state of ResInterfaceOperation looses focus, the requested document elements are identified and their values are returned to the requesting mobile agent (as shown in the upper-right area of the state chart in Fig. 7.3). Finally, focus is returned to the "history" state within ResInterfaceOperation – to the state, which was active before the interruption. Identification of requested elements and return of their values is simulated in the Resource Interface Agent active object: execution of the agent's statechart is delayed. The duration of this delay corresponds to the size of the document and the number of requested elements (Table 7.2).

The LinksChange macro-state corresponds to Resource Interface agent making changes to the consistency links file at the request of a mobile agent. Because of the necessary atomicity of this operation (propagation of concurrent changes to the link set file), LinksChange state is separated from ResInterfaceOperation and doesn't allow servicing of external events. Since events are queued upon receipt, their processing resumes once the Resource Agent leaves the LinksChange macro-state.

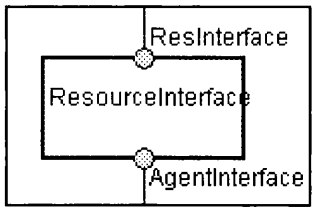


Fig. 7.3a. Structure of the Resource Interface Agent active object.

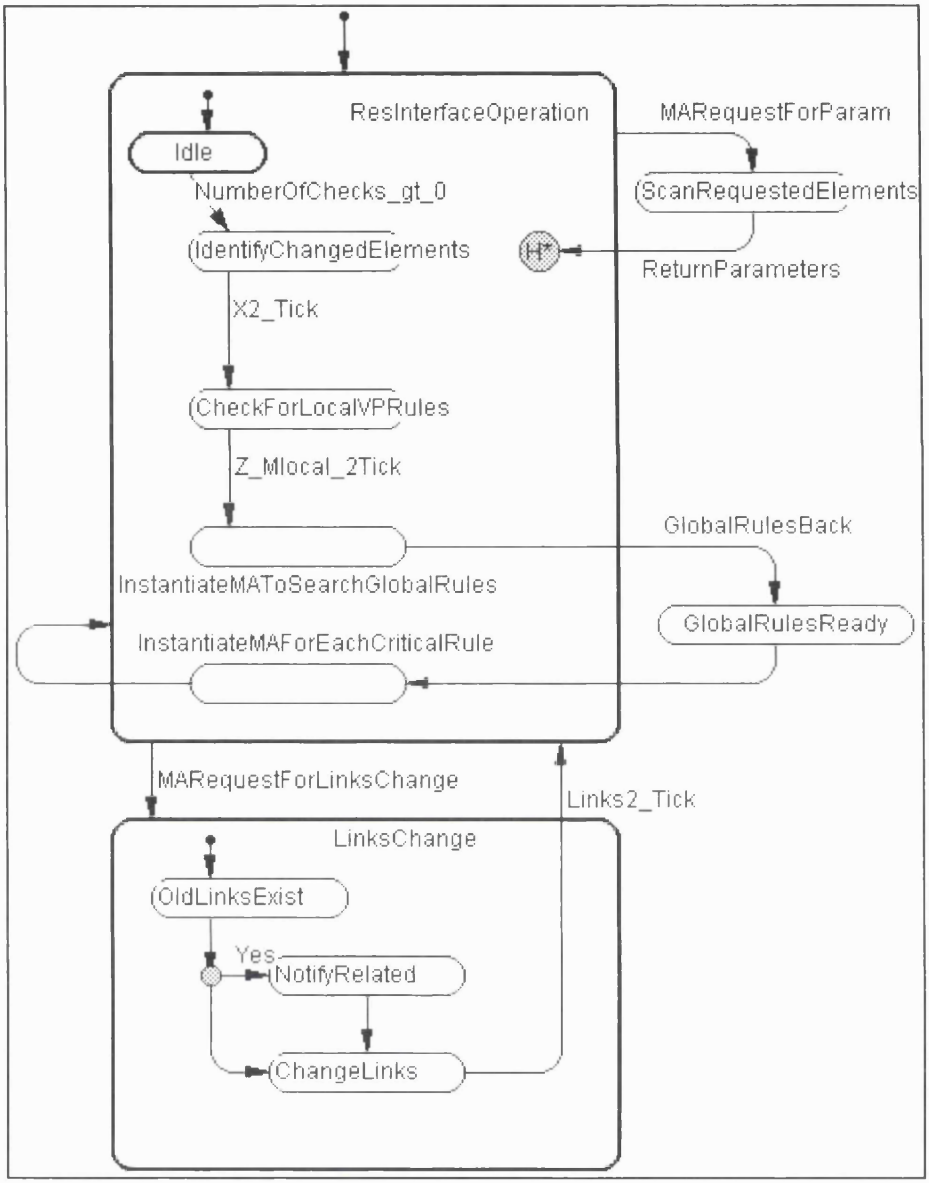


Fig. 7.3b. Behaviour of the Resource Interface active object.

Table 7.2. Partial code: Resource Interface Agent active object.

```
// FULL CODE of the model is provided in Appendix B.
InstantiateMAForEachCriticalRule
Enter: for (int i=0; i++; i<Mlocal) {
        TMAgent ag = new TMAgent(
      new TNotificationMessage("Check"), new TNotificationMessage(Z),
          m_in.rule[i]);
        ag.dispatch(IP_DOMAINSERVER);
      }
Exit:
Transition: -> ResInterfaceOperation
ResInterfaceOperation
Transition: -> ScanRequestedElements
Delay: if ((TNotificationMessage m=AgentInterface->
Get())=="ParamRequest")
Transition: -> LinksChange
Delay: if ((TNotificationMessage m=AgentInterface->
Get())=="LinksChange")
```

7.3.3 Consistency Checking Mobile Agent active object

Because of the inherent mobility of the consistency checking agent, the corresponding active object is not connected to any other stationary component in the outline of the architectural model (Fig. 7.1). Instead, the relevant active object is shown in Fig. 7.4.

Consistency Checking Mobile Agent is invoked by the Resource Interface Agent during the transition InstantiateMAForEachCriticalRule->ResInterfaceOperation (Fig. 7.3b). Table 7.2 specifies that the following parameters are passed to the agent constructor: goal identifier "Check", structure with references to changed document elements Z serialised as a NotificationMessage, and a relevant consistency rule out of a set of rules (m_in), selected by the resource agent.

```
TMAgent ag = new TMAgent(
   new TNotificationMessage("Check"), new TNotificationMessage (Z),
   m_in.rule[i]);
ag.dispatch(IP_DOMAINSERVER);
```

The state chart in Fig. 7.4 consists of four composite states. 'Active' state includes initialisation of the mobile agent and its operation at *source* and *destination* domains. The model assumes, that documents are located at different network hosts in different domains. Thus, two domains (called "source" and "destination") constitute a minimum unit of mobile agent's itinerary, where inter-domain migration is required. Modelling of additional domains is carried out by additional executions of the "Migrate" transition (Fig. 7.4) from within the AtDestinationDomain super-state.

The three other composite states are event handlers. AgentDataReceived handles collaborative communication between active mobile agents, where agents merge their already collected data. Redundant handles identification of the current agent's redundancy if another agent is found to be checking the same consistency rule at a given domain. Failure handler defines agent's persistency at unforeseen situations and its communicability with the source domain.

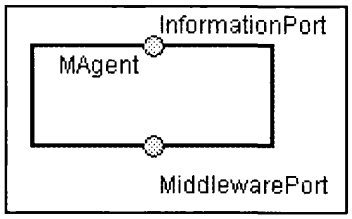


Fig. 7.4a. Structure of the Consistency Checking Mobile Agent active object.

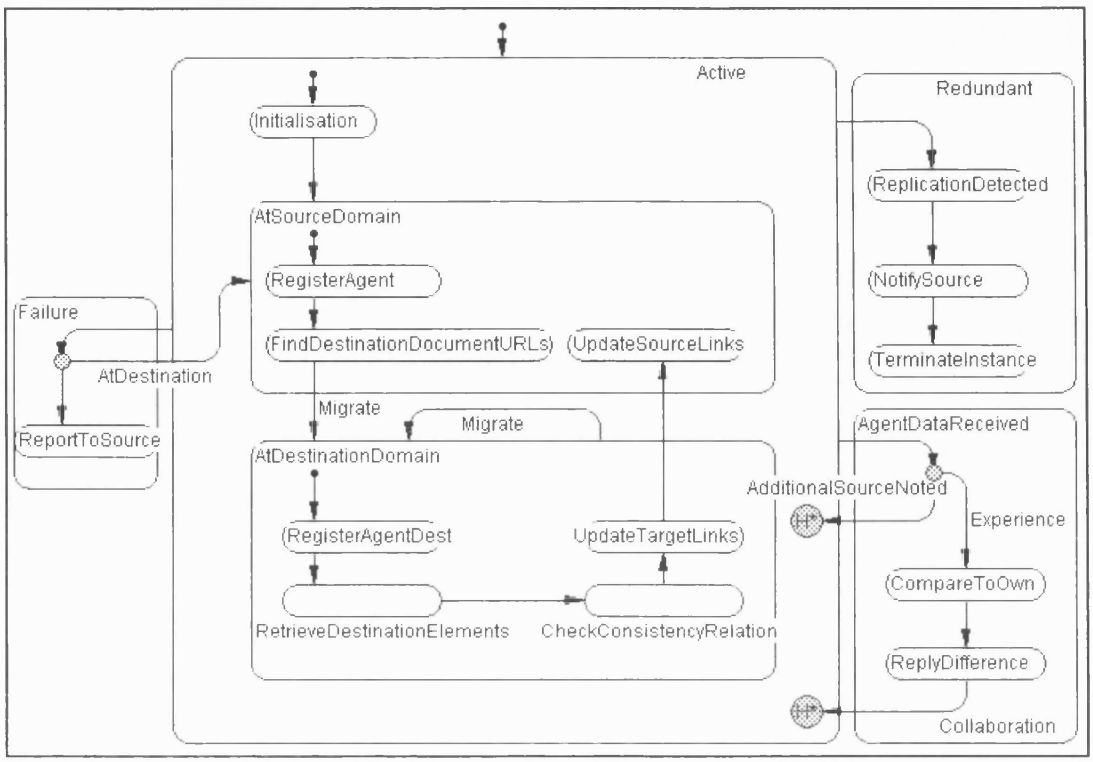


Fig. 7.4b. Behaviour of the Consistency Checking Mobile Agent active object.

The agent model embodies the essence of an incremental consistency checking algorithm in the "Active" state. State FindDestinationDocumentURLs accounts for identification of document's locations, RetrieveDestinationElements – for retrieval of document elements, specified by the consistency rule being checked. If the agent's itinerary has not been completed, then the agent model re-enters the composite state AtDestination via transition Migrate. CheckConsistencyRelation is reached when the itinerary has been completed, where the rule operators are executed on collected document elements. Generation and update of consistency links occur in states UpdateTargetLinks at the destination domain and UpdateSourceLinks at the source domain. The full code for state transitions of the mobile agent active object can be found in Appendix B, where execution operations, entrance and exit conditions for each of the agent's states are listed.

Due to the distributed nature of the consistency checking task, execution of the agent's algorithm takes place at the "source" domain, and one or more of "destination" domains. Migrate transition links the AtSourceDomain and AtDestinationDomain composite states, and allows re-visiting the AtDestinationDomain composite state for additional destinations. To ensure that the location and activities of all mobile agents can be identified and logged, upon each migration the agent undergoes a mandatory registration procedure (RegisterAgent and RegisterAgentDest) with a domain agent of the domain, into which the agent has migrated. During registration, the agent deposits its unique agent identifier and a reference to the consistency rule being checked into corresponding databases, managed by the domain agent.

Agents, checking the same consistency rule in the domain, engage in co-operative information exchange. The domain agent identifies redundant agents in this case and facilitates information

exchange between them by notifying each agent about existence of other redundant agents. Redundant and AgentDataReceived states of the mobile agent model are reached in response to the domain agent's notification. AgentDataReceived state of a consistency checking agent compares other agents' itineraries and extent of their completion with its own. The agent replies with the difference (CompareToOwn, ReplyDifference) or complements its itinerary (AdditionalSourcesNoted transition). If the current agent has completed a lesser part of the itinerary than its redundant counterparts, the agent terminates execution (TerminateInstance) within the Redundant macro-state.

The deployed approach to multi-agent co-operation and information exchange attempts to introduce a certain measure of agent intelligence and adaptability to the operating environment and to improve efficiency of distributed consistency checks.

Mobile agent's failure handling mechanism aims to enable persistent and *pro-active* achievement of the agent's goals. Failures can be classified into the following categories:

- Failure at source (i.e., document not found) – the agent reports to the source domain, this information is filed in the consistency links document (i.e., an inconsistent link is created if the document is required by the rule).
- Failure at one of destinations (destination host or document not found) – agent must persistently re-query the destination domain agent for information about the target document location. This information could have been updated after the initial incorrect itinerary was received.
- Failure at source or destination when a document element was not found – an inconsistent link is created as part of the normal operation of the consistency checking algorithm.

The Failure macro-state addresses these failure categories, and re-directs the execution flow of the agent's state transition model appropriately.

Migration state calls on the agent middleware to carry out simulation of agent's serialisation and physical transfer of its code and data via a TCP/IP network. After migration, the agent returns to the AtDestinationDomain state (if the agent itinerary has not yet been completed) or to AtSourceDomain within the Active macro-state.

7.3.4 Domain Agent active object

The model of the Domain Agent encompasses instances of TDomain class and instances of four classes – information repositories (Fig. 7.5): instance of consistency rule table (class TRulesDatabase), document name lookup table (class TNameLookup), agent lookup table (class TAgentsList), and event list table (class TEventsList). The purpose and structures of these information repositories are described in Chapter 6. All instances are connected through corresponding communication ports and participate in the behaviour of the Domain Agent (Fig. 5.7b).

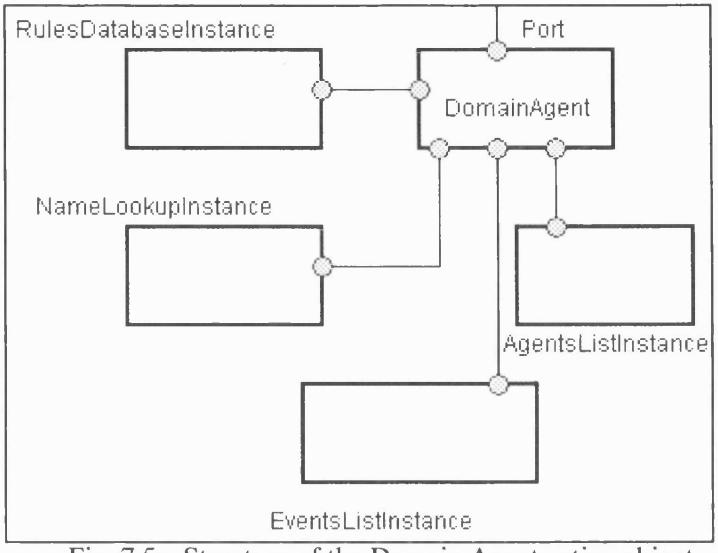


Fig. 7.5a. Structure of the Domain Agent active object.

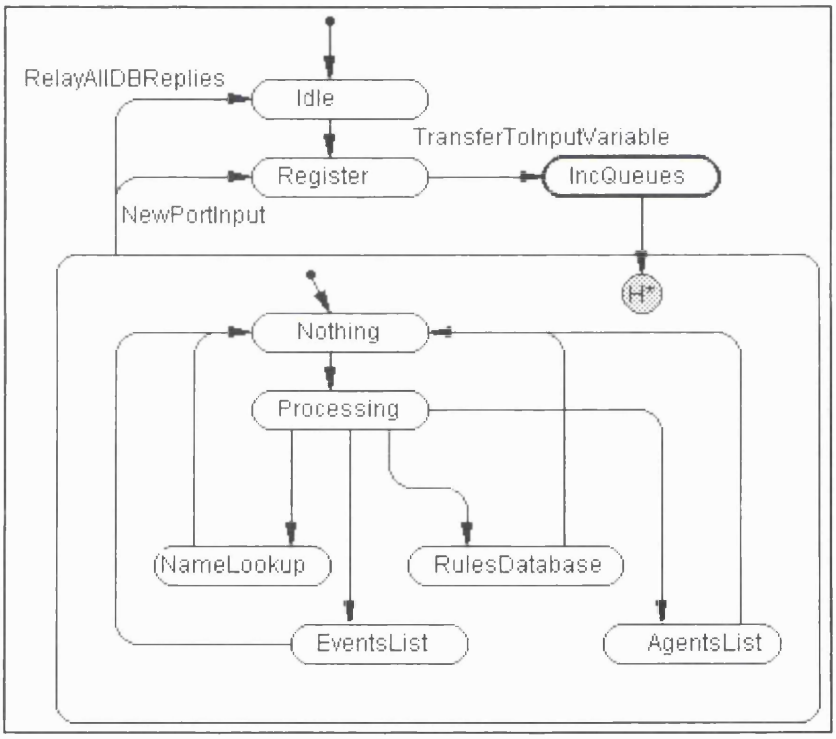


Fig. 7.5b. Behaviour of the Domain Agent active object.

The Domain Agent serves as a data repository of the distributed architecture; it functions primary in a "request-response" fashion. The state chart (Fig. 7.5b) is subsequently composed of message sorting and queuing (Idle, Register, IncQueues states) and an operational, information processing macro-state (Processing). The latter state executes requests for information to the four information repositories and forwards the responses in an acceptable message format to the enquiring active architecture components. Processing of queries is interrupted by incoming messages, which trigger NewPortInput transition from the macro-state to the message sorting Register state, and are sorted into a number of queues by the IncQueues state. A transition to the history state within the composite state Process enables the agent to resume processing of the ongoing queries, interrupted by incoming messages.

7.3.5 Agent Middleware active object

The Agent Middleware active object is based on a simulation of an agent transfer protocol over TCP/IP network. Most architecture components access agent middleware through the access ports. The implementation of the active object exposes methods for data queuing and re-transmission. These methods are invoked automatically when messages are exchanged between architectural components (such as mobile agent and domain agent). The methods cause a transmission delay when messages between components are exchanged through the middleware.

The Middleware active object supports generation of "Failure" messages in case the addressee is not found or has gone offline (Fig. 7.6, transition Report_Failure). Those messages are in turn processed by relevant components (i.e., the mobile agent). The middleware retries the transmission a number of times (one can specify a number of retries).

The ports Port1 and Port2 on the structure diagram of the mobile agent middleware are connected to the Resource Interface Agent and the Domain Agent, respectively. The Mobile Agent component re-connects to Port1 after each migration, thereby acquiring direct local access to the Resource Interface Agent, which uses the same port. The Gateway Port connects different domains within the model of the architecture. When the simulation is carried out in a multiple-domain configuration, the port is jointly used by replicated domain models, represented by class TSystem in Fig. 7.1. Inter-domain traffic of mobile checking agents is routed through that port between the models of different domains.

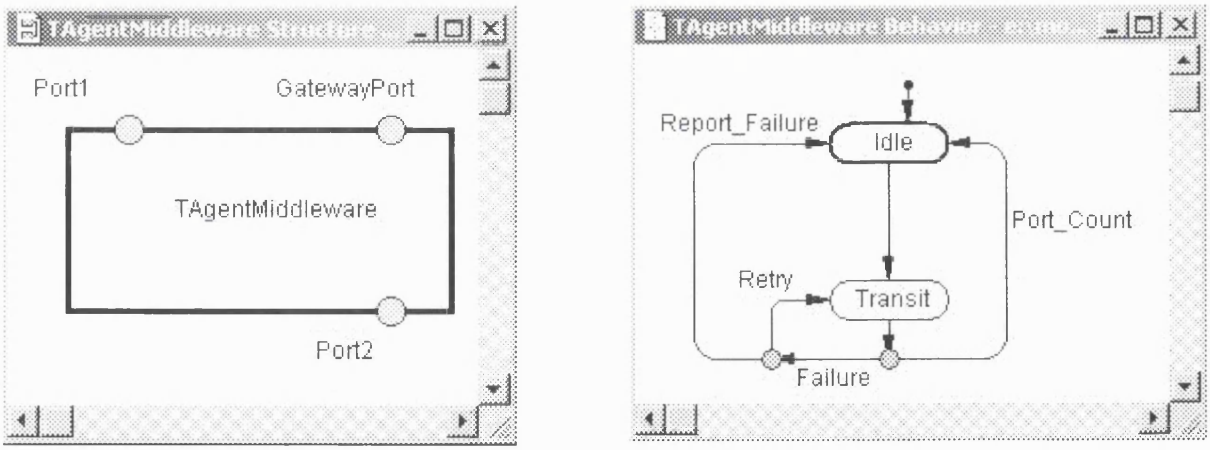


Fig. 7.6. Agent Middleware active object: its structure and behaviour.

7.4 Evaluation Results

Evaluation of the software agent architecture on the state transition model demonstrates that the proposed architecture is capable of carrying out distributed consistency management services with mobile agents. We also take advantage of the COVERS development environment's capability to simulate "execution" of statecharts, and demonstrate that the model is capable of delivering suitable responses in terms of consistency checking performance. In order to evaluate the performance of the

model, we have benchmarked the model in a number of configurations. For each configuration, we have also verified that all consistency checks complete and asynchronous interaction between independent components does not result in synchronisation mismatches.

The configuration parameters for simulation tests were chosen in order to resemble those of real-life scenarios (i.e., UML model development). The evaluation of the model is carried out through variation of the parameters and observation of the resulting timing of a distributed consistency check. Fig. 7.7 depicts the set of configuration parameters being varied during the execution of the model.

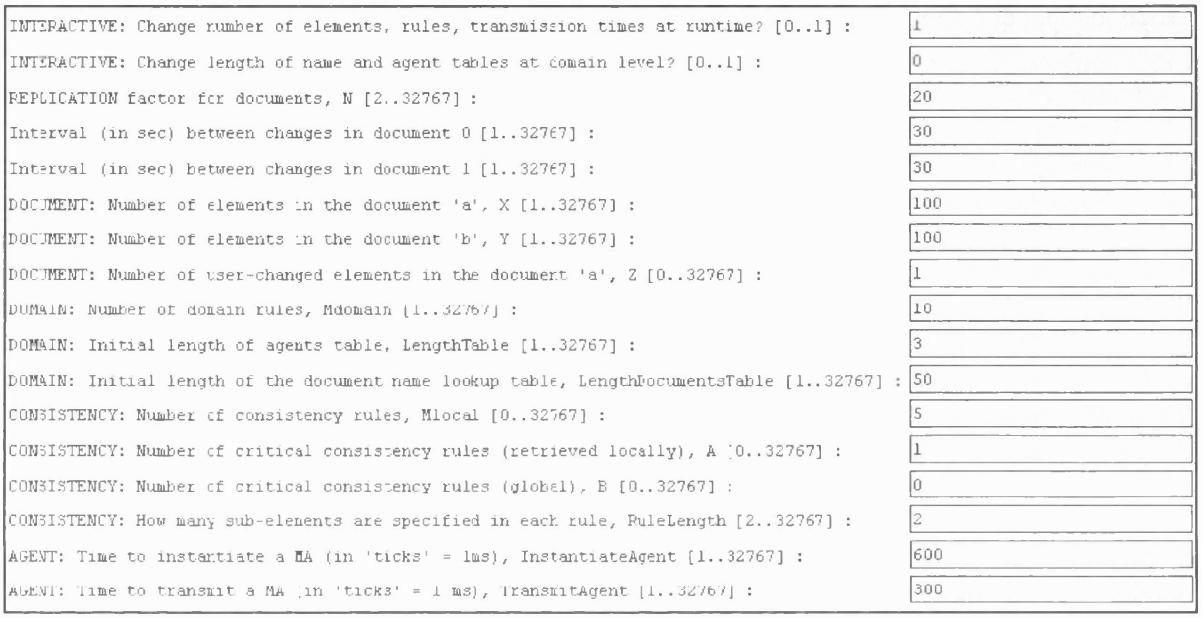


Fig. 7.7. Variation of model parameters.

We have classified the variant parameters in Fig. 7.7 into a number of categories (shown capitalized in Fig.7.7), as follows.

- Interactive mode – enables interactive run of the model, where simulation parameters can be changed at runtime throughout the duration of the execution. Disabling this mode results in a "batch" execution of the model in an automated mode without user involvement.
- Replication factor – varies the number of participating document instances in the model.
- Timing interval – used in randomisation of intervals between events of document changes, which are raised by document instances during model operation.
- Document – sets the number of structural elements in each document instance.
- Domain – allocates memory for the information tables at the domain agent, allows us to simulate the effect of information centralisation in the domain.
- Consistency – varies the total number of consistency rules, selected (relevant) rules, and their allocation (de-centralisation) configuration.

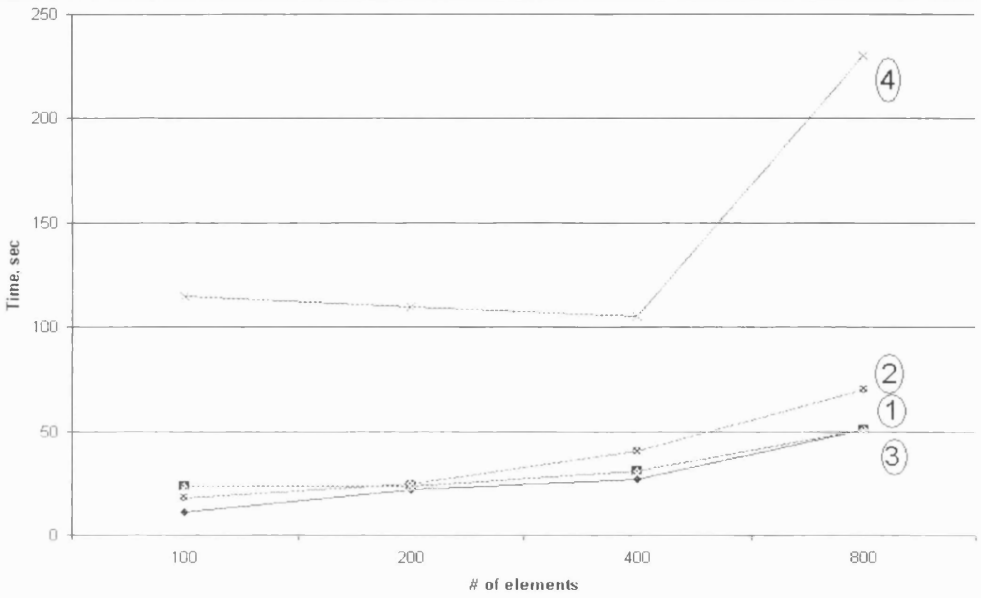
- Mobile agent – performance characteristics of the execution environment and the agent middleware. We have measured these basic characteristics in the test runs of the mobile agent middleware – the IBM Aglets.

Fig. 7.8 depicts the performance graphs of the architecture model. The data is collected in an experiment, where a number of distributed documents (1..50) are affected by changes (1..50 elements changed out of 100..800 elements per document). These changes trigger selection of applicable consistency rules (1..20 rules) from the consistency rule database (15..150 rules). Each rule contains a number of operators, called "rule elements" (2..20), which are executed on the collected data. These configuration conditions correspond to software engineering projects with small to medium sized documents, when a modest amount of changes is made for each incremental check, and those changes affect a significant number of consistency rules, which are concurrently checked across a number of documents.

Each successive graph of the three graphs represented in Fig. 7.8 demonstrates the consistency checking times, which the model of the software agent architecture achieves in varying conditions. The number of consistency rules being concurrently checked varies from 1 in Fig. 7.8a, to 10 in Fig. 7.8b, to 20 in Fig. 7.8c, and the number of documents, on which the rules are checked also increases from 10 to 25, and to 50, respectively. As the number of documents and rules increase, our model of the incremental consistency checking algorithm spends more time on individual checks (51-230 sec, 80-275 sec, 87-384 sec, respectively). From the graphs, we observe that durations of each distributed check are lower than the duration of a check of a single consistency rule, taken in the proportion to the increase in the number of documents and consistency rules. To a large extent, the observed performance improvement of the distributed incremental checking with the software agent architecture is due to use of concurrent checking at distributed locations, rather than use of traditional checks at a central location. The simulation data constitutes an initial step towards evaluating scalability of the distributed software agent architecture.

In addition to the initial performance evaluation of the distributed architecture, the simulation model allows us to observe performance of an incremental consistency check. Having demonstrated different durations of distributed checks with an increase in the total number of checked consistency rules and documents, each graph in Fig. 7.8 also shows a variation in durations of incremental checks with the increase of documents' size (from 100 to 800 elements for each document). Each graph also contains four sets of benchmark data (marked lines 1-4), which result from variation of the number of changed document elements (1-50 elements) and of the number of operators in the executed consistency rules (20-150 rules, totalling 40-3000 rule operators). In the experiment, we observe that an increase in the number of document elements or rule operators does not trigger as high a proportional growth in the duration of consistency checks. In certain conditions, performance unexpectedly improves (demonstrative examples are Fig. 7.8a, line 4, and Fig. 7.8c, line 1). Since our model simulates a number of concurrent distributed checks, and total check time depends on a number of randomised parameters

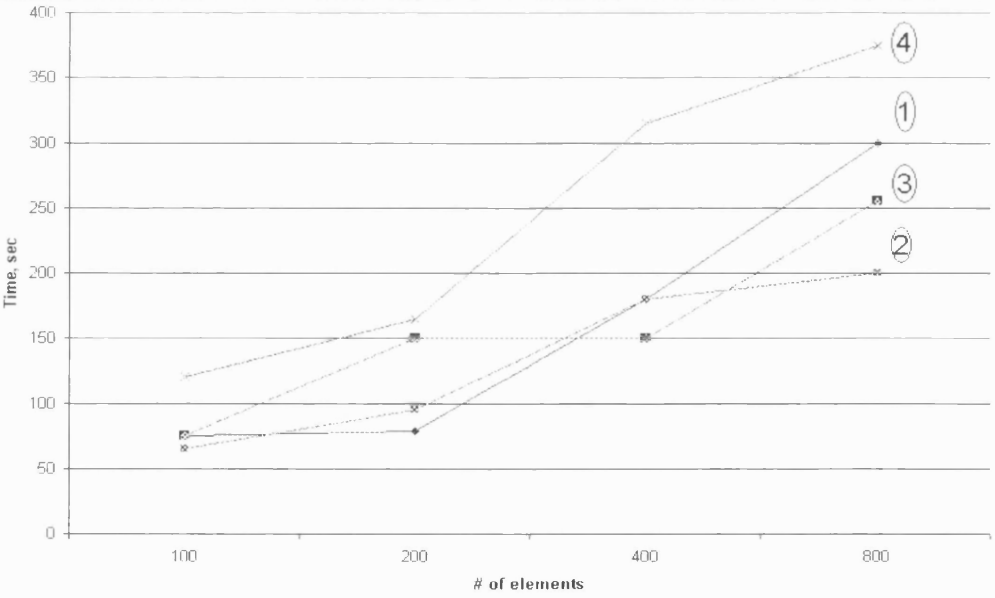
(i.e., agent migration delay, domain agent response time, etc.), we cannot at this point draw an optimistic conclusion solely on the basis of the simulation and the observed experimental data.



1 checked consistency rule, 10 participating documents

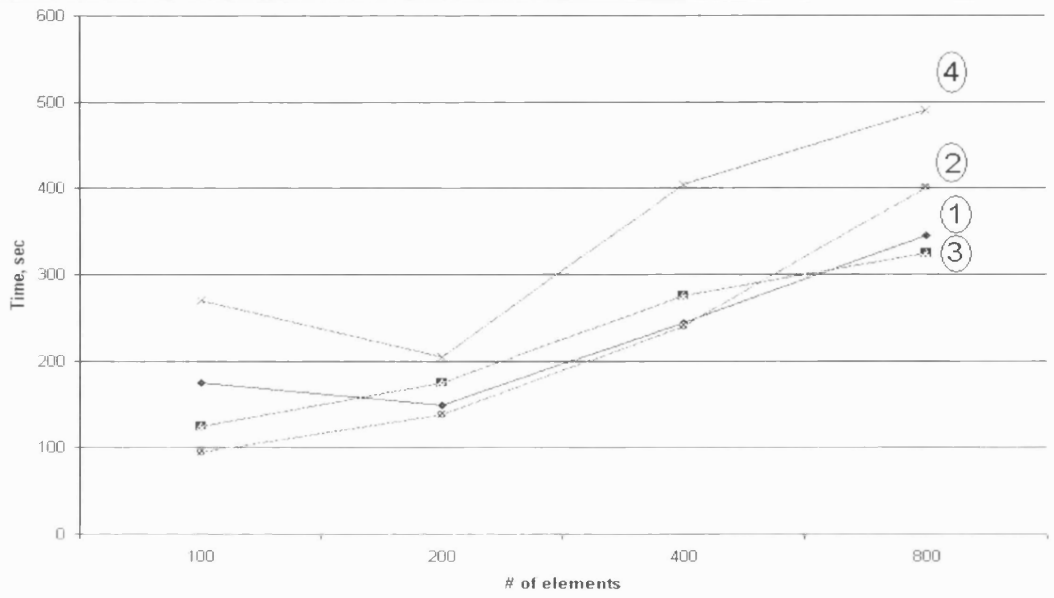
Legend:
(1) 20 rules in total, total 40 rule expressions to compare and choose a relevant rule from, 1 changed element checked in each document
(2) 150 rules total - 3000 rule elements, 1 changed document element checked
(3) 20 rules total - 40 rule elements, 50 changed document elements checked
(4) 150 rules total - 3000 rule elements, 50 changed document elements checked

Fig. 7.8a. Performance evaluation of the architecture model.



10 checked consistency rules, 25 documents

Fig. 7.8b. Performance evaluation of the architecture model (continued).



20 checked consistency rules, 50 documents

Legend:
(1) 20 rules in total, total 40 rule expressions to compare and choose a relevant rule from, 1 changed element checked in each document
(2) 150 rules total - 3000 rule elements, 1 changed document element checked
(3) 20 rules total - 40 rule elements, 50 changed document elements checked
(4) 150 rules total - 3000 rule elements, 50 changed document elements checked

Fig. 7.8c. Performance evaluation of the architecture model (continued).

Experimental data (Fig 7.8) allows us to characterise the performance "penalty" of the distributed consistency checking architecture, incurred by successive migrations of the mobile checking agent. An observed increase of this penalty with the increase in the number of distributed documents can be explained. On each successive migration, a mobile agent carries all the data, collected previously to the current location, as well as the data, collected at the current location. In the architecture model, migration delay grows as the agent size increases after retrieval of document elements. Therefore, we observe a growing timing increment for each consecutive migration of an agent, which adds up to the performance penalty over a number of agent migrations.

We suggest for an implementation of the architecture that lengthy itineraries for mobile agents should be avoided, and the number of agent migrations should be minimised for each individual agent. Performance advantages of the architecture will be realised when an itinerary for a given consistency rule is divided, and a number of checks are carried out concurrently with a number of agents checking documents at the distributed locations.

Fig. 7.9 summarises the experimental data on performance of the model of the distributed agent architecture and includes as a reference performance figures of a stationary, centralised consistency checker. The data for centralised checks is taken from [Revheim 2000]. Rows 1-3 of the table (Fig. 7.9) exhibit performance measurements of the distributed architecture, rows 4-7 – of the centralised approach.

Experiment No.	Number of Rules	Number of Documents	Max. Number of Elements in a Document	Total Rules, Rule Elements	Time of Consistency Check, Sec.
(1)	1	10	800	20-150 rules, 2-20 elements per rule	51 – 230
(2)	10	25	800		80 – 275
(3)	20	50	800		87 – 384
(4)	1, stand-alone	10	700	1 rule, 2 conditions, 2-20 elements per rule	55
(5)		20	540		435
(6)		10	1200		396
(7)		20	540		641

Fig. 7.9. Performance of the model and a centralised consistency checker.

Comparing the performance of a centralised checker with the simulation data of the distributed software agent architecture, we are in a position to highlight once again some significant characteristics of the architecture, that have a potential to improve the performance of distributed checks.

- Concurrent execution of numerous consistency rules. In rows 2 and 3 (Fig. 7.9), execution of all consistency rules was carried out concurrently, rather than sequentially as in rows 4 – 7.
- Concurrent checking of numerous distributed documents. Multiple mobile checking agents, concurrently checking the same rule at different locations can collaborate and exchange values of document elements, concurrently retrieved from the distributed documents.

During the execution of the simulation model, we have observed that collaboration between mobile agents in exchange of retrieved document elements provides acceleration of consistency checks for documents with lower number of elements, affected by a consistency check (100-200 elements), and while the number of different consistency rules checked concurrently is not very large (around 10). In other conditions, an overhead of information exchange between numerous mobile agents outweighs the penalty of repeated requests for document element values. In such conditions, the state "AgentDataReceived" of the mobile agent's statechart (Fig. 7.4b), responsible for information exchange, is active most of the time, rather than the agent's normal operational state "Active". Therefore, it would be an advantage to disallow inter-agent collaboration when the number of agents checking a consistency rule exceeds a certain threshold (i.e., 10 agents), and instead make these agents re-request element values from the documents.

7.5 Summary

In this chapter a model of the distributed software agent architecture for consistency management has been presented. Construction of this model has taken us through the design stage of the architecture, and allowed us to express component functionality and interactions between the components through state transitions of active objects, which implement the corresponding components of the architecture.

Execution of the state transition model was simulated on a range of model configuration parameters. The collected experimental data serves as an initial step in evaluation of scalability of the architecture

with the growth in the number of documents, their size and with the increase in the number of consistency rules. The evaluation conditions were chosen to resemble those of a distributed software engineering activity. As a result of the evaluation, we were able to measure the range of durations of consistency checks, and identify characteristics of the architecture, which we can draw on in the implementation prototype, in order to improve performance of distributed consistency checks.

In the following chapters, we continue evaluation of the software agent architecture on an implementation prototype. We consider the performance evaluation of the model in this chapter as an initial evaluation step, rather than the final judgement. Chapter 10 contains performance benchmarking of the implementation prototype and further supports our argument in support of the distributed architecture.

The main purpose that the architecture model serves is the representation of structure and logic of the architectural components in state transition diagrams. Using the model, we observed execution of transitions within each component and interactions between components that these transitions trigger. Consequently, the constructed model is a step in verification of the architecture: the model completes execution of all started consistency checks. Performance evaluation is a third significant, but complementary benefit of the model. Promising evaluation results have encouraged us to continue development of the architecture and to implement a working prototype, which we demonstrate in the following Chapter 8.

Chapter 8 Scenarios: Distributed Deveiopment of the Break Planner Appiication

8.1 Introduction

In a running scenario throughout this thesis, we consider the development process of a "Break Planner" application [Bergner, et al. 1997]. We demonstrate deployment of the software agent architecture for consistency checking throughout the distributed development of this application.

Three scenarios, presented in this chapter, serve a dual function. First and foremost, they constitute a part of evaluation of the software agent architecture, proposed in the thesis. The scenarios reflect a number of situations during the application development, and demonstrate the results of distributed consistency checks that the architecture prototype carries out. The discussion of distributed consistency checks and of their results in each scenario demonstrates the distributed incremental checking algorithm [Chapter 5, 5.4.2] in action. In addition, we re-visit the instrumentation in the software agent architecture of the concepts, demanded by the functional requirements for distributed consistency checking [Chapter 3], this time on the practical examples. Furthermore, some significant characteristics of the architecture, which are not directly demanded by the general requirements, are demonstrated in the scenarios. Among such characteristics are concurrent multi-agent checks, disconnected operation and replication of key architectural components.

Secondly, the scenarios give a detailed explanation of the operation of all architectural components. Particular examples of document distribution, domain hierarchy and mobile agent migration are provided in order to better explain the nature of interactions between the components. To a certain extent, these scenarios constitute a validation for the proposed architecture. In the provided examples, we explain the nature of changes in particular documents, the subject of individual consistency rules, relevant to those changes, particulars of the consistency checking process with the software agent architecture, and the content of the resulting consistency links.

All scenarios in this chapter describe the operation of an architecture prototype, which we have implemented to evaluate the architecture. We have chosen to informally introduce the prototype in these scenarios, in order to be able to follow up in Chapter 9 with a detailed prototype description, outlining the internal event-based operation of each prototype component.

8.2 Application Domain

All scenarios constitute application of the software agent architecture for consistency checking to the domain of checking well-formedness of UML models. The Universal Modeling Language standard [Booch, et al. 1999] provides description techniques for application development, and defines their syntax and semantics. An important feature for a software engineers is the ability of the development tools to provide checking of the UML diagrams under development for conformance to the standard.

At present, UML development tools do not provide support for checking full compliance of a UML model to the standard. As one example, Rational Rose [Rational 2000] does not prohibit developers from leaving association ends unnamed, which violates one of the UML standard well-formedness rules [Booch, et al. 1999]. At the same time, the Rose tool provides support for some of other UML rules, for instance, regarding uniqueness of attribute and method names in classifiers within their namespace.

The motivation behind each scenario is to offer software engineers a facility to check conformance of the developed UML model to all well-formedness rules, specified in the standard [Booch, et al. 1999]. The scenarios demonstrate how the proposed software architecture for distributed consistency management can be deployed by software engineers during development to complement the current UML model standard compliance checking services, provided by the development tools.

The domain of checking UML well-formedness rules was chosen, because these rules include a variety of consistency relations of different types between a relatively extensive number of different constructs that exist in a UML model. Through investigation of such a non-trivial application domain, we are able to demonstrate the usability of our approach in the relatively complex scenarios.

8.3 An Overview of the Scenarios

The application in development will assign supervision of multiple school breaks to the teachers in a school. Assignment of teachers is specified in a break plan, where each break must be supervised by a teacher, and the number of breaks each teacher has to supervise is proportional to the time they spend teaching. Teachers can set exclusion times, when they cannot supervise breaks because of other duties.

The intended application must support plan editors and staff editors, responsible for respective data. The application allows a user to create and delete break plans, assign teachers to breaks, and manage the list of teachers. In addition, the tool computes some statistics, for example, the number of breaks a teacher still needs to be assigned to.

The application must be constructed in such a way, that a number of plan editors and staff editors could concurrently edit break plans, using individual client software, which accesses the database on the application "server".

8.4 Application Development Team

Throughout scenarios I and II, the development team consists of Anne and Bob. Bob starts constructing application classes from a customer specification, and Anne initially designs associations between the classes. Anne and Bob carry out their development activity at the separate workstations A and B, respectively; these workstations are included in the domain "core development", the domain machine, running the domain agent. The domain is named CDS – the "core development server" (Fig. 8.1).

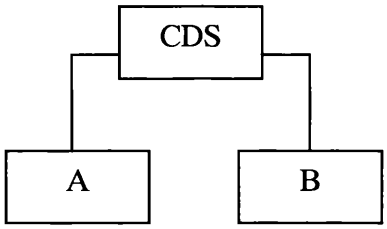


Fig. 8.1. Initial structure of the CDS domain.

8.5 Analysis and Design of the Break Scheduler Application

The following scenarios I and II are concerned with the analysis and design stage of the development of the break scheduler application. In these scenarios, we describe developer experience and architecture prototype operation on the example of the initial development of the Analysis Class Diagram (Fig. 8.4 and also in Appendix C, C1), which forms part of the UML model of the distributed break scheduler application. The model is being developed in a distributed fashion; new model elements, created by Anne and Bob, are distributed across the network hosts in the following way (Fig. 8.2 and Fig. 8.3a-b). This particular distribution is not a specific requirement for correct operation of the proposed architecture for distributed consistency checking; this simple distribution is used to facilitate understanding of the prototype's operation.

XMI files Contained at Host A	XMI files Contained at Host B	UML Well-formedness Consistency Rules Contained at the Domain CDS
Model Associations	Model Classes	Association, AssociationEnd, AssociationClass, BehavioralFeature, Classifier, Class, GeneralizableElement, Generalization, Interface, Method, NameSpace, StructuralFeature, Type
	Model Namespace	
Document universe: list of all local documents	Document universe: list of all local documents	

Fig. 8.2. Distribution of XMI documents for Scenarios I and II (summary).

Document Name	Element Description
Association0.xml	Association, Organiser-Staff
Association1.xml	" , Organiser (organises) BreakPlan
Association2.xml	" , Staff-Statistics
Association3.xml	" , Staff-Teacher
Association4.xml	" , Teacher (supervises) Break
Association5.xml	" , Period (generalises) Break
Association6.xml	" , Teacher–Exclusion Time
Association7.xml	" , BreakPlan-Break
Association8.xml	" , ExclusionTime-Period

Fig. 8.3a. Document distribution (host A).

Document Name	Element Description
Class0.xml	Class Organiser
Class1.xml	" Staff
Class2.xml	" Statistics
Class3.xml	" BreakPlan
Class4.xml	" Break
Class5.xml	" Teacher
Class6.xml	" Period
Class7.xml	" ExclusionTime
Class8.xml	" Account
Namespace0.xml	NameSpace of the model

Fig. 8.3b. Document distribution (host B).

For creation of the UML model in this and following scenarios, we use the Rational Rose tool [Rational 2000]. For consistency checking of UML well-formedness rules, the UML model is exported to XMI via Unisys XMI converter for Rational Rose [Rational 1999]. We then applied the UMLXMI utility [Appendix F, F.5] to the resulting XMI model file, which enabled us to extract model elements into separate XMI files. The consistency checks are then executed by our architecture prototype on these files after they have been distributed (Fig. 8.3a-b).

As an alternative to use of an exported UML model, we also considered development of the model directly in XMI. Such approach is less transparent for a user than development with Rational Rose that provides a graphic user interface for editing the model. We have found the alternative unrealistic, because developers without extensive knowledge of the XMI format will certainly find it difficult and error prone to develop in a complex-structured format, such as XMI.

We found the graphical UML notation of the application model (i.e., Fig. 8.4 and Appendix C) useful during development and for illustration of the scenario; this notation would not have been easily unavailable from XMI.

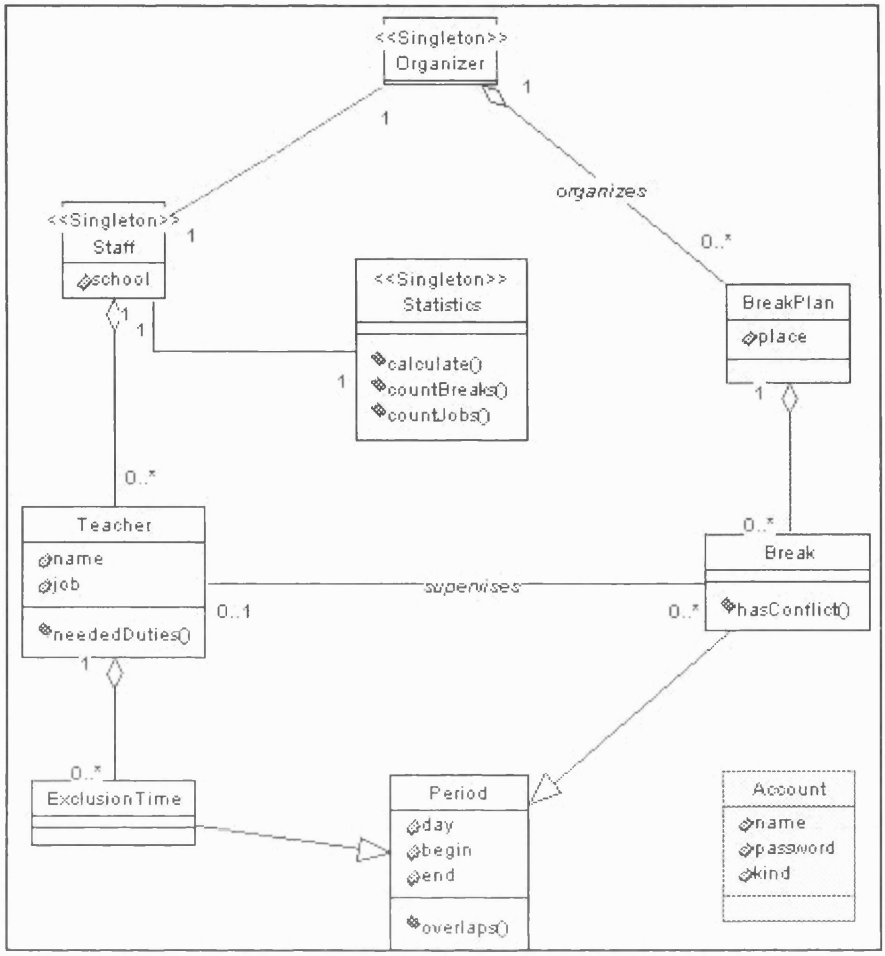


Fig. 8.4. The Analysis Class Diagram considered in the Scenarios I and II.

The majority of inconsistencies of the UML model, detected in the scenarios are allowed by the Rational Rose tool, because the tool does not impose all well-formedness constraints of the UML standard. For example, names of association ends are not displayed on the diagrams, and are effectively not used in the tool. However, the UML standard requires uniqueness of names of association ends. When developers are required to produce a compliant UML model, all inconsistencies detected with respect to UML well-formedness rules must be corrected. It is thus appropriate to highlight these scenarios as a definite advantage of joint deployment of the proposed consistency checking prototype with industrial UML development environments.

Scenarios I, II and III form a series of scenarios, where the structure of hosts and domains and complexity of distribution of the model elements increases from one scenario to the next. The architecture prototype is intended to accommodate configurations beyond those demonstrated in these scenarios. In this respect, the scenarios can serve as recommended deployment configurations, but operation of the prototype is not limited to these configurations.

8.6 Scenario I: Local Consistency Checking at a Host Within a Single Domain

8.6.1 Event: Creation of a New Document

In accordance to the incremental checking algorithm [Chapter 5, 5.3], every time a new document is added to the set of documents at a given host, initial exhaustive consistency checking of that document is carried out to determine applicability of all consistency rules to the new document. In the context of our scenarios, UML model elements are represented as separate documents. Thus, each time an additional UML element is created, a complete check of UML well-formedness rules is carried out on the XMI document, corresponding to that model element.

Initial checks establish the sets of rules, initially applicable to every XMI document. Rule applicability information is saved into a document table [Chapter 6, 6.3.1], maintained by the domain agent of the CDS domain. The information is used during incremental checks to create itineraries for mobile checking agents, which include all documents, known to be relevant to the rule being checked.

A list of consistency rules, relevant to classes, associations and other elements of the UML model, located at domain CDS, is provided in Fig. 8.5. Consistency rules are referred to by their unique identifiers; a complete set of UML well-formedness rules used for all scenarios can be found in Appendix A. This initial relevance of rules to documents is established at initialisation of the incremental checker (line 2, Fig. 5.7 in Chapter 5).

Classes Class0, ..., Class6.xml, set of relevant consistency rule identifiers:
[n1, n2, b1, b2, cs2, cs3, cs4, cs5, cs6, c1, d1, d2].
These consistency rules refer to model elements: Class (identifier c1), Classifier (cs2-cs5), Name Space (n1,n2), Behavioural Feature (b1,b2), Data Type (d1,d2).
Associations Association1, ..., Association6.xml, set of relevant consistency rule identifiers:
[n1, n2, ae1, ae2, a1, a2, a3, a4].
These consistency rules refer to model elements: Associations (identifiers a1-a4), Association Ends (ae1,ae2), Name Space (n1,n2).
AssociationEnd: [n1, n2, ae1, ae2, a2, a3, a4]
Generalization: [gen1]
Classifier, DataType, Behavioural Feature: [n1, n2, b1, b2, cs2, cs3, cs4, cs5, cs6, c1, d1, d2]
Structural Feature: [gen1, n1, n2, b1, b2, cs2, cs3, cs4, cs5, cs6, c1, d1, d2]
GeneralizableElement: [n1, n2, g2, g4, gen1]

Fig. 8.5. Relevance of UML well-formedness rules to model elements.

8.6.2 Response: Consistency Check

In addition to selection of relevant rules, creation of a new document triggers an incremental consistency check on this document. The document is considered empty prior to creation, and after creation, the contents of this document change.

In our scenario, Bob at host "B" of domain CDS is responsible for development of classes and operations. His user profile is granted a right to create new documents and make changes to existing documents, therefore the Resource Interface Agent at host B allows him to create a new instance of a class. When Bob creates the class Teacher (Class5.xml, Fig. 8.3b) for the analysis class diagram (Fig. 8.4 and Appendix C, C.1), a consistency rule "c1" is identified relevant to the class (Fig. 8.5), and an incremental check of this rule is triggered. At this point, we will consider execution of one consistency rule on one document; more sophisticated examples follow.

Description of the rule "c1": if a Class is concrete, all the Operations of the Class should have a realizing method in the full descriptor. The XML representation of this rule in the consistency rule language is provided in [Appendix A, A5].

```
<?xml version="1.0" encoding="UTF-8"?>
<XMI>
  <XMI.content>
  <Model_Management.Model xmi.id="G.1">
   <Foundation.Core.Namespace.ownedElement>
    <Foundation.Core.Class xmi.id="S.10001">
      <Foundation.Core.ModelElement.name>Teacher
      </Foundation.Core.ModelElement.name>
      <Foundation.Core.ModelElement.visibility xmi.value="public"/>
      <Foundation.Core.GeneralizableElement.isRoot xmi.value="true"/>
      <Foundation.Core.GeneralizableElement.isLeaf xmi.value="true"/>
 <Foundation.Core.GeneralizableElement.isAbstract xmi.value="false"/>
      <Foundation.Core.Class.isActive xmi.value="false"/>
      <Foundation.Core.ModelElement.namespace>
        <Model_Management.Model xmi.idref="G.1"/>
      </Foundation.Core.ModelElement.namespace>
      <Foundation.Core.Classifier.feature>
         <Foundation.Core.Operation xmi.id="S.10004">
         <Foundation.Core.ModelElement.name>neededDuties
         </Foundation.Core.ModelElement.name>
        <Foundation.Core.ModelElement.visibility xmi.value="public"/>
        <Foundation.Core.Feature.ownerScope xmi.value="instance"/>
       <Foundation.Core.BehavioralFeature.isQuery xmi.value="false"/>
       <Foundation.Core.Operation.specification/>
       <Foundation.Core.Operation.isPolymorphic xmi.value="false"/>
      <Foundation.Core.Operation.concurrency xmi.value="sequential"/>
         </Foundation.Core.Operation>
     </Foundation.Core.Classifier.feature>
     </Foundation.Core.Class>
   </Foundation.Core.Namespace.ownedElement>
  </Model_Management.Model>
 </XMI.content>
</XMI>
```

Fig. 8.6a. Initial version of class Teacher.

8.6.3 Result: Generated Consistency Links

For class Teacher, an inconsistent link was generated by rule "c1", because there is an operation "neededDuties()" defined for this class in the class diagram [Appendix C, C.1], and naturally, this operation has no method specification at this early stage in project development. For class Teacher, there

is no method specification in the features of the class (sub-element Foundation.Core.Method does not exist within the Foundation.Core.Classifier.feature element of class Teacher, Fig. 8.6a). An inconsistency link, generated in this case is shown in Fig. 8.6b. The link connects two existing elements, demanded by the rule "c1": the class and the operation.

```
<xlinkit:ConsistencyLink ruleid="class.xml#/consistencyrule[1]">
<!— Class rule c1-->
    <xlinkit:State>inconsistent</xlinkit:State>
<!—Inconsistency : operation neededDuties doesn't have a realising method in the descriptor
of the class Teacher, to which it belongs. -- >
    <xlinkit:Locator
xlink:href="atp://B/Class5.xml#/XMI.content[1]/Model_Management.Model[1]/
Foundation.Core.Namespace.ownedElement[1]/Foundation.Core.Class[1]" xlink:label=""
xlink:title=""/>
<!—this is the Teacher class, xmi.id= "S.10001" -- >
    <xlinkit:Locator
xlink:href="atp://B/Classifier.feature0.xml#/XMI.content[1]/Model_Management.
Model[1]/Foundation.Core.Namespace.ownedElement[1]/Foundation.Core.Class[1]/
Foundation.Core.Classifier.feature[1]/Foundation.Core.Operation[1]" xlink:label=""
xlink:title=""/>
<!—this is the neededDuties operation, xmi.id = " S.10004" -- >
</xlinkit:ConsistencyLink>
```

Fig. 8.6b. Inconsistent link between an operation in the class and the class descriptor.

8.6.4 Processing of the Event

We now describe the sequence of events and their processing within the software agent architecture prototype. In detail, we comment on actions of the components of the architecture, which resulted in generation of an inconsistency link (Fig. 8.6b).

Having created a new class Teacher (Fig. 8.6a), file name Class5.xml, Bob has saves the file by overwriting the previous (empty) version. The resource interface agent at host "B" is monitoring file Class5.xml for change, because Bob has included the file in consistency checks by adding a reference to it in the DocumentSet.xml, which describes the local document set – the "document universe".

After a change in Class5.xml is detected, the resource interface agent executes stage 2.1 of the incremental checking algorithm (Chapter 5, Fig. 5.7), and computes a tree-wise difference (TreeDiff) between the current version of the XML file and its previous version. Among numerous modifications, the previously empty class Teacher acquires a new *neededDuties* operation. The TreeDiff, concerning addition of the operation, shows the added XMI content and gives the XPath expression to the parent of the changed element (*xpathparent* attribute of *addsubtree* element in Fig.8.7).

```
<?xml version="1.0" encoding="UTF-8"?>
<treediff>
  <addsubtree
xpathleftsibling="/XMI/XMI.content[1]/Model_Management.Model[1]/Foundation.
Core.Namespace.ownedElement[1]/Foundation.Core.Class[1]/Foundation.Core.
Classifier.feature[1]/text()[1]"
xpathparent="/XMI/XMI.content[1]/Model_Management.Model[1]/Foundation.
Core.Namespace.ownedElement[1]/Foundation.Core.Class[1]/Foundation.Core.
Classifier.feature[1]">
    <Foundation.Core.Operation xmi.id="S.10004">
<Foundation.Core.ModelElement.name>neededDuties
</Foundation.Core.ModelElement.name>
        <Foundation.Core.ModelElement.visibility xmi.value="public"/>
        <Foundation.Core.Feature.ownerScope xmi.value="instance"/>
        <Foundation.Core.BehavioralFeature.isQuery xmi.value="false"/>
        <Foundation.Core.Operation.specification/>
        <Foundation.Core.Operation.isPolymorphic xmi.value="false"/>
        <Foundation.Core.Operation.concurrency xmi.value="sequential"/>
</Foundation.Core.Operation>
  </addsubtree>
</treediff>
```

Fig.8.7. Document change is indicated in the TreeDiff.

The resource interface agent finds an intersection of the XPath expression in *xpathparent* with XPaths in all consistency rules (stage 2.1, Fig. 5.7, Chapter 5). The result is a set of consistency rules, which are relevant to this particular change, as shown in the activity log of the prototype (Fig. 8.8).

```
1. RA: Class8.xml changed.
2. RA: Processing message change
3. RA: Finding relevant rules for the following change:
   /XMI/XMI.content/Model_Management.Model/Foundation.Core.Namespace.
   ownedElement/Foundation.Core.Class/Foundation.Core.Classifier.feature
4. RA: relevant rules to the change identified :[c1, t1, t2]
5. DA: Processing message newChecker
6. DA: Processing message newChecker
7. DA: Processing message newChecker
8. DA: Processing message getRuleApplicability
9. DA: Processing message getRuleApplicability
10.   DA: Processing message getRuleApplicability
Legend:
RA – log record from one a resource interface agent;
DA – log record from the domain agent.
```

Fig. 8.8. Activity log of the prototype: selection of relevant consistency rules.

In addition to the rule c1, two *type* well-formedness rules have been identified as relevant - t1 and t2 [Appendix A, A.15]. These two rules *may* be relevant to the current TreeDiff XPath, because they are based on the XPath expression, containing an *id()* function, and the parent is also //Foundation.Core.Class, alike the *class* rule. A lightweight rule selection algorithm [Appendix F, F.2] has selected the *type* rules in this case. The mechanism for computing an intersection between a TreeDiff XPath and the XPath from the consistency rule is described in greater detail in Appendix F.

Type rules are based on the following XPath expression:
Foundation.Core.Class[id(Foundation.Core.ModelElement.stereotype/
Foundation.Extension_Mechanisms.Stereotype/@xmi.idref)]

XPath expression – basis of the UML well-formedness rule for Type.

For checking of each of the rules "c1", "t1" and "t2", a mobile consistency checking agent is instantiated at host "B" (lines 5-7, Fig. 8.8). Each agent on instantiation receives as a parameter the DOM tree of the changed document (class Teacher), the rule identifier of the relevant rule (rule id), and the DOM tree of the relevant consistency rule. The mobile agent then requests an itinerary from the domain agent, which lists any documents, relevant to the rule that the agent must check for consistency. The log of the working prototype shows three messages "getRuleApplicability" processed by the domain agent (lines 8-10, Fig. 8.8): these messages contain itinerary queries from mobile agent instances.

Based on the relevance of consistency rules to distributed documents, the domain agent compiles itineraries for each mobile agent (Fig. 8.9). Mobile consistency agents, checking type rules "t1" and "t2" receive empty itineraries (lines 1-8 in Fig. 8.9), because comprehensive initial consistency checks have determined that no currently existing documents are relevant to these rules. Therefore an over-estimation of rule applicability, resulting from a light-weight rule selection process, is *corrected* in the mobile agent's itinerary.

```
1. DA: getRuleApplicability (t1) gives null itinerary
2. CA 1e28: null itinerary received from Domain, terminating.
3. DA: getRuleApplicability (t2) gives null itinerary
4. CA 514b: null itinerary received from Domain, terminating.
5. CA 514b: Processing message dispose
6. CA 1e28: Processing message dispose
7. DA: Processing message deleteChecker
8. DA: Processing message deleteChecker
9. CA 6599: got itinerary
10.   CA 6599: checking for local documents...
11.   CA Already processed document atp://B:4434//XMI\Class5.xml
12.   CA Reading files specified in itinerary:
13.   CA Adding XMI\DataType0.xml, …, XMI\DataType16.xml
14.   CA Adding XMI\BehavioralFeature.parameter0.xml, …,
      XMI\BehavioralFeature.parameter3.xml
15.   CA Adding XMI\StructuralFeature.type0.xml, …,
      XMI\StructuralFeature.type9.xml
16.   CA Adding XMI\Class0.xml, …, Class4.xml, Class6.xml, … Class8.xml
17.   CA Adding XMI\Classifier.feature0.xml, …,  XMI\Classifier.feature6.xml
18.   CA Adding XMI\StructuralFeature.type0.xml, …,
      XMI\StructuralFeature.type9.xml
Legend:
CA – log record from a consistency checking mobile agent. The 4-digit number
distinguishes individual agent instances and corresponds to the last 4
digits of a 64-bit unique agent identifier generated by Aglets framework;
DA – log record from the domain agent.
```

Fig. 8.9 Mobile consistency agents receive itineraries from the domain agent.

The mobile consistency checking agent (identifier CA 6599), checking rule "c1", receives an itinerary (line 9, 10 in Fig. 8.9), where all relevant documents to rule c1 are listed. The agent then processes all documents, specified in this itinerary (lines 11-18, Fig. 8.9). These documents are stored locally at host "B", therefore the consistency check occurs locally and migration in this case is unnecessary. A following Scenario II considers a distributed consistency check.

Since all documents in the itinerary for rule "c1" are available locally to the mobile agent, the agent carries out link generation at host "B" (line 1, Fig. 8.10). The consistency links are saved in a link file (line 3, Fig. 8.10), which is opened by the user interface agent in a browser (line 4, Fig. 8.10).

```
1. CA 6599: executing link generation...
2. CA 6599: Time for rule (c1) = 220 milliseconds.
3. CA 6599: Links written to 'Links/m1435882505.xml'.
4. UI: Processing message links, launching 'Links/m1435882505.xml'
5. DA: Processing message deleteChecker
Legend:
CA – log record from a consistency checking mobile agent;
UI – log record from the user interface agent;
DA – log record from the domain agent.
```

Fig. 8.10. Activity log of the prototype. Local link generation for rule "c1".

The incremental check considered in this section is triggered by an event, where Bob creates a new class (Teacher). In this section, we have described the actions, carried out by architectural components as a response to detected changes in the Teacher class when Bob had created this class anew. From Bob's point of view as a user, he saved a new version of the Teacher class, and after a short period of time (less than a second total time, 0.22 seconds for consistency checking, Fig. 8.10), an automatically generated inconsistent link opened in a browser window. The XML representation of this inconsistent link is shown in Fig. 8.6b. Bob navigates the link and establishes that the link corresponds to the lack of implementation of operation *neededDuties* in the class Teacher on the analysis diagram [Appendix C, C.1]. He decides *not* to correct the inconsistency at this stage, as the method is to be implemented at a later stage of project development.

Consistency links establish consistent or inconsistent status of relations between documents. *Inconsistent* links generally indicate that further work is needed to achieve compliance with consistency rules. The scenario I has described an inconsistent link in some detail. *Consistent* links allow developers to navigate between the related UML elements in the model. In a simple example, Anne can get an overview of Bob's classes by navigating consistent links from a common project namespace to each individual class. Having created a number of associations between the classes, Anne can navigate consistent links, which connect the associations and the related classes, to verify the results of her work.

Generation of consistent links allows developers to get a "look and feel" of a complete UML model that they are working on. Consistent links externally connect distributed XMI representations of model elements into a single 'browseable' model. Consistency checking provides developers with views on the model from the development viewpoint of a particular part of the system (i.e., 'class view' or 'association view').

8.7 Scenario II: Distributed Consistency Checking Within a Single Domain

8.7.1 Distributed Check: Generation of Consistency Links

Throughout the analysis stage of the break scheduler application development, the software agent architecture for consistency checking provides Bob with a number of consistent links in addition to the inconsistent links, discussed in Scenario I. One of such links results from an association, which Anne is working on at host "A". This link has been created in a *distributed* fashion, unlike the local link already presented in Scenario I. However, this scenario relies on understanding of the local consistency check from Scenario I.

The developed architecture prototype is coherent: selection of distributed or local checks is carried out automatically, and creation of local or distributed links depends on locations of the documents being checked. Separation of two scenarios is for clarity of description only.

```
<xlinkit:ConsistencyLink ruleid="association.xml#/consistencyrule[4]">
    <xlinkit:State>consistent</xlinkit:State>
    <xlinkit:Locator
xlink:href="atp://A/XMI/AssociationEnd6.xml#/XMI.content[1]/
Model_Management.Model[1]/Foundation.Core.Namespace.ownedElement[1]/
Foundation.Core.Association[1]" xlink:label="" xlink:title=""/>
<!—this is an association between Teacher and ExclusionTime classes,
association's xmi.id = G.14 -- >

    <xlinkit:Locator
xlink:href="atp://B/XMI/Namespace.ownedElement0.xml#/XMI.content[1]/
Model_Management.Model[1]/Foundation.Core.Namespace.ownedElement[1]/
Foundation.Core.Association[1]/Foundation.Core.Association.connection[1]/
Foundation.Core.AssociationEnd[1]/Foundation.Core.AssociationEnd.type[1]/
Foundation.Core.Class[1]" xlink:label="" xlink:title=""/>
<!—this is the Teacher class, class xmi.id=S.10001 -- >

      <xlinkit:Locator
xlink:href="atp://B/XMI/Namespace.ownedElement0.xml#/XMI.content[1]/
Model_Management.Model[1]/Foundation.Core.Namespace.ownedElement[1]/
Foundation.Core.Association[1]/Foundation.Core.Association.connection[1]/
Foundation.Core.AssociationEnd[2]/Foundation.Core.AssociationEnd.type[1]/
Foundation.Core.Class[1]" xlink:label="" xlink:title=""/>
<!—this is the ExclusionTime class, class xmi.id=S.10017 -- >
</xlinkit:ConsistencyLink>
```

Fig. 8.11. Consistent link between a namespace and the ends of an association.

A consistent link (Fig. 8.11) is generated by UML well-formedness consistency rule for associations (rule id "a4") [Appendix A, A.1]. This rule is relevant to UML association elements and concerns association ends, classifiers and the model namespace.

Description of the rule "a4" is the following. The connected Classifiers of the AssociationEnds should be included in the Namespace of the Association. The XML representation of the rule in the consistency rule language is provided in [Appendix A, A.1].

8.7.2 Event: Document Change

In the same way as Bob's, Anne's user profile at host "A" grants modification rights, therefore the resource interface agent at host "A" lets Anne save the changes she made to the Association6.xml document, which contains an association between Teacher and ExclusionTime classes [Appendix C, C.1]. In a similar fashion to Scenario I, the resource interface agent at host "A" launches an incremental check and computes a TreeDiff (Fig. 8.12) between the Association6.xml document and its previous version.

```
<?xml version="1.0" encoding="UTF-8"?>
<treediff>
  <addsubtree insertOrder="1"
xpathleftsibling="/XMI/XMI.content[1]/Model_Management.Model[1]/
Foundation.Core.Namespace.ownedElement[1]/Foundation.Core.Association[1]/
Foundation.Core.Association.connection[1]/text()[1]"
xpathparent="/XMI/XMI.content[1]/Model_Management.Model[1]/
Foundation.Core.Namespace.ownedElement[1]/Foundation.Core.Association[1]/
Foundation.Core.Association.connection[1]">
    <Foundation.Core.AssociationEnd xmi.id="G.15">
      <Foundation.Core.ModelElement.name/>
        <Foundation.Core.ModelElement.visibility        xmi.value="public"/>
        <Foundation.Core.AssociationEnd.isNavigable      xmi.value="true"/>
        <Foundation.Core.AssociationEnd.isOrdered        xmi.value="false"/>
        <Foundation.Core.AssociationEnd.aggregation      xmi.value="shared"/>
<Foundation.Core.AssociationEnd.multiplicity>1..1
</Foundation.Core.AssociationEnd.multiplicity>
        <Foundation.Core.AssociationEnd.changeable       xmi.value="none"/>
        <Foundation.Core.AssociationEnd.targetScope     xmi.value="instance"/>
        <Foundation.Core.AssociationEnd.type>
          <Foundation.Core.Class xmi.idref="S.10001"/> <!-- Teacher -->
      </Foundation.Core.AssociationEnd.type>
    </Foundation.Core.AssociationEnd>
  </addsubtree>
</treediff>
```

Fig. 8.12. TreeDiff: an association end element is added to the UML model.

8.7.3 Processing of the Event by the Software Agent Architecture

The sequence of events in this case closely resembles that of the example considered in case of the local consistency check [Scenario I], with some notable exceptions, which we will discuss in detail.

In a similar way to the case of a local consistency check in Scenario I, the Resource Interface Agent at host "A" identifies consistency rules (Fig. 8.13), relevant to the change shown in the TreeDiff (Fig. 8.12). The consistency rules concerning associations are selected [a1, a2, a3 and a4], and the rule relevance database is then updated by domain agent (lines 3-5, Fig. 8.13). The resource interface agent then instantiates a mobile consistency checking agent for each relevant consistency rule. Execution of all four selected rules by a *single* agent instead of separate agents would have been carried out if a rule execution policy demanded such joint execution.

The mobile agents register in the agents list at the domain (lines 6-9, Fig. 8.13), and each of them queries the rule relevance database at the domain agent for an itinerary of relevant documents to be processed (lines 10-13).

1. RA: XMI\AssociationEnd0.xml changed.
2. RA: Processing message change
3. RA: Finding relevant rules for change
4. DA: Processing message updateRelevant
5. **RA: relevant rules to the change identified :[a1, a2, a3, a4]**
6. DA: Processing message newChecker
7. DA: Processing message newChecker
8. DA: Processing message newChecker
9. DA: Processing message newChecker
10. DA: Processing message getRuleApplicability
11. DA: Processing message getRuleApplicability
12. DA: Processing message getRuleApplicability
13. DA: Processing message getRuleApplicability
14. CA 6f9d NOTaClone haveNOclone **:got itinerary**:
15. CA 6f9d NOTaClone haveNOclone :checking for local documents...
16. CA 6f9d removing atp://A/XMI/Association6.xml (already processed)
17. CA 6f9d **Adding** Association0.xml, ... Association5.xml
18. CA 6f9d NOTaClone haveNOclone **:preparing to** clone and **migrate to** atp://**B**:4334/
19. CA 6f9d NOTaClone haveNOclone :run completed, standby - scanning messages.
20. CA 6f9d NOTaClone haveNOclone :Processing message **clone**
21. CA **clone** a77e IAMaClone haveNOclone :clone created, onClone() is called.
22. CA 6f9d NOTaClone haveACLONE :Processing message **setclone**
23. CA **clone** a77e IAMaClone haveNOclone :**dispatching** to next URL: atp://**B**:4334/

Fig. 8.13. Activity log of the prototype at host A: identification of changes, relevant rules, processing of local relevant documents, migration to host B.

In the remainder of this scenario, we consider the mobile consistency agent, checking rule "a4" (agent identifier 6f9d). The process of execution of distributed checks for different rules is similar,

therefore we consider in detail the consistency rule "a4". The mobile agent receives its itinerary (lines 14-16), and processes all local documents at host "A" from this itinerary (line 17). Nodesets, corresponding to XPath expressions in the consistency rule, are collected from these documents and preserved by the consistency agent until all itinerary documents have been processed and consistency links can be generated.

In addition to documents from host "A", an itinerary of the consistency agent 6f9d also contains some documents, located at host "B". For the purpose of being able to carry out a concurrent distributed check, we propose and use in this and the following scenarios a *cloning and migration approach* instead of the simple agent migration. In order to migrate to the next host "B", from the itinerary, the agent sends a message "clone" to itself (line 18). The message is placed in a queue (line 19) and processed after any previously received message (line 20). In this way, should it become necessary for a user or the domain agent to terminate checking of the consistency rule, a "dispose" message will be processed by a mobile checking agent before the next migration.

The Scenario II considers a sequential distributed check, schematically represented in (Fig. 8.14). In a sequential check, the last agent clone processes all remaining documents at the final host in the itinerary (host D in the example in Fig, 8.14, host B in the example in Fig. 8.13) and carries out link generation at that host. Consequently, nodesets collected by the parent agents at the previous hosts (hosts A, B, C in Fig. 8.14) have to be made available to the clone prior to link generation. Therefore, in a sequential check, the clone agent inherits a copy of all data, collected by the parent agent.

The entry point in the code of the clone agent (agent identifier a77e) is the *onClone()* method (line 21, Fig. 8.13). Here, the clone sets the proxy to the parent agent and sends its own proxy to the parent via message "setclone", which is processed by the parent agent (line 22). The clone then dispatches to the next host in the itinerary (line 23). This cloning process is similar to the cloning example in Figs. 4.2 and 4.3 (Chapter 4).

In order to enable concurrent distributed checking (discussed in Scenario III), checker agents are cloned before migrating to the distributed hosts. The parent agent and all clones, checking the same consistency rule, constitute an agent "family", where each member "knows" at least its parent and its clones (if there are any), and establishes a relationship of trust with these agents. The composition of an agent family ensures that all its members can be reached via any member. Such ability to efficiently reach all distributed agents without a need to identify their locations is essential for controlling a distributed check. Retraction of an ongoing distributed check in the event of a change in the consistency rule is an example where reacheability of all agents in the family is required. Agent families provide an effective solution to management of distributed agents for resolution of redundant consistency checks (section 8.8.5.5-6).

A significant task of a distributed consistency check is to save the generated consistency links at all hosts, where relevant documents reside. When a clone of the checking agent is already running at each host, propagation of links is simplified. The set of clones, located at distributed locations, receive the links via messaging and locally save them in a file on the host. If cloning were not used, the checking

agent would need to migrate to each individual host and locally save the links there, an unnecessary and a lengthy operation.

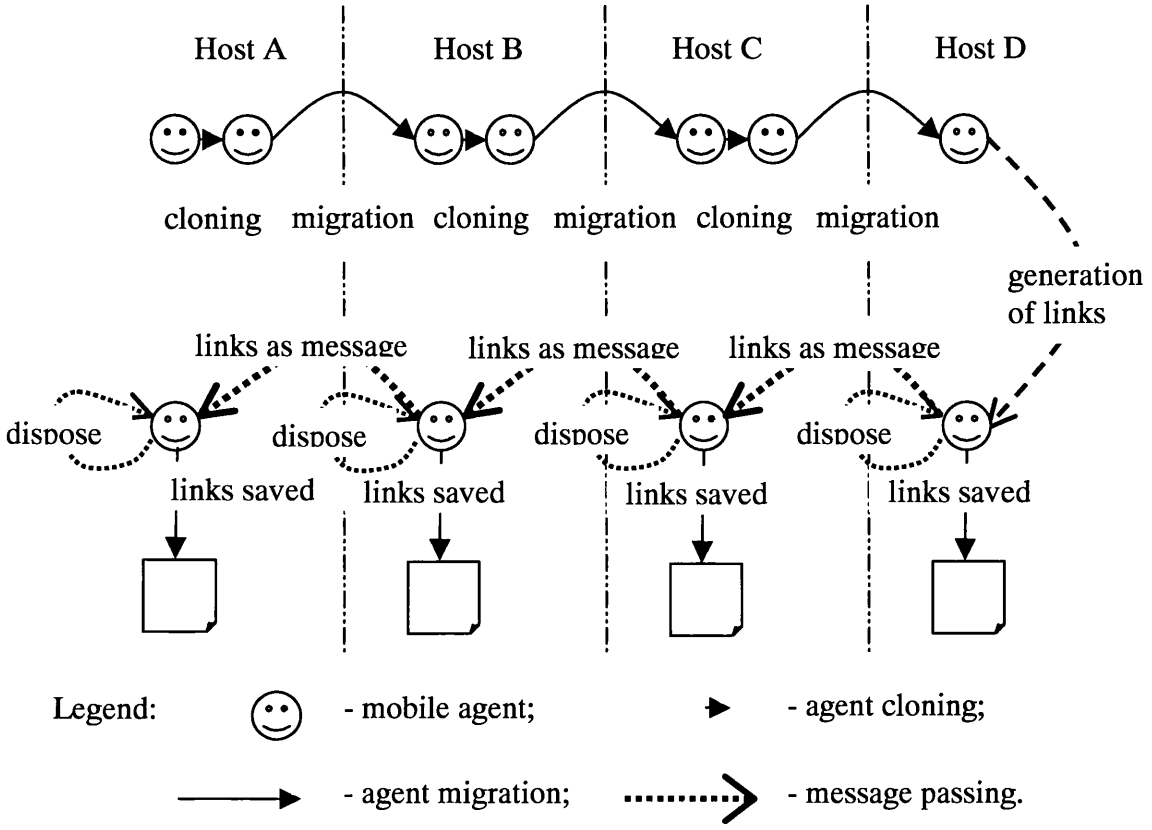


Fig. 8.14. Sequence of cloning and migration actions in a distributed check.

Upon arrival at the next destination host "B", the mobile checking agent processes all local documents, specified in its itinerary (lines 2-5, Fig. 8.13). In the case of consistency rule "a4", relevant documents at host "B" are UML elements-namespaces (lines 3 and 5, Fig. 8.15).

```
1. CA clone a77e IAMaClone haveNOclone :Arrived at destination
2. CA clone a77e IAMaClone haveNOclone : file specified in itinerary, adding
3. XMI\Namespace0.xml
4. CA clone a77e IAMaClone haveNOclone :file specified in itinerary, adding
5. XMI\Namespace1.xml
6. CA clone a77e IAMaClone haveNOclone :executing link generation...
7. CA clone a77e Time for rule (a4) = 721 milliseconds
8. CA clone a77e Links written to 'Links/432725405.xml'
9. CA clone a77e IAMaClone haveNOclone :run completed, scanning messages.
10. CA clone a77e IAMaClone haveNOclone :Processing message dispose
11. CA clone a77e IAMaClone haveNOclone :Clone proxy is null - nothing to shut down.
```

Fig. 8.15. Activity log of the prototype at host B: processing of local relevant documents, link generation, propagation of links to host A.

When all documents from the itinerary have been processed, link generation can be carried out (line 6, Fig. 8.15). By this time, the mobile checking agent has collected all nodesets, corresponding to XPath expressions in the consistency rule. Rule operators are now applied to the values of elements, contained in the nodesets. As a result, a consistent or an inconsistent state of the consistency relation is determined.

In this scenario, generation of the consistency links is carried out at the location of the last document, specified in agent itinerary. However, the implementation prototype is flexible to allow the mobile agent to carry out computation of the consistency status at an arbitrary location. Addition of a URL of the desired processing host as the last element of the agent itinerary will result in the required behaviour. This option would be important for application domains, where execution of consistency rule operators is computationally heavy and it may be desirable to off-load it from an all-purpose client workstation to a more powerful, dedicated server. The resulting behaviour, indeed, does not equal to the centralised processing. In our case, only nodesets from relevant documents, rather than the complete documents themselves, are transported to a specified location for computation of the consistency status.

Generation of links in the Scenario II for the consistency rule "a4" took 0.7 seconds (line 7, Fig. 8.15), and links have been saved locally at host "B". These links have been sent as a message to the *parent* mobile agent at host "A", which also saved the links locally at that host (line 1, Fig. 8.16). The clone disposed of itself (line 10-11, Fig. 8.15), making sure that there were no "children" clones still active (line 11, Fig. 8.15). The agent family structure thus ensures that all its agent members are disposed of when a check has been completed.

```
1. CA: Processing message links, saving links as Links/432725405.xml
2. UI: Processing message links, launching Links/432725405.xml
3. DA: Processing message deleteChecker
```

Fig. 8.16. Activity log of the prototype at host A: storing of consistency links, disposal of the parent mobile agent.

The parent agent at host "A" now un-registers from the agent list at the domain agent (line 3, Fig. 8.16). Throughout the duration of a check of a consistency rule, the registration record in the agents table prevents a re-launch of the consistency rule until the current check has finished. Agent un-registration releases the "lock" on checking of the consistency rule "a4". Any subsequent change to the documents, relevant to this consistency rule, would trigger a new consistency check. The software agent architecture for distributed consistency checking thus ensures that checks are executed in a way similar to transactions: during the check of a particular rule, current versions of documents are used to establish the consistency status.

Any document modifications trigger the resource interface agents to raise events in the architecture. If a number of modification events are raised at a host *after* this host has been visited by a mobile checking agent, but *before* checking of a rule has completed, these events are queued by resource agents for future execution. Once the lock on a rule is lifted at the domain agent level (line 3, Fig. 8.16), new consistency checking agents are instantiated to process the queued events. When a host is visited by a

mobile agent, and the latest document modifications at that host are relevant to the rule being checked, the modifications are incorporated into the current consistency check and the corresponding events are removed from the event queue.

This approach allows users to divide a *stream* of document changes into transactions, where a number of changes are grouped into individual consistency checks. The generated consistency links constitute the result of each transaction and reflect a consistency status of the set of document changes, joined in this transaction.

The durations of individual checks (for example, the timings in Fig. 8.17) are the durations of transactions and constitute minimal intervals of time between consecutive transactions. If a document is modified more frequently than the duration of transactions, multiple changes are likely to be considered jointly within one transaction. In any case, with event queues in place, individual changes will never be lost, no matter how frequently they are introduced.

```
Forall: 741msec, 67 calls, 11.05 msec/call
Exists: 441msec, 106 calls, 4.16 msec/call
Equals: 0msec, 0 calls, NaN msec/call
XPath: 101msec, 94 calls, 1.07 msec/call
Comm: 4176msec, 4 calls, 1044.0 msec/call
Agent information processing: 531msec, 39 calls, 13.61 msec/call
```

Fig. 8.17. Individual statistics for the agent family (parent and all clones).

Consistency links - a result of the distributed consistency check, can be browsed by both Anne and Bob after the check is completed. The generated link is shown in Fig. 8.11.

8.8 Scenario III: Distribution of the Break Scheduler Application.

This section demonstrates how the mobile agent architecture and its prototype handle multiple document changes across numerous hosts in different domains. Two previous scenarios I and II elaborated on examples from analysis and design stage: development of the class diagram and associations between classes in a distributed break scheduler application [Bergner, et al. 1997]. In this section, distribution of the scheduler application is considered.

One of the requirements of the distributed break scheduler is that plan editors may work at home over the Internet with a Java-capable browser [Bergner, et al. 1997, p. 14]. RMI has been chosen by application developers as a distribution architecture, and application logic is now being divided into server-side and client-side objects. Each object, which must be accessed by the client, is split up into an interface and an implementation class. "Loose coupling" principle is to be preserved in doing this. In a Java application, implementation methods are usually hidden by using *private* annotation. In the context of RMI, clients are provided with restricted interfaces, containing subsets of the full class signature. Such restriction of the client's functionality can be seen in the class diagram [Appendix C, C.2]. The

client interfaces contain only parts of the functionality of their corresponding server implementation classes.

The developed application has to support "update on change" of schedules on all connected clients. The standard Java Observable pattern (java.util.Observable) cannot be used in this case, and a remote version has to be used instead, because an observer and observable objects are on different sides of the client/server gap. A freely available implementation of remote observable pattern is available from Caribou Lake Software [Caribou 1997], and is shown on the class diagram as COM.cariboulake.util.RemoteObservable [Appendix C, C.2].

8.8.1 Distribution of UML Model Elements

The Scenarios I and II in the beginning of this chapter were concerned with the initial project development stage. Since then, the development team on-site has expanded and now consists of the server sub-team and the client sub-team. There is also an off-site team at Caribou Lake Software, where development the RemoteObservable pattern for the distributed break scheduler application is carried out.

The new distribution configuration is as follows (Fig. 8.18).

Hosts: A, B, D, domain CDS, developers Anne, Bob, Dimitry – server sub-team
Hosts: CL1, CL2, domain BCL, developers Caroline and Karl – client sub-team
Hosts at Caribou Lake: O, domain CAR, developer Olga – Caribou Lake software team.
Gateway hosts: INT – internal between CDS and BCL domains, EXT – external with CAR.

A schematic representation of the hierarchy of domains and network hosts is given in Fig. 8.19. In this Scenario III, we elaborate on distributed consistency checks between the hosts in different domains. Consequently, a new type of a software agent – the gateway domain agent - is participating in this scenario. Gateway domain agents ensure that mobile checking agents, performing *global* consistency checks, propagate freely between the domains, whereas *in-domain* checks are not allowed to leave the domain.

The two previous scenarios related to consistency checks within a single domain CDS. Both scenarios work in a multi-domain configuration as well. However, we will elaborate on checks between domains to demonstrate the function of gateway domains and present important features of the architecture, such as collaboration between mobile checking agents and the disconnected operation.

Host	Documents, stored at the host (UML elements saved in XMI, other documents)
A	Associations, generalizations between numerous classes within CDS domain
B	Classes Staff/StaffImpl, Teacher/TeacherImpl, Serialization interface, UnicastRemoteObject interface, generalizations between these classes, name space.
CDS	UML well-formedness consistency rules, table of documents within CDS domain, table of relevance of consistency rules to documents (continuously updated). CDS runs Domain Agent

D	Interface Remote, Interface ExclusionTime and ExclusionTimeImpl, PeriodImpl, Break/BreakImpl, BreakPlan/BreakPlanImpl, Organiser/OrganiserImpl, BreakPlanner
INT	Runs Gateway Domain agent between CDS and BCL domains
BCL	UML well-formedness consistency rules, table of documents within BCL domain, table of relevance of consistency rules to documents (continuously updated). BCL runs Domain Agent.
CL1	StatisticsViewImpl, associations and generalizations between classes within BCL domain
CL2	StatisticsView, BreakPlannerView
EXT	Runs Gateway Domain agent between INT and CAR
CAR	UML well-formedness consistency rules, table of documents within CAR domain, table of relevance of consistency rules to documents (continuously updated). CAR runs Domain Agent.
O	COM.cariboulake.util.Observable/COM.cariboulake.util.RemoteObservable

Fig. 8.18. Document distribution for Scenario III.

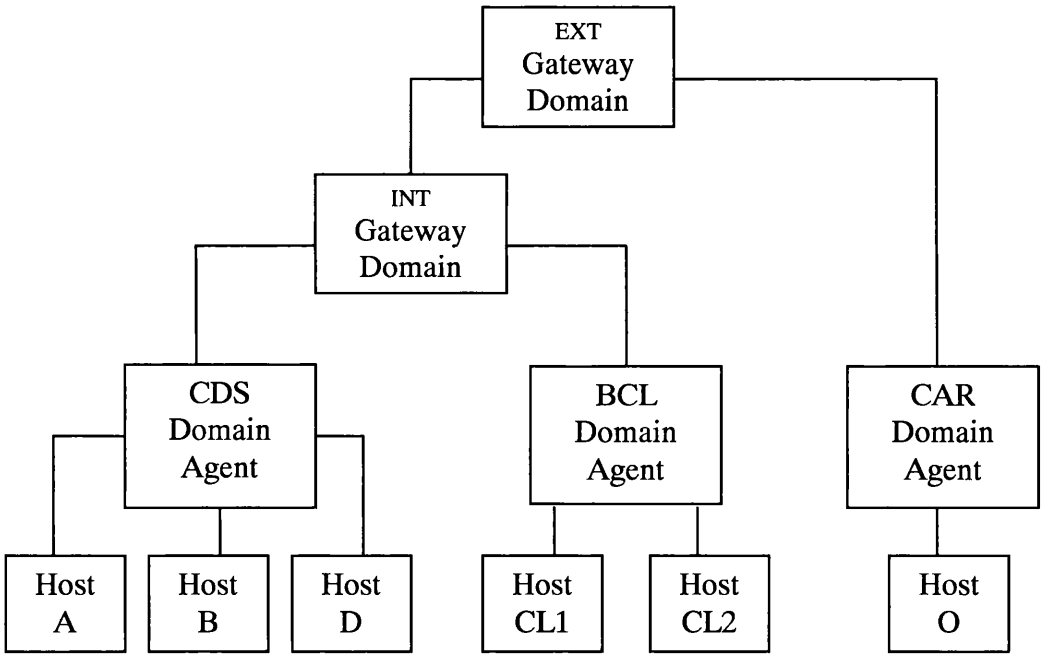


Fig. 8.19. Multi-domain hierarchy of network hosts.

The multi-domain configuration in Fig. 8.19 assumes that all participating hosts (depicted as leaves of the tree hierarchy) run resource interface agents and contain local document universes (DocumentSet.xml) of participating documents.

8.8.2 Inter-domain Agent Migration Policies

When documents are grouped into domains by relation to a particular project module, in-domain consistency checks are usually most often executed. In addition to these, the software agent architecture also provides flexibility for carrying out inter-domain, global consistency checks when multiple domains

exist. For example, a global check of *generalizations* well-formedness consistency rule [Appendix A, A.11] across all three domains in this scenario would establish consistency status of all generalizations.

However, allowing certain inter-domain global checks may or may not be desirable. If all domains in a hierarchy relate to the departments, working closely together within an organisation (like domains CDS and BCL in our scenario), traffic limitations and security concerns may not be too severe; therefore, inter-domain checking could be allowed. On the other hand, when development teams from *different* organisations collaborate (scheduler development team and Caribou Lake Software team in this scenario), it would be natural to establish the limits of each organisation as the borders for consistency checks. Migration policies for consistency checking mobile agents are thus established to control inter-domain checking.

In this scenario, we also consider examples of unilateral migration policies: checks from CDS domain are allowed to propagate to the CAR domain, yet inter-domain checks from CAR to CDS and CAR to BCL are not. Let us suppose, that development of RemoteObserver class is an open source project, and Caribou Lake Software has opened their UML models to consistency checks from other organisations. At the same time, the distributed break scheduler application development is a commercial project, thus checks from Caribou Lake are not allowed to propagate either to CDS or BCL domains.

Within the scheduler project, consistency checks from the application development domain CDS are allowed to propagate out of the project organisation boundaries into the CAR domain. In addition, consistency checks of generalizations (consistency rule "gen1"), originating from the domain BCL, are allowed to propagate to CAR.

Fig. 8.20 summarises mobile agent migration policies, which establish allowed directions of consistency checks between the domains in this scenario. A table of agent routing policies for each gateway domain is provided in Fig. 8.21.

As follows from routing policies for agent migration (Fig. 8.21), the gateway domains EXT and INT operate as "firewalls", protecting the inner domains CDS and BCL from incoming traffic from outside of the organisation (policies 5,6,9, Fig. 8.21). At the same time, both firewalls allow some outward agent migration into the outer world (policies 7 and 8) and into the external domain CDS (policies 3 and 4). The internal firewall INT allows exchange of agents between CDS and BCL domains (policies 1,2).

The "clone and migrate" pattern used in the architecture prototype makes the job of specifying policies for gateway domain agents easier: no inward agent migration is required for the final part of agent's missions: only messaging is used to propagate results of the checks – the consistency links.

We do not elaborate on message routing here, because the prototype implementation allows direct agent-to-agent message exchange via a persistent proxy of an agent [Lange and Oshima 1998]. Direct messaging feature of the agent framework, Aglets, becomes available only to those agents, which *receive* another agent's proxy either from that agent or from the agent *place*.

Mobile consistency checking agent originating **from**	Mobile consistency checking agent **destination**	Example routing path	Propagation of the check is allowed / not allowed
Domain CDS: Hosts A, B, or D	Domain BCL: Hosts CL1 or CL2	B → INT → CL1	Allowed (reason: within same organisation)
Domain BCL: Hosts CL1 or CL2	Domain CDS: Hosts A, B, or D	CL2 → INT → D	Allowed
Domain CDS: Hosts A, B, or D	Domain CAR: Host O	A → INT →EXT → CAR → O	Allowed
Domain BCL: Hosts CL1 or CL2	Domain CAR: Host O	CL1 → INT → EXT → CAR → O	Allowed only of consistency rule type "gen1".
Domain CAR: Host O	Domain BCL	O → EXT	Not allowed
Domain CAR: Host O	Domain CDS	O → EXT	Not allowed

Fig. 8.20. Summary of the policies for inter-domain consistency checks.

Policy no	From Host	To Host	Checks of following rule types are allowed:
Gateway Domain INT:			
1	*.CDS	*.BCL	*
2	*.BCL	*.CDS	*
3	*.CDS	*.CAR	*
4	*.BCL	*.CAR	Gen1
5	*	*.CDS	None
6	*	*.BCL	None
Gateway Domain EXT:			
7	*.CDS	*	*
8	*.BCL	*	*
9	*	*	None
Legend:	* - a wildcard, meaning "any".		

Fig. 8.21. Routing policies for mobile agent inter-domain consistency checks.

At the architectural level, routing of messages between agents in the domain occurs through gateway domain agents. And in this case, message routing policies can be set up in a way, similar to agent routing policies, discussed in this section.

8.8.3 Distributed Inter-Domain Consistency Check

All three domains of the network configuration, used for this scenario, host documents, containing UML elements generalizations. In this scenario we elaborate on checking of the consistency rule for well-formedness of generalizations [Appendix A, A.11].

At this point in project development, the scheduler application class diagram [Appendix C, C.2] contains a relatively large number of interface classes and implementations of the application

functionality. For instance, generalizations located at the server development domain CDS include "TeacherImpl implements Teacher interface", "OrganiserImpl implements Organiser", and a number of other pairs of classes. The client development domain, BCL, contains "StatisticsViewImpl implements StatisticsView interface". An off-site CAR domain contains "COM.cariboulake.util.Observable implements COM.cariboulake.util. RemoteObservable interface". All generalizations, referred to by this scenario, are found on the class diagram [Appendix C, C.2]

Let us suppose, that Dimitry creates a generalization OrganiserImpl implements Organiser, document XMI/GeneralizationG320.xml at host D, domain CDS. In the beginning of Scenario III, we consider a consistency check of this generalization and the distributed checks of other generalizations, relevant to the consistency rule "gen1". Checking the consistency rule "gen1" is a goal of an instantiated mobile consistency checking agent (for example, agent id b47e).

8.8.4 Inter-Domain Document Location Discovery

The inter-domain document location discovery process (example in Fig. 8.22 for the rule "gen1") collects the location information for all distributed documents, relevant to the selected consistency rule, across numerous domains. The mobile checker agent requests the itinerary from the CDS domain agent, and receives the result of an inter-domain document location discovery. The itinerary includes URLs of *all* distributed documents, relevant to checking of the consistency rule "gen1" (Fig. 8.23).

The itinerary, resulting from an inter-domain location discovery, includes URLs of relevant documents from all accessible domains (CDS, BCL, CAR), to which routing policies allow propagation of mobile checking agents from the current domain. In other words, this itinerary is compiled in such a way, which allows mobile agents from one domain to find out about related documents in other domains (hence the name "inter-domain document location discovery"). Compilation of such an itinerary requires collaboration between the domain agent CDS and the gateway domain agents INT and EXT, as well as collaboration between these gateway domains and all other domain agents, connected to them – the domains BCL and CAR (Fig. 8.19).

When Caroline creates a generalization "StatisticsViewImpl implements StatisticsView" on the host CL1 of domain BCL, an inter-domain location discovery procedure is executed for this event also. Due to the symmetry of agent routing policies for the domains BCL and CDS (Fig. 8.21), exactly the same combined itinerary would result in this case (in step 10, Fig. 8.22), as the itinerary for the mobile agent b47e (Fig. 8.23).

The routing policies prohibit outward checking of any consistency relations, except between generalizations, from the domain BCL into the external domain CAR (policy 4, Fig. 8.21). Checking of all relations, however, is allowed from domain CDS (policy 3, Fig. 8.21). For example, if an itinerary were requested for any of association rules (a1-a4), the resulting itinerary returned to the CDS domain would *contain* the association O.CAR/XMI/AssociationG63.xml between COM.cariboulake.util.Observable and COM.cariboulake.util.RemoteObserver. The itinerary for the

same rule, requested from the domain BCL, would *not* contain this association, because routing policies for BCL prohibit checking into the domain CAR (policy 4, Fig. 8.21). Itinerary for the same consistency rule of well-formedness of associations, requested within the domain CAR, would result in selection of only *one* association between two mentioned classes of that domain, and would *not* include any associations from CDS or BCL domains.

1. Mobile agent b47e sends request for itinerary for rule "gen1" to CDS.
2. CDS compiles "local" itinerary in the same way as in scenarios I and II.
3. CDS "knows" that it's connected to the domain agent INT, and forwards the itinerary request from b47e to INT.
4. INT receives the request for documents, relevant to rule "gen1", from agent b47e in domain CDS.
5. Mobile agent routing policies at gateway domain INT allow routing of agents from CDS to BCL, therefore INT queries domain agent BCL for itinerary of documents, relevant to consistency rule "gen1". If routing restrictions applied, no such query would be made for domain BCL. BCL replies to INT with the itinerary.
6. Mobile agent routing policies at gateway domain INT allow routing of agents from CDS to gateway domain EXT, therefore INT queries EXT for itinerary of documents, relevant to consistency rule "gen1".
7. Mobile agent routing policies at gateway domain EXT allow routing of agents from CDS to external domain CAR, therefore EXT queries CAR for itinerary of documents, relevant to consistency rule "gen1".
8. CAR replies with the itinerary to EXT, EXT forwards that itinerary to INT.
9. INT concatenates itineraries from BCL (step 5) and EXT (step 8) and returns the itinerary to CDS.
10. CDS combines this itinerary, received from INT with the local itinerary (step 2), and returns the combined itinerary to mobile agent b47e.

Fig. 8.22. Inter-domain location discovery process example for consistency rule "generalizations".

A.CDS/XMI/GeneralizationG304.xml, A.CDS/XMI/GeneralizationG305.xml, ...
D.CDS/XMI/GeneralizationG320.xml, ...
CL1.BCL/XMI/GeneralizationG335.xml, CL1.BCL/XMI/GeneralizationG336.xml
O.CAR/XMI/GeneralizationG332.xml

Fig. 8.23. Result: inter-domain itinerary for rule "gen1", checked from the CDS domain.

The consistency rule "gen1" demands that all generalizable elements are children of an element of the same type. The agent b47e checks all generalizations in its itinerary (Fig. 8.23), and creates a number of consistency links between the distributed documents. Below, we consider the linked model elements in more detail.

The *result of the local consistency check.* An inconsistent link is created at host D.CDS, where Dimitry created a generalization "OrganiserImpl implements Organiser" (Fig. 8.24). Other locally generated inconsistent links at the host D.CDS include "BreakImpl implements Break", and "BreakPlanImpl implements BreakPlan".

The *result of a distributed in-domain check.* A number of inconsistent links are also created at the host B.CDS: for instance, "TeacherImpl and Teacher", "StaffImpl and Staff". Once again, the reason for inconsistency is in construction of a generalization with a mismatch in the types of the parent and the child element.

Result of a distributed inter-domain check. An inconsistent link is generated by the mobile agent b47e at host O, domain CAR, with respect to the generalization "COM.cariboulake.util.Observable implements COM.cariboulake.util.RemoteObservable". The agent security policies of the CAR domain allow mobile agents from external domains to execute at the domain, therefore *generation* of this link will be allowed. However, if necessary, such policies can prohibit a mobile agent from *saving* consistency links into the local linkbase of the CAR domain.

In practice, all inconsistent of generalizations, identified in this scenario, would normally be tolerated at this early stage in development, rather than result in immediate correction actions by the developers since violation of this requirement does not invalidate the UML model. The UML well-formedness rules require all generalizations to contain equal *Foundation.Core.Generalization.supertype* and *Foundation.Core.Generalization.subtype* sub-elements. On numerous occasions in the original UML model [Bergner, et al. 1997] on which the scenarios are based, and in the particular generalization created by Dimitry, the generalization sub-elements differ (*Foundation.Core.Interface* and *Foundation.Core.Class*), and result in an inconsistent link (Fig. 8.24).

Generalization: Domain CDS, host D, OrganiserImpl implements Organiser
Consistency Link:

```
<xlinkit:ConsistencyLink ruleid="generalization.xml#/consistencyrule[1]">
        <xlinkit:State>inconsistent</xlinkit:State>
<xlinkit:Locator
xlink:href=" D.CDS/XMI/GeneralizationG320.xml#/XMI.content[1]/
Model_Management.Model[1]/Foundation.Core.Namespace.ownedElement[1]/
Foundation.Core.Generalization[17]" xlink:label="" xlink:title=""/>
</xlinkit:ConsistencyLink>
```

XMI Source:

```
<Foundation.Core.Generalization xmi.id="G.320">
    <Foundation.Core.ModelElement.name/>
    <Foundation.Core.ModelElement.visibility xmi.value="public"/>
    <Foundation.Core.Generalization.discriminator/>
    <Foundation.Core.Generalization.subtype>
      <Foundation.Core.Class xmi.idref="S.10065"/> <!-- OrganizerImpl -->
    </Foundation.Core.Generalization.subtype>
    <Foundation.Core.Generalization.supertype>
      <Foundation.Core.Interface xmi.idref="S.10115"/> <!-- Organizer -->
    </Foundation.Core.Generalization.supertype>
</Foundation.Core.Generalization>
```

Fig. 8.24. XMI Source of generalization "OrganiserImpl implements Organiser" and the resulting inconsistent link.

8.8.5 Multi-Agent Collaboration

Multi-agent collaboration takes place, when a number of agents combine the nodesets they collected *concurrently* at numerous hosts across the network. All nodesets are then used to compute the status of a consistency relation. Use of multi-agent collaboration allows the software architecture to execute a consistency check concurrently on a number of documents at different locations. The collaborative multi-agent approach, introduced in this section, extends the sequential distributed approach, considered in Scenario II. The concurrent approach continues to capitalize on the cloning and migration behavioural pattern, introduced in the section 8.7.3 (Fig. 8.14).

8.8.5.1 Concurrent rule checking at distributed locations

Let us suppose, that Caroline introduces a modification to the generalization "StatisticsViewImpl implements StatisticsView" on the host CL1 of the domain BCL. As we considered on an example in 8.8.4, the itinerary of a mobile agent for checking the relevant consistency rule "gen1" contains a number of documents from different hosts. In this case, the itinerary is exactly the same as the one in Fig. 8.23.

Consistency rules, which deal with sub-elements of a particular UML model element (i.e., the generalization well-formedness rule "gen1"), rather than with sub-elements of a number of different UML elements (i.e., the namespace well-formedness rules "n1", "n2"), constitute one example, where concurrent checking may yield a significant efficiency advantage. "Gen1" is a consistency rule, where generation of a consistency link requires processing of one document instance: all parameters, required by the rule, can be found in this document. Multi-agent checking also provides efficiency advantages for a case, where elements for the check have to be collected from multiple documents.

In the multi-agent collaborative checking approach, the complete itinerary (Fig. 8.23) is divided into sub-itineraries; each sub-itinerary ideally contains documents from a single host. The original agent produces agents-clones, and each agent executes one sub-itinerary. If the rule allows an agent to complete the check at the distributed locations (i.e., the generalizations rule), links are generated locally at each host (Fig. 8.25). Otherwise, collected nodesets are transported to a selected network host, where results are combined for link generation (Figs. 8.28-8.30).

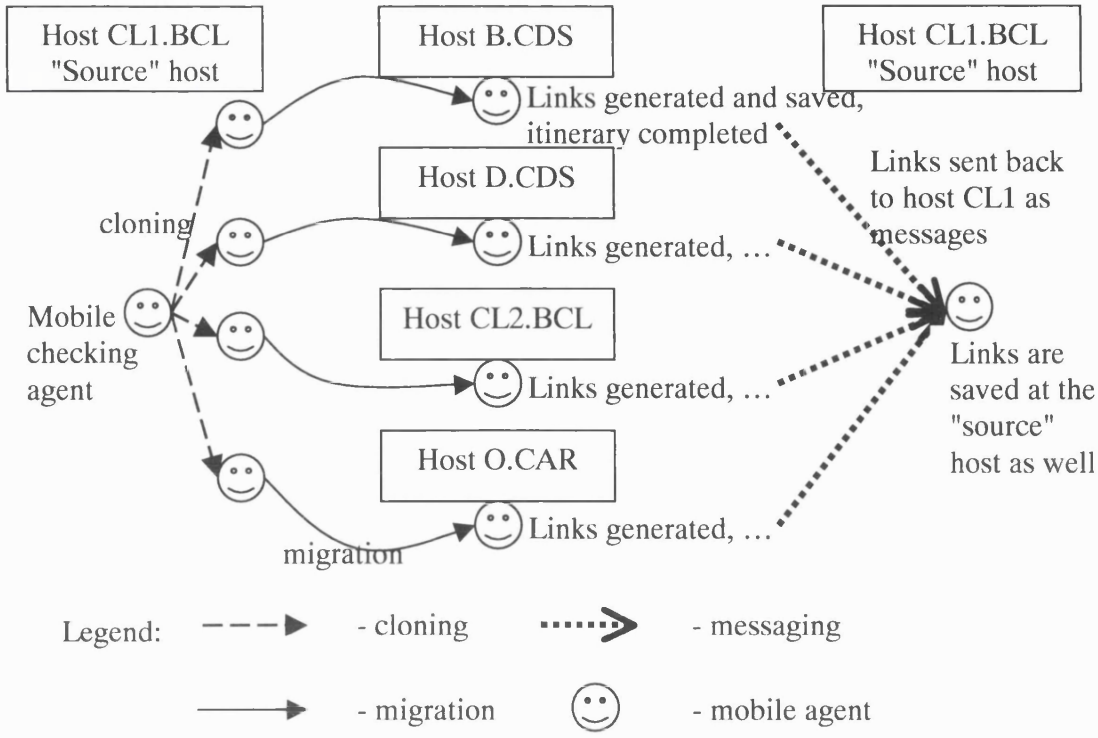


Fig. 8.25. Multi-agent collaboration scenario for concurrent checking of a rule at each participating host.

Multi-agent collaboration in the case of concurrent distributed rule checking is demonstrated in creation of multiple sub-itineraries and their distributed and concurrent execution. The resulting links are then combined and saved at the originating host (Fig. 8.25)

8.8.5.2 Concurrent distributed retrieval and exchange of nodesets

Most often, checking of a consistency rule requires processing of a *number* of different documents for generation of each link, for instance, the namespace well-formedness rule "n2" [Appendix A, A.14]. In such a case, link generation cannot be carried out at the distributed locations, and all collected nodesets need to be exchanged between distributed agents before link generation can commence.

In execution of the consistency rule "n2", a number of associations are considered together with the namespace elements. In this part of the scenario, we executed a consistency check of rule "n2" across the domain CDS, and found an inconsistency: one association was repeated in the UML model twice. We will demonstrate the exchange of data between mobile agents on this example.

Let us suppose, that Dimitry has created an association between the use case *Manage Users* and the use case *System Administrator* [Appendix C, C3]. Dimitry saved the new association on his workstation: D.CDS/XMI/AssociationG146.xml. However, he didn't know, that Anne has created the same association some time ago; the association already exists in the file A.CDS/XMI/AssociationG137.xml. The namespace document is located at host B.CDS (Fig. 8.18) – we use the same distribution of documents across domain CDS throughout the Scenario III.

The documents, participating in a consistency check of the rule "n2", are the following: A.CDS/XMI/AssociationG137.xml, D.CDS/XMI/AssociationG146.xml, and B.CDS/XMI/ Namespace.-xml. These relevant files are included in the itinerary for checking of the consistency rule "n2".

```
<xlinkit:ConsistencyLink ruleid="namespace.xml#/consistencyrule[2]">
<xlinkit:State>inconsistent</xlinkit:State>
<xlinkit:Locator
xlink:href="B.CDS/XMI/Namespace.xml#/XMI.content/Model_Management.Model/Foundatio
n.Core.Namespace.ownedElement/Model_Management.Model" xlink:label="" xlink:title=""/>
<xlinkit:Locator xlink:href="A.CDS/XMI/AssociationG137.xml#/XMI.content/
Model_Management.Model/Foundation.Core.Namespace.ownedElement/
Model_Management.Model/Foundation.Core.Namespace.ownedElement/
Foundation.Core.Association" xlink:label="" xlink:title=""/>
<xlinkit:Locator xlink:href="D.CDS/XMI/AssociationG146.xml#/XMI.content[1]/
Model_Management.Model/Foundation.Core.Namespace.ownedElement/
Model_Management.Model/Foundation.Core.Namespace.ownedElement/
Foundation.Core.Association" xlink:label="" xlink:title=""/>
</xlinkit:ConsistencyLink>
```

Fig. 8.26 Inconsistent link: two copies of the same association exist in the UML model.

Let us suppose that collaboration between mobile agents within the domain CDS is allowed. In certain cases, or for large domains, collaboration may be disabled to avoid excessive cloning of agents and, as a result, unnecessary "clogging" of resources within the domain.

The mobile agent (for example, agent id 437d), instantiated with the goal to check rule "n2", will then create the following three sub-itineraries (Fig. 8.27).

```
Initial itinerary:
A.CDS/XMI/AssociationG137.xml,
D.CDS/XMI/AssociationG146.xml,
B.CDS/XMI/Namespace.xml

Sub-itineraries:
(1) A.CDS/XMI/AssociationG137.xml
(2) B.CDS/XMI/Namespace.xml
(3) D.CDS/XMI/AssociationG146.xml
```

Fig. 8.27. Generation of sub-itineraries.

The document in sub-itinerary number 3 would have already been processed by agent 437a at this point, because the creation of the corresponding file has triggered the consistency check. The two clones of agent 437d are then created: let us suppose they have identifiers 437a and 437b. Sub-itineraries 1 and 2 are then "given" to each of these clones, and executed concurrently at distributed locations (Fig. 8.28). This concludes Stage 1 of multi-agent collaboration.

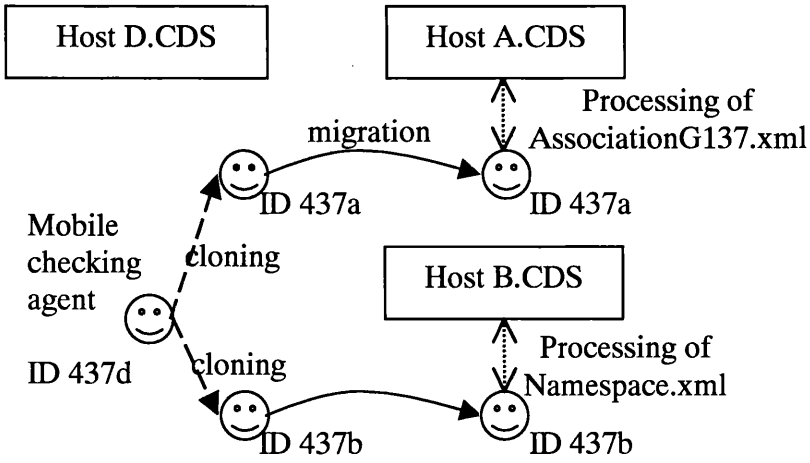


Fig. 8.28. Stage 1 of multi-agent collaboration: execution of sub-itineraries.

By the end of stage 1 of multi-agent collaboration, each of the agents 437a, 437b and 437d have already collected the nodesets from each of three documents, relevant to consistency rule "n2". Because all three nodesets (Fig. 8.28) are required for link generation of rule "n2", none of the checker agents alone is able to compute the consistency status of the rule. In order to carry out link generation, it is necessary to collect all nodesets in one place. Often the host, from which the consistency check has originated, functions as a collection point for the partial nodesets (Fig. 8.29).

In the prototype, a simple approach for collection of nodesets is implemented: since only *one* instance of each consistency rule can be checked at a given domain at any time due to agent registration, all partial nodesets are collected into a separate repository for each rule identifier (Fig. 8.29).

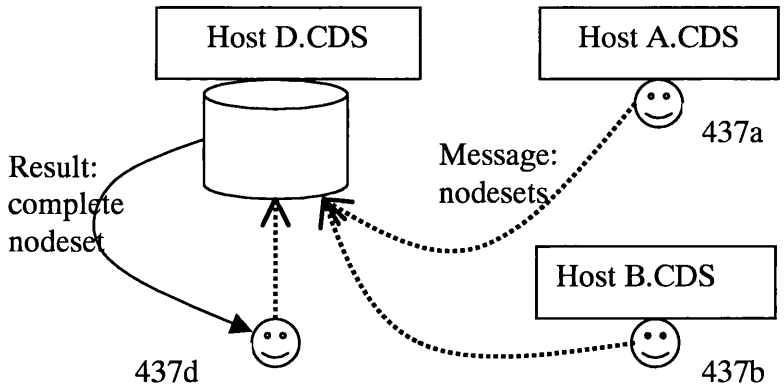


Fig. 8.29. Stage 2 of multi-agent collaboration: collection into the repository.

Once collection of nodesets from all three mobile agents is complete, all nodesets are forwarded to the original checker agent at host D (ID 437d). This concludes Stage 2 of the multi-agent collaboration. The mobile agent carries out the generation of links, and sends the links out to its clones (Fig. 8.30). All three agents save links locally and terminate.

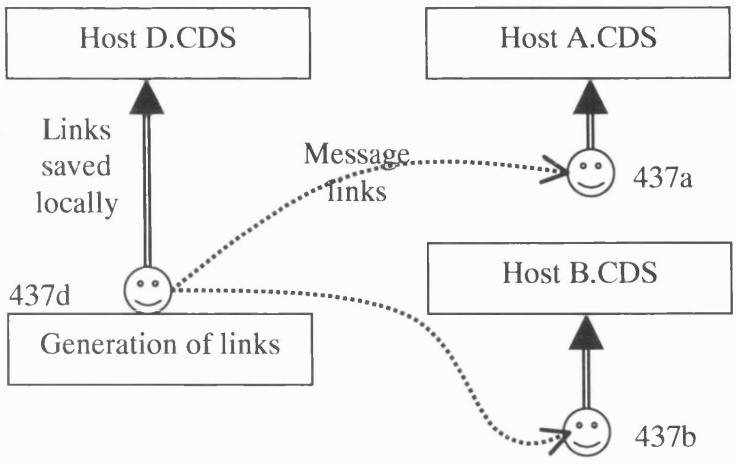


Fig. 8.30. Stage 3 of multi-agent collaboration: generation of links and their propagation.

8.8.5.3 Comments

In this section, we introduced collaborative concurrent checking of consistency rules on a number of distributed documents. Multi-agent collaboration is potentially most beneficial in the scenario, where checking of a consistency rule requires access to single documents (i.e., checking of well-formedness of generalizations). In this case, division of the distributed itinerary into a number of sub-itineraries, local to each participating host, results in a minimum communication overhead, because the link generation can be completed locally at each host. The results of a check – consistency links, are then forwarded from each collaborating agent to the host, where the consistency check originated.

The multi-agent collaboration approach was also presented here for a more challenging scenario, when checking of a consistency rule requires collection of nodesets from a *number* of documents before link generation can be executed. The proposed 3-stage approach capitalises on concurrent nodeset selection by multiple agents at distributed locations, followed by concatenation of nodesets at a specified repository. Messaging, rather than active migration, is extensively deployed, for reasons of efficiency, between trusted members of the agent family. The proposed multi-agent collaboration schemes have a potential to yield efficiency benefits in comparison to traditional, sequential "clone-and-migrate" approach.

Another case, in which the multi-agent collaboration is necessary, is in identification and disposal of "redundant" mobile checking agents.

8.8.5.4 Identification of redundant mobile checking agents

The proposed software agent architecture for consistency checking ensures, that within a particular domain, only one agent family is checking a particular consistency rule at any time across all relevant documents.

Let us suppose, that in a multi-domain configuration (Fig. 8.19) a consistency check has started in a different domain and continues after migration into the current domain. For instance, a new consistency

check (mobile agent ID 535b) of a namespace well-formedness rule "n1" originated at host CL1 of the domain BCL. The agent 535b has already collected the nodeset, relevant to this consistency rule, from the document CL1.BCL/XMI/AssociationG60.xml – an association between Organiser and StatisticsViewImpl. At this time, a clone of the mobile agent (clone ID 535c) has migrated to the host A.CDS for checking of the next document in its itinerary A.CDS/XMI/AssociationG17.xml (Fig. 8.31).

For this example, we consider the complete itinerary of all associations, which extends the shortened itinerary we considered in Fig. 8.27.

CL1.BCL/XMI/AssociationG60.xml, (Organiser and StatisticsViewImpl)
A.CDS/XMI/AssociationG17.xml, (TeacherImpl supervises BreakImpl)
D.CDS/XMI/AssociationG51.xml, (OrganiserImpl and StaffImpl)
D.CDS/XMI/AssociationG54.xml, (OrganiserImpl and BreakPlanner)
D.CDS/XMI/AssociationG66.xml (BreakPlannerView and StatisticsViewImpl)
B.CDS/XMI/Namespace.xml

Fig. 8.31. Complete itinerary of documents, related to consistency rule "n1", from all participating domains

At the same time, let us suppose, that a consistency check of the same consistency rule "n1" has already started independently in the current domain CDS. Dimitry changed the association between OrganiserImpl and BreakPlanner, document D.CDS/XMI/AssociationG54.xml. A mobile agent (agent ID 247d) has collected the nodeset from this document, and also from other related documents at host D.CDS: D.CDS/XMI/AssociationG51.xml and D.CDS/XMI/AssociationG66.xml (Fig. 8.31). An instantiated clone (agent ID 247a) then migrates to the next host in the itinerary – A.CDS, for processing of A.CDS/XMI/AssociationG17.xml "TeacherImpl supervises BreakImpl".

At this point of time, there are two agent families, carrying out checks of consistency rule "n1" within the domain CDS. First family belongs to the CDS domain: agent 247d at host D.CDS and its clone 247a at host A.CDS; the second family originated from domain BCL: agent 535b at host CL1.BCL and its clone agent 535c at host A of the CDS domain.

Allowing both consistency checks to proceed uninterrupted would not be efficient, since in this case some documents would be processed by both agent families (i.e., more than once): for instance, all associations at host D.CDS in Fig. 8.31. Therefore, only one consistency check of the two running checks should be allowed to proceed.

The domain agent CDS *identifies redundancy* in checking of consistency rule "n1" when the cloned agent 535c migrates into the domain CDS from domain BCL and registers with the domain agent CDS. Upon instantiation of a mobile agent for a *new* consistency check, or upon entering by a mobile agent of a *new domain*, the agent sends a proxy of itself and an identifier of the consistency rule being checked to the domain agent (message newChecker, Fig. 8.8). Likewise, upon such registration of agent 535c, the domain agent CDS attempts to log 535c for consistency rule "n1" in the "agents table" database, and identifies that agent 247a has already started checking of rule "n1" at the domain. A message "redundant", containing proxies of detected agents, 535c and 247a, is then sent to both agents.

8.8.5.5 Resolution of redundant checks

Receipt of the "redundant" message from the domain agent pauses the normal execution of each redundant mobile agent. In response to the "redundant" message, all agents forward the nodesets, that they have so far collected, and their partially completed itineraries to the domain agent. Having done so, all redundant agents clear their itineraries and are "paused", awaiting further directions from the domain agent.

The multi-agent collaboration now actively involves the domain agent: all collected nodesets are concatenated, and a remaining itinerary is generated by *intersection* of itineraries of redundant agents (Fig. 8.32). The domain agent is used as the host, where the repository for concatenating nodesets is located.

The resulting concatenated nodeset and the remaining itinerary are returned to one of the redundant mobile checking agents. Usually there is no difference, which agent of all redundant agents is selected to continue the check. Let us suppose, that agent 535c is selected to execute the remaining itinerary that now contains a number of generalization documents at host A.CDS and B.CDS (Fig. 8.32).

Agent 535c, originates from domain BCL, selected to **continue execution**	Agent 247a, originates from domain CDS	Result
Uncompleted Part of the Itinerary: A.CDS/AssociationG17.xml, D.CDS/AssociationG51.xml, D.CDS/AssociationG54.xml, D.CDS/AssociationG66.xml, B.CDS/Namespace.xml	*Uncompleted Part of the Itinerary*: CL1.BCL/AssociationG60.xml, A.CDS/AssociationG17.xml, B.CDS/Namespace.xml	*Remaining Itinerary* (operation AND): A.CDS/AssociationG17.xml, B.CDS/Namespace.xml
Collected nodeset: From AssociationG60.xml at CL1.BCL	*Collected nodeset*: From AssociationG51.xml, AssociationG54.xml, AssociationG66.xml at D.CDS	*Collected concatenated nodeset* (operation OR): From AssociationG60.xml at CL1.BCL From AssociationG51.xml, AssociationG54.xml, AssociationG66.xml at D.CDS

Fig. 8.32. Concatenation of itineraries and nodesets from redundant consistency checking agents.

The agent 535c continues its execution of the remaining itinerary: it clones itself and migrates to all hosts in this itinerary. All other redundant agents remain awaiting a response from the domain agent.

When execution of the remaining itinerary is completed and consistency links are generated, the agent 535c follows the normal link propagation pattern: it saves the links at the current location, and forwards the links as a message through all agent clones down to the checking agent, which started the check (agent 535d at host CL1.BCL). In addition, the agent 535c also forwards the links to the domain agent CDS. The domain agent re-forwards the links to all awaiting redundant agents (agent 247a).

The redundant agent 247a is waiting for response from domain agent CDS. When links are forwarded to 247a, its itinerary is empty, therefore, it proceeds to carry out normal link propagation through its agent "family", i.e., to its parent agent 247d at host D.CDS.

8.8.5.6 Comments

Identification and management of redundant consistency checking agents is an important feature of the proposed software agent architecture for distributed consistency checking. Management of redundant checks is in a certain way similar to conflict resolution: numerous agent itineraries and the nodesets need to be reconciled in such a way, that no acquired information is lost. From then on, the completion of the consistency check relies on only one *family* of agents to finish the check.

The approach to identification and resolution of redundancy heavily relies on the concept of the *agent family* in the software agent architecture. In the example we have considered, redundancy identification halted execution of an agent from one family, and the execution of another agent from a different agent family was allowed to proceed. The same mechanism applies without modification for a case, where in each agent family, a number of agents concurrently execute checks at numerous locations. The "Redundant" message, received by one agent in the family, is forwarded through the hierarchy of parents and clones within that family and reaches all family members, which are all paused by the receipt of this message. The following receipt of the remaining itinerary by a member of the agent family re-commences the check of the remaining documents. When one agent family has generated links, the links are collected at the domain, where redundancy was detected, and are forwarded to all other redundant agent families on standby. Within each of these families, the links are distributed between the member agents via messaging.

The redundancy identification and resolution task once again highlights the efficiency in management of distributed concurrently operating agents that the agent family construct provides in the architecture. Without knowing a complete list of all distributed agents or having any centralised control point, the architecture is able to effectively reach and manage an arbitrary number of distributed agents within each family via a *single* family member by use of the established "peer-to-peer" family relationships. The redundancy resolution example is thus another "killer" application for the agent family concept, in addition to the approach of terminating distributed checks for consistency rules, which have been modified (as mentioned in the section 8.7.3).

The proposed approach to identification and resolution of redundant checks is based on the multi-agent collaboration scheme, which we presented. The approach is based on the architectural features, demonstrated throughout this chapter, in the Scenarios I, II and at the beginning of Scenario III, specifically agent cloning and migration, messaging, and propagation of links and events via messaging throughout the agent family.

In the remainder of the Scenario III and of this chapter, we address other useful features of the software agent architecture: firewalls, disconnected operation of domains and replication of domain agents.

8.8.6 Firewalls and mobile agent security

The software agent architecture and its implementation prototype allow system administrators to configure agent migration policies in order to regulate the inflow and outflow of agents from any domain. Section 8.8.2 of this chapter described the inter-domain agent migration policies in detail.

Here we present an example of a firewall at the gateway domain EXT, which has been set up to protect the distributed break scheduler project from external consistency checking agents. The corresponding prohibitive policy (policy number 9, Fig. 8.21) disallows mobile agent migration from within the organisation into the outside world for all but two development domains: BCL and CDS (policies 7 and 8).

Let us extend the example from the section 8.8.3-8.8.4, where we considered a distributed inter-domain consistency check of UML elements Generalizations, and consider operation of firewalls on this example. Let us suppose that Olga modified the generalization O.CAR/XMI/GeneralizationG332.xml COM.cariboulake.util.Observable implements COM.cariboulake.util.RemoteObservable. Due to the prohibitive migration policies at the gateway domain EXT, the itinerary for checking well-formedness of generalizations rule "gen1" contains only documents from the domain CAR, unlike the complete itinerary shown in Fig. 8.23.

Creation and configuration of firewalls is an effective security measure. In our architecture, agent migration policies at gateway domain agents allow creation of firewalls, which regulate migration of mobile agents between the domains. Perhaps even more importantly than that, inter-domain document discovery in the software agent architecture is constructed in such a way, that one can effectively protect information about *names* and *locations* of documents from becoming known outside of the firewall (i.e., when the routing policies are referenced in steps 5-7 of the inter-domain location discovery algorithm, Fig. 8.22).

The system administrator should configure migration policies in such a way, which will not hinder teamwork across project and organisation boundaries. As we have demonstrated in this example, any changes to the freely available software component COM.cariboulake.util.RemoteObservable do *not* become instantly "known" to developers of the distributed break scheduler application in the current configuration of gateway domain agents EXT and INT. By restricting the visible itinerary of documents, the EXT domain prevents consistency checks of the CAR domain from propagating into the domains BCL and CDS of the application development project. Changes in CAR become "known" to developers in BCL and CDS domains "on-demand", rather than "on change": CAR-related consistency links are re-generated when a document, relevant to any Caribou Lake component, is changed in the scheduler project. In our case, this kind of behaviour is desired, because it saves our main developers' time for tracking each change in the package from Caribou Lake Software.

8.8.7 Disconnected operation

The proposed software agent architecture for distributed consistency checking provides support for disconnected operation of individual hosts and groups of hosts. 'Disconnected operation' implies, that the consistency checking services within the domain will continue operating, should the network connection between the domain and the rest of the network become unavailable either willingly through disconnection into the off-line mode, or unwillingly due to a network fault.

By providing for disconnected operation, the software agent architecture caters for needs of remote users, operating from home with the intermittent connection over a modem, and also for needs of larger organisations, where distributed offices may become temporarily disconnected from each other because of network maintenance work. In each case, consistency checking functionality within the disconnected domain can be provided: all documents within that domain continue to be involved in in-domain consistency checks in a normal way, as if disconnection has not taken place.

Inter-domain consistency checking between the disconnected domain and the outer world is paused during the disconnection period. However, consistency links, which were generated previously by consistency checks when connection was still available, continue to exist within the disconnected domain. Therefore, throughout the whole disconnection period, all users of the disconnected domain can analyse the state of the document universe - all consistent and inconsistent links, which were created just before the moment of disconnection.

The remainder of this section will briefly outline on the example of disconnection of the domain BCL in the network configuration of the distributed break scheduler application development project (Fig. 8.19). Let us suppose, that the network connection between BCL and the outside world is temporarily unavailable. In this case, gateway domain INT, and all of domains CDS and CAR become unreachable from within BCL.

The complete itinerary for checking well-formedness of generalizations "gen1" consistency rule (Fig. 8.23) across all domains includes documents from host CL1, domain BCL, and a number of documents from domains CDS and CAR:

CL1.BCL/XMI/GeneralizationG335.xml, CL1.BCL/XMI/GeneralizationG336.xml
A.CDS/XMI/GeneralizationG304.xml, A.CDS/XMI/GeneralizationG305.xml, ...
D.CDS/XMI/GeneralizationG320.xml, ...
O.CAR/XMI/GeneralizationG332.xml

During disconnection of the domain BCL from the outside world, inter-domain document location discoveries via the gateway domain INT are no longer available. Since during this time the gateway INT cannot be accessed, only consistency checking of documents, local to the domain BCL, is carried out. The partial itinerary for checking of consistency rule "gen1" within the disconnected domain BCL is shown below (Fig. 8.33).

CL1.BCL/XMI/GeneralizationG335.xml, CL1.BCL/XMI/GeneralizationG336.xml

Fig. 8.33. Partial itinerary for checking rule "gen1" at the disconnected domain BCL.

Once connection of BCL domain to the outside world is restored, the gateway INT becomes accessible, and any subsequent inter-domain document location discovery requests will become processed, and will return complete, multi-domain itineraries.

Thus any consistency checks, starting *after* re-connection either from within the previously disconnected domain or from outside of this domain will be able to propagate between the domains in a normal way. Disconnected operation is a normal operational mode of the software agent architecture.

8.8.8 Replication

In the above example of a disconnected operation we discussed the effects of a network disconnection between the domain BCL and the gateway domain INT. If the software at the gateway domain INT crashes, or the network host INT is down, an effect, logically equal to the disconnected operation, will be forced on all sub-domains of INT – the domains BCL and CDS. Even though users from BCL and CDS will continue to be able to access documents in each other's domains in a distributed fashion (unlike the physical network disconnection discussed above), the scope of consistency checks will be limited to individual domains throughout the outage of gateway domain INT.

In a way, similar to the outage of a gateway domain, an interruption of a domain agent will postpone any *future* consistency checks in this domain until the point in time, when the domain agent becomes operational again. However, the software architecture does not allow all distributed consistency checks already underway to correctly complete in this case: the mobile checking agents, which already possess their multi-domain itineraries, will be able to carry out nodeset collection from distributed hosts in a normal fashion, without involvement of the local domain agent.

With respect to stability, each individual domain agent and gateway domain agent is a single point of failure within the domain or within the network, respectively. Replication of domain and gateway domain agents is proposed as a measure to leverage the possible risks.

The software agent architecture is able to combine two replication schemes. Firstly, backup domain or gateway domain agent *software* can be launched on the same host; this is replication in software. Secondly, "backup" agents can be run on alternative hosts, thus reducing the risk of domain or gateway domain agent unavailability due to hardware failures. The second approach is the physical replication.

The ability of the prototype to run multiple copies of a domain agent on the same network host is due to use of a separate Java virtual machine for execution of each separate agent instance, and to allocation of different agent transfer protocol [Karjoth, et al. 1997] ports for separate domain agent instances. The latter feature is provided by the Aglets mobile agent framework [IBM 1998]: both in agent migration and messaging, complete host addresses with URL and ATP port number are used. Using this feature, multiple agents, executing on one network host, can be individually addressed.

The software agent architecture and its working prototype implement an automatic switch-over mechanism between the main and the "backup" domain and gateway domain agents, for both types of the software and physical replication.

8.9 Summary

In this chapter, we have introduced and discussed a number of scenarios, which help explain the process of distributed consistency checking with the software agent architecture.

Scenario I has introduced an example of UML model distribution across several hosts, and discussed the process of a local consistency check, triggered by an event - document modification, made by a developer. This scenario complements introduction of the incremental checking algorithm in Chapter 5. Scenario II considered a consistency check with a distributed itinerary, where migration between hosts in a domain was required. This scenario introduced the "clone and migrate" approach, which gives mobile checking agents presence at all hosts, concerned with the check. The resulting "family" of connected agent clones improves control of distributed checks, where all agents in the family can be reached via any member of the family. This approach is also superior to conventional migration, as it allows resulting consistency links to be efficiently propagated to the distributed locations via messaging along the family of present distributed agents.

Scenario III considered a distributed consistency check between documents in distributed domains. On an example, we introduced mobile agent migration policies, by which execution of inter-domain consistency checks can be controlled, and a process of inter-domain document location discovery, which provides information about distributed document locations without a need for a central repository for such information. An approach to concurrent execution of distributed checks was described, where agents collaborate by exchanging nodesets, collected at different locations. In the remainder of the scenario, we considered the disconnected operation of the software agent architecture, which facilitates distributed development by improving fault tolerance of consistency checks within the domains and enabling operation via intermittent network connections.

Chapter 9 Implementation Prototype

9.1 Introduction

Chapter 7 has introduced the model of the software agent architecture for distributed consistency management and outlined the internal structure of each architectural component and presented sequences of performed operations within these components through state transition diagrams. Chapter 8 described several scenarios, where the working implementation prototype of the software agent architecture is demonstrated and evaluated in a single-host, multiple-host, and multiple-domain configurations. Discussion of the scenarios is complemented by explanation of some of the features of the prototype, and provides detailed examples of collaboration of architectural components in different situations. In that way, the scenarios served to clarify operation of the architectural components on these particular examples.

This chapter will focus in greater detail on the structure of the implementation prototype. Following the event-oriented design of the software agent architecture, in this Chapter we look at architectural components from the view point of events, which are created and processed by each component. Event orientation is supported by the message model used in the Aglets mobile agent framework, which was used to construct the implementation prototype.

Construction of the prototype follows design of components, elaborated in the state charts, which were used for state-based modelling in Chapter 7. Development and evaluation of the working prototype allowed us to optimise the architecture in a number of ways. Most notably, cloning-migration pattern has being introduced as a more efficient alternative to agent migration, proposed in the original architectural design.

9.2 Resource Interface Agent

In this section, all events, processed by the resource interface agent, are described. This agent is launched at each host, where exist one or more documents, participating in the consistency checks.

9.2.1 Startup

On startup, domain discovery is carried out to obtain the addresses of the main and backup domain agents. At initialisation, all consistency rules present at the host in the local document universe are applied to all local documents. This is carried out in preparation for incremental checking and aims to establish which rules apply to documents initially. The array of document names and vector of

applicable to each document rules are sent to the domain agent with message "documentRuleApplicability".

Resource Interface agent creates a thread, which monitors each of participating documents for change. When a datestamp of the document changes, the thread runs an XML TreeDiff on the changed document, and passes message "change" with parameters – document name and TreeDiff, back to the Resource Interface agent. The backup copy of the document is then replaced by its current version.

The Resource Interface agent completes initialisation and starts processing any incoming messages.

9.2.2 Handled messages

9.2.2.1 Change – change detected on a document

The message carries as parameters: the XML TreeDiff between the old version of a document and the new version, and the name of the changed document. This message is sent by the thread, which monitors changes in all participating documents at this host.

If a document has been deleted completely, or a new document has been added to the local document universe, "addDocument" or "deleteDocument" messages respectively are sent to the domain agent.

Analysis of individual changes in the TreeDiff is carried out by intersection with XPath expressions in consistency rules [Chapter 5, 5.3]. As a result, lists of consistency rules are produced, which are relevant to document changes. An "updateRelevant" message is sent to the domain agent, that updates the table relevance of rules to documents.

For each relevant consistency rule, a new message "newChecker" is sent to itself (Resource Interface agent).

9.2.2.2 NewChecker – instantiation of a new consistency checking agent

Resource Interface agent instantiates a new mobile checking agent, which takes as parameters the name of a changed document and the identifier of the relevant consistency rule.

The "newChecker" message is sent to the domain agent, in which the proxy to the instantiated agent is sent, together with the identifier of the consistency rule being checked. This information is stored at domain agent level, and used for detection of redundant consistency checking agents.

The proxy is a location-independent reference to a mobile agent, by which messages can be sent to this agent, regardless of agent's location. One important feature of the Aglet mobile agent framework [IBM 1998, Lange and Oshima 1998] we used for the prototype is that proxy is guaranteed to remain persistently valid until the relevant agent has terminated.

A copy of the mobile agent proxy and the consistency rule identifier are also stored locally, in order to allow the Resource Agent to terminate the check if necessary (i.e., upon request of the user).

9.2.2.3 Dialog – displays the status of the document and consistency checks

Message triggers display of document histories and other statistical information: a number of changes to each participating document at this host, a number of consistency checks, and file names of relevant link files.

9.2.2.4 Links – local saving of generated consistency links

This message is normally received from the mobile checking agent, and contains the generated consistency links. This XML document is serialised by the Resource Interface agent into a local file on the host.

The message is appended by a parameter – name of the file, and is forwarded to the User Interface agent at the host. It is at the discretion of that agent, whether links should be displayed in a browser at this time.

9.2.2.5 Update – checking of all local documents for consistency

This message is normally received from the user interface agent; the message is sent by the user. The message results in checking all consistency rules on all local documents.

This message triggers the resource interface agent to send a number of "newChecker" messages to itself, each message containing one identifier of a consistency rule. Identifiers of all consistency rules, existing at this host, are used. This triggers consistency check of all consistency rules on all relevant participating documents.

9.2.2.6 Dispose – disposal of the resource interface agent

This message can be sent by the user-administrator of the host, when the host needs to be disconnected from the consistency checking framework or taken down, permanently or temporarily.

The Resource Interface agent terminates all ongoing consistency checks, which have originated at this host. "Dispose" messages are sent to mobile checking agents, using locally saved proxies.

Document monitoring thread is stopped, and backup copies of participating documents are removed.

A message "updateRelevant" is sent to the domain agent, which lists all local documents with no relevant consistency rules. A number of "deleteDocument" messages are sent to the domain agent, in order to remove information about names of local documents. This is done in order to disengage participation of the local documents in the consistency checking framework.

9.3 Domain Agent

9.3.1 Startup

The agent initializes the data structures [Chapter 6, 6.3], identifies addresses of replicated domain agents [Chapter 8, 8.8.8], and awaits any incoming messages.

9.3.2 Handled Messages

9.3.2.1 DocumentRuleApplicability –incremental checking infrastructure

This message establishes the relation of applicability within the document table of the domain agent. Both parameters, the array or consistency rule identifiers and an array of document names (more precisely, URLs also containing host name), are saved into the document table.

The data structure is constructed in a way, which allows querying the document table for list of documents, relevant to the rule, (agent itinerary) or for the list of consistency rules, relevant to the document (used when updating rule relevance after a document has been changed).

9.3.2.2 UpdateRelevant – document changes require checking of new rules

Contains a document name and a list of relevant consistency rules. Used as a convenient update mechanism of the document table.

The domain agent updates both representations of the document table: list of consistency rules, relevant to the particular document, and the list of documents, relevant to each consistency rule.

9.3.2.3 GetRuleApplicability – mobile checking agent itinerary constructor

Each mobile consistency agent, checking a particular consistency rule, receives the itinerary with names (URLs) of documents, relevant to this rule. This message has a parameter with the identifier of the consistency rule being checked.

The domain agent searches the local document table for documents, currently registered as relevant to the consistency rules. This is an in-domain itinerary.

If the domain agent is connected to a gateway domain, the GetRuleApplicability message is forwarded there. The result of an inter-domain document location discovery [Chapter 8, 8.8.4] constitutes an inter-domain part of the itinerary.

The in-domain and inter-domain parts are joined together and returned to the requesting mobile checking agent, in a message "itinerary".

9.3.2.4 AddDocument and DeleteDocument – changes in document universe

The AddDocument message notifies of the expansion of the document universe, which now includes a new document in the domain. DeleteDocument removes all information about this document from the document table.

Any mobile checking agents, querying the domain after this point in time, will get updated information with respect to new and deleted documents.

9.3.2.5 NewChecker – launch of a new consistency check

The message contains a consistency rule identifier and a proxy of the mobile checking agent. These are saved into the "agent table", which is maintained by the domain agent.

If the domain agent finds, that there is already a mobile checking agent in the domain, checking the same consistency rule, *redundancy* is detected between these mobile checking agents. The domain sends out "redundant" messages to all mobile agents involved in checking of this consistency rule. A sequence of multi-agent collaboration actions are then followed [Chapter 8, 8.8.5].

9.3.2.6 DisposeChecker – ending of a consistency check

Upon successful completion of a consistency check, or on termination by the user, mobile checking agents notify the domain agent with the DisposeChecker message.

This message lifts the locks on checking of the consistency rule by removing the mobile agent concerned from the agents table.

9.3.2.7 Redundancy – avoiding redundant consistency checks

Each "redundancy" message is a reply from the mobile checking agent to the message, sent out by the domain agent when redundancy of consistency check has been identified. Thus, normally, a number of messages of this kind will be received: one from each "family" of mobile agents.

The message parameters are: identifier of the consistency rule, proxy of the mobile agent checking the rule, a partially completed itinerary, and a vector of nodesets already collected throughout execution of the itinerary.

An example is given in [Chapter 8. 8.8.5.4-8.8.5.5] on how domain agent concatenates nodesets and computes intersection of itineraries. The concatenated nodeset and the remaining itinerary are sent to one of the redundant mobile checking agents.

9.3.2.8 Links – redistribution of links after redundant consistency checks

Since only one of redundant mobile checking agent families is allowed to continue the consistency check [Chapter 8. 8.8.5.5], the resulting consistency links need to be distributed to other redundant

families at the end of the check. This message "Links" is sent by the mobile agent upon having completed the check.

The message is forwarded by the domain agent to each of redundant mobile checking agents, registered in the agent table.

9.4 Mobile Consistency Checking Agent

9.4.1 Startup

Upon instantiation, the mobile agent receives a consistency rule identifier and the name of a changed document from the resource interface agent.

In order to find out locations and names of documents, which it needs to process before link generation can be executed, the checking agent requests itinerary from the domain agent, by sending message "getRuleApplicability", with a parameter – consistency rule identifier.

9.4.2 Handled Messages

9.4.2.1 Itinerary – list of documents with locations for consistency check

The message contains a parameter – array of document names (with URLs). In a multiple-domain configuration, the returned itinerary contains documents from the current host, current domain and other domains (inter-domain location discovery).

The checker agent processes all documents from the itinerary, which are available on the local host. Processing includes execution of XPath expressions from the consistency rule, being checked, on each of the documents from the itinerary. The result – node sets, are saved into a vector together with the corresponding XPath expressions. Values of elements from these node sets are ultimately used to establish consistency or inconsistency status of the consistency rule.

In order to continue processing of documents, which are located on other network hosts, the mobile agent must be transferred to the next host, specified in the itinerary. At design stage of the architecture, it was decided that agents will migrate between hosts, i.e. will transfer their code and data from one host to the other. The implementation prototype uses a "clone and migrate" approach, which allows propagation of the consistency links after their generation across all the hosts in the mobile agent's itinerary.

Before migration, the mobile checking agent creates a clone of itself, and passes on the collected nodesets and the itinerary to the agent clone. The "parent" agent remains at the current location, and is awaiting the consistency links in a form of a message "Links" from the agent clone. Whenever the clone completes the itinerary and has computed links, one agent migration operation is "saved" for each host in the itinerary since messaging is used instead.

Clone and migrate pattern is deployed at each host, specified in the itinerary. In the Thesis, we often refer to the set of agents and their clones, which are checking the same consistency rules across a number of hosts as an agent "family". Note, that the domain agent has a proxy to the father of the family; at each level in the family the father has a proxy of the son, and the son has a proxy of the father. Message "Links", and other control messages, such as "dispose", are passed within the family from father agents to clones, and the whole set of agents is always "connected" to each other.

The IBM Aglet mobile agent framework, which we use for the implementation, allows the implementation prototype to extensively benefit from the use of the "clone and migrate" pattern. During installation of the prototype, agent code is distributed to all participating nodes. Thus, during migration between such hosts, mobile Aglet agents only transfer the *state* of the code and any data that the agent has acquired; the code base already exists at all hosts.

To conclude, throughout the design of the architecture and development of the working prototype, our approach to agent mobility has evolved. Initially, we relied on "strong" mobile agents, where all code was going to be transferred together with the data. The "clone and migrate" approach proved to be more efficient for our application domain, where propagation of links across all previously visited hosts is a necessity. Furthermore, it has become possible to use *state of code* migration, which further enhanced the performance.

9.4.2.2 Clone – clone oneself and migrate the clone to the next host

Message "clone" is sent by the mobile checking agent to itself in order to initiate the cloning procedure. The procedure consists of a sequence of calls to the methods of the agent class, which are associated with cloning [Chapter 4, 4.7.5].

9.4.2.3 SetClone and SetParent

Each message contains a parameter – proxy of the clone agent or the parent agent. The messages are used to set the proxy of the clone in a parent agent, and the proxy of the parent in the clone [Chapter 4, Example in 4.7.5]. These proxies are used to keep all members of the agent family "in touch" by exchanging messages between one another.

9.4.2.4 Links – propagation of consistency links across distributed hosts

The message contains a parameter – internal representation of the XML document, in which consistent and inconsistent links are registered.

This message can be received from the clone to the parent mobile checking agent. In this case, the message establishes, that the clone has completed execution of the itinerary, and has terminated after the message had been sent. The parent agent under consideration saves (serialises) the links into a local file on the current host. Then, the message is forwarded to the parent of the current agent, if one exists, then the mobile agent terminates [Chapter 8, 8.7.3, Fig. 8.14].

The "Links" message can also be received from the domain agent in a case, when "redundant" consistency checks were detected [Chapter 8, 8.8.5.5]. In this case, the mobile agent serialises the links into the local file, and forwards the message to any clones and to the parent, if one exists.

The message can be received by the clone from the parent agent. This case also corresponds to resolution of "redundant" consistency checks. The domain agent stored a proxy to the *parent* of the agent family and, therefore, forwarded the "Links" message to that parent. Since then, the message is propagating through agent family, down the hierarchy from parent to each clone. Similarly to the previous cases, consistency links have to be serialised into a local file, and the message needs to be forwarded to any existing clones of the current agent.

A mechanism is in place within the implementation prototype of the software agent architecture, where policies can restrict the ability a mobile agent to save consistency links at the current domain. The Security section below considers the issue in more detail.

9.4.2.5 Dispose

Upon termination, the mobile checking agent must un-register from the agents table at the domain agent. Thus, if this instance of an agent has registered with the domain agent (this happens either for the root parent agent of the agent family or for a clone, which has migrated into a different domain), then the agent sends "deleteChecker" message to the domain agent.

9.4.3 Cloning Procedure

Cloning is an event in lifespan of the Aglets mobile agent, which triggers a sequence of calls to methods of the agent class [Chapter 4, 4.7.5]. OnCloning is executed by the parent before cloning begins, OnClone is executed by the clone after creation, and OnCloned – by the parent after creation. The sequence of calls to these pre-defined methods is defined by the Aglets mobile agent framework; these calls can be considered events, because similar to messages-events, they are processed by methods of the mobile agent's class.

9.4.3.1 OnCloned

The parent agent establishes a link to the clone: proxy of the clone is saved in the agent.

9.4.3.2 OnClone

The clone establishes a link to the parent agent: proxy of the parent is saved in the clone agent.

The clone selects the next document from the itinerary, and migrates to the host, where the document is located.

9.4.4 Migration

9.4.4.1 OnArrival

Upon arrival to a host of a destination domain, different from the original domain, the mobile agent identifies the address of a domain agent. Domain discovery process is followed: the agent reads the configuration file of the local resource interface agent, and determines URL addresses of the primary domain agent and of any backup domain agents.

Having acquired the address of the domain agent, the mobile agent sends it the message "newChecker" with a parameter – an identifier of the consistency rule being checked. This is used to establish whether another agent is already checking the same consistency rule - "redundant" consistency checking within the domain.

The checking agent then processes all local documents at the current host, which are specified in its itinerary. If at this point of time, all documents from the itinerary have been process, the mobile checking agent executes generation of consistency links between the distributed documents in the itinerary. Technically, generation is carried out between the collected nodesets, which are identifiable by the name of each relevant distributed document instance, and the XPath path, leading to the node or a set of nodes. At the end of link generation, the mobile checking agent sends to itself the message "Links", with a parameter, containing the resulting internal representation of consistency links.

If the agent's itinerary still contains one or more documents after the ones processed at this host, the mobile agent then once again clones itself and its clone migrates to the next host in the itinerary [Section 9.4.3].

9.4.5 Redundant Consistency Checks

Redundant consistency checks may occur in any domain of a particular configuration of the architecture. When redundancy is identified, one or more mobile checking agent *families* are involved in checking of the same consistency rule across a number of documents in the domain. An example of the redundant check and resolution is given in [Chapter 8, 8.8.5].

9.4.5.1 Redundant – resolution of redundant checks

This message, sent by the domain agent, identifies the current mobile checking agent as redundant. The agent pauses its execution, and awaits further messages from the domain.

One of two kinds of messages can be received after "redundant" message. The first kind, "itinerary", contains the new itinerary to be executed in a normal fashion. This itinerary is an intersection of partially completed itineraries from all redundant consistency agents. The second parameter is the concatenated nodeset, containing the nodeset, collected by all redundant agents. Both parameters *replace* the respective itinerary and collected nodeset data structures.

The second kind of message is "Links". The message confirms, that execution of the consistency check has been completed by another agent, and the produced links now need to be forwarded to all agents within the agent family of the current mobile agent. Processing of this message is carried out in the normal fashion.

9.5 Gateway Domain Agent

9.5.1 Startup

Upon initialisation, the gateway agent loads mobile agent migration policies, which establish inter-domain migration patterns of consistency checking agents.

The domain agent also performs discovery of all domain agents, to which it is connected. Discovery is carried out in the same fashion, as mobile checking agents after migration into the new domain [Section 9.4.4.1].

9.5.2 Handled Messages

9.5.2.1 GetRuleApplicability – initiation of the inter-domain location discovery

This message is received from the domain agent of the domain, where a new consistency check has just started. The domain agent forwards this message to all connected domain agents, except the one where the message has originated.

9.5.2.2 Itinerary – distributed itineraries

As a reply to the GetRuleApplicability message, domain agents reply with partial itineraries, which contain documents within their domains. The gateway domain agent concatenates all such itineraries, and sends the result with the "itinerary" to the domain agent, which requested the distributed itinerary. All itineraries are marked with the consistency rule identifier, thus confusion does not occur when multiple consistency rules are processed in the inter-domain document location discovery.

9.6 User Interface Agent

User interface agent is the front interface of the system to the user. There are a number of policies, which distinguish users of different kinds: administrator, power user, user and guest. The User Interface Agent is in charge of exposing the functionality of the software agent architecture adequately to the role of each particular user.

9.6.1 Startup

Upon startup, user login panel is provided. Users, successfully logged in, are assigned the role of the administrator, power user, user, or guest, depending on the security profile of the user.

The user interface consists of the control panel, which allows users to query the status of individual agents, running on this host, and engage in dialog with the agents by sending messages to the agents. Depending on the rights of the user, one or more of these functions become "greyed out" (disabled). For instance, a "create" control, which invokes the function of the Aglets framework and instantiates an agent class, is only available to administrators.

9.6.2 Handled Messages

9.6.2.1 Links

The user interface agent opens the browser window, and displays consistency links to the user. An appropriate stylesheet can be applied to the XML document, which contains the links, if required.

9.7 Security

The software agent architecture for distributed consistency management aims to support security features at the user level, at the mobile agent level and at the message level. Flexibility, functionality and ability to carry out distributed consistency checks were primary concerns during development of the software agent architecture; therefore, initial security support is in place at this time. During the discussion, we outline some directions for extension of security features in the future work on the architecture.

At the user level, the user interface agent is capable of maintaining user profiles and allowing users to log into the system. Resource interface agent is in place to enforce user security policies with respect restrictions on modification of documents, which participate in consistency checks, restrictions on changes to consistency rules, and to the document universe.

At the level of execution of mobile agents within the domain, the domain agent follows trust policies between domains. These policies establish whether mobile agents can save generated links in this domain. Resource interface agents enforce these policies.

At the level of migration of mobile agents between domains, the gateway domain agent maintains migration policies, which control inter-domain mobile agent migration.

At the level of exchange of messages – events between agents of different types, or between different instances of the same type, verification procedures are in place that discard unwanted and unexpected messages.

Below, we consider the implementation prototype of the software agent architecture, and describe each of the levels of security in more detail. We discuss security-related messages, through which all

components of the architecture communicate and enforce security constraints on the users and mobile checking agents.

9.7.1 User-level Security

Different types of users can be specified in the security settings file ".java.policy", which sets configuration parameters for the Java virtual machine (Java JDK or JRE versions 1.2 or later). Administrators, power users, users and guests can log in to the user interface agent, in order to be able to change documents, monitor consistency checks, view consistency links, and perform other actions, allowed by the security settings of the relevant type of user. In the prototype, user authentication is carried out with respect to Java security permissions, specified in the policy file.

The resource interface agent maintains holds the file handles to all files, participating in the system. These include all documents, participating in the document universe, the document universe file itself, consistency rule documents, and consistency link files. When the prototype is running, all system files and user documents are monitored for change by a "watchdog" thread of the resource interface agent, and by default the thread "locks" all such files and prohibits modifications.

9.7.1.1 Messages "Login" and "Release"

User authentication gives users permission for modification of system files. After a user has logged in, the user interface agent sets the security policy for this user, and sends a message "login" to the resource interface agent with the parameter – user login name. For each of participating files, Java security class java.security.AccessController is used by resource interface agent to decide whether an access to a system resource is to be allowed or denied, based on the security policy currently in effect.

If access is allowed, message "release" is sent to the thread, which is monitoring the file for change. This causes the thread to release file handle in the intervals between checks for document update. The thread also starts a decrementing "timeout" counter; if the user doesn't access the file during this timeout, file lock is automatically replaced.

When the file lock is lifted, the user can save all modifications to the file. The modifications are then identified by the watchdog thread, and consistency checks or system reconfiguration actions proceed in a normal fashion, as defined by the architecture.

9.7.1.2 Messages "Logout" and "Lock"

When the user logs out from the user interface agent, the message "logout" is sent to the resource interface agent. The latter sends the message "lock" to the watchdog thread, and all file locks are replaced. The system is awaiting login of the next user.

9.7.1.3 Comments

The proposed file locking mechanism is designed to be deployed in a single user per host scenario. Such scenarios are intended uses of the software agent architecture for consistency management, where individual users are modifying local files at their workstations and are interested to establish the status of consistency relations between such distributed collections of files. The locking mechanism in this way corresponds to the target use scenarios.

For the case of multiple users modifying documents on the same host (i.e., in case of users logged in via telnet or remote login), more sophisticated file locking can be built, where sophisticated reconciliation of concurrent modifications to a document by several users can be carried out. Deployment of distributed authoring protocols like WebDAV [Goland, et al. 1999] can be recommended for such setting.

From the standpoint of the software agent architecture, multiple user on a single host access can be provided: the resource interface agent is the component, responsible for monitoring of all participating documents and system files and establishing access permissions to them. In the implementation of the architecture, a simple and lightweight locking mechanism is being demonstrated, which is based on standard Java security mechanisms. Extension of this mechanism to include multiple user modification access on the same host is a part of the future work.

9.7.2 Execution-level Security of Mobile Agents

Execution-level security policies are used in the prototype to establish whether a mobile checking agent from one domain is allowed to save the generated consistency links on a host in another domain. In multi-domain configurations of the architecture, similar to the example in Scenario III [Chapter 8, 8.8.1], migration of consistency checking agents between domains will occur. Thus, security policies are in place and are enforced by the resource interface agents, which define mobile agents' rights on saving of consistency links and display of these links to the user currently logged in at the host.

When a mobile checking agent migrates into the domain, the domain agent registers the agent identifier in the "agents' table". At the same time, security policies are checked; it is established if this agent is allowed to save links in this domain and display these links to users. These permissions are also saved in the "agents' table".

When the mobile checking agent has generated consistency links and sends the message "Links" to the resource interface agent, the latter queries the domain agent for security permissions of this agent – message "getAgentPermissions". After a reply, message "agentPermissions", has been returned, if permissions allow, the links are saved locally at this host.

User interface agent acts in a similar way to the resource interface agent: upon receipt of message "Links", it requests permissions from the domain agent and displays the links in the browser if permissions allow.

In order to make the explanation of security policies and multi-agent interaction more transparent, the execution-level security issue is described in this paragraph, rather than in the sections, related to the user interface agent, resource interface agent, domain agent and mobile checking agent. The messages-events "getAgentPermissions" and "agentPermissions" are sent and received in the same way, as other messages relevant to operation of these architectural components. Processing of these messages is described in this section.

9.7.3 Migration-level Security of Mobile Agents

Migration-level security is enforced via migration policies, which are set at the gateway domain level. These policies specify all allowed paths for migration of mobile agents into and from the current domain.

The explanatory example of inter-domain migration policies, provided in Scenario III [Chapter 8, 8.8.2], demonstrated the role of gateway domain agents in managing the traffic of migrating agents between domains. In order to simplify understanding of the concept, the gateway agent was demonstrated as performing the role of a router on the migration path between two individual network hosts in different domains.

The software agent architecture for distributed consistency checking deploys the gateway domains as routers, which are envisaged not only to control migration, but also to be able to perform conversion of the mobile agent's code and data in order for that agent to run on heterogeneous hardware and operating system platforms in different domains [Chapter 6. 6.5.3].

The implementation prototype is built on the agent transfer protocol (ATP) of the Aglet framework, which relies on serialisation of java code and agent's data, and on socket-based connection for migration of agents between the two hosts. The protocol does not provide a transparent way to configure programmable routers on the path of the connection.

In order to support migration-level security in the implementation prototype, the domain agent component assumes enforcement of the migration policies for all agents, migrating into the domain. Immediately after migration, when the migrated mobile agent registers with the domain agent, the latter requests the migration security policy for the domain of the incoming mobile agent. The domain agent sends the message "getAgentPermissions" to the gateway domain agent, with a parameter – originating domain of the mobile agent. A reply from the gateway domain – message "agentPermissions" specifies whether migration of this agent is allowed by the migration policy or not.

If migration from a particular domain is not allowed, a "dispose" message is then sent to the migrating mobile agent, and that instance of the agent terminates its execution within the current domain. Any remaining documents in the agent's itinerary are then processed by other agent clones in the family, originating from the "unwanted" domain.

9.7.4 Message-level Security

Message level security relates to identification of unwanted (possibly malicious) and unexpected messages, which are exchanged by instances of components of different kinds.

Unwanted messages are erroneous messages-events, which trigger execution of an unnecessary in current conditions or an unwillingly malicious action by the receiving component. For instance, continuous stream of "update" messages from the user interface agent triggers continuous checking of all consistency rules, the action requiring a lot of resources at all participating hosts. This stream may be a result of a failure of the component (user interface agent) or malicious user action, which triggers generation of these messages.

Unexpected messages are messages-events, which are easier to detect than unwanted messages. Unexpected messages are legitimate messages - part of interaction protocols, which repeat, arrive out of sequence, or are lost and need to be re-requested.

The components, which exchange messages, can be trusted (i.e., legitimately instantiated, registered components within the domain) or untrusted (i.e., mobile checking agents of a different domain, which engage in direct message exchange with components of current domain).

9.7.4.1 Message authentication

For each message in the Aglets mobile agent framework, in addition to the type of the message and its parameters, the address of the originating agent and the type of that agent can be determined.

Stationary agents (user interface agents, resource interface agents, and the domain agent) of the current domain are considered "trusted" agents within that domain. Trust relationships need to be established between the gateway domain agent and all domain agents, connected to the former. By default, agents of "foreign" domains are not trusted agents; however, relationships of trust can be established on the individual (agent-to-agent) and group (domain-to-domain) level.

"Trust" is the notion, supported by the Aglets mobile agent framework, and the prototype builds on this support. With respect to message authentication, a message is successfully authenticated if it originates from a trusted agent. In such a way, messages like "updateRelevant" (discussed in sections 9.2.2.2, and 9.2.2.5) require some form of authentication; they need to be checked they've been sent by a trustworthy resource interface agent.

However, authenticated messages are still subject to security issues, because they are not always "expected".

9.7.4.2 Unexpected messages from trusted agents

The implementation prototype of the software agent architecture for consistency checking builds communication between all components on the principle of messaging, message-event queues and processing of these messages in the order of their arrival. However, processing of messages of particular

kinds, discussed throughout this Chapter, in most cases triggers a well-defined sequence of message exchanges between the components. These message sequences relate to sequence of actions, performed by each component, which is prescribed within the software agent architecture by state transition component models [Chapter 7].

Therefore, the first class of security threats at the message level, out-of-sequence messages from trusted components, is handled at the level of *components* of the *architecture*. Out-of-sequence messages in the prototype implementation of the architecture are "caught" in exception handlers, of which records in the logs are made for monitoring purposes.

9.7.4.3 Messages from un-trusted agents

For another class of message level security threats, messages from un-trusted agents, an initial solution exists at the *implementation* level. The simplest approach is in place: non-authenticated messages from un-trusted agents are not processed by any components of the prototype.

However, this approach, effective from the security viewpoint, somewhat lacks efficiency from the viewpoint of future extension of architecture functionality. Any communication with a component within the domain from outside of the set of "trusted" external domains requires *migration* of the communicating agent into this domain. If migration policies do not allow such migration, communication becomes impossible. At the same time, it may, for example, be useful for a developer to be able to query the domain agent for a status of consistency checks within that domain, once the developer is located at a different domain (i.e., at a conference).

9.7.4.4 Unwanted messages from trusted agents

Third level – identification of unwanted messages from trusted agents, is a largely unsolved issue. While a number of checks can be hard-coded in the implementation prototype for most common patterns of unwanted messages, the solution of the problem is clearly in introduction of additional, higher-level authentication mechanisms between components or additional architectural components in charge of coordination. In the future work, we aim to consider possible architectural approaches to this issue.

9.8 Summary

In this chapter, we have elaborated on the types of events, which architectural components create and process. In this description of the implementation prototype, we have intentionally chosen a detailed degree of elaboration of individual events, in order to be precise in describing the functionality that has been achieved so far in the implementation of the software agent architecture.

This chapter concludes a set of chapters, describing the software agent architecture that we have introduced in this thesis. Based on the functional requirements demanded of a distributed consistency checking architecture, the architecture components were introduced in Chapter 6, and their internal structure was represented as a state transition model in Chapter 7. In Chapter 8, we have demonstrated in

detail the operation of an implementation prototype on a number of local and distributed consistency checking scenarios of increasing complexity. Our discussion in this chapter contributes to clarity and completeness of prototype description and takes the preceding chapters as a prerequisite.

This chapter demonstrates that the software agent architecture is extensively event-oriented, which enables a substantial degree of extensibility and flexibility. Inter-component communication via asynchronous event notification lessens coupling of components, and allows us to expand, contract and cluster the distributed system at runtime. Disconnected operation of domains is an important example of such flexibility, which is made possible by the asynchronous nature of the adopted event model. Composition of the architecture from loosely coupled components also allows us to extend the architecture by adding components of new types with additional functionality, replace existing components with their updated versions, and exclude unnecessary component types (i.e., exclude the user interface agent for automated operation in the background).

The following Chapter 10 presents a quantitative, performance evaluation of the prototype and a qualitative evaluation of some of the characteristics of the architecture.

Chapter 10 Evaluation

The operation of the implementation prototype of the software agent architecture for distributed consistency checking has been demonstrated on a number of scenarios in Chapter 8. The prototype and the incremental consistency checking method were tested in a single-host local configuration, multiple host configuration within a single domain, and multiple host setup with multiple domains.

The purpose of this chapter is to highlight and summarise the strengths and weaknesses of the proposed software agent architecture. Firstly, we will assess whether the architecture meets the functional requirements, specified in Chapter 4. In doing so, we will refer to the evaluation scenarios from Chapter 8, and map the particular features of the architecture, demonstrated on these scenarios, to the functional requirements that we demanded.

Secondly, we discuss, from a user's point of view, the performance level that the current implementation of the architecture prototype has achieved. In this section, we compare performance results of exhaustive consistency checks with incremental checks and comment on different positioning of both types of checks within the development life cycle of the project. We compare user experience in side-by-side evaluation of the centralised checking tool and the implementation of the distributed software agent architecture for consistency checking.

10.1 Qualitative features

10.1.1 Elegance

Elegance of the proposed software agent architecture for consistency checking can be established by the degree of intuitiveness, understanding and transparency of the process for users, that deployment of this architecture establishes. In related work, software and mobile agent solutions for the complex task of distributed consistency checking that the architecture aims to address, tend to involve a multitude of different types of mobile and stationary software agents, operating at different hosts and meeting places [Prestegard, et al. 1999].

In the proposed architecture, an explicit effort has been made to make the architecture simpler and more transparent to use for the end user throughout the project life cycle. At all architecture development stages from functional requirements (Chapter 4), through analysis and initial design (Chapter 6 and Chapter 7), in development of the implementation prototype (Chapter 9) and evaluation on scenarios (Chapter 8), each of the four architectural components was targeted at particular requirements, which it implements or satisfies (Chapter 6, 6.5).

Instead of inventing new component names, all architectural components are described in a simple fashion, where names correspond to the role, performed by each component in the system.

Decomposition of the consistency checking task into a number of clearly identifiable roles, performed by components, aids in understanding of the architecture.

The architecture does not define a unique namespace, as most component names have been previously used in software agent systems. At the same time, our use of common namespace does not diminish our contribution in construction and evaluation of the software agent architecture for distributed consistency checking. The architecture can hardly be judged as trivial: functionality that the components provide in response to the events, that the architecture processes, and complex interactions between multiple instances of different types of components across numerous hosts provide solutions for complex scenarios (Chapter 8).

Locality of consistency checks, which is an important principle that the architecture builds on, facilitates intuitiveness and transparency of the architecture for its users. Reactive incremental checks originate locally at the host where modifications are made to the documents, and pro-active diagnostics of inconsistencies is provided via the generated links. The software agent architecture prototype provides automatic execution of consistency checks in the background, which enables users to concentrate on the nature of documents and consistency relations, rather than on the mechanism for invocation and control of distributed consistency checks.

10.1.2 Manageability

One of the difficulties in carrying out distributed consistency checks is the question of how to make consistency rules available and update them at locations, where these rules are required. Another challenge that the architecture tackles is compilation of a list of distributed documents, related to a particular consistency rule, without having a single centralised repository for document relevance and location information.

Solutions to both of these problems impact manageability of the architecture. For the latter problem, an automatically updated database of relevant documents and consistency rules is maintained at each domain agent in the domain hierarchy. The approach is focused on efficiency of updates to relevance information, since documents are expected to be often modified and their relevance to consistency rules will change. The trade off for this solution is centralisation of information within each domain and a possibility of bottleneck occurring at domain agent, since the information is always being requested and updated there. Availability of multiple-domain configurations is a step towards de-centralisation of relevance information.

Consistency rules are expected to be modified far less frequently than the documents, as the consistency relations that the rules express correspond to an established convention or a standard. The set of consistency rules is thus distributed to all participated hosts. In addition, since multiple rules are accessed for each incremental check, it is an advantage to make them available locally. In this approach, the trade off is the difficulty in updating consistency rules at all distributed locations in an atomic fashion.

The solutions for the problem of rule distribution and the problem of centralisation of document and rule relevance information have been proposed in the software agent architecture, and have been found satisfactory in a number of consistency checking scenarios [Chapter 8]. At the same time, we do not consider these solutions final, and investigation of efficient distribution mechanism must be continued.

Tool support for creation and management of consistency rules is another issue for the future development of the architecture and its implementation. It is envisaged, that the rule editor, analogous to the one created for the previous version of the rule language [Zisman, et al. 2000], can also be used to trigger updates of instances of the changed rule across all hosts.

10.1.3 Rule Applicability Policies

Deployment of system policies to specify applicability of consistency rules may be useful in a case, where checking of additional consistency rules, which are not detected as relevant by the incremental checking algorithm, is required for a particular document change. Some examples are execution of a sequence of consistency rules and a dependency of execution of one rule on the result of an execution of another one.

Use of policies may also be needed if a particular change does not trigger execution of a certain consistency rule in the general incremental checking algorithm of rule selection, but execution of this rule is required by an external factor (i.e., a demand of the customer). An example of such policy is execution of consistency rules at regular intervals of time (i.e., daily at the end of the working day).

The resource interface agent and the domain agent [Chapter 6] provide support for execution of rule applicability policies. The implementation prototype provides the basic implementation of rule application policies, which is inherent in the incrementality of consistency checks. As such, a consistency rule is checked every time it is found relevant to a document change. Future work will include further support for specification of more complex policies and investigation of how conflicts between policies can be managed.

10.1.4 Flexibility and Dynamic Reconfiguration

Dynamic reconfiguration occurs automatically for addition and removal of documents and consistency rules. Resource interface agents monitor changes in configuration files, which describe the rule set and the document set at each host, and are able to react to changes in those files. During reconfiguration, the information on relevance of consistency rules to documents, which is stored by the domain agent, is updated for all new documents by resource interface agents.

The architecture also offers flexibility in removal and addition of hosts and domains at runtime, without termination of ongoing distributed consistency checks. Upon creation of a new domain, default mobile agent routing policies are assigned to the new domain at the gateway domain agent level, and thus the new domain becomes available to incoming distributed consistency checks.

Dynamic reconfiguration feature can be made more efficient for a particular reconfiguration scenario, where documents are moved between different hosts. Current prototype sequences the removal action of the document and creation of a new document at a different location. Rule relevance information in this case is re-generated, whereas it could be reused instead. Future work will investigate how the existing approach can be optimised.

10.1.5 Grouping of Resources Into Domains and Support of Domain Hierarchies

The domain is a unit of the architecture, which has the following characteristics:

- It is a unit of organisation for a set of resources;
- Security boundary;
- Agent migration and cloning boundary;
- Document namespace and rule relevance boundary;
- Administration boundary;
- Policy boundary.

Domains are collections of hosts, participating in distributed consistency checks, which are served by a domain agent. The documents, located at these hosts, are "grouped" in the domain. These documents are normally related to the same sub-set of the global project, of which the domain forms a part.

Domain serves as a document namespace and rule relevance boundary: all information about the documents in a domain and relevance of consistency rules to these documents is contained and maintained by the domain agent within that domain. All relevant queries are processed inside the domain, and only a small part of the domain information is replicated by the gateway domain. In this way, in-domain consistency checks are processed in the most efficient way.

Capacity for disconnected operation, discussed below, builds on these two characteristics of a domain: being a unit of organisation for a set of resources and serving as the document namespace and rule relevance boundary.

The security boundary is maintained by use of user profiles and application of inter-domain agent migration policies, which restrict access to in-domain resources for users and incoming mobile agents, respectively. Across-domain security policies implement user authentication procedures. The domain is also an initial point of contact and mandatory registration for all migrating mobile agents.

Related to the security boundary, the domain administration boundary keeps the policies, agents of the domain and their configuration under the control of domain administrators.

The rule applicability policies have an effect within the current domain and by default do not apply to other domains. The domains are created for purpose of differentiation of consistency rule application policies and grouping of resources. In addition, creation of the domain hierarchy usually reflects:

- Organizational structure (i.e., project and sub-project domains within the development organisation [Scenarios 1-3, Chapter 8]);
- Delegation of authority (i.e., user rights: to change particular documents, and run particular sets of consistency checks on them)
- Capacity (i.e., to reflect difference between development of user documentation and mainstream design and development of the software itself).
- Clustering (i.e., bringing together developers working on different parts of the project to facilitate cooperation in development).
- Administration (i.e., to distributed administrative roles across a number of persons).

Scalability of the architecture is enhanced, when a number of domains in a hierarchy are used. In comparison to a single-domain configuration, in a multiple-domain configuration the amount of information about in-domain objects, stored and processed at each domain, is reduced. Queries for location of documents, relevant to a particular consistency rule, are distributed across multiple domain agents. If an administrator finds that the domain agent is operating at maximum load, re-destribution of some of in-domain resources to other domains will improve operating efficiency.

Availability of consistency rules and their applicability can be controlled at each individual domain. An ability to control the span of consistency rules by producing different rule sets for different domains and to set up rule applicability policies for each domain gives system administrators additional flexibility.

10.1.6 Disconnected Operation

All domains support disconnected operation mode, which allows in-domain consistency checks to complete, even if connection to any outside domains is not available. This feature is useful for developers working from home and connecting to the organisation's networks via a modem, or for geographically distributed teams of developers, when faults may temporarily put the connections out of order.

The prototype implementation of the software agent architecture takes a simple approach to disconnected operation. In the "disconnected" mode, any consistency checks, which may involve relevant documents from other domains, complete within the limits of the domain. Any consistency checks, started after connection is re-established, will span across the whole document universe. In the area of disconnected operation, future work on the architecture will be focused on reusing results of completed in-domain checks and completing these checks across external domains after connection becomes available again. It is envisaged that persistency feature of the mobile consistency checking agents can be effectively used to achieve such "delayed" execution of inter-domain checks while in the disconnected mode.

10.1.7 Support for Transactions

Incremental checking approach, coupled with event-orientation of the software agent architecture, enabled us to provide *elementary* support for transactions in the distributed consistency management system. With incremental checking, it becomes possible to relate individual change or a set of changes on a document to the current state of all related documents by means of generated consistency links. In the context of consistency checking, a transaction relates a document modification with the generated linkset. Each transaction starts when a modification is introduced, and finishes when consistency links are saved locally on the host where modified document is located.

A number of such transaction records are created throughout the life cycle of each document. A mechanism is already in place in the architecture prototype, which allows a user to browse the history of transactions within a particular domain. Document changes for each transaction are accessible via a resource interface agent, which keeps history of document versions, consistency links are stored at the document's location, and event history, maintained by domain agent at each domain, keeps track of starting and ending time for each transaction.

The collected information allows a user not only to store transactional information, but also to construct a tool, which will enable a user to browse through the state of generated consistency links and roll-back one or more transactions in order to return to the desired state of the system. This is a novel feature - one of contributions, which the software agent architecture makes to the continuous development and evolution of consistency checking apparatus.

At this time, the software agent architecture for distributed consistency checking provides only initial level of support for transactions. A number of problems need further investigation, including detection of relevance and dependency between different transactions. It is important, for instance, to have an ability to transparently update the state of one transaction in the history, taking into account the changes that occurred to related documents since that transaction had taken place. At the same time, transaction-awareness of the proposed architecture is positively a useful addition to the core architectural features, demanded in the functional requirements.

10.1.8 Balance of Requirements

A balance of functional requirements is met in the proposed architecture. For this evaluation, we revisit the targets – functional requirements, that each of the architectural components was aimed at (Chapter 6).

Functional requirements on location of consistency rules and applicability policies (Chapter 3, 3.3), requirement for document monitoring (3.4) and for timing of consistency checks (3.5) are satisfied by the resource interface agent. Use of an interface component not solely for access to the underlying resource, but for identification of changes and facilitation of communication between mobile checking components and resources is a novel contribution of the software agent architecture.

The architecture uses a mobile component for collecting data from participating documents across numerous sites (requirement 3.8). This component is called "mobile checking agent", for the autonomy and inter-cooperation with other static components.

The novelty of the proposed distributed checking approach is in ability of the components to engage in communication and co-operative activities. In identification and resolution of redundant consistency checks, mobile checker agents engage in exchange of information previously collected from distributed locations. A scheme for replication based on cloning and migration has been proposed; such scheme is a novel contribution to the software agent domain. The scheme is particularly useful for applications, where information, collected from distributed locations, is analysed and certain actions at these locations must be carried out depending on a result of the analysis.

Deployment of mobile components for execution of distributed consistency checks is novel in itself. Joint exploitation of mobility and locality of consistency checks is a novel contribution of this thesis to the software engineering domain.

In comparison with the software agent architecture, the predecessor framework for centralised consistency checking [Nentwich, et al. 2000b] cannot satisfy functional requirements for distributed document monitoring (Chapter 3, 3.4) and processing locality (3.8), without use of additional components - distributed watchdogs and link generators. Requirement for timing of consistency checks (3.5) can be only partially met by a centralised checker, which must be launched by the user, who thereby personally chooses the timing of checks. In the proposed architecture, automation of this process is readily provided.

The functional requirements (Chapter 3) express the principles, on which the proposed software architecture is constructed. While these requirements cannot be met by a centralised architecture for consistency checking, the proposed distributed software agent architecture satisfies them and, in addition, provides a number of useful features which facilitate distributed cooperative work. The most important of such qualitative features have been discussed in this qualitative evaluation section. Prototype evaluation scenarios (Chapter 8, Scenarios I-III) demonstrate these features in operation. Comparison of performance characteristics of the basic centralised checking architecture and the distributed one is carried out in the following section.

10.2 Quantitative Performance Evaluation

This section, performance evaluation of the implementation prototype, consists of three parts. The first part compares the performance of an incremental check with that of an exhaustive check. Here we aim to determine a particular configuration, in which usage of one or the other would be more efficient from the performance viewpoint.

In the second part, we contrast a centralised consistency check, where processing of consistency rules and generation of links occurs on a single host, with a distributed consistency check, where participating documents are processed locally at the hosts where they are stored. Here we determine

document distribution patterns, which can be used to improve efficiency of distributed consistency checks. The evaluation involves variation in the configuration parameters, such as the number of hosts in the consistency check, the number of documents (ultimately the number of document elements) at each host, and the number of software agents, concurrently carrying out distributed consistency checks of the same consistency rule.

In the final part of the evaluation, we compare centralised exhaustive checking of distributed documents with incremental distributed checking of the same documents. Our evaluation and the collected performance data aim to highlight the different ways, in which these different consistency checking methodologies are intended to be used.

The UML model that we use for performance evaluation of our prototype is the BMS model [Rational 1998b]. This model serves as a development example for the Rational Rose environment, and we therefore consider it representative of the typical UML development project. BMS is a relatively large UML model, containing more than 10700 elements.

In all performance evaluation studies of this chapter, we use the same set of XMI files, which has been derived from the BMS by the UMLXMI utility [Appendix F, F.5]. The set consists of 362 XMI files in total, more than 5 megabytes in size altogether. Having a model represented as a set of files rather than one large file was essential for evaluation of configurations, where the model is distributed across a number of hosts. The same document set is also used for the first part of performance evaluation at a single host.

All performance evaluations were conducted on a notebook with the Intel Pentium II processor operating at 266 MHz, equipped with 128 megabytes of RAM memory. The testing was carried out under OS Windows 2000, running Java Runtime Environment version 1.3.0-C. Evaluation of distributed checks was made on a number of machines of a similar configuration, connected via a 100 Mbit Ethernet LAN.

10.2.1 Exhaustive Consistency Check vs. Incremental Check

Exhaustive consistency check of all 34 UML well-formedness consistency rules [Appendix A] on the document set that we used took 737 seconds. 63 seconds were spent on parsing of XMI files, and total of 674 seconds on execution of consistency rules (Fig. 10.1).

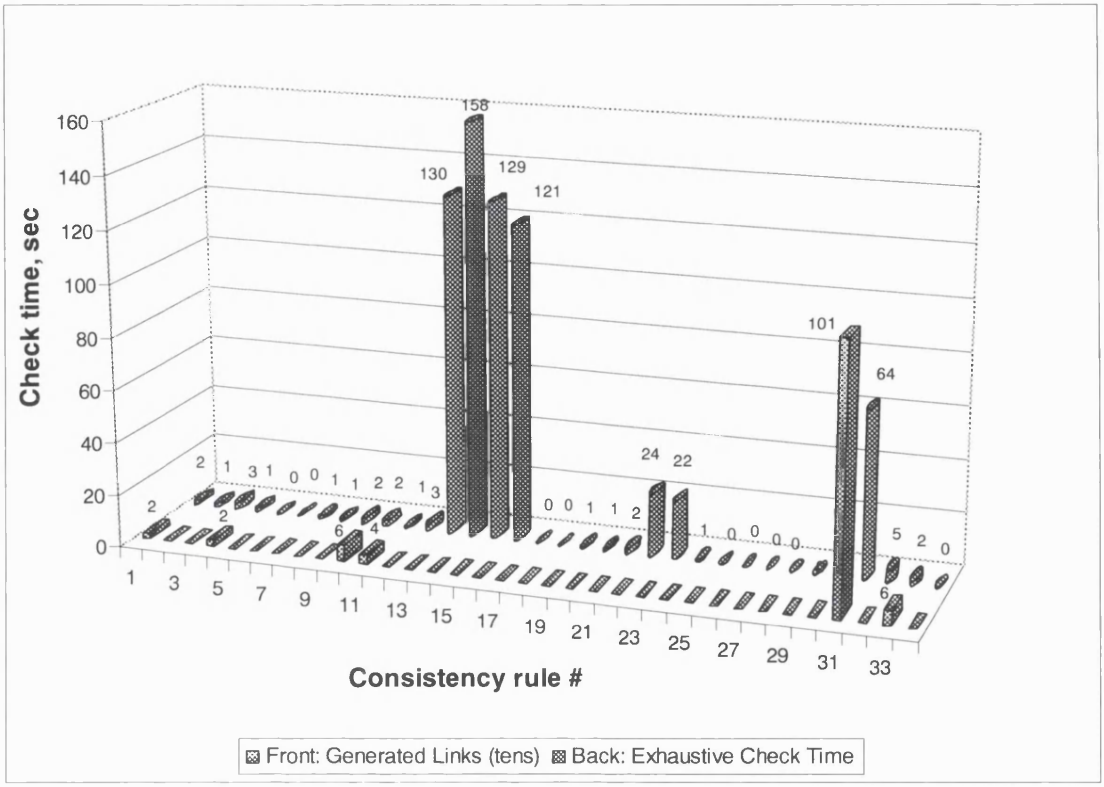


Fig. 10.1. Timings of exhaustive checks of UML well-formedness consistency rules.

It is worth noting, that 5 megabytes of XMI files were represented by over 50 megabytes of DOM trees in memory during consistency checking. During an exhaustive check, DOM trees of all documents remain in memory throughout the whole execution of the check. The test machine had enough RAM to complete the exhaustive check of this model, although a consistency check of a larger model [Rational 1998b] did not complete within the reasonable time limits (30 minutes).

Carrying out an exhaustive check of a relatively large BMS model required all the memory available to the operating system on the test machine, and in addition almost 100% usage of the CPU for over 12 minutes. Therefore, the exhaustive consistency check took the complete control of the test machine for the duration of the check and made it difficult for a user to continue any productive work during this time.

Analysis of durations of consistency checks for individual rules (last row, Fig. 10.1) and numbers of generated links (front row) for each of checked consistency rules (Fig.10.1) allows us to conclude, that the most time during the consistency check *of this particular model* was spent on execution of consistency rules, which did not produce any inconsistent links. Out of the total time spent (674 sec), only 10% (71 sec) was spent on execution of consistency rules, which produced the total of 1204 inconsistent links. The four longest-running consistency rules (14,15,16 and 17 on Fig. 10.1), which check the well-formedness relations UML classifiers (rule identifiers cs3,cs4,cs5,cs6, Appendix A, A6), produced 24 consistent, and no inconsistent links. In the consecutive checks after the initial check has been completed, the consistent links are normally not very useful to the developer, as they do not identify any problems with the UML model. The further development of the UML model is most likely

to be concerned with correction of the identified inconsistencies, rather than with further investigation of well-formedness of classifiers. Such development will necessarily result in a number of successive consistency checks, following the incremental changes – corrections. In every following exhaustive check of the model, unnecessary checking of the consistency rules cs3-cs6, which are always executed by the exhaustive checker, demands 90% of the total resources, spent on the check, and produces the result of little value to the developer in the incremental development process. Use of incremental checking allows a user to avoid execution of the rules, which are not relevant to the incremental document changes.

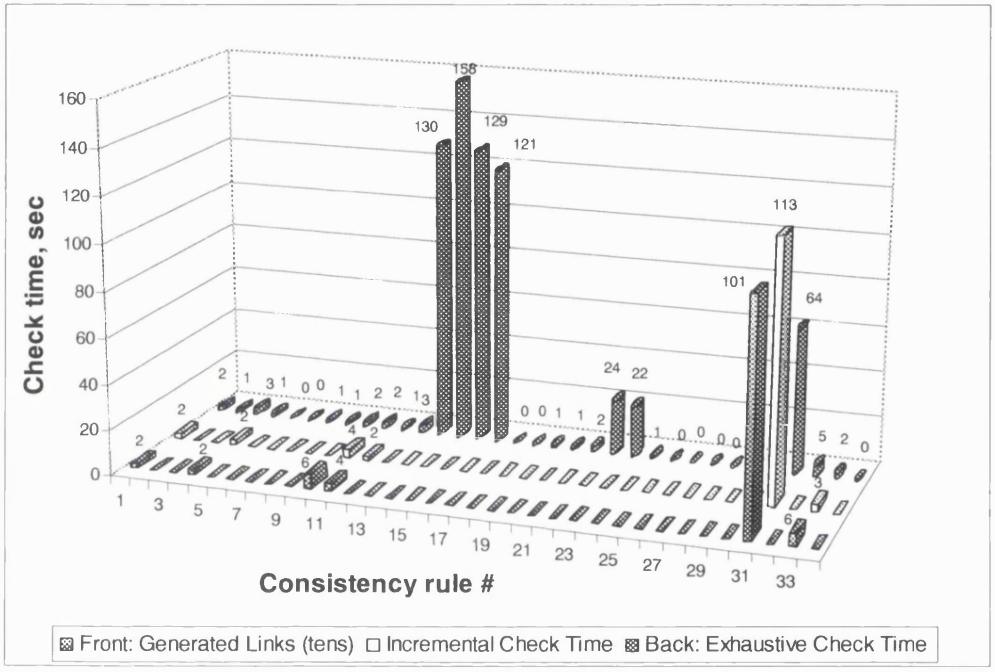


Fig. 10.2. Check times for incremental and exhaustive checks.

Comparison results for execution timings of incremental and exhaustive checks are shown in Fig. 10.2. For each consistency rule, the incremental check produced the same consistent and inconsistent links as the exhaustive check. The timings of incremental checks exceed the execution timings of individual rules in the exhaustive checks by a margin. This margin is used to compute the tree-wise difference (TreeDiff) between the updated XML document and its backup version, and to use this difference for selection of relevant consistency rules. In the experiment, the margin was within 0.5 to 3 seconds for checks of different rules, where the longest margin is comparable with the execution time for some of the individual rules.

A difference between the timing of an incremental check of a certain rule and the execution time of that rule in an exhaustive check depends on a number of parameters. Efficiency of an incremental check improves in a configuration with multiple smaller XMI files as compared to the configuration with one large model. Tree-wise differencing comparisons between documents, used in an incremental check, bear a polynomial performance degradation with the increase in the length of the files.

For large models, consisting of a number of files, the incremental check achieves better performance than the exhaustive counterpart because of more efficient use of memory resources. During the incremental check, only XML DOM trees of those documents, which are relevant to the rule being checked reside in the memory; whereas the exhaustive check must keep all DOM trees of all documents in memory until the check has been completed.

For a user, interested in a way that a particular document change will affect consistency relations of a document with the rest of document universe, an incremental check will almost always be more useful than an exhaustive consistency check. However, if a number of modifications need to be processed in "batch", exhaustive checking may produce results faster, because margins of incremental checks in this case add up. The balance point between the two is determined by differencing the number and the execution times of the consistency rules that *need* to be checked in the batch and the respective parameters for consistency rules that are *not relevant* to any changes represented in the batch.

Obviously, when each and every consistency rule needs to be checked, the exhaustive checking is preferable to the incremental approach. Likewise, in a situation, where consistency rules take approximately equal time to complete, and *almost all* consistency rules are relevant to changes in the batch, exhaustive checking will often be preferable. However, in a situation where one or more consistency rules in the configuration takes considerably longer than others to execute, use of incremental checking becomes more appropriate from the performance standpoint.

In our example, any batches of changes, which do *not* include relevance to one or more of the longest-running consistency rules (rule identifiers cs3, cs4, cs5, cs6), will be processed *more efficiently* with the *incremental* checking. When all four of the mentioned rules need to be checked, incremental checking will still produce results faster, but with a smaller performance margin. This difference in performance will be further reduced if all four longest-executing rules need to be checked in addition to one or more of remaining consistency rules.

10.2.2 Distributed Check vs. Centralised Check of Distributed Documents

Distributed consistency checks are those when documents are processed locally at the hosts where they are stored. In addition to the parsing and rule execution time, discussed in the sub-section above, the timing of a distributed check includes the time for migration between distributed hosts and a communication overhead between the architectural components, involved in the distributed check. A mathematical model, which aims to describe timings of distributed checks for different configurations of a distributed system, is provided in Appendix E.

Some of the most influential configuration parameters, on which the performance of a distributed check depends, are the following:

I. Number of hosts for migration, between which agent migration is required.

II. Number of documents, participating in the check, number of elements in these documents.

III. Number of distributed agents, concurrently performing consistency checks of the consistency rule.

In order to determine the trends for better document distribution patterns, we consider impact of all these parameters on performance of distributed checks carried out by the implementation prototype.

10.2.2.1 Number of hosts of distribution

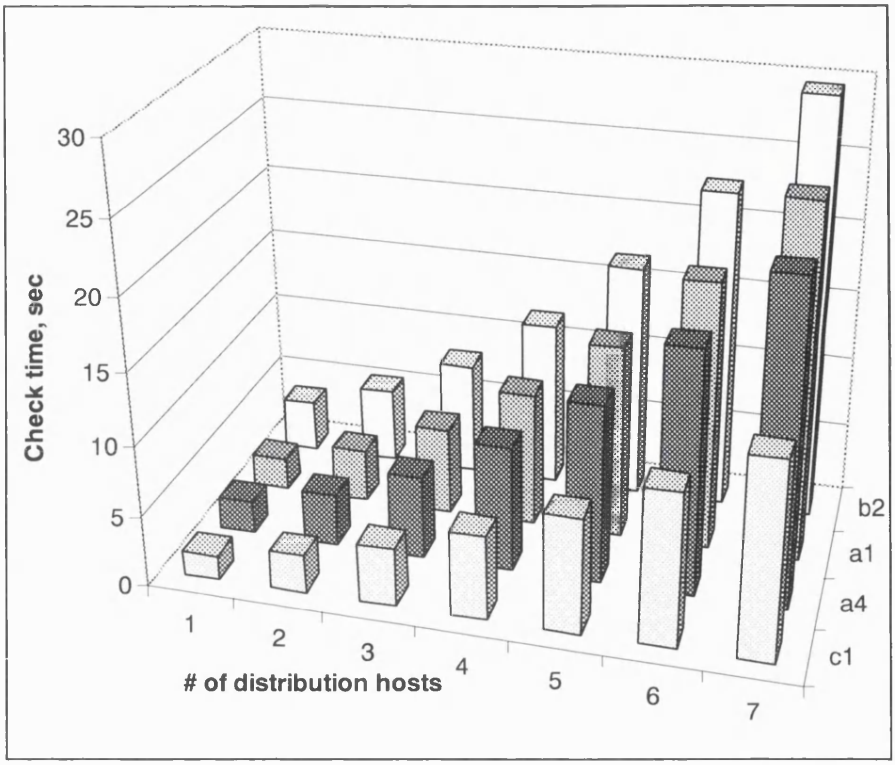
The testing configuration used the file set, produced by UMLXMI utility from the BMS model. Tests were carried out in distribution configuration from 1 to 7 hosts, where in the configuration of 1-3 hosts all hosts belonged to the same domain, and in configuration of 4-7 hosts there were two domains of 3 hosts and 1...4 hosts, respectively. In all cases of distributed checks, our experiment has not indicated a substantial difference in performance of consistency checks between a configuration, when all documents participate in a single domain, and a multiple-domain configuration.

Four consistency rules were executed in all configurations, UML well-formedness rules relating to associations (a1 and a4), behavioural features (b2) and classes (c1). Each of these four rules is applicable to the BMS model, and produces a number of consistency links (Fig. 10.1).

Each file in the set of XMI files, distributed to different hosts in the experiment, represents a sub-tree of an element of the BMS model. Obviously, the most of UML well-formedness consistency rules, used for evaluation of the software agent architecture in this thesis, relate less then four different UML model elements. Therefore, rarely would more than four distributed hosts be visited for checking of a consistency rule. However, in order to evaluate different document distribution patterns, in the experiment link generation was *deferred* until all hosts in each configuration (1 to 7 hosts) were visited.

In this section, we benchmark performance of a distributed consistency check, carried out by *one* mobile checking agent family. Deployment of *multiple* collaborating mobile agents, running concurrently at different hosts, is considered below [Section 10.2.2.3].

Fig. 10.3 demonstrates the durations of consistency checks of different rules for different system configurations, depending on the number of distribution hosts. The height of each bar of the graph corresponds to the total time it took to carry out the check when documents were distributed across the corresponding number of hosts.



Legend: a1, a4 – UML associations well-formedness rules [Appendix A, A.1]
b2 – behavioural feature consistency rule [Appendix A, A4]
c1 – class consistency rules [Appendix A, A.5]

Fig. 10.3a. Check timings of distributed consistency checks by number of hosts.

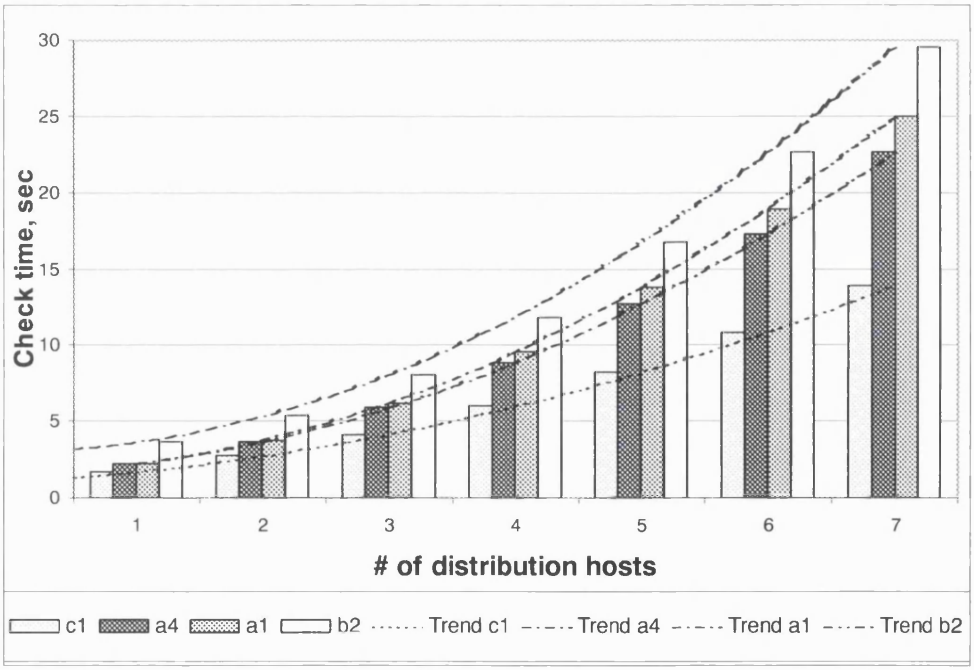


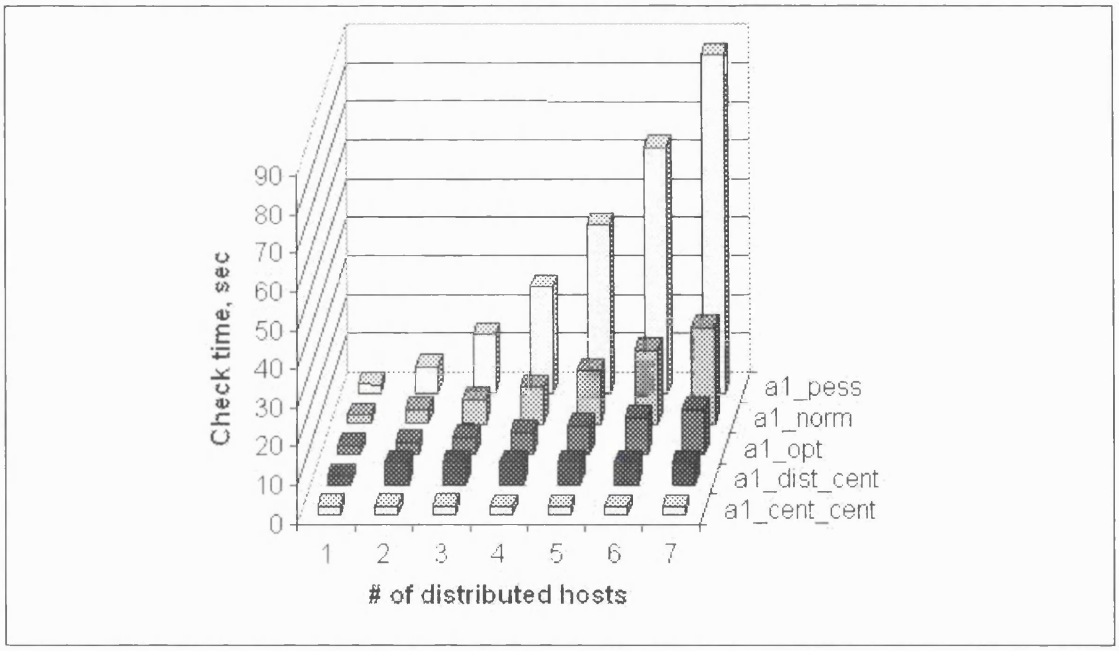
Fig. 10.3b. Check timings - plain graph with trend lines.

Increase in check times (Fig. 10.3) with the growth of a number of hosts is attributed to two primary factors. Serialisation and de-serialisation of nodesets, which a mobile agent collects from the documents at each host, results in growth of time spent on migration from the first agent's migration in the consistency check to the last migration. The need to carry around already collected nodesets at each migration increases migration durations for the multi-host consistency checks (Fig. 10.3). The second factor is the size of the agent code and the state of the code that needs to be transmitted between the hosts. In the implementation prototype, agent migration results in transfer of the *state* of the code only, given that agent classes are already installed at the destination, and the mobile agent framework overhead for migration is approximately 0.6 seconds on the test machine.

A relatively even distribution across the hosts of XMI files of each category (associations, behavioural features, classes) was used in the experiment. Thus, approximately equal size of nodesets was selected by mobile checking agents at each host. In sequential checking of hosts, this resulted in relatively low initial value, and linear increase of the timing increment, related to serialisation of collected data.

Performance-wise, the most *inefficient* distribution of documents across numerous hosts would obviously consist of the *majority* of files (and therefore, selected nodesets) located at the *first host* in the agent's itinerary, and only one file, resulting in a small selected nodeset, at each of the following visited hosts. In this case, almost all information for consistency check needs to be carried through all hosts in the itinerary. This pessimistic scenario results in a large timing increment for the first agent migration. This timing increment will slowly increase and remain large for each subsequent migration of the checking agent. In the end, distributed check performance results are unacceptably poor (Fig. 10.4, "a1_pess").

In the *optimistic* case of the most efficient distribution of documents, the *largest* portion of the collected *nodeset* is generated from documents, located at the *last visited* host. A much smaller timing increment at each migration in this case results in distributed check performance, which is much closer to that of a centralised consistency check (Fig. 10.4, "a1_opt" and "a1_cent"). The fastest checking result is expectedly demonstrated in a centralised check where all documents are accessible locally and no distribution is involved (Fig. 10.4, "a1_cent_cent").



Legend: a1_cent_cent – consistency checking of documents locally at the same host, no distribution overhead is involved;
a1_dist_cent – centralised check of distributed documents, transferred to the centralised location for checking;
a1_opt – distributed check with the software agent architecture – recommended distribution configuration;
a1_norm – distributed check, random distribution;
a1_pess – distributed check, worst-case distribution scenario.

Fig. 10.4. Performance comparison of distributed and centralised checks of rule a1.

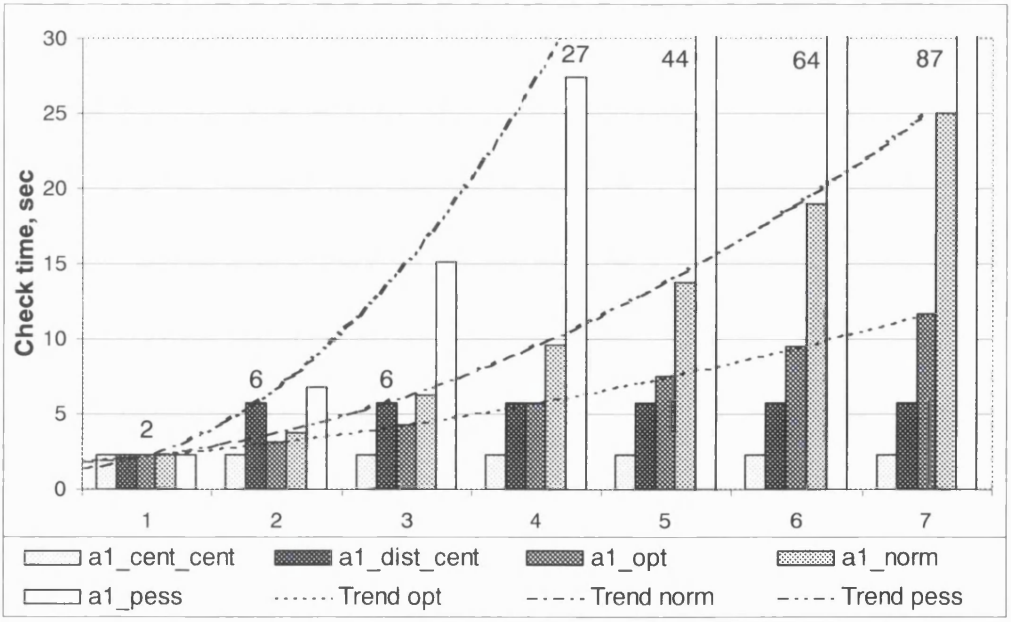


Fig. 10.4b. Performance comparison: plain graph with trend lines.

Fig. 10.4 contrasts performance of a distributed check in the experimental, optimistic and pessimistic configurations (rows 3, 4, 5 on the graph) with performance of a centralised check and performance of a consistency checking web-service, where centralised processing of documents is deployed (rows 1 and 2, respectively). The experimental data, shown on the graph for all configurations of the distributed check, takes a parabolic shape, which corresponds to the findings of a quantitative model [Appendix E]. While the scale on the graph in Fig. 10.4 does not allow us to see a deviation of the experimental data from the parabolic theoretical curve, our measured deviation constituted 8-11%. Deviation between experimental timings for association well-formedness rules a1 and a4 were within 3-5%.

From the analysis of checking performance in different document distribution configurations, we can conclude that certain optimisation of agent itineraries can improve efficiency of distributed multiple-host sequential checks. Ultimately, such optimisation would require guessing or computing the size of nodesets, which are to be selected at each host in the itinerary, before execution of the itinerary by a mobile checking agent actually takes place. In practice, nodeset sizes can be estimated, given that number of elements in each of the documents from agent itinerary is known to the agent before migration. In such a case, each itinerary can be sorted, so that hosts with larger size of estimated collected nodesets are processed *last* in a sequential check. Such optimisation will bring the total check times closer to the result of the optimistic scenario.

It should be noted, however, that not for all consistency rules the size of selected nodesets is proportional to the size of an XML document. The implementation prototype, therefore, does not perform agent itinerary optimisation in accordance with any "prediction" scheme. Future work needs to be carried out to establish more reliable criteria for estimation of the size of selected nodesets.

Experimental prototype has demonstrated, that the major part of migration time is spent on serialization and de-cerialization of nodesets. Therefore, future work on optimisation of distributed consistency checking with respect to timings of migration between different hosts will concentrate on nodeset serialisation process. The current prototype deploys standard Java generic serialization mechanism, which we intend to optimise by implementing nodeset compression for improvement in performance of the agent migration.

As a follow-up to the experiment, we have compared the nodeset serialization times on experimental machines (Intel Pentium II 266 MHz processor) with respective times on faster workstations (Intel Pentium III 850 and 650 MHz processors, equal amount of RAM memory), which at the time of writing were already considered lower-range PC configurations. We detected a speed gain in nodeset serialization of a factor of 2: such performance improvement results in slashing the times of migration between hosts almost by half.

10.2.2.2 Number of document elements to be checked

With the increase of number of elements, participating in a consistency check, check timings grow in a near-linear fashion (Figs. 10.5, 10.6, 10.7).

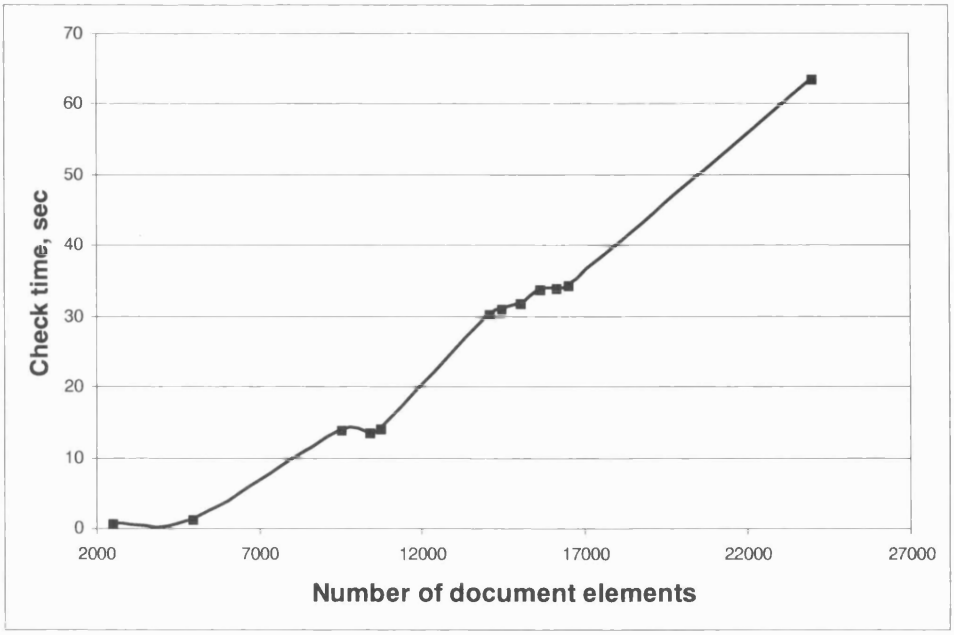


Fig. 10.5. Execution of consistency rule: well-formedness of Namespace.

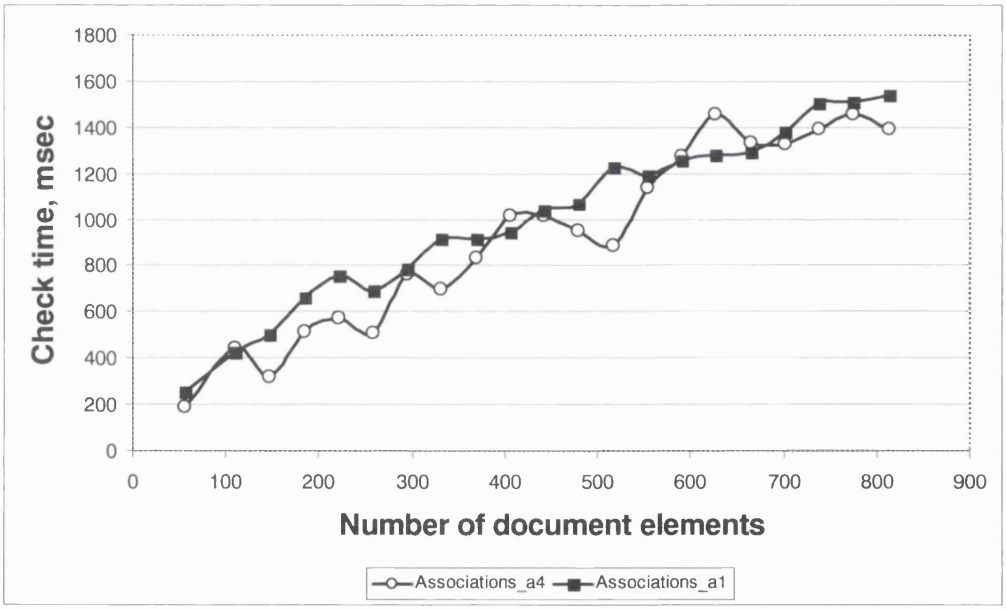


Fig. 10.6. Execution of consistency rules: well-formedness of Associations.

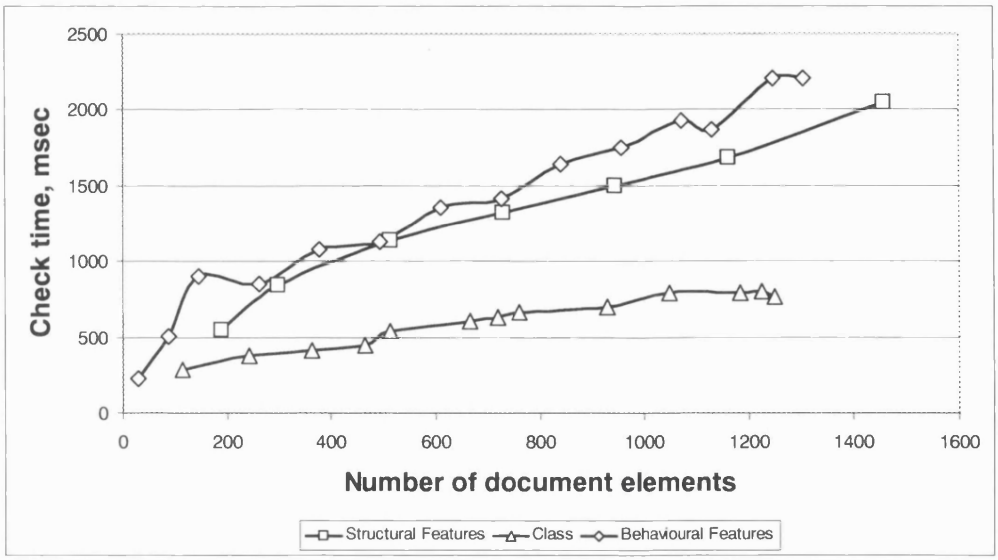


Fig. 10.7. Execution of consistency rules: Classes and Behavioural Features.

The experimental data concerns the performance of the XLinkit exhaustive checker, which was tested on the individual consistency rules. The data constitutes the minimal durations of time spent on execution of rule operators and link generation with respect to the number of elements in the checked documents. The benchmarks do not take into account document parsing time and any distribution overhead. All the other performance graphs in this quantitative performance evaluation include the distribution overhead, and its effect is specifically addressed in Section 10.2.2.1, depending on the number of distribution hosts.

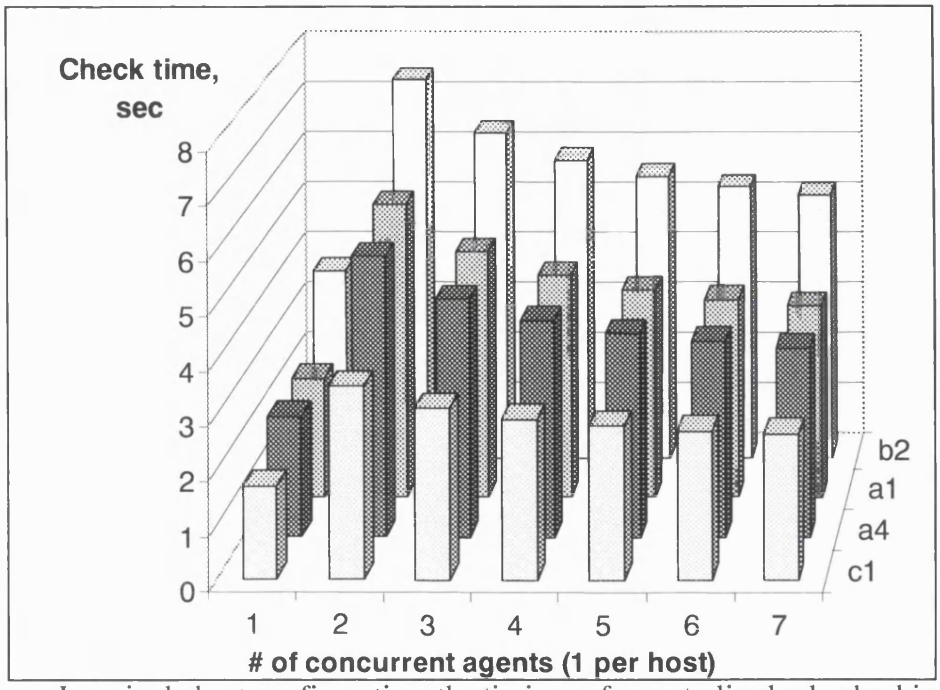
10.2.2.3 Number of mobile checking agents working concurrently

Concurrent checking with multiple mobile agents (discussed in the Scenario III in Chapter 8) divides the complete agent itinerary into sub-itineraries between a number of mobile agents, and each agent processes documents in the sub-itineraries.

Performance of the most *pessimistic* scenario of a concurrent collaborative check is the same as in a single-agent scenario [10.2.2.1], where one mobile agent migrates through most of the hosts in the itinerary. This agent then takes the longest to complete nodeset collection, and its performance alone determines the timing of concurrent consistency check.

In the most *optimistic* scenario of a collaborative multi-agent check, each mobile agent performs collection of nodesets at one host and sends the collected nodesets to a selected location, where link generation takes place. The maximum time spent by any agent on selecting nodesets and the time of transferring a collected nodeset of a maximum size determine the timing of the complete consistency check in this case.

In the optimistic scenario mobile agents migrate only once and collected nodesets are transmitted only once. In the pessimistic scenario, nodesets migrate with the agent each time a different host needs to be visited. Improvement in the timing of an optimistic scenario in comparison to the pessimistic one is due to the difference in two approaches.



Legend: In a single host configuration, the timings of a centralised rule checking are provided for comparison.
a1, a4 – UML associations well-formedness rules [Appendix A, A.1]
b2 – behavioural feature consistency rule [Appendix A, A4]
c1 – class consistency rules [Appendix A, A.5]

Fig. 10.8. Performance of concurrent multi-agent checks.

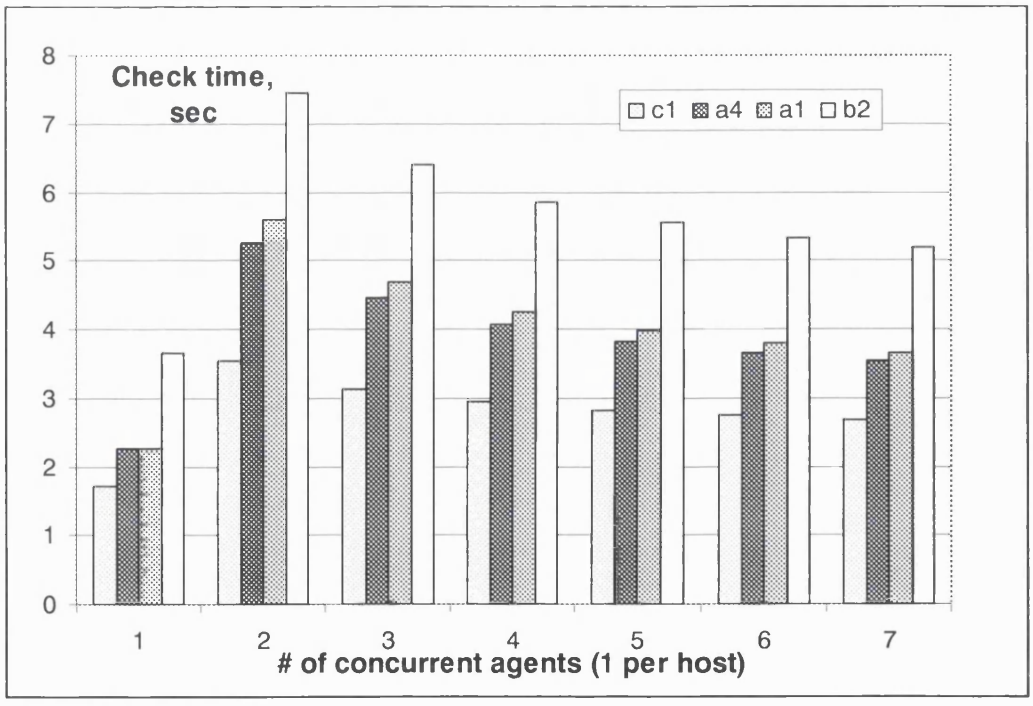


Fig. 10.8b. Performance of concurrent multi-agent checks.

Fig 10.8 demonstrates the performance gains from deployment of multiple agents, concurrently collecting nodesets from distributed hosts. The first column shows the timing of a centralised check of each respective consistency rule; distributed multi-agent check timings gradually decrease as the number of checking agents grows. The most substantial timing improvements are demonstrated for consistency rules, which select and transport larger nodesets during the check.

10.2.3 Centralised Exhaustive Consistency Check vs. Distributed Incremental Check

Fig. 10.9 summarises the experiments above and demonstrates the performance difference of the best multi-agent distributed check, worst single-agent check and centralised checking of distributed documents (XLinkit web service). The graph clearly indicates the performance advantage that multi-agent checking provides. A sequential single-agent check (last row, Fig. 10.9) was not "on par" with any of the faster alternatives.

In the example we followed, multi-agent distributed check achieves similar performance to that of a centralised checker of distributed documents. However, this result results from the "best case scenario", when each consistency rule was executed individually, and all resources of distributed hosts were made available to the distributed check. In operational conditions, however, concurrent execution of different rules on the same host is expected, and network bandwidth will then be shared as well between multiple agent families. Thus, multi-agent performance in a particular configuration may vary from the "best case scenario".

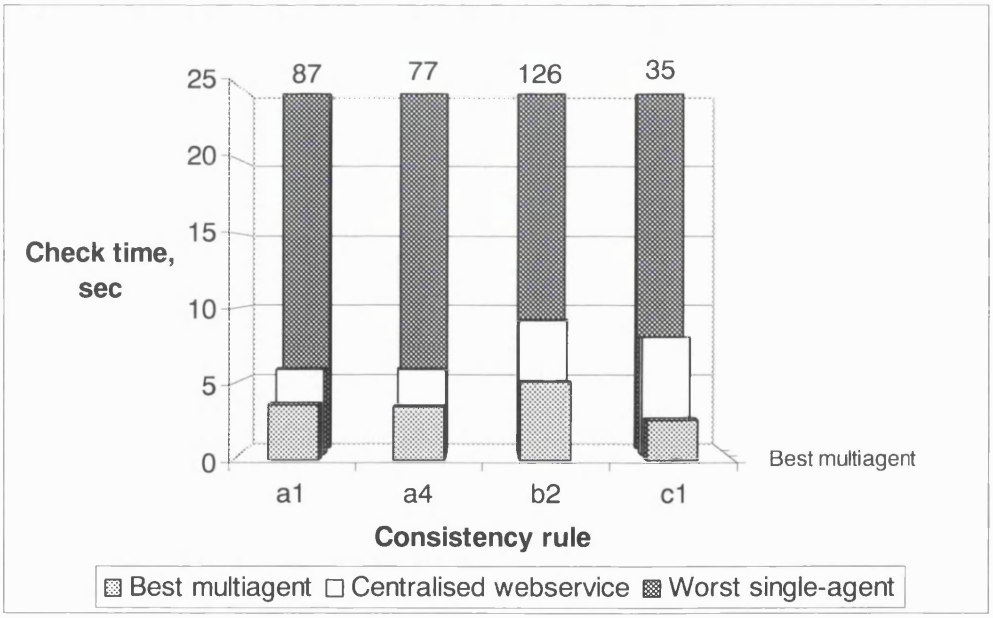


Fig. 10.9. Performance of best and worst distributed checking approaches compared with centralised checking of distributed documents.

10.3 Summary

In this chapter, we have summarised important qualitative features of the software agent architecture for distributed consistency checking. Elegance of the architecture stems from clarity of roles, assigned to architectural components. Although the architecture shares component namespace, defined in the software agent community, operation of each component and nature of inter-component interactions are not trivial. In addition to manageability of document updates and distribution of consistency rules, we have discussed architecture flexibility and support for dynamic reconfiguration, and clustering of resources into domains, organised into a domain hierarchy. We also focused on the disconnected operation capability and initial support for consistency checking transactions that the architecture makes available to its users - software engineers. We further re-iterated the architecture's achievement of the demanded requirements, which correspond to the roles of each architectural component, and pointed out novelty in a number of contributions of the proposed software agent architecture.

In the quantitative performance evaluation, we contrasted an incremental and exhaustive consistency checks on a number of selected consistency rules from the application domain of UML well-formedness checking. The experimental data supports our argument that incremental checks are most suitable for day-to-day, incremental project development. In the representative UML model construction example that we have considered, an incremental check of the elements, which have been detected inconsistent and are thus likely to be extensively modified, measured 10% of the total checking time spent on the exhaustive check. In a larger model example, an exhaustive check did not complete within 30 minutes of standalone execution on the test machine, in which case use of incremental checking was mandatory.

We further contrasted a distributed and a traditional centralised check of distributed documents, and a performance impact of configuration parameters, such as the number of distribution hosts, document size and the number of concurrently executed mobile agents. We have supported our argument on efficiency of concurrent distributed checking by the experimental data, and suggested to optimise the itineraries of individual agents with respect to size of collected nodesets and to shorten the itineraries in favour of the increase in the number of concurrently operating agents.

Finally, we contrasted a centralised exhaustive and an incremental distributed check and established a favourable document distribution pattern, which results in a performance improvement of the incremental check. The experimental data has proved that performance advantage of the distributed software architecture over the traditional approach can be achieved in the recommended configuration.

Chapter 11 Conclusions and Future Work

11.1 Conclusions

This thesis has advocated an approach to carrying out distributed consistency checks, different from the traditional centralised processing or client-server architectures. We have illustrated the need for a distributed framework for consistency checking, in which checks occur locally at the hosts, where documents are located, and the proposed software agent architecture meets the requirements for such framework. The technology used to realise the work in this thesis is based around a component architecture, in which various functions draw from concepts of software agency and mobility. Below we briefly outline some of important results, presented in the thesis.

11.1.1 The Software Agent Architecture for Distributed Consistency Checking

We have developed a distributed architecture for consistency checking. An important characteristic of the architecture, that distinguishes it from traditional approaches, is that consistency checks of distributed documents occur locally at the hosts, where the documents are located. Static and mobile software agents are deployed to cooperatively carry out distributed consistency checks.

The architecture provides flexibility in configuration, and integrates with existing development environments. Transparency of distributed checks for the user and simplicity of the architecture were some of the key factors in design and implementation of the architecture.

One of the advantages of the proposed approach is in improved security features with respect to the traditional approach: in the software agent architecture, complete documents are never transmitted across the network. In addition, user-level, document-level and mobile agent migration-level security is supported.

11.1.2 Architecture Model

We have developed an architecture model based on state-transition diagrams of architectural components. Basic functions of component algorithms, event handling and inter-component communication are outlined in the state transition diagrams. The model provides initial evaluation of the architecture: the model development environment allowed us to "execute" the state transition diagrams, replicate the components and estimate the effect of different configurations on predicted performance of a distributed consistency check.

11.1.3 Incremental Checking

We have established applicability of the incrementality concept to the underlying consistency checking framework, on which the architecture is constructed. Incremental consistency checks extend the functionality of the traditional consistency management framework, by establishing relations between individual changes in one or more documents and the current state of all the documents in the document universe. We argue, that incremental checks are useful at all development stages, and complement the ability of the consistency framework to generate global consistency views by performing exhaustive checks.

Ability to carry out individual incremental checks, complemented by the approach to local checking of distributed documents, form the base for the proposed software agent architecture. The architecture concept, where an incremental consistency check serves as a response to an event - document change, allowed us to simplify architecture design by supporting event-orientation of all components. Event-orientation, in turn, enables the architecture to provide initial support for transactions by maintaining histories of document changes, their consistency status, and making rollback available to the users.

In our contribution, we have proposed a general algorithm for incremental consistency checking for the consistency framework, underlying the software agent architecture. We also provide an implementation of this algorithm, which is integrated in the working architecture prototype. Evaluation of the proposed incremental checking approach is carried out within qualitative and quantitative evaluations of the architecture. The quantitative performance evaluation results suggest, that in certain conditions, more often than exhaustive checking, distributed incremental checks in the proposed software agent architecture can be used for providing near "real-time" feedback to the developer throughout the development process. We value this significant result as a contribution of this thesis, complementary to the primary contribution – the software agent architecture for distributed consistency checking.

11.1.4 Validation of the Architecture

In order to establish further validity of the approach, we have developed an implementation prototype. Operation of the prototype is validated on the three scenarios, which correspond to stages of a real software development project. We demonstrate and explain features of the architecture on particular examples of the scenarios in a single-domain, multi-host and multiple-domain configurations. In addition to scenarios, the internal structure of the prototype implementation is explained through events and actions, which these events trigger in the architectural components.

We complete the evaluation by demonstrating how the proposed architecture conforms to the specified functional requirements. We also carry out performance benchmarking, by comparing centralised and distributed consistency checks, single-host incremental and exhaustive consistency checks, and finally, centralised exhaustive and distributed incremental consistency checks of a set of

consistency rules. As a conclusion of performance evaluation, we give recommendations on optimal configuration of document distributions and confirm performance advantages of multi-agent concurrent collaborative checks.

Performance evaluation results are extended in the quantitative performance model of the software agent architecture. The latter determines the relations between measurable configuration parameters (i.e., document parsing time, number of concurrent software agent, etc.). These relations can be used by to predict performance of a particular architecture configuration before the actual system roll-out, and can advise on deployment of either traditional centralised checking of distributed documents or distributed multi-agent checking, for each particular configuration.

11.2 Open Questions and Future Work

11.2.1 Consistency Checking Framework

We aim to carry out research in specifying and carrying out inconsistency resolution actions in the current framework, following the generation of inconsistent links. An approach to comprehensive inconsistency resolution is a significantly complex area of research in itself. As an initial step, investigation of propagation of element values over inconsistent links is under development. In this approach, a developer points out the element, which is believed to cause inconsistency, and resolution is carried out, whereby the value of this element is changed, satisfying the consistency relation and making an inconsistent link consistent.

Future work on the software agent architecture, which builds on the consistency checking framework, will then investigate whether conflict resolution role can be delegated to existing mobile checking components, or a separate mechanism needs to be created. It is clear, however, that efficient inconsistency resolution will make use of the distributed family of mobile agents, running at the hosts, where inconsistent documents are located. It is then a question whether generation of possible resolution actions can be carried out throughout the duration of the distributed check, or will necessarily have to be postponed until the final result of the check is known. Evaluation of individual operators, comprising a consistency rule, throughout distributed nodeset collection may make it possible to generate initial suggestions on resolution actions during the check, rather than after its completion. This approach can then facilitate resolution actions in the software agent architecture.

Another development area for the consistency checking framework is in specification of rule dependencies. Support for conditional checking of consistency rules has been suggested in the research group for some time already. From the standpoint of the software agent architecture for consistency checking, it is suggested that rule applicability policies can be defined in such a way, that multiple rules are checked at the same time. The question still remains on how to decidably determine inter-relation of several consistency rules, so that applicability policies can be generated and updated automatically.

11.2.2 Integration of the Software Agent Architecture

One interesting area for future development of the software agent architecture is tighter integration of the architecture, its prototype with existing software development environments and workflow tools. While currently consistency links serve as a feedback that the architecture provides to a developer, the usability of the prototype would be extended if links could be interpreted within the native environment of the tool, which the developer is using.

When incremental checking technology matures and its performance is further improved, an ability to provide instant feedback of consistency status *during* document authoring should prove to be extremely effective in the development process. Modern development environments enable real-time syntactic checks, where, for instance, the syntax in a program module is checked in real time as the developer is authoring the program. The software agent architecture already supports the automated execution of consistency checks, and the performance evaluation suggests, that partly due to the locality of document access, in certain conditions checking performance is sufficient to provide near-realtime incremental checks. Thus, it would be desirable to further investigate how the software agent architecture can be used for real-time consistency checking and for provision of feedback on results of these checks in a useful form, within the user's development environment.

11.2.3 Distributed Software Agent Architecture

The software agent architecture makes use of replication of consistency rules to the distributed hosts, where the documents, relevant to these rules, are located. At the same time, a central point of software agent coordination at each domain – the domain agent – maintains information on applicability of different rules to distributed documents. The scenarios developed and the performance evaluation carried out, verify that this approach is satisfactory.

Future work will investigate possible advantages and drawbacks of allowing arbitrary location of the consistency rules and information of their applicability. Such approach could be useful in an "ad-hoc" distributed development network, or may benefit open source developers, working on a project without an established constant development team.

11.2.4 Incremental Checking

Consistency rules in the current iteration of the framework for link generation, used in this thesis, are based on expressions, formulated in the XPath language. The general incremental checking algorithm we have specified, requires that each document change is compared to expressions in consistency rules in order to determine, whether a rule is applicable to that change. In practice, the process of selection of relevant rules compares the XPath expressions in the rules with the XPath expression to the changed element, the latter being generated by the TreeDiff.

The current implementation of the incremental checking algorithm uses a light-weight approach, which provides string-wise analysis of XPath expressions and relatively simple document DOM tree navigation to aid in selection of relevant rules. When complex XPath expressions are used, this approach is proved to select consistency rules, which, in addition to relevant rules, selects the rules that are not necessarily applicable to the particular document change under consideration.

Future work will enable the rule selector algorithm to compare XPath expressions of arbitrary complexity. Efficiency of this algorithm for rule selection will then be compared to the efficiency of a consistency check to determine whether a performance advantage can be derived from more precise estimation of relevant rules for the incremental check.

Another area of incremental checking development, currently investigated in the group, is in establishment of rule applicability to a document *type*. In relevant rule selection process, the type definition (DTD) can be navigated during the selection of relevant consistency rules. This approach is attractive for application areas, where DTD is sufficiently constrained, so that sequence of allowed elements can be easily determined. The approach was not used in this thesis, because the document DTD of the application domain studied here, UML DTD in its original form, does not include sufficient constraints so that the DTD can be navigated, alike a document instance.

11.3 Closing Remarks

This thesis is an attempt to tackle the software engineering problem of consistency checking in a distributed setting. We have successfully applied the current state of the art in the continuously developing mobile agent technology to a practical consistency checking task.

From the functional requirements, through the design and modelling, to the construction of the implementation prototype, an effort has been made to simplify and generalize the approach. Yet, the proposed software agent architecture has demonstrated the complexity of an infrastructure, which is required in order to be able to engage mobile agents in a non-trivial distributed activity and coordinate the distributed agents in the process.

Though the focus of this thesis is on distributed consistency checking, it may shed light on a larger question whether mobile agents are an appropriate infrastructure for providing distribution support in software engineering tools. It is clear that the agent technology can be successfully applied to consistency checking and in other software engineering areas, but such application requires a careful architectural consideration and a radical rethinking of the underlying algorithms.

Bibliography

[Ainsworth, et al. 1994] Ainsworth, M., Cruickshank, A. H., Groves, L. J., and Wallis, P. J. L., Viewpoint Specification And Z, In *Information and Software Technology*, 36(1): 43-51, 1994.

[Alderson 1988] Alderson, A., A space-efficient technique for recording the versions of data, In *Software Engineering Journal*, November 1988: 240-247, 1988.

[Alderson and Elliott 1989] Alderson, A. and Elliott, A., The Eclipse Tool-Builder's Kit and the HOOD Toolset. In Software Engineering Environments: Research and Practice, Bennett, K.H., Ed., Ellis Horwood, 1989.

[Alderson 1991] Alderson, A., Meta-CASE Technology. In Software Development Environments and CASE Technology, vol. LNCS 509, pp. 81-91, Endres, A. and Weber, H., Eds., Berlin: Springer-Verlag, 1991.

[Apache 2000a] Apache, Xalan XSLT Processor. http://xml.apache.org/, 2000.

[Apache 2000b] Apache, Xerces Java XML Parser. http://xml.apache.org/~andyc/xerces2/, 2000.

[Armitage, et al. 1998] Armitage, S., Stevens, R., and Finkelstein, A., Implementing a Compliance Manager, In *Requirements Engineering*, 3(2): 98-106, 1998.

[Asperti and Busi 1996] Asperti, A. and Busi, N., Mobile Petri Nets. Technical Report UBLCS-96-10, Department of Computer Science, University of Bologna, 1996.

[Baldwin 1993] Baldwin, D., Applying Multiple Views To Information Systems: A Preliminary Framework, In *Database*, 24(4): 15-30, 1993.

[Balzer 1991] Balzer, R., Tolerating Inconsistency. In *Proc. of 13th Int. Conference on Software Engineering*, Austin, Texas, USA, pp. 158-165, IEEE Computer Society Press, 1991.

[Bergner, et al. 1997] Bergner, K., Rausch, A., and Sihling, M., Using UML for Modeling a Distributed Java Application. TUM-I9735, Techniche Universitat Munchen, Munchen, July 1997.

[Boiten, et al. 1996] Boiten, E., Bowman, H., Derrick, J., and Steen., M., Issues In Multiparadigm Viewpoint Specification. In *Proceedings of the International Workshop on Multiple Perspectives in Software Development (Viewpoints '96)*, A., Finkelstein and G., Spanoudakis, Eds., ACM SIGSOFT Foundations of Software Engineering 4, pp. 162-166. 1996.

[Booch 1991] Booch, G., Object-Oriented Design With Applications, The Benjamin/Cummings Publishing Company, Inc., 1991.

[Booch 1994] Booch, G., Object-Oriented Analyis And Design With Applications, 2nd ed. The Benjamin Cummings Publishing Company, Inc., 1994.

[Booch, et al. 1999] Booch, G., Jacobson, I., and Rumbaugh, J., The Unified Modelling Language User Guide: UML, Addison Wesley, 1999.

[Bowman, et al. 1996] Bowman, H. A., Boiten, E. A., Derrick, J., and Steen, M. W. A., Viewpoint Consistency In ODP, A General Interpretation. In *Formal Methods for Open Object-based Distributed Systems (FMOODS)*, Najim, E. and Stefani, J.B., Eds., pp. 189-204. Chapman & Hall, 1996.

[Bowman, et al. 1995] Bowman, H. A., Derrick, J., and Steen, M. W. A., Some Results On Cross Viewpoint Consistency Checking. In *Open Distributed Processing III*, Raymond, K. and Armstrong, L., Eds., pp. 399-412. Chapman & Hall, 1995.

[Box 1998] Box, D., Essential COM, Addison Wesley Longman, 1998.

[Bray, et al. 1998] Bray, T., Paoli, J., Sperberg-McQueen, C. M., and Maler, E., Extensible Markup Language. Recommendation. Wold Wide Web Consortium. *http://www.w3.org/TR/2000/REC-xml-20001006,* 1998.

[Caribou 1997] Caribou, Lake Software, Remote Observer Classes. *http://www.cariboulake.com/products/products.html,* 1997.

[Castelfranchi 1995] Castelfranchi, C., Guarantees for Autonomy in Cognitive Agent Architectures. In *Intelligent Agents: Theories, Architectures and Languages*, Wooldridge, M. and Jennings, N. R., Eds., Lecture Notes in Artificial Intelligence, vol. 890, pp. 56-70. Springer-Verlag, 1995.

[Cerutti and Pierson 1993] Cerutti, D. and Pierson, D., Distributed computing environments, McGraw-Hill, 1993.

[Chess, et al. 1997] Chess, D., Harrison, C., and Kershenbaum, A., Mobile Agents: Are They a Good Idea? In *Mobile Object Systems, Towards a Programmable Internet*, Vitek, J. and Tschudin, C., Eds., Lecture Notes in Artificial Intelligence, vol. 1222, Springer-Verlag, 1997.

[Chin 1991] Chin, D. N., Intelligent Interfaces As Agents. In *Intelligent User Interfaces*, Sullivan, J. W. and Tyler, S. W., Eds., pp. 177-206. ACM Press, 1991.

[Clark and DeRose 1999] Clark, J. and DeRose, S., XML Path Language (XPath) Version 1.0. Recommendation. World Wide Web Consortium. *http://www.w3.org/TR/1999/REC-xpath-19991116,* 1999.

[Colouris, et al. 2000] Colouris, G., Dollimore, J., and Kindberg, T., Distributed Systems: Concepts And Design, 3nd ed. Addison-Wesley, 2000.

[Covers 1998] Covers, Covers - Tool for Construction of State Transition Models. *http://www.xjtek.com,* 1998.

[Cowen, et al. 1993] Cowen, G., Derrick, J., Gill, M., Girling, G., Herbert, A., Linington, P. F., Rayner, D., Schulz, F., and Soley, R., Prost Report Of the Study On Testing For Open Distributed Processing. Girling, G., Ed., APM Ltd., 1993.

[Cugola, et al. 1996] Cugola, G., Ghezzi, C., Picco, G., and Vigna, G., Analyzing Mobile Code Languages. In *Mobile Object Systems - Second International Workshop, MOS'96*, Linz, Austria, J., Vitek and C., Tschudin, Eds., Lecture Notes in Computer Science, 1222, pp. 93-109, Springer, July 1996.

[Dasgupta, et al. 1999] Dasgupta, P., Narasimhan, N., Moser, L. E., and Melliar-Smith, P. M., MAgNET: Mobile Agents for Networked Electronic Trading. University of California, Santa Barbara, CA, March 1999. http://alpha.ece.ucsb.edu/~pdg/research/ papers/MAgNEThtml/MAgNET.html.

[DeRose, et al. 2000] DeRose, S., Maler, E., and Orchard, D., XML Linking Language (XLink). Version 1.0. Candidate Recommendation. World Wide Web Consortium, 20 December 2000. www.w3.org/TR/xlink/.

[Despeyroux 1988] Despeyroux, T., TYPOL - A Framework to Implement Natural Semantics. Technical Report 94, INRIA, Roquencourt, 1988.

[Donzeau-Gouge, et al. 1984] Donzeau-Gouge, V., Kahn, G., Lang, B., and Melese, M., Document structure and modularity in Mentor. In *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, Penn., ACM SIGSOFT Software Engineering Notes, vol. 9(3), pp. 141-148, ACM, 1984.

[Easterbrook, et al. 1994] Easterbrook, S., Finkelstein, A., Kramer, J., and Nuseibeh, B., Co-ordinating Distributed Viewpoints: the Anatomy Of a Consistency Check, In *International Journal on Concurrent Engineering: Research and Applications, 2(3),*: 209-222, 1994.

[Ellmer, et al. 1999] Ellmer, E., Emmerich, W., Finkelstein, A., Smolko, D., and Zisman, A., Consistency Management of Distributed Documents Using XML And Related Technologies. Research Note 99-94, University College London, Dept. of Computer Science, London, 1999. Submitted for Publication.

[Emmerich 2000] Emmerich, W., Engineering Distributed Objects, John Wiley & Sons, 2000.

[Emmerich, et al. 1999] Emmerich, W., Finkelstein, A., Montangero, C., Antonelli, S., Armitage, S., and Stevens, R., Managing Standards Compliance, In *Transactions on Software Engineering*, 25(6) 1999.

[Engels, et al. 1987] Engels, G., Nagl, M., and Schafer, W., On the Structure of Structure oriented Editors for Different Applications. In *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Palo Alto, Cal., ACM SIGPLAN Notices, vol. 22(1), pp. 190-198, ACM,

[Fifin, et al. 1994] Fifin, T., Weber, J., Wiederhold, G., Genesereth, M., Fritzon, R., McKay, D., McGuire, J., Pelavin, R., Shapiro, S., and Beck, C., Specification Of the KQML Agent-Communication Language, In *Artificial Intelligence*, 2(3-4): 189-208, 1994.

[Finkelstein, et al. 1994] Finkelstein, A., Kramer, J., and Nuseibeh, B., Inconsistency Handling in Multi-Perspective Specifications, In *IEEE Transactions on Software Engineering*, 20(8): 569-578, 1994.

[Finkelstein, et al. 1992] Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., and Goedicke, M., Viewpoints: a Framework For Integrating Multiple Perspectives In System Development, In *International Journal on Software Engineering and Knowledge Engineering, Special issue on Trends and Research Directions in Software Engineering Environments*, 2(1): 31-58, 1992.

[Finkelstein and Smolko 2000] Finkelstein, A. and Smolko, D., Software Agent Architecture for Consistency Management in Distributed Documents. In *Proceedings of SCI 2000 - World Multiconference on Systemics, Cybernetics and Informatics and ISAS 2000 - 6th Internatiomal Conference on Information Systems Analysis and Synthesis*, Orlando, USA, vol. IX, pp. 715-719, International Institute of Informatics and Systemics, July 2000.

[Finkelstein, et al. 1996] Finkelstein, A., Spanoudakis, G., and Till, D., Managing Interference. In *Proceedings of ACM SIGSOFT 96 Workshop, Viewpoints 96*, pp. 172-174, ACM Press,

[Franklin and Graesser 1997] Franklin, S. and Graesser, A., Is it an Agent, or Just a Program? In *Intelligent Agents III*, Muller, J. P., Wooldridge, M. , and Jennings, N. R., Eds., Lecture Notes on Artificial Intelligence, vol. 1193, pp. 21-36. Springer-Verlag, 1997.

[Frappier, et al. 1995] Frappier, M., Mili, A., and Desharnais, J., Program Construction By Parts. In *Mathematics of Program Construction*, B., Möller, Ed., Lecture Notes in Computer Science, vol. 947, pp. 257-281. Springer-Verlag, 1995.

[Fuggetta, et al. 1998] Fuggetta, A., Picco, G. P., and Vigna, G., Understanding Code Mobility, In *IEEE Transactions on Software Engineering*, 24(5): 342-361, 1998.

[Geddis, et al. 1995] Geddis, D. F., Genesereth, M. R., Keller, A. M., and Singh, N. P., Infomaster: A Virtual Information System. In *Proceedings of the Intelligent Information Agents Workshop, Fourth International Conference on Information and Knowledge Management*, Baltimore, Maryland, USA, 1995.

[Genesereth and Fikes 1992] Genesereth, M. R. and Fikes, R. E., Knowledge Interchange Format Version 3.0 Reference Manual. Interlingua Working Group of the DARPA Knowledge Sharing Initiative,available from http://logic.stanford.edu/kif/.

[Genesereth and Ketchpel 1994] Genesereth, M. R. and Ketchpel, S. P., Software Agents, In *Communications of the ACM*, 37(7): 48-53, 1994.

[Giladi and Shoval 1994] Giladi, R. and Shoval, P., An Architecture Of an Intelligent System for Routing User Requests In a Network Of Heterogeneous Databases, In *Journal of Intelligent Information Systems*, 3: 205-219, 1994.

[Goland, et al. 1999] Goland, Y. Y., Whitehead, E. J., Faizi, A. J., Carter, S. R., and Jensen, D., Extensions for Distributed Authoring on the World Wide Web - WEBDAV. Request for Comments: 2518, Internet Engineering Task Force, February 1999. http://www.ics.uci.edu/~ejw/ authoring/protocol/rfc2518.txt.

[GOODSTEPTeam 1994] GOODSTEPTeam, The GOODSTEP Project: General Object-Oriented Database for Software Engineering Processes. In *Proc. of the Asia-Pacific Software Engineering Conference*, Tokyo, Japan, K., Ohmaki, Ed., pp. 410-420, IEEE Computer Society Press, 1994.

[Goose 1995] Goose, S., Distributed Open Hypermedia Systems. PhD Thesis, Department of Electronics and Computer Science, University of Southampton, 1995.

[Gray 1996] Gray, R. S., Rus, D. and Kotz, D., Transportable Information Agents. Technical Report TR96-278, Department of Computer Science, Dartmouth College, USA, 1996.

[Habermann and Notkin 1986] Habermann, A. N. and Notkin, D., Gandalf: Software Development Environments, In *IEEE Transactions on Software Engineering*, 12(12): 1117-1127, 1986.

[Harel 1987] Harel, D., Statecharts: A Visual Formalism for Complex Systems, In *Science of Computer Programming*, 8(3): 231-274, 1987.

[Haugeneder and Steiner 1998] Haugeneder, H. and Steiner, D., Co-operating Agents: Concepts and Applications. In *Agent Technology: Foundation, Applications and Markets*, Jennings, R. and Wooldridge, M., Eds., Springer, 1998.

[IBM 1997] IBM, TabiCan. IBM Corporation, 1997. http://www.tabican.ne.jp.

[IBM 1998] IBM, Aglets SDK. IBM Corporation, July 1998. http://www.trl.ibm.co.jp/aglets/.

[Johnson and Fisher 1982] Johnson, G. F. and Fisher, C. N., Non-syntactic attribute ow in language based editors. In *Proc. of the 9th Annual ACM Symposium on Principles of Programming Languages*, pp. 185-195, ACM Press, 1982.

[Karjoth, et al. 1997] Karjoth, G., Lange, D. B., and Oshima, M., A Security Model for Aglets, In *IEEE Internet Computing*, 1(4): 68-77, 1997.

[Karnik and Tripathi 1998] Karnik, N. and Tripathi, A., Design Issues in Mobile-Agent Programming Systems, In *IEEE Concurrency*, 6(3): 52-61, 1998.

[Kastens 1980] Kastens, U., Ordered Attributed Grammars, In *Acta Informatica,*, 13(3): 229-256, 1980.

[Kay 1984] Kay, A., Computer Software, In *Scientific American*, 251(3): 191-207, 1984.

[Kiniry and Zimmerman 1997] Kiniry, J. and Zimmerman, D., A Hands-On Look at Java Mobile Agents, In *IEEE Internet Computing*, 1(4) 1997.

[Knuth 1968] Knuth, D. E., Semantics of Context-Free Languages, In *Mathematical SystemsTheory*, 2(2): 127-145, 1968.

[Kotonya and Sommerville 1992] Kotonya, G. and Sommerville, I., Viewpoints For Requirements Definition, In *IEE/BCS Software Engineering Journal*, 7(6): 375-387, 1992.

[Labrou and Finin 1994] Labrou, Y. and Finin, T., A Semantics Approach for KQML - a General Purpose Communication Language for Software Agents. In *Proceedings of the 3rd International Conference On Information and Knowledge Management (CIKM'94)*, Gaithersburg, MD, USA, pp. 447-455, ACM Press, 1994.

[Lange and Aridor 1997] Lange, D. B. and Aridor, Y., Agent Transfer Protocol. IBM Tokyo Research Laboratory Draft ATP/0.1, March 1997.

[Lange and Oshima 1998] Lange, D. B. and Oshima, M., Programming and Deploying Java Mobile Agents with Aglets, Addison Wesley, 1998.

[Larsen, et al. 1995] Larsen, K. G., Steffen, B., and Weise, C., A Constraint Oriented Proof Methodology Based On Modal Transition Systems. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95),*, E., Brinksma, Ed., Lecture Notes in Computer Science, vol. 1019, pp. 17-40, Springer-Verlag, 1995.

[Laurel 1990] Laurel, B., Interface Agents: Metaphors with Character. In *The Art of Human-Computer Interface Design*, Laurel, B., Ed., pp. 355-365. Addison-Wesley, 1990.

[Lewerentz and Schurr 1988] Lewerentz, C. and Schurr, A., GRAS, a management system for graph-like documents. In *Proc. of the 3rd Int. Conf. on Data and Knowledge Bases*, pp. 19-31, Morgan Kaufmann, 1988.

[Lieberman 1995] Lieberman, H., Letizia: An Agent that Assists Web Browsing. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Montreal, Canada, pp. 924-929, AAAI Press/The MIT Press, 1995.

[Linington 1995] Linington, P. F., RM-ODP: The Architecture. In *Open Distributed Processing II*, Raymond, K. and Armstrong, L., Eds., pp. 15-33. Chapman & Hall, 1995.

[Lux, et al. 1993] Lux, A., de Greef, P., Bomarius, F., and Steiner, D., A Generic Framework for Human Computer Co-operation. In *Proceedings of International Conference on Intelligent and Co-operative Information Systems*, Huns, M., Papazoglou, M., and Schlageter, G., Eds., pp. 89-97, IEEE Computer Society Press, 1993.

[Magedanz, et al. 1996] Magedanz, T., Rothermel, K., and Krause, S., Intelligent Agents: An Emerging Technology for Next Generation Telecommunications? In *Proceedings of the INFOCOM Conference. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation*, San Francisco, CA USA, vol. 2, pp. 464-472, IEEE, March 1996.

[Mayfield, et al. 1995] Mayfield, J., Labrou, Y., and Finin, T., Desiderata for Agent Communication Languages. In *Proceedings of the AAAI Symposium on Information Gathering from Heterogeneous, Distributed Environments*, Technical Report SS-95-08, Knoblock, C. and Levy, A., Eds., pp. 64-70, 1994.

[Metasoft 2001] Metasoft, ISDE Metasoft Ltd. 329-331 London Road, Camberley, Surrey GU15 3HQ, UK.

[Meyers 1991] Meyers, S., Difficulties in Integrating Multiview Development Systems, In *IEEE Software*, 8(1): 49-57, 1991.

[Milliner and Papazoglou 1994] Milliner, S. and Papazoglou, M., Reassessing the Roles of Negotiation and Contracting for Interoperable Databases. In *Proceedings of the International Workshop on Advances in Databases and Information Systems*, Russian ACM SIGMOD, pp. 169--202, May 1994.

[Milojicic 1999] Milojicic, D., Trend Wars: Mobile Agent Applications, In *IEEE Concurrency*, 7(3): 80-90, 1999.

[Milojicic, et al. 1998] Milojicic, D., Breugst, M., Busse, I., Campbell, J., Covaci, S., Friedman, B., Kosaka, K., Lange, D., Ono, K., Oshima, M., Tham, C., Virdhagriswaran, S., and White, J., MASIF - The OMG Mobile Agent System Interoperability Facility. In *Proceedings of Mobile Agents - Second International Workshop, MA '98*, Stuttgart, Germany, K., Rothermel and F., Hohl, Eds., Lecture Notes in Computer Science, vol. 1477, pp. 50-67, Springer, September 1998.

[Mobility 2001] Mobility, The Mobility Mailing List for discussion of mobility. *http://mobility.lboro.ac.uk*, 2001.

[Mullery 1979] Mullery, G. P., CORE - a Method for Controlled Requirement Specification. In *Proceedings of the 4th International Conference on Software Engineering*, pp. 126-135, IEEE Computer Society, 1979.

[Nagl 1985] Nagl, M., An Incremental and Integrated Software Development Environment, In *Computer Physics Communications*, 38: 245-276, 1985.

[Navathe, et al. 1986] Navathe, S., Elmasri, R., and Larson, J., Integrating user views in database design, In *IEEE Computer*, 19(1): 50-62, 1986.

[Nentwich, et al. 2000a] Nentwich, C., Capra, L., Emmerich, W., and Finkelstein, A., http://www.xlinkit.com, 2000.

[Nentwich, et al. 2000b] Nentwich, C., Capra, L., Emmerich, W., and Finkelstein, A., xlinkit: a Consistency Checking and Smart Link Generation Service. Research Note RN/00/66. Submitted for Publication, University College London, London, December 2000.

[Nentwich, et al. 2001a] Nentwich, C., Emmerich, W., and Finkelstein, A., Checking Distributed Software Engineering Content. Department of Computer Science Research Note RN/01/11. Submitted for publication, University College London, UK, March 2001.

[Nentwich, et al. 2001b] Nentwich, C., Emmerich, W., and Finkelstein, A., xlinkit: links that make sense. Department of Computer Science Research Note RN/01/6, University College London, UK, 2001.

[Nuseibeh 1994] Nuseibeh, B. A., A Multi-Perspective Framework for Method Integration. PhD thesis, Imperial College, University of London, 1994.

[Nuseibeh 1995] Nuseibeh, B., Meta-CASE Support for Method-Based Software Development. In First International Congress on Meta-CASE, Sunderland, UK, 5-6th January 1995.

[Nwana and Wooldridge 1996] Nwana, H. S. and Wooldridge, M., Software Agent Technologies, In *British Telecom Technology Journal*, 14(4): 68-78, 1996.

[ObjectSpace 1997] ObjectSpace, Voyager, General Magic Odyssey, IBM Aglets: A Comparison. Technical White Paper. ObjectSpace, Incorporated, June 1997.

[OMG 1995] OMG, Object Management Group, The Common Object Request Broker: Architecture and Specification Revision 2.0. OMG, 492 Old Connecticut Path, Framingham, MA 01701, USA, 1994.

[OMG 2000a] OMG, Mobile Agent Facility formal specification, v.1.0. Object Management Group, 492 Old Connecticut Path, Framingham, MA 01701, USA, 2000.

[OMG 2000b] OMG, Unified Modeling Language Specification, 2000.

[OMG 2000c] OMG, XML Metadata Interchange (XMI) Specification 1.1, November 2000.

[Oshima and Karjoth 1997] Oshima, M. and Karjoth, G., Aglets Specification (1.0). IBM Tokio Research Laboratory, May 1997.

[Panurak 1998] Panurak, V. D. H., Practical and Industrial Applications of Agent-Based Systems. Available at http://www.erim.org/~van/apps98.pdf.

[Papaioannou 2000] Papaioannou, T., On the Structuring of Distributed Systems: The Argument for Mobility. Ph.D. Thesis, Loughborough University, 2000.

[Papaioannou and Edwards 1999] Papaioannou, T. and Edwards, J. M., Using Mobile Agents To Improve the Alignment Between Manufacturing and its IT Support Systems, In *International Journal of Robotics and Autonomous Systems*, 27: 45-57, 1999.

[Picco 1998] Picco, G. P., Understanding, Evaluating, Formalizing, and Exploiting Code Mobility. Politecnico di Torino: 1998.

[Prestegard, et al. 1999] Prestegard, G., Hanssen, A. A., Brandstadmoen, S., and Nymoen, B. S., DIAS - Distributed Intelligent Agent System. Project report. NTNU, Trondheim, Norway, 1999.

[Rational 1998] Rational, Insurance Network Fees and Claims System, Rational Inc. *http://www.rational.com/support/downloadcenter/addins/media/rose/insurance.zip,* 1998.

[Rational 1999] Rational, XMI Converter for Rational Rose, Rational Inc. *http://www.rational.com/products/rose/forms/xmisupport/xmisupport.jsp,* 1999.

[Rational 2000] Rational, Rational Rose versions 98- 2000.

[Redmond 1997] Redmond, F., DCOM: Microsoft Distributed Component Object Model, IDG Books Worldwide, 1997.

[Reeves, et al. 1995] Reeves, A., Marashi, M., and Budgen, D., A software design framework or how to support real designers, In *IEE/BCS Software Engineering Journal*, 10(4): 141-155, 1995.

[Reps and Teitelbaum 1981] Reps, T. W. and Teitelbaum, T., The Cornell Program Synthesizer: A syntax-directed programming environment, In *Communications of the ACM*, 24(9): 449-477, 1981.

[Reps and Teitelbaum 1984] Reps, T. W. and Teitelbaum, T., The Synthesizer Generator., In *ACM SIGSOFT Software Engineering Notes*, 9(3): 42-48, 1984. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Penn.

[Reps and Teitelbaum 1988] Reps, T. W. and Teitelbaum, T., The Synthesizer Generator - a system for constructing language based editors, Springer, 1988.

[Revheim 2000] Revheim, T., Consistency Checking of Heterogeneous Distributed Documents. Master Thesis. University College London, 2000.

[RMI 1998] RMI, Java Remote Method Invocation Specification. JavaSoft, Revision 1.50, JDK 1.20 Edition. 1998.

[Seregeundo, et al. 1996] Seregeundo, G. D. M., Muhugusa, M., Tschudin, C., and Harms, J., Formalization of Agents and Multi-Agent Systems. The Special Case of Category Theory. Technical Report 109, Teleinformatics, University of Geneva, 1996.

[Sibertin-Blanc 1994] Sibertin-Blanc, C., Cooperative nets. In *LCNS,* vol. 815, Valette, R., Ed., Springer-Verlag, 1994. pp. 471-490.

[Simon 1996] Simon, E., Distributed Information Systems – from client/server to distributed multimedia, McGraw-Hill, 1996.

[Sloman and Kramer 1987] Sloman, M. and Kramer, J., Distributed Systems and Computer Networks, Prentice Hall, 1987.

[Smolko 1999] Smolko, D., Consistency Issues in Document Management of Distributed Group Work. In *Proceedings of Doctorate Symposium, Automated Software Engineering '99, 14th IEEE International Conference*, Orlando, USA, IEEE Computer Society, 1999.

[Smolko 2001] Smolko, D., Design and Evaluation of the Mobile Agent Architecture for Distributed Consistency Management. In *Proc. 23rd Int. Conference on Software Engineering*, Toronto, Canada, pp. 799-801, IEEE Computer Society, 2001.

[Spanoudakis, et al. 1997] Spanoudakis, G., Finkelstein, A., and Emmerich, W., Viewpoints 96: International workshop on multiple perspectives in software development (SIGSOFT 96)., In *ACM SIGSOFT Software Engineering Notes*, 22(1): 39-41, 1997.

[Steels 1990] Steels, L., Cooperation between distributed agents through self-organisation. In *Decentralized AI*, Demazeau, Y. and Muller, J. P., Eds., pp. 175-196. Amsterdam: Elsevier Science Publishers B.V. (North Holland), 1990.

[Steen 1998] Steen, M., Consistency And Composition Of Process Specifications. Ph. D. Thesis, University of Kent at Canterbury. 1998.

[Steen, et al. 1999] Steen, M., Derrick, J., Boiten, E., and Bowman, H., Consistency of Partial Process Specifications. In *Algebraic Methodology and Software Technology, 7th International Conference, AMAST '98*, Amazonia, Brasil, Haeberer , A. M., Ed., Lecture Notes in Computer Science, vol. 1548, Springer, January 1999.

[Vigna 1997] Vigna, G., Mobile Code Technologies, Paradigms, and Applications. Ph.D. Thesis Politecnico di Milano, Italy, 1997.

[Waldo, et al. 1994] Waldo, J., Wyant, G., Wollrath, A., and Kendall, S., A note on distributed computing. Sun Microsystems Technical Report SML 94-29, Sun Microsystems, 1994.

[Welland, et al. 1990] Welland, R. C., Beer, S., and Sommerville, I., Method Rule Checking in a Generic Design Editing System, In Software Engineering Journal, 5(2): 110-115, 1990.

[Westfechtel 1989] Westfechtel, B., Revision Control in an Integrated Software Development Environment, In *ACM SIGSOFT Software Engineering Notes*, 17(7): 96-105, 1989.

[Wooldridge and Jennings 1995] Wooldridge, M. and Jennings, N. R., Intelligent Agents: Theory and Practice, In *Knowledge Engineering Review*, 10(2): 115--152, 1995.

[Zave and Jackson 1996] Zave, P. and Jackson, M., Where do operations come from : A multi-paradigm specification technique, In *IEEE Transactions on Software Engineering*, 22(7): 508-528, 1996.

[Zisman, et al. 1999] Zisman, A., Finkelstein, A., Ellmer, E., Smolko, D., and Emmerich, W., Method and Apparatus for Monitoring and Maintaining the Consistency of Distributed Documents. International Patent, The U.K. Patent Office Patent No GB2351165, University College London, 18/06/1999. Application No GB9914232.5.

[Zisman, et al. 2000] Zisman, A., Emmerich, W., and Finkelstein, A., Using XML to Build Consistency Rules for Distributed Specifications. In *Proceedings of 10th International Workshop on Software Specification and Design*, San Diego, CA, USA, pp. 141 -148, ACM, November 2000.

Appendix A UML Well-Formedness Consistency Rules

A.1 Associations

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE consistencyruleset SYSTEM 'consistencyrule.dtd'>
<?xml-stylesheet type='text/xsl' href='rule.xsl'?>
<?cocoon-process type='xslt'?>
<consistencyruleset>

<globalset id="associations"
xpath="//Foundation.Core.Association[@xmi.id]"/>

<consistencyrule id="a1">
  <description>
The AssociationEnds must have a unique name within the Association
  </description>

  <forall var="a" in="$associations">
      <forall var="x"
in="$a/Foundation.Core.Association.connection/Foundation.Core.
AssociationEnd">
          <forall var="y"
in="$a/Foundation.Core.Association.connection/Foundation.Core.
AssociationEnd">
              <implies>
                  <equal
op1="$x/Foundation.Core.ModelElement.name/text()"
op2="$y/Foundation.Core.ModelElement.name/text()"/>
                  <same op1="$x" op2="$y"/>
              </implies>
          </forall>
      </forall>
  </forall>
</consistencyrule>

<consistencyrule id="a2">
  <description>
      At most one AssociationEnd may be an aggregation or composition
  </description>

  <forall var="a" in="$associations">
      <forall var="x"
in="$a/Foundation.Core.Association.connection/Foundation.Core.
AssociationEnd">
          <implies>
              <notequal
op1="$x/Foundation.Core.AssociationEnd.aggregation/@xmi.value"
op2="'none'"/>
              <not><exists var="y"
in="$a/Foundation.Core.Association.connection/Foundation.Core.
AssociationEnd">
```

```
                        <and><notequal
op1="$y/Foundation.Core.AssociationEnd.aggregation/@xmi.value"
op2="'none'"/>
                            <not><same op1="$x" op2="$y"/>
                            </not>
                        </and>
                    </exists>
                </not>
            </implies>
        </forall>
    </forall>
</consistencyrule>

<consistencyrule id="a3">
    <description>
        If an Association has three or more AssociationEnds, then no
        AssociationEnd may be an aggregation or composition
    </description>

    <forall var="a" in="$associations">
        <not><exists var="x"
in="$a/Foundation.Core.Association.connection/Foundation.Core.
AssociationEnd" mode="instance">
                <exists var="y"
in="$a/Foundation.Core.Association.connection/Foundation.Core.
AssociationEnd" mode="instance">
                    <exists var="z"
in="$a/Foundation.Core.Association.connection/Foundation.Core.
AssociationEnd" mode="instance">
                        <and>
                            <and>
                                <not>
                                <same op1="$x" op2="$y"/>
                                </not>
                                <and>
                                    <not>
                                    <same op1="$y" op2="$z"/>
                                    </not>
                                    <not>
                                    <same op1="$x" op2="$z"/>
                                    </not>
                                </and>
                            </and>
                            <or>
                                <notequal
op1="$x/Foundation.Core.AssociationEnd.aggregation/@xmi.value"
op2="'none'"/>
                                <or>
                                    <notequal
op1="$y/Foundation.Core.AssociationEnd.aggregation/@xmi.value"
op2="'none'"/>
                                    <notequal
op1="$z/Foundation.Core.AssociationEnd.aggregation/@xmi.value"
op2="'none'"/>
                                </or>
                            </or>
                        </and>
                    </exists>
                </exists>
            </exists>
        </not>
    </forall>
</consistencyrule>
```

```
<consistencyrule id="a4">
  <description>
      The connected Classifiers of the AssociationEnds should be included in
the Namespace of the Association
  </description>

  <forall var="a" in="$associations">
      <forall var="x"
in="$a/Foundation.Core.Association.connection/Foundation.Core.
AssociationEnd/Foundation.Core.AssociationEnd.type/*[1]">
          <exists
var="y" in="$a/ancestor::Foundation.Core.Namespace.ownedElement/*[@xmi.id=
$x/@xmi.idref]">
          </exists>
      </forall>
  </forall>
</consistencyrule>

</consistencyruleset>
```

A.2 AssociationClass

The XML header of this and of all further rulesets are omitted to save space, as they are the same as the header of A.1, Associations.

```
<consistencyruleset>
<globalset id="associationclasses"
xpath="//Foundation.Core.AssociationClass[@xmi.id]"/>

<consistencyrule id="ac1">
  <description>
      The names of the AssociationEnds and the StructuralFeatures do not
overlap
  </description>

  <forall var="a" in="$associationclasses">
      <forall var="s" in="$a/Foundation.Core.StructuralFeature/*">
          <not>
              <exists
var="c" in="$a/Foundation.Core.Association.connection/Foundation.Core.
AssociationEnd">
                  <equal
op1="$s/Foundation.Core.ModelElement.name/text()"
op2="$c/Foundation.Core.ModelElement.name/text()"/>
              </exists>
          </not>
      </forall>
  </forall>
</consistencyrule>

<consistencyrule id="ac2">
  <description>
      An AssociationClass cannot be defined between itself and something
else.
  </description>

  <forall var="a" in="$associationclasses">
      <not>
```

```
          <exists var="c"
in="$a/Foundation.Core.Association.connection/Foundation.Core.
AssociationEnd">
              <equal

op1="$a/@xmi.id"
op2="$c/Foundation.Core.AssociationEnd.type/*[1]/@xmi.idref"/>
          </exists>
      </not>
  </forall>
</consistencyrule>

</consistencyruleset>
```

A.3 AssociationEnd

```
<consistencyruleset>
<globalset id="associationends"
xpath="//Foundation.Core.AssociationEnd[@xmi.id]"/>

<consistencyrule id="ae1">
  <description>
      The Classifier of an AssociationEnd cannot be an Interface or a
      DataType if the association is navigable from that end
  </description>

  <forall var="e" in="$associationends">
      <implies>
          <or>
            <exists
var="x" in="$e/Foundation.Core.AssociationEnd.type/Foundation.Core.
DataType"/>
            <exists
var="x" in="$e/Foundation.Core.AssociationEnd.type/Foundation.Core.
Interface"/>
          </or>
          <not>
              <exists
var="x" in="$e/../Foundation.Core.AssociationEnd/Foundation.Core.
AssociationEnd.isNavigable[@xmi.value='true']"/>
          </not>
      </implies>
  </forall>
</consistencyrule>

<consistencyrule id="ae2">
  <description>
      An Instance may not belong by composition to more than one composite
Instance
  </description>

  <forall var="e" in="$associationends">
      <implies>
          <equal
op1="$e/Foundation.Core.AssociationEnd.aggregation/@xmi.value"
op2="'composite'"/>
          <or>
              <equal
op1="$e/Foundation.Core.AssociationEnd.multiplicity/text()"
op2="'1..1'"/>
              <equal
op1="$e/Foundation.Core.AssociationEnd.multiplicity/text()"
```

```
op2="'0..1'"/>
          </or>
      </implies>
  </forall>
</consistencyrule>
</consistencyruleset>
```

A.4 BehavioralFeature

```
<consistencyruleset>
<globalset id="behaviors"
xpath="//Foundation.Core.BehavioralFeature.parameter/.."/>

<consistencyrule id="b1">
  <description>
      All parameters should have a unique name
  </description>

  <forall var="b" in="$behaviors">
      <forall var="x"
in="$b/Foundation.Core.BehavioralFeature.parameter/Foundation.Core.
Parameter">
          <forall var="y"
in="$b/Foundation.Core.BehavioralFeature.parameter/Foundation.Core.
Parameter">
              <implies>
                  <equal
op1="$x/Foundation.Core.ModelElement.name/text()"
op2="$y/Foundation.Core.ModelElement.name/text()"/>
                  <same op1="$x" op2="$y"/>
              </implies>
          </forall>
      </forall>
  </forall>
</consistencyrule>

<consistencyrule id="b2">
  <description>
      The type of the Parameters should be included in the Namespace of the
Classifier
  </description>

  <forall var="b" in="$behaviors">
      <forall var="x"
in="$b/Foundation.Core.BehavioralFeature.parameter/Foundation.Core.
Parameter/Foundation.Core.Parameter.type/*[1]">
          <exists var="y"
in="$b/ancestor::Foundation.Core.Namespace.ownedElement/*[@xmi.id=
$x/@xmi.idref]"/>
      </forall>
  </forall>
</consistencyrule>
</consistencyruleset>
```

A.5 Class

```
<consistencyruleset>
<globalset id="classes" xpath="//Foundation.Core.Class[@xmi.id]"/>

<consistencyrule id="c1">
```

```
  <description>
      If a Class is concrete, all the Operations of the Class should have a
realizing method in the full descriptor
  </description>

  <forall var="c" in="$classes">
      <implies>
          <equal
op1="$c/Foundation.Core.GeneralizableElement.isAbstract/@xmi.value"
op2="'false'"/>
          <forall var="o"
in="$c/Foundation.Core.Classifier.feature/Foundation.Core.Operation">
              <exists var="m"
in="$c/Foundation.Core.Classifier.feature/Foundation.Core.Method">
                  <equal
op1="$o/@xmi.id"
op2="$m/Foundation.Core.Method.specification/Foundation.Core.
Operation/@xmi.idref"/>
              </exists>
          </forall>
      </implies>
  </forall>
</consistencyrule>
</consistencyruleset>
```

A.6 Classifier

```
<consistencyruleset>
<globalset id="classifiers"
xpath="//Foundation.Core.Classifier.feature/.."/>

<consistencyrule id="cs2">
  <description>
      No Attributes may have the same name within a Classifier
  </description>

  <forall var="c" in="$classifiers">
      <forall var="x"
in="$c/Foundation.Core.Classifier.feature/Foundation.Core.Attribute">
          <forall var="y"
in="$c/Foundation.Core.Classifier.feature/Foundation.Core.Attribute">
              <implies>
                  <equal
op1="$x/Foundation.Core.ModelElement.name/text()"
op2="$y/Foundation.Core.ModelElement.name/text()"/>
                  <same op1="$x" op2="$y"/>
              </implies>
          </forall>
      </forall>
  </forall>
</consistencyrule>

<consistencyrule id="cs3">
  <description>
      No opposite AssociationEnds may have the same name within a Classifier
  </description>

  <forall var="c" in="$classifiers">
      <forall var="x"
in="id($c/Foundation.Core.Classifier.associationEnd/Foundation.Core.
AssociationEnd/@xmi.idref)/../Foundation.Core.AssociationEnd
[Foundation.Core.AssociationEnd.type/*[1]/@xmi.idref!=$c/@xmi.id]">
```

```
          <forall var="y"
in="id($c/Foundation.Core.Classifier.associationEnd/Foundation.Core.
AssociationEnd/@xmi.idref)/../Foundation.Core.AssociationEnd
[Foundation.Core.AssociationEnd.type/*[1]/@xmi.idref!=$c/@xmi.id]">
             <implies>
                  <not><same op1="$x" op2="$y"/></not>
                  <notequal
op1="$x/Foundation.Core.ModelElement.name/text()"
op2="$y/Foundation.Core.ModelElement.name/text()"/>
             </implies>
          </forall>
      </forall>
  </forall>
</consistencyrule>

<consistencyrule id="cs4">
  <description>
      The name of an Attribute may not be the same as the name of an
opposite AssociationEnd or a ModelElement contained in the Classifier
  </description>

  <forall var="c" in="$classifiers">
      <forall var="a"
in="$c/Foundation.Core.Classifier.feature/Foundation.Core.Attribute">
          <and>
               <forall var="x"
in="id($c/Foundation.Core.Classifier.associationEnd/Foundation.Core.
AssociationEnd/@xmi.idref)/../Foundation.Core.AssociationEnd
[Foundation.Core.AssociationEnd.type/*[1]/@xmi.idref!=$c/@xmi.id]">
                    <notequal
op1="$a/Foundation.Core.ModelElement.name/text()"
op2="$x/Foundation.Core.ModelElement.name/text()"/>
               </forall>
               <forall var="y"
in="$c/Foundation.Core.Namespace.ownedElement/*">
                    <notequal
op1="$a/Foundation.Core.ModelElement.name/text()"
op2="$y/Foundation.Core.ModelElement.name/text()"/>
               </forall>
          </and>
      </forall>
  </forall>
</consistencyrule>

<consistencyrule id="cs5">

  <description>
      The name of an opposite AssociationEnd may not be the same as the name
of an Attribute or ModelElement contained in the Classifier
  </description>

  <forall var="c" in="$classifiers">
      <forall var="x"
in="id($c/Foundation.Core.Classifier.associationEnd/Foundation.Core.
AssociationEnd/@xmi.idref)/../Foundation.Core.AssociationEnd
[Foundation.Core.AssociationEnd.type/*[1]/@xmi.idref!=$c/@xmi.id]">
          <and>
              <forall var="a"
in="$c/Foundation.Core.Classifier.feature/Foundation.Core.Attribute">
                    <notequal
op1="$a/Foundation.Core.ModelElement.name/text()"
op2="$x/Foundation.Core.ModelElement.name/text()"/>
               </forall>
```

```
                <forall var="y"
in="$c/Foundation.Core.Namespace.ownedElement/*">
                    <notequal
op1="$y/Foundation.Core.ModelElement.name/text()"
op2="$x/Foundation.Core.ModelElement.name/text()"/>
                </forall>
            </and>
        </forall>
    </forall>
</consistencyrule>

<consistencyrule id="cs6">
    <description>
        For each Operation in a specification realized by a Classifier, the
Classifier must have a matching Operation.
    </description>

    <forall var="c" in="$classifiers">
        <forall var="x"
in="id($c/Foundation.Core.Classifier.specification/*[1]/@xmi.idref)/
Foundation.Core.Classifier.feature/Foundation.Core.Operation">
            <exists var="y"
in="$c/Foundation.Core.Classifier.feature/Foundation.Core.Operation">
                <and>
                    <equal
op1="$x/Foundation.Core.ModelElement.name/text()"
op2="$y/Foundation.Core.ModelElement.name/text()"/>
                    <equal
op1="$x/Foundation.Core.BehavioralFeature.parameter/Foundation.Core.
Parameter/Foundation.Core.ModelElement.name/text()"
op2="$y/Foundation.Core.BehavioralFeature.parameter/Foundation.Core.
Parameter/Foundation.Core.ModelElement.name/text()"/>
                </and>
            </exists>
        </forall>
    </forall>
</consistencyrule>
</consistencyruleset>
```

A.7 Component

```
<consistencyruleset>

<globalset id="components"
xpath="//Foundation.Auxiliary_Elements.Component"/>

<consistencyrule id="co1">
    <description>
        A Component may only contain other Components
    </description>

    <forall var="c" in="$components">
        <forall var="o"
in="$c/Foundation.Core.Namespace.ownedElement/*">
            <equal
op1="name($o)"
op2="'Foundation.Auxiliary_Elements.Component'"/>
        </forall>
    </forall>
</consistencyrule>
</consistencyruleset>
```

A.8 Constraint

```
<consistencyruleset>
<globalset id="constraints" xpath="//Foundation.Core.Constraint"/>

<consistencyrule id="cs1">
  <description>
      A Constraint cannot be applied to itself
  </description>

  <forall var="c" in="$constraints">
      <forall var="x"
in="$c/Foundation.Core.Constraint.constrainedElement/*">
          <notequal op1="$c/@xmi.id" op2="$x/@xmi.idref"/>
      </forall>
  </forall>
</consistencyrule>
</consistencyruleset>
```

A.9 DataType

```
<consistencyruleset>
<globalset id="datatypes" xpath="//Foundation.Core.DataType"/>

<consistencyrule id="d1">
  <description>
A DataType can only contain Operations, which all must be queries
  </description>

  <forall var="d" in="$datatypes">
      <forall var="x" in="$d/Foundation.Core.Classifier.feature/*">
          <and>
              <equal
op1="name($x)" op2="'Foundation.Core.Operation'"/>
              <equal
op1="$x/Foundation.Core.BehavioralFeature.isQuery/@xmi.value"
op2="'true'"/>
          </and>
      </forall>
  </forall>
</consistencyrule>

<consistencyrule id="d2">
  <description>
      A DataType cannot contain any other model elements
  </description>

  <forall var="d" in="$datatypes">
      <not>
          <exists var="x"
in="$d/Foundation.Core.Namespace.ownedElement/*"/>
      </not>
  </forall>
</consistencyrule>
</consistencyruleset>
```

A.10 Generalizable Element

```
<consistencyruleset>
```

```
<globalset id="generalizableelements"
xpath="//Foundation.Core.GeneralizableElement.generalization/.."/>

<consistencyrule id="g1">
  <description>
      A root cannot have any Generalizations
  </description>

  <forall var="g"
in="//Foundation.Core.GeneralizableElement.
isRoot[@xmi.value='true']/..">
      <not>
          <exists var="x"
in="$g/Foundation.Core.GeneralizableElement.generalization"/>
      </not>
  </forall>
</consistencyrule>

<consistencyrule id="g2">
  <description>
      No GeneralizableElement can have a parent Generalization to an element
which is a leaf
  </description>

  <forall var="g" in="$generalizableelements">
      <not>
          <exists var="p"
in="id(id($g/Foundation.Core.GeneralizableElement.generalization/
Foundation.Core.Generalization/@xmi.idref)/Foundation.Core.
Generalization.supertype/*[1]/@xmi.idref)/Foundation.Core.
GeneralizableElement.isLeaf[@xmi.value='true']">
          </exists>
      </not>
  </forall>
</consistencyrule>

<consistencyrule id="g4">
  <description>
      The parent must be included in the namespace of the
      GeneralizableElement
  </description>

  <forall var="g" in="$generalizableelements">
      <forall var="p"
in="id($g/Foundation.Core.GeneralizableElement.generalization/
Foundation.Core.Generalization/@xmi.idref)/Foundation.Core.
Generalization.supertype/*[1]">
          <exists var="x"
in="$g/ancestor::Foundation.Core.Namespace.ownedElement/*">
              <equal
op1="$p/@xmi.idref" op2="$x/@xmi.id"/>
          </exists>
      </forall>
  </forall>
</consistencyrule>
</consistencyruleset>
```

A.11 Generalization

```
<consistencyruleset>
<globalset id="generalizations" xpath="//Foundation.Core.Generalization"/>
```

```
<consistencyrule id="gen1">
  <description>
       A GeneralizableElement may only be a child of a GeneralizableElement
of the same kind
  </description>

  <forall var="g" in="$generalizations">
      <equal
op1="name($g/Foundation.Core.Generalization.supertype/*[1])"
op2="name($g/Foundation.Core.Generalization.subtype/*[1])"/>
  </forall>
</consistencyrule>
</consistencyruleset>
```

A.12 Interface

```
<consistencyruleset>
<globalset id="interfaces" xpath="//Foundation.Core.Interface"/>

<consistencyrule id="i1">
  <description>
      An Interface can only contain Operations.
  </description>

  <forall var="i" in="$interfaces">
      <forall var="f" in="$i/Foundation.Core.Classifier.feature/*">
          <or>
              <equal
op1="name($f)" op2="'Foundation.Core.Operation'"/>
              <equal
op1="name($f)"
op2="'Behavioral_Elements.Common_Behavior.Reception'"/>
          </or>
      </forall>
  </forall>
</consistencyrule>

<consistencyrule id="i2">
  <description>
      An Interface cannot contain any ModelElements
  </description>

  <forall var="i" in="$interfaces">
      <not>
          <exists var="x"
in="$i/Foundation.Core.Namespace.ownedElement/*"/>
      </not>
  </forall>
</consistencyrule>

<consistencyrule id="i3">
  <description>
      All Features defined in an Interface are public
  </description>

  <forall var="i" in="$interfaces">
      <forall var="f" in="$i/Foundation.Core.Classifier.feature/*">
          <equal
op1="$f/Foundation.Core.ModelElement.visibility/@xmi.value"
op2="'public'"/>
      </forall>
  </forall>
```

```
</consistencyrule>
</consistencyruleset>
```

A.13 Method

```
<consistencyruleset>
<globalset id="methods" xpath="//Foundation.Core.Method"/>

<consistencyrule id="m1">
  <description>
      If the realized Operation is a query, then so is the method.
  </description>

  <forall var="m" in="$methods">
      <implies>
          <equal
op1="id($m/Foundation.Core.Method.specification/Foundation.Core.
Operation/@xmi.idref)/Foundation.Core.BehavioralFeature.isQuery/
@xmi.value"
op2="'true'"/>
          <equal
op1="$m/Foundation.Core.BehavioralFeature.isQuery/@xmi.value"
op2="'true'"/>
      </implies>
  </forall>
</consistencyrule>

<consistencyrule id="m2">
  <description>
      The signature of the Method should be the same as the signature of the
realized Operation
  </description>

 <forall var="m" in="$methods">
      <equal
op1="$m/Foundation.Core.BehavioralFeature.parameter/Foundation.Core.
Parameter/Foundation.Core.Parameter.type/*[1]/@xmi.idref"
op2="id($m/Foundation.Core.Method.specification/Foundation.Core.
Operation/@xmi.idref)/Foundation.Core.BehavioralFeature.parameter/
Foundation.Core.Parameter/Foundation.Core.Parameter.type/*[1]/
@xmi.idref"/>
  </forall>
</consistencyrule>

<consistencyrule id="m3">
  <description>
      The visibility of the Method should be the same as for the realized
Operation
  </description>

  <forall var="m" in="$methods">
      <equal
op1="$m/Foundation.Core.ModelElement.visibility/@xmi.value"
op2="id($m/Foundation.Core.Method.specification/Foundation.Core.
Operation/@xmi.idref)/Foundation.Core.ModelElement.visibility/
@xmi.value"/>
  </forall>
</consistencyrule>
</consistencyruleset>
```

A.14 NameSpace

```
<consistencyruleset>
<globalset id="namespaces"
xpath="//Foundation.Core.Namespace.ownedElement/.."/>

<consistencyrule id="n1">
  <description>
If a contained element, which is not an Association or Generalization has a
name, then the name must be unique in the Namespace.
  </description>

  <forall var="n" in="$namespaces">
      <forall var="x"
in="$n/Foundation.Core.Namespace.ownedElement/
*[(not (name(.)='Foundation.Core.Association')) and
(not (name(.)='Foundation.Core.Generalization'))]">
        <forall var="y"
in="$n/Foundation.Core.Namespace.ownedElement/
*[(not (name(.)='Foundation.Core.Association')) and
(not (name(.)='Foundation.Core.Generalization'))]">
            <implies>
                <equal
op1="$x/Foundation.Core.ModelElement.name/text()"
op2="$y/Foundation.Core.ModelElement.name/text()"/>
                <same op1="$x" op2="$y"/>
            </implies>
        </forall>
      </forall>
  </forall>
</consistencyrule>

<consistencyrule id="n2">
  <description>
      All Associations must have a unique combination of name and associated
Classifiers in the Namespace
  </description>

  <forall var="n" in="$namespaces">
      <forall var="x"
in="$n/Foundation.Core.Namespace.ownedElement/Foundation.Core.
Association">
          <forall var="y"
in="$n/Foundation.Core.Namespace.ownedElement/Foundation.Core.
Association">
              <implies>
                  <and>
                      <equal
op1="$x/Foundation.Core.ModelElement.name/text()"
op2="$y/Foundation.Core.ModelElement.name/text()"/>
                      <equal
op1="$x/Foundation.Core.Association.connection/Foundation.Core.
AssociationEnd/Foundation.Core.AssociationEnd.type/*[1]/@xmi.idref"
op2="$y/Foundation.Core.Association.connection/Foundation.Core.
AssociationEnd/Foundation.Core.AssociationEnd.type/*[1]/@xmi.idref"/>
                  </and>
                  <same op1="$x" op2="$y"/>
              </implies>
          </forall>
      </forall>
  </forall>
</consistencyrule>
```

```
</consistencyruleset>
```

A.15 Type

```
<consistencyruleset>
<globalset id="types"
xpath="//Foundation.Core.Class[id(Foundation.Core.ModelElement.
stereotype/Foundation.Extension_Mechanisms.Stereotype/@xmi.idref)/
Foundation.Core.ModelElement.name/text()='Type']"/>

<consistencyrule id="t1">
  <description>
      A Type may not have any methods
  </description>

  <forall var="t" in="$types">
      <not> <exists var="x"
in="$t/Foundation.Core.Classifier.feature/Foundation.Core.Method"/>
      </not>
  </forall>
</consistencyrule>

<consistencyrule id="t2">
  <description>
      The parent of a type must be a type
  </description>

  <forall var="t" in="$types">
      <forall var="p"
in="id(id($t/Foundation.Core.GeneralizableElement.generalization/
Foundation.Core.Generalization/@xmi.idref)/Foundation.Core.
Generalization.supertype/*[1]/@xmi.idref)">
          <equal
op1="id($p/Foundation.Core.ModelElement.stereotype/Foundation.
Extension_Mechanisms.Stereotype/@xmi.idref)/Foundation.Core.
ModelElement.name/text()"
op2="'Type'"/>
      </forall>
  </forall>
</consistencyrule>
</consistencyruleset>
```

Appendix B State Transition Model Pseudo-Code

This Appendix contains the pseudo-code for state transition model of the software agent architecture. Each state is specifies the name, entry and exit actions, time delay caused by execution of the state in the model, and an inter-state transition. The transition indicates a destination state, reached after the transition is completed. Conditional transitions allow selection of one destination state out of a possible set of states, depending on values of one or more parameters. Transitions can have model execution delays assigned to them.

The pseudo-code corresponds to the state transition diagrams of all architectural components, constructed within the COVERS development environment. This pseudo-code was created from the C++ source code of the executable model of the architecture, generated within the COVERS environment.

B.1 Document Active Object

```
GoStraightToChange
Entry:
// if system flag of changes in model parameters is raised,
// request new value of event generation frequency
   if (change==1) ExecuteModificator();
Exit: // Generate a "Change" event straight away
Transition: -> Changed

Changed – document has been changed. Event must be raised.
Entry:
// Event has occurred. Send new event (notification message)
// "Check" through Port
  Port->Send(new TNotificationMessage("Check"));
Exit:
Transition: -> AskForRate

AskForRate
Entry: if (change==1) ExecuteModificator();
Exit:
Transition: -> Idle

Idle – document is awaiting next event.
Transition: -> Changed:
Transition delay: 1000*(Rate+Rate*TExponentDistr( Rate ))
// Calculation of a delay until the next change - randomised with
// an exponential distribution model parameter Rate in milliseconds.
```

B.2 Resource Interface Active Object

```
Idle
Entry:
Exit:
Transition: -> IdentifyChangedElements:
Condition: NumberOfChecks>0

IdentifyChangedElements
// Compares a backup copy with current version of the original,
// tree-wise, X – number of elements
Enter:  Z:=X*(1.0+TNormalDistr(X));
// Computes the number of "changed" elements
Transition: -> CheckForLocalVPRules
Delay: (2*X)*(1.0+TExponentDistr(1))*Tick
// Twice the randomised number of elements in this document (X)
// Tick is a time (in ms) required to access each tree element

CheckForLocalVPRules
// Scans the local rule database for consistency rules,
// relevant to changed elements
Transition: -> InstantiateMAToSearchGlobalRules
Delay: Z*Mlocal*2*(1.0+TExponentDistr(1))*Tick
// Twice the number of "changed" elements by the number of rules,
// randomised

InstantiateMAToSearchGlobalRules
Enter: TMAgent ag = new TMAgent(
new TNotificationMessage("Rules"),new TNotificationMessage(Z) );
// Instantiation of a new Mobile agent with a goal "Rules" (to search
// globally for relevant rules to changes in "Z")
Transition: -> GlobalRulesReady
Delay: TNotificationMessage m_in;
       while ((m_in = AgentInterface->Get())!="Rules")
         AgentInterface->Send(m);
// awaits the returning message from Mobile Agent with results
// from global Rules collection

GlobalRulesReady
Enter: Mlocal+=m_in.numberOfRetrievedRules;
// Added retrieved global consistency rules to local relevant rules

InstantiateMAForEachCriticalRule
Enter: for (int i=0; i++; i<Mlocal)
{
        TMAgent ag = new TMAgent(
      new TNotificationMessage("Check"), new TNotificationMessage(Z),
          m_in.rule[i]);
        ag.dispatch(IP_DOMAINSERVER);
      }
Exit:
Transition: -> ResInterfaceOperation

ResInterfaceOperation
Transition: -> ScanRequestedElements
Delay: if ((TNotificationMessage m=AgentInterface
->Get())=="ParamRequest")
Transition: -> LinksChange
Delay: if ((TNotificationMessage m=AgentInterface
->Get())=="LinksChange")
```

```
ScanRequestedElements
Enter:
// for each of consistency rules requested in the message,
// traverse the document tree and return elements
    TXDocument return;
    for (i=0; i++; i<m.numberOfRetrievedRules) {
      TXDocument consRule=m.rule[i];
      return.addElement(source.parse(consRule));
    }
Delay: Z*X*(1.0+TExponentDistr(1))*Tick
Exit: AgentInterface->Send( new TNotificationMessage("ParamRequest",
return));
Transition: -> ResInterfaceOperation (Deep History State)

OldLinksExist (LinksChange)
Transition: conditional
if (links!=null) -> NotifyRelated
else -> ChangeLinks
// Notify related document owners of a change in respective links
// so that the other end of the links could be updated

NotifyRelated
// Asynchronous notification of all "destinations" of existing links
// about the upcoming change
// asynchronous – no delay is required
Transition: -> ChangeLinks

ChangeLinks
// Update changed links within CLinks file
Enter:
     TXDocument CLinks;
     for (i=0; i++; i<m.numberOfLinks) {
      TXDocument consLink=m.link[i];
      CLinks.addElement(consLink);
    }
Transition: -> ResInterfaceOperation
Delay: m.numberOfLinks*2*(1.0+TExponentDistr(1.0))*Tick;
```

B.3 Mobile Agent Active Object

```
Initialisation (Active)
Enter: ConnectPort(MiddlewarePort, TAgentMiddleware.Port1);

// Dynamic MAgent object hooks up its communication ports to static
// architectural components
Exit:  InformationPort.Send ( new TNotificationMessage (
         "Migrate"+targetIPaddress));
// Sends "migration" message to itself, targetIPaddress is set by the
// dispatch() method after construction, is defined by a parameter
Transition: -> AtSourceDomain
Delay: AgentMiddleware.transmitDelay(this)*(1.0+TExponentDistr(1.0))
// Transmition of the whole MA to the host of the source Domain

RegisterAgent (AtSourceDomain)
Enter:
// Connect the InformationPort to domain agent's Port
ConnectPort(InformaionPort, TDomainAgent.Port);
// Pass agent's goal parameters (from constructor) to
```

```
// domain agent
InformationPort.Send(new TNotificationMessage("Check"), new
TNotificationMessage (Z), Rule);
Transition: -> FindDestinationDocumentURLs

FindDestinationDocumentURLs
Exit: Disconnect(InformationPort, TDomainAgent.Port);
      InformationPort.Send ( new TNotificationMessage (
         "Migrate"+m.targetIPaddress));
// Sends "migration" message to itself, targetIPaddress is received
// from the Domain agent
Transition: -> AtDestinationDomain
Delay: while (TNotificationMessage m = InformationPort.Get())!="URL");

RegisterAgentDest (AtDestinationDomain)
Enter: Connect(InformationPort, ResInterfaceAgent.AgentInterface);
// Connects to ResInterfaceAgent port for interaction
Transition: -> RetrieveDestinationElements

RetrieveDestinationElements
Enter: InformationPort.Send( new TNotificationMessage ("ParamRequest",
Rule.target );
// Requests target parameter values from target ResInterfaceAgent
localProcessingDone = true ;
Transition: -> CheckConsistencyRelation
Delay: while (TNotificationMessage m = InformationPort.Get())!=
"ParamRequest");
// waits until ResInterface sends back parameter values

CheckConsistencyRelation
Enter: TXDocument ConsistencyLinks =
       ConsistencyLinksGenerator(Z, m.paramValues, Rule);
// Runs link generator with parameters: source element values,
// destination element values, consistency rule
Transition: -> UpdateTargetLinks
Delay: sizeof(Z)*sizeof(m.paramValues)*sizeof(Rule)*
(TExponentDistr(1.0)+1.0) * Tick

UpdateTargetLinks
Enter: InformationPort.Send(new TNotificationMessage("LinksChange"),
       ConsistencyLinks );
Exit: Disconnect(InformationPort, ResInterfaceAgent.AgentInterface);
// Leaving AtDestinationDomain, thus port will have to be re-connected
// after migration
      InformationPort.Send(new TNotificationMessage("Migrate"),
      IP_HOMEDOMAIN);
// Migration back to source domain, IP_HOMEDOMAIN constant has been
// initialised at agent construction
Transition: -> UpdateSourceLinks

UpdateSourceLinks (AtSourceDomain)
Enter: Connect(InformationPort, ResInterfaceAgent.AgentInterface);
       InformationPort.Send(new TNotificationMessage("LinksChange"),
       ConsistencyLinks );
Transition: final state

Active
Enter: TMAgent(TXDocument Z, ConsistencyRule Rule);
// constructor called as normal upon creation
Transition: -> ReplicationDetected (Redundant)
Delay: while ((TNotificationMessage m=InformationPort.Get())!="Redundant");
Transition: -> branch_state (AgentDataReceived)
```

```
Delay: while ((TNotificationMessage m=InformationPort.Get())!="AgentData");
Transition: -> Migrate
Delay: while ((TNotificationMessage m=InformationPort.Get())!="Migrate");
Transition: -> branch_state (Failure)
Delay: while ((TNotificationMessage m=InformationPort.Get())!="Failure");
Transition: ~TMAgent(); // deconstructor

ReplicationDetected (Redundant)
Enter:
// Connect to Middleware and transmit own "experience"
// if target paramValues have already been collected, transmit
// them with a target documentURL + source values + home documentURL
if (paramValues!=null)
MiddlewarePort.Send(new TNotificationMessage ("AgentData"),
m.agentIPAddress, paramValues, targetIPaddress, Z, IP_HOMEDOMAIN);
// else if only target URL is known
else if (targetIPaddress!=null)
MiddlewarePort.Send(new TNotificationMessage ("AgentData"),
m.agentIPAddress, paramValues, Z, IP_HOMEDOMAIN);
// else only source values + home documentURL
MiddlewarePort.Send(new TNotificationMessage ("AgentData"),
m.agentIPAddress, Z, IP_HOMEDOMAIN);

Disconnect(InformationPort);
Transition: -> NotifySource

NotifySource
Enter: MiddlewarePort.Send(new TNotificationMessage ("Redundant"),
IP_HOMEDOMAIN, paramValues, targetIPaddress);
Transition: -> TerminateInstance

TerminateInstance
Enter: this.~TMAgent(); // de-constructor

branch_state (AgentDataReceived)
// add the target URL received from collaborating agent
// to the list of target URLs
Enter: targetIPaddress = m.targetIPaddress;
Transition: -> deep_history_state (Active) "AdditionalSourceNoted"
Delay: do { TNotificationMessage m=InformationPort.Get() }
       while (m.paramValues!=null);
Transition: -> CompareToOwn

CompareToOwn
// Compares received parameter values to own
Enter: TXDocument d=TreeDiff(paramValues, m.paramValues);
Transition: -> ReplyDifference
Delay: sizeof(paramValues)*sizeof(paramValues)*(1.0+TExponentDistr(1.0));

ReplyDifference
Enter: MiddlewarePort.Send(new TNotificationMessage("AgentData"),
m.agentIPaddress, d);
Transition: -> deep_history_state (Active)

Migrate
Condition: localProcessingDone == true;
Enter: localProcessingDone = false;
```

```
Transition: AtDestinationDomain
Delay: AgentMiddleware.transmitDelay(this)*(1.0+TExponentDistr(1.0))

branch_state (Failure)
// if failure caused at target domain, go back to home domain
Transition: -> AtSourceDomain
Delay: while (currentIP!=targetIPaddress);
Transition: -> ReportToSource

ReportToSource
Enter: MiddlewarePort.Send(new TNotificationMessage("Failure"), this);
Transition: this.~TMAgent(); // de-constructor
```

B.4 Domain Agent Active Object

```
Idle
Enter: TNotificationMessage m;
m = Port.Get();
Transition: -> Register

Register
Transition: -> IncQueues:
if (Port->Count()) Input=strdup((Port->Get())->GetText());

IncQueues
Enter:
if (!strnicmp(Input,"Rules",strlen("Rules"))) {
  RulesQueue++;
  EventsQueue++;
} else if (!strnicmp(Input,"Name",strlen("Name"))) {
  NamesQueue++;
} else if (!strnicmp(Input,"Events",strlen("Events"))) {
  EventsQueue++;
} else if (!strnicmp(Input,"Agents",strlen("Agents"))) {
  AgentsQueue++;
};
Transition: -> Deep history (Active)

Active
Nothing
Transition: ->Processing

Processing:
Transition: conditional:
If (RulesQueue>0) -> RulesDatabase
If (AgentsQueue>0) -> AgentsList
If (EventsQueue>0) -> EventsList
If (NamesQueue>0) -> NameLookup

RulesDatabase
Enter: RulesQueue--;
RulesDatabaseInterface->Send  ( new TNotificationMessage("Request") );
Delay: Mdomain*2*(5.0+TExponentDistr(1))
Transition: -> Nothing
Port->Send (new TNotificationMessage("Rules"));

AgentsList
Enter: AgentsQueue--;
redundant = AgentsListInterface->Send (new TNotificationMessage("Request"));
```

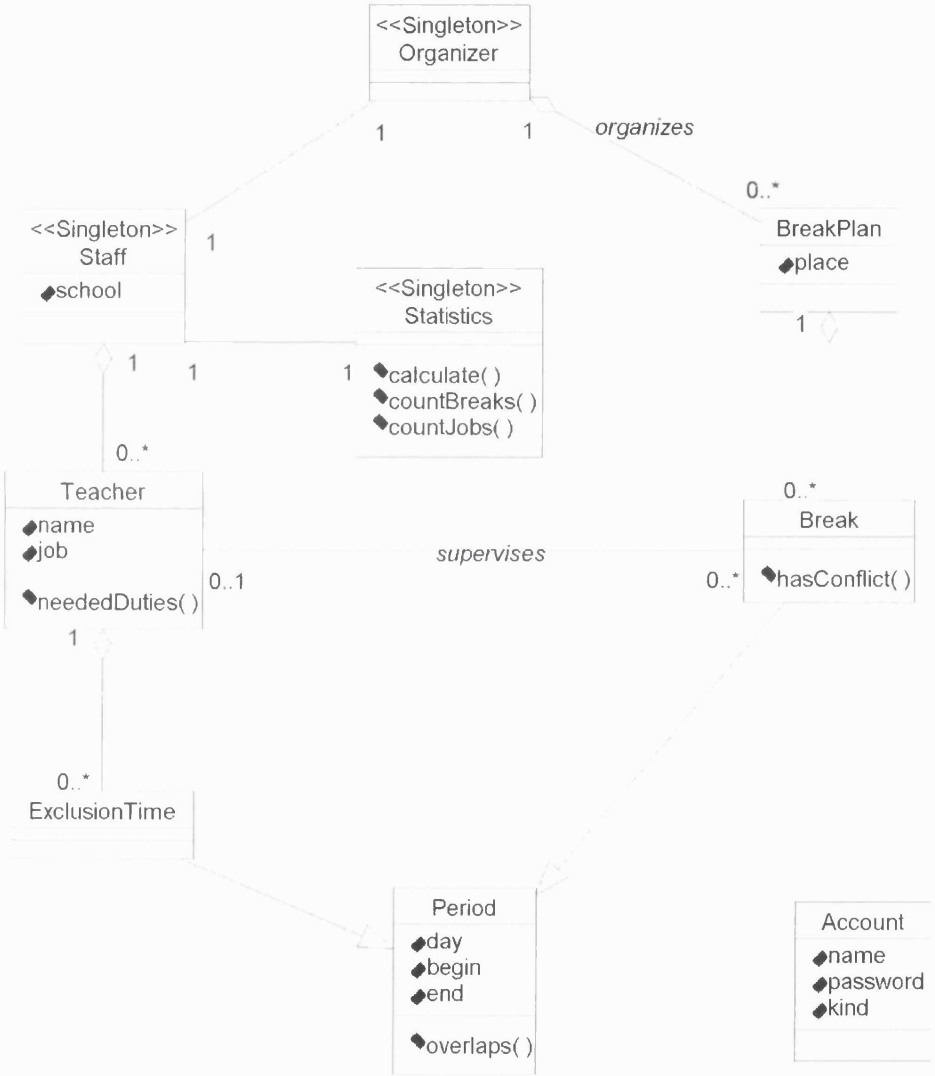
```
Delay: LengthTable*(1.0+TExponentDistr(1))
Transition: -> Nothing
If (redundant!=null) Port->Send(new TNotificationMessage("Redundant"+m));

EventsList
Enter:EventsQueue--;
EventsListInterface->Send (new TNotificationMessage("Request"));
Delay: LengthDocumentsTable*(1.0+TExponentDistr(1))
Transition: -> Nothing

NameLookup
Enter: NamesQueue--;
NameLookupInterface->Send (new TNotificationMessage("Request"));
Delay: LengthDocumentsTable*(1.0+TExponentDistr(1))
Transition: -> Nothing

Transition: RelayAllDBReplies (Active -> Idle)
Condition: (AgentsListInterface->Count()) ||(RulesDatabaseInterface-
>Count())
||(EventsListInterface->Count()) ||(NameLookupInterface->Count())
Action: if (AgentsListInterface->Count()) Port->Send(AgentsListInterface-
>Get());
if (RulesDatabaseInterface->Count()) Port->Send(RulesDatabaseInterface-
>Get());
if (EventsListInterface->Count()) Port->Send(EventsListInterface->Get());
if (NameLookupInterface->Count()) Port->Send(NameLookupInterface->Get());

Transition: NewPortInput (Active->Register)
Condition: (Port->Count())
```

B.5 Agent Middleware Active Object

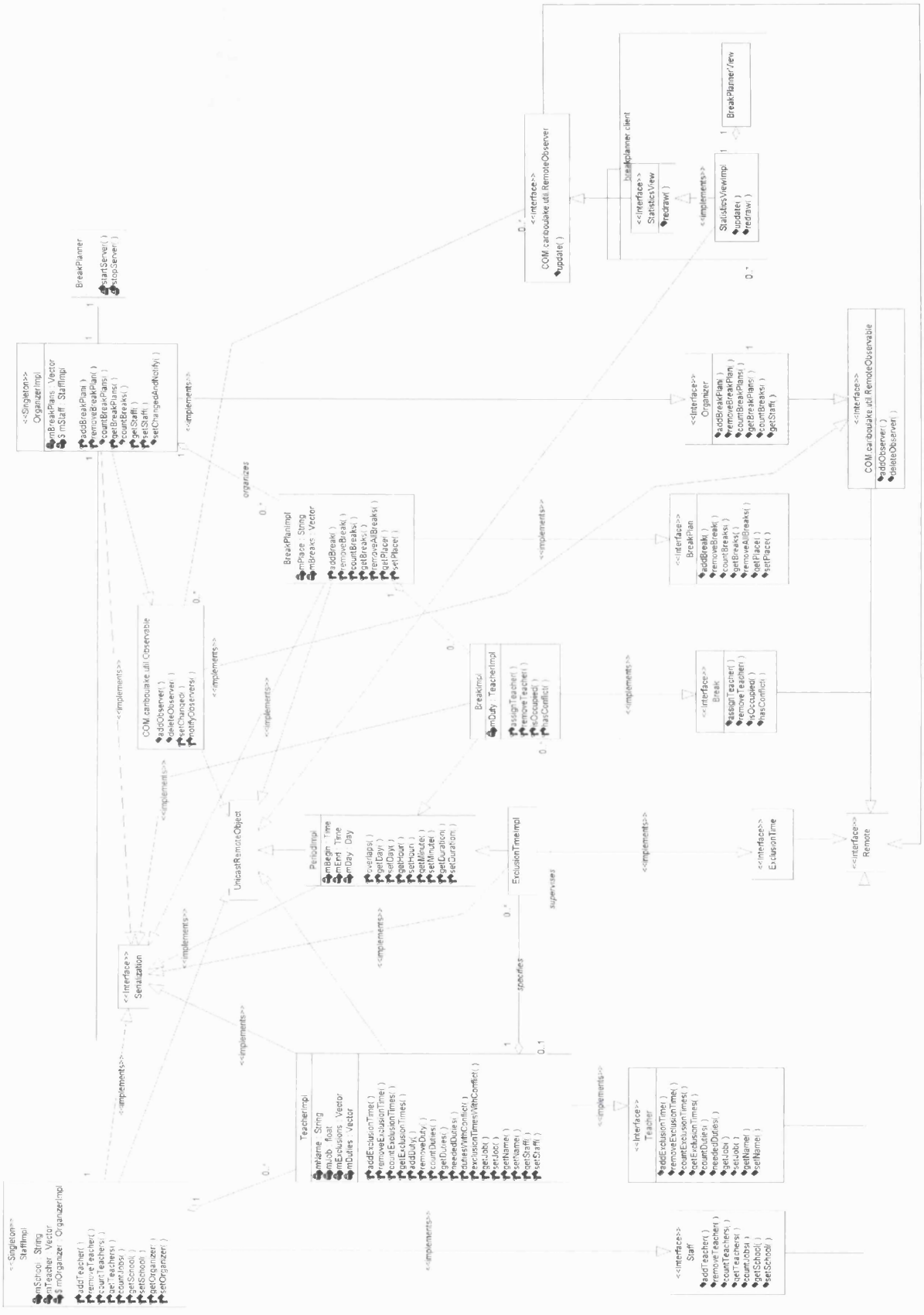
```
Idle
Enter: TNotificationMessage m;
m = Port1.Get();
if (m==null) { fromPort=0; m = Port2.Get(); }
else fromPort=1;
if (m==null) { m = GatewayPort.Get(); if (fromPort==0) fromPort=3; }
else fromPort=2;
Transition: -> Transit
Delay: AgentMiddleware.transmitDelay(m)*(1.0+TExponentDistr(1.0))

Transit
Enter:
      Switch(fromPort) {
      Case 0: break;
      Case 1: Port2.send(m); GatewayPort.send(m); break;
      Case 2: Port1.send(m); GatewayPort.send(m); break;
      Case 3: Port1.send(m); break;
      }
Transition: -> Idle
// Depending on direction Port1->Port2+GatewayPort, or Port2 or
// GatewayPort -> Port1, the corresponding transition Port1_Count or
// Port2_Count is followed in the state chart
```

Appendix C Break Planner Application: Selected UML Diagrams

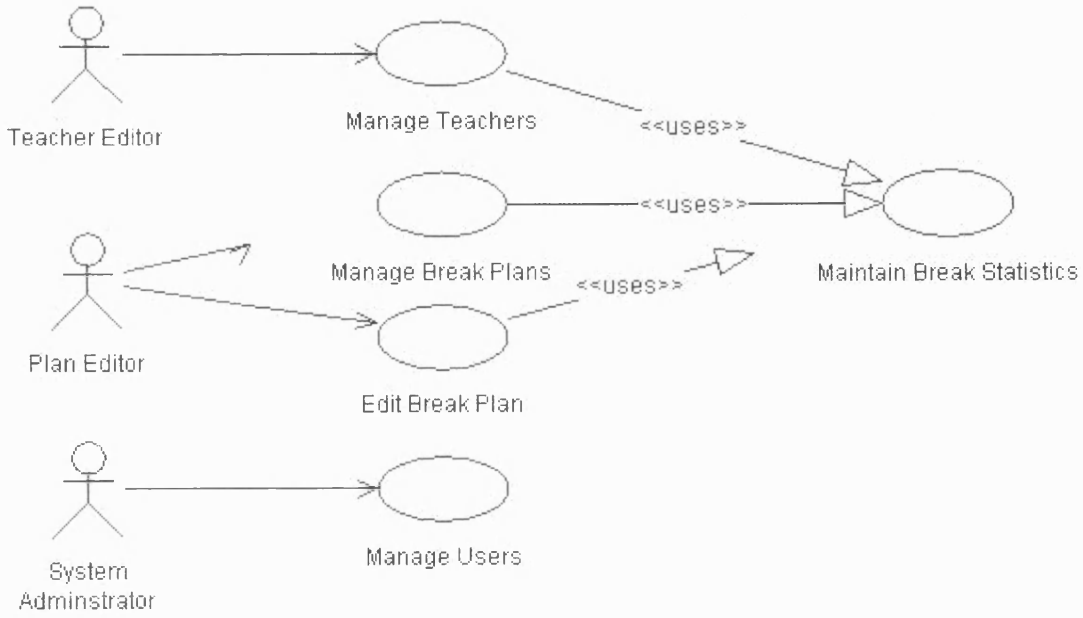
C.1 Analysis Class Diagram



C.2 Distributed Break Planner Application Class Diagram



C.3 Break Scheduler Application - Use Cases



Appendix D Review of Mobile Agent Frameworks

D.1 D'Agents

D'Agents [Rus, et al. 1997] is a multi-language agent system being developed at Dartmouth College, Hanover, Massachusetts. The first versions were released in 1994, based on a proprietary script language [Kotay and Kotz 1994], and in 1995 using Tcl [Ousterhout 1994] under the project name TIAS [Harker 1995]. From 1995 to 1997 the system was renamed Agent Tcl since Tcl was the only supported programming language [Gray 1995, Gray 1996, Gray, et al. 1997, Kotz, et al. 1997]. Since 1998 the project is known as D'Agents and now supports Java, Python and Scheme programming languages [Gray, et al. 1998].

From the start, D'Agents were aimed to provide support for strong mobility. In D'Agents, a call to a single function, *agent_jump,* triggers an agent's migration between network hosts. The D'Agents interpreter packages up the complete state of the agent and sends it to a destination machine. D'Agents offer support for strong mobility in Java, however a modified Java Virtual Machine (JVM) is required to run the code. This disadvantage may limit usability of D'Agents support for Java, as the proprietary interpreter may become out of date with evolution of the Java language.

D'Agents consists of four levels. The lowest level is an interface to transport mechanisms, the next level is a server that runs on each machine, while the top level consists of the execution environments, one for each supported agent language. This last level comprises the agents themselves, which execute in the interpreters and use facilities provided by the server.

D'Agents implements the following basic concepts: agents, server, migration, and messaging.

Agents

An agent in D'Agents is a program written in any of the supported script languages, and run on top of the framework's interpreter. Each agent is being assigned a unique name in the system.

Servers

Like most mobile agent systems, D'Agents run a server on each machine that the agent can visit. On a server, each agent runs as a separate process in the interpreter. D'Agents server keeps track of the agents that are running on its machine and maintains a hierarchical namespace for agents, thus allowing agents to send messages to each other within this namespace. D'Agents can back up their states to a nonvolatile store provided by the server. In case of server failure and a subsequent restart of interpreters, the server restores the agents' states from the storage, and execution of the agents continues.

Mobility

Agents can migrate from their home server to a given list of destinations. Possessing strong mobility, D'Agents call a single function, which triggers the capturing of a complete agent's state and forwarding of this state to the destination machine.

The D'Agents server is in charge of looking after agent dispatch and arrival. Incoming agents are authenticated with the identity of the agent's owner and passed on to the interpreter for execution. After the agent's state information is loaded into the interpreter's execution environment, the agent is restarted at the exact point at which it left off. Agents can be transferred over the network by plain TCP/IP sockets or email.

Communication

Agents are allowed to send messages to each other within the server's namespace. Two reserved types of messages are defined: an event message and a direct connection. An event message provides synchronous notification of an important occurrence while a connection message requests or rejects the establishment of a direct connection. A direct connection is a named message stream between agents and is more convenient and efficient than message passing for long interactions. The server buffers incoming messages, selects the best transport mechanism for outgoing messages, and creates a named message stream once a connection request has been accepted.

Agent can also communicate directly locally or remotely through procedure calls using a special Agent RPC defined in AIDL (Agent Interface Definition Language).

Security

Security in D'Agents focuses on the protection of the host against malicious agents; server control of the system resources is achieved via stationary agents - resource managers. Untrusted Tcl agents are run in a special Tcl interpreter, whilst all system resources are managed by a trusted process – a trusted interpreter. The latter asks an appropriate resource manager to determine if the visiting agent should have access to the resource. The trusted interpreter then enforces the manager's policy decision, either proceeding with the resource access or throwing a security exception back to the untrusted interpreter. Agents and messages, which are transferred over the network, are encrypted and signed to maintain their privacy and to authenticate the agent to the new host.

D.2 Mole

The mobile agent system Mole [Strasser, et al. 1997a] was developed at the University of Stuttgart, Germany [Mole 1999] in 1995. It was one of the first mobile agent systems implemented in Java [Strasser, et al. 1997b].

Mole only supports weak mobility, which is justified by the advantage that Mole gives through its ability to work with any standard, un-modified JVM [Baumann, et al. 1997]. The Mole approach contrasts with D'Agents, where strong mobility has been achieved through modifications of the JVM. Deployment of weak or strong mobility in mobile agent systems is still an open question for the agent community, thus both approaches co-exist and allow the developers to exploit advantages of the either approach.

Mole provides the notions of places, the executing environment, where user agents are able to meet and communicate. They can interact with the underlying operating system resources via service agents, which are always stationary. A number of communication mechanisms are supported, including badges, sessions and events. In addition to places and messaging services, Mole includes a resource manager for accounting and resource control, a simple local directory service where agents can find other agents, and a graphical agent monitor for examining agents, places, and messages sent between them.

Mole overcomes the Java applet restrictions of being able to connect only to the server, from which applets are loaded, by using relay components in the server engines. These relays redirect communication data to or from the browser engine to remote server engines. Mole implements the following basic concepts: agent, place, migration, badge, session, and shadows.

Agents

Mole agents are clusters of self-contained objects, containing references to their execution environment, through which Mole framework commands are executed. Agents move from place to place to access services that places provide and to meet other agents.

Agents in Mole can be identified in two ways: through globally unique identifiers, generated by the system, so-called *badges*. Badge identifiers represent the role (i.e., goal) of an agent at a given time, and so long as the agent provides functionality associated with this role, it "wears" the badge. An agent can wear several badges at the same time. Badges are used to group cooperative agents performing a user task and to allow communication between them.

Servers

An agent system consists of a set of places, each one assigned to an agent community, providing access to a set of services and often implementing a certain "pricing" policy. A place is a server engine that represents an execution environment for agents where they can execute and communicate with other agents, use resources and services of the underlying system. The system agents are in charge of negotiating the use of resources; the Master Control Process schedules all execution threads in the Mole system. Mole supports disconnected operation through the concept of associated resources, if the system is not connected to the network permanently.

Each place has a simple local directory service, where agents register the services that they are capable of. Agents wishing to use other agents' services retrieve the identifier of a suitable agent from the directory service, and establish communication directly with the target agent.

Mobility

Mole provides weak mobility through use of Java and RMI. The *MigrateTo* command of the framework takes a URL of the destination place server, and results in serialization of the agent's state, transmission to the destination and subsequent agent re-instantiation. Code base mobility is supported: if any of the Java classes needed are not available locally, the target location requests these classes either from a code server or from the source location if no code server has been specified. When migrated agent resumes its thread successfully, a confirmation message is sent back to the source location. The source location then removes any resources pertaining to the agent from the system.

Agents can be terminated remotely by use of a shadow protocol [Baumann and Rothermel 1998]. When an agent is created, a corresponding *shadow* is generated at a place. After agent's migration or at regular intervals of time, the Mole system updates the shadow links for all agents. If the shadow no longer exists, the agent is declared to be an orphan and is removed.

Communication

Agents wishing to communicate with each other must establish a session before the actual communication can be started. After the session setup, the agents can interact by remote method invocation or by message passing. Communication is not restricted to agents at the same server place. Message forwarding is not supported: communication sessions must be ended and restarted after an agent migrates to the next place. Badges facilitate selection of target agents for communication.

Security

Mole enforces the "sandbox" security model, in which mobile agents have very limited access to the underlying system. Stationary service agents control system resources, maintaining security and providing abstractions of the resources to Mole agents operating in the agent system.

D.3 Hive

Hive is a distributed agent platform and a decentralized system for building distributed applications, making local system resources available to such network applications, and taking advantage of mobile code [Minar, et al. 1999]. Hive developers at the MIT Media lab, are using it to provide the infrastructure for connecting their numerous "Things That Think" [Gershenfeld 1999] research initiatives. Hive is built using the standard Java features of object serialization and interpretation used by so many mobile agent frameworks and therefore supports weak mobility.

The Hive architecture consists of the following three abstractions: cells, shadows and agents. A cell is the executing environment in which agents are hosted. Cells also contain shadows, which are placeholders for local resources, for example a display or printer. The designers of Hive have made particular efforts to address the problems of agent description and Hive supports both a syntactic and semantic ontology. Inter-agent communication in Hive has been achieved by using RMI as the communication mechanism. This allows the methods of Hive agents to be executed remotely. While this approach is simple and uses built in capabilities of the Java language, it has the disadvantages of loss of control and security. In the author's opinion, it also blurs and lowers the abstraction level of the mobile agent to one of merely a mobile object. If an agent's methods can be called and executed remotely, then any notion of autonomy for the agent has been lost. Hive thus embodies a hybrid abstraction, drawing elements from the autonomous agents research arena, and from contemporary RPC distributed systems.

This hybrid abstraction has caused the Hive team some considerable difficulties in achieving their goals [Minar, et al. 1999]. However, the ontological descriptions supported by Hive are superior to many if not all of the other frameworks reviewed.

D.4 Concordia

Concordia is in development since 1997 at the Mitsubishi Electric Information Technology Center America (MEITCA). A framework for developing and executing mobile agents in Java [Castillo, et al. 1998, Koblick 1999, Walsh, et al. 1998, Wong, et al. 1997], Concordia has been used in a variety of applications, including an electronic auction house [Huai and Sandholm 1999].

Concordia consists of the following components: a server for executing and transferring agents, and an administration manager for remote server administration. Concordia implements the following basic concepts: agent, context, service point, collaboration and policies.

Agents

Concordia agents migrate between network nodes and interact with information agents locally at each node. As components of the same application, agents can form one or more collaboration units, known as agent groups. Concurrent execution of collaborative units allows tackling complex tasks, and collaboration allows agents to correlate their results and adjust goals accordingly.

Servers

Concordia server consists of various modular components that provide an integrated environment for mobile agents. The agent manager controls the creation, destruction and execution of agents. A queue manager schedules transport of agents over the network. Concordia servers provide services through one or more stationary information agents, which act as facilitators, or service bridges. These service bridges effectively perform a role of interfaces and make it possible for agents to interact with existing native applications or legacy systems. A persistent store manager ensures that agents recover successfully from system crashes. An event manager provides for the collaboration of agent groups, a security manager controls the local security policies, and a directory manager enables agents to locate the application servers they wish to interact with on each host.

Mobility

Each server provides for agent mobility through a queue manager. Fault-tolerant transmission is guaranteed through caching of a migrating agent in the local system message queue until it is confirmed to have been received by the remote host. This store-and-forward mechanism makes Concordia suitable for disconnected operation. Like most mobile systems, Concordia transfers the agent's internal state in migration. Concordia supports a class loader, which packages Java byte code into a special data structure that travels with the agent's state. Migration process runs on Java RMI and can be secured with SSL.

The migration schedule of an agent is described in its itinerary, which is composed of multiple destinations. Each itinerary record describes a location, to which the agent is to travel, and the work the agent has to accomplish at that location. The agent can modify its itinerary, but the itinerary is not part of the agent. This mechanism allows multiple entry points into agents executed at multiple locations.

Communication

Concordia provides two forms of inter-agent communication: distributed notification of events and agent collaboration. Distributed events are scheduled and managed by the event manager. Before an agent can receive selected events, it must register with the event manager by sending a list of events it is interested in receiving and a reference to a location where it wishes events to be sent. Agent collaboration, which makes use of this event mechanism, allows agents to interact in a group. Such collaboration requires each agent in the group to participate and to access a collaboration point. Collaboration concurrency is resolved through a blocking scheme: when an agent arrives at the meeting point, it posts the results of its execution to the agent group and blocks until all group agents have posted their results as well. In addition to "strong" collaboration with a blocking scheme, meeting points can be set up to support "weak" collaboration, where collaboration is allowed to continue even if some of the agents in the group fail to arrive at the collaboration point.

Security

Security in Concordia is provided at numerous levels, including agent storage protection through encryption, transmission protection with digital certificates for identity verification and symmetric key exchanges to authenticate sender and receiver. The Concordia resource protection model extends the capabilities of the Java environment and provides flexible user-based access control.

D.5 Grasshopper

Grasshopper [Grasshopper 1999] by IKV++ GmbH is a Java mobile agent development and runtime platform built on top of CORBA [IKV 1999a, IKV 1999b]. It is the first mobile agent system that is compliant with the CORBA Mobile Agent System Interoperability Facility (MASIF) [Milojicic, et al. 1998].

Grasshopper framework includes the distributed agent environment, providing the basic system functionality, region registry registers for all agents and places in a region, and the platform management tools for agents, places, and monitoring of events and threads.

Grasshopper implements the following basic concepts: agents, agencies, places, regions, and authorities.

Agents

Agents in Grasshopper have an authority (i.e., a person or an organization), for which the agent acts. Agents are named by their authority, identity and an agent type; a combination of these uniquely identifies the agent. An agent's identity identifies an agent's instance among all agents of the same authority.

Servers

A server in Grasshopper (an "agency") contains the one or more core agencies and one or more places for visiting agents. The core agency provides minimal functionality required by the server in order to support execution of agents. The following services can be made available in the agency: a communication service, a registration service for all local places and agents, a management service for monitoring and control of agents and places of the agency, a security service, and a persistence service to store agents and places on a persistent medium.

Mobility

Grasshopper supports weak mobility, agent serialization and transport use the underlying CORBA services, Java RMI, or are based on socket connections. To achieve secure communication, RMI and the plain socket connection can be protected with SSL.

Communication

Communication service is part of the core agency and is responsible for all remote interactions that take place between the distributed components of Grasshopper. Supported are: location-transparent inter-agent communication, agent migration, and localization of agents by means of the region registry. The communication service provides synchronous and asynchronous communication, multicast communication, and remote method invocation.

Security

Grasshopper security engine extends the features of JDK 1.2 by making use of identity-based group-based access control policies. Access controller manages agent's access rights and is capable of identifying the agent's owner behind the agent's authority. With this information it contacts the policy object to extract the set of permissions for the owner, which are then applied to the agent.

D.6 MESSENGERS

MESSENGERS [MESSENGERS 1999], developed at the University of California, Irvine, is an autonomous, object-based environment intended primarily for development of dynamic applications, such as distributed simulations [Bic, et al. 1997, Bic, et al. 1996, Fukuda, et al. 1997, Fukuda, et al. 1998]. MESSENGERS classifies itself as a system for self-migrating threads.

MESSENGERS programming language is built on a subset of C and requires an interpreter, running as a daemon at each node in the network, and a set of libraries. The language contains statements belonging to one of the following classes: computation, navigation, and system function invocation.

MESSENGERS implements the following basic concepts: messengers agents, nodes, and hops.

Agents

Agents (Messengers) are compiled into byte code, interpreted at runtime. The agent framework provides a number of system functions, including explicit and implicit cloning (multithreading).

Servers

Each network node runs the Messenger interpreter, and can contain one or more logical nodes. Messengers can call precompiled C functions at logical nodes or spawn separate processes. Messengers can create, change or delete logical nodes.

Mobility

Messengers navigate through a logical network based on their schedule. A number of addressing methods exist: migration destinations can be specified by several parameters, including logical and physical nodes, link names, and wildcards. At migration, a replica of the Messenger is propagated to all nodes that match the specification.

For every Messenger agent, an interpreter continues to run the code until it encounters a navigational instruction. At that point the interpreter passes the Messenger to the appropriate destination nodes of the logical network. If destination logical nodes are located at the same physical node, the Messenger is simply moved to the appropriate queue, where it awaits its turn, as the interpreter is multiplexed between serving agents at the different logical nodes. If the destination is found at a different node, the Messenger is sent there using TCP/IP sockets.

Communication

Messengers communicate by invocation of precompiled C functions at logical nodes.

Security

MESSENGERS provides virtually no security features, since its general purpose is to use the power of local area networks to speed up computational tasks.

D.7 Tacoma

Tromso And COrnell Mobile Agents (TACOMA) [TACOMA 1999] was developed at the University of Tromso, Norway and Cornell University, USA with the focus on fault-tolerance, scheduling and management, security, and accounting of agents [Johansen, et al. 1995]. The first implementation was built in 1993 on the basis of experience from the StormCast project [Johansen, et al. 1998] and was entirely implemented using Unix and C. Current versions support various operating systems, and a number of programming languages such as C, C++, Java, ML, Perl, Python, Scheme, and Tcl/Tk.

TACOMA builds on two following mechanisms: meeting and folders. The source of a program with all corresponding data is stored in folders and moved from host to host through stationary firewall agents on each host.

TACOMA implements the concepts of agent, folder, briefcase, and cabinet.

Agents

A TACOMA agent has a collection of folders associated with it - a "briefcase". The agent "carries" the briefcase while moving around the network. A *code* folder contains one or more scripts in a supported language, and a *data* folder has the data, associated with the code from the code folder. In addition to mobile briefcases, an agent can set up stationary folders called file cabinets, which are used as permanent data repositories on a host.

Servers

Servers provide transport services for migrating agents through a firewall agent (tac) and a re-launching agent (tac exec), which passes agent code on to the appropriate interpreter depending on the language used by the agent. Such architecture provides effective agent migration support and doesn't compromise security features of the agent framework.

Mobility

TACOMA agents migrate from one host to the other as they are a part of the "briefcase". An agent stores required data in the briefcase and invokes TACOMA migration command. This opens a TCP connection with the firewall agent at the destination site, URL of which is specified in the *host* folder of the briefcase. After migration, the receiving tac firewall agent *meets* the TACOMA agent with whatever agent identified in the *contact* folder of the briefcase. This mechanism can facilitate efficient collaboration between agents.

Communication

Instead of communicating by means of messages, agents use briefcases with the data they wish to share. One significant drawback of TACOMA is that place-to-place communication is not supported: if the agents are located at different places, they must co-locate by moving before they can communicate. In other words, communication between active agents can only be performed using shared cabinets and can only involve agents residing on the same server. As described above, *host* and *contact* folders of briefcases are used to specify the URL of the migration destination place and the ID of the agent to be contacted.

Security

Use of firewall agents in TACOMA allows to implement a number of security mechanisms. One of the ways to restrict access to untrusted agents is to specify a set of TACOMA hosts, from which agents are accepted. Further, execution of agents can be confined in a subtree of the file system.

D.8 Telescript

Developed in the early 1990s by General Magic, Telescript is a proprietary, object-oriented, type-safe language conceived for the development of distributed applications [White 1997]. It was the first commercial implementation of the mobile agent concept and has been the most widely adopted agent system for years in the industry [Doemel 1996]. Indeed, General Magic first coined the term mobile agents and applied for a patent on the concept in 1993. In 1997, it received US patent 5,603,031 [Kiniry and Zimmerman 1997].

Telescript framework consists of three major components: anent programming language and an interpreter, engines and communication protocols, which are also used for agent migration. One of the driving factors for Telescript's language design was support for strong mobility, the other – support for higher levels of security. Telescript implemented compilation of agent source files into byte code, which is interpreted at run time, before Java became available. Success of Java forced General Magic to re-develop Telescript, and release Odyssey [Kiniry and Zimmerman 1997], a new agent system implemented solely in Java that uses the same design framework.

Telescript implements the following basic concepts: agent, place, travel, meeting, authorities, and permits.

Agents

All agents in Telescript must are derived from the Agent parent class. An agent has several public methods that can be accessed during a meeting. Each agent has a *telename*, which is data that denote the agent's identity as well as its authority, and a *ticket*, which is data that specify the agent's destination and the other terms of the trip (for example, the means by which it must be made and the time by which it must be completed).

Servers

The Telescript engine manages and executes stationary agents, called places, as well as mobile agents, which visit these places and access services, provided by the places. Each place has a telename, denoting the place's identity and authority.

Mobility

In Telescript agents decide when to travel from one place to another by executing Telescript's *go* instruction. The instruction takes a *ticket* as a parameter; the latter gives a telename and teleaddress (URL) of the destination.

The Telescript agent migration protocol involves the two places concerned in authentication through firewall agents (at the lower level) and communication of the migrating agent's briefcase between from one place to the other (at the higher level). Fault tolerance in migration, is implemented through exceptions: the agent receives an exception raised by the invoked *go* instruction and can handle it appropriately. When migration succeeds, the agent's next instruction is executed at its destination.

Communication

Agents can interact locally at the place by executing the *meet* instruction of Telescript. As a result of the meeting, each agent obtains a reference to the other. Then, both method invocation and shared memory mechanisms can be used for communication [Vigna 1997]. The meet instruction requires a petition, a reference to the agent to be met a number of meeting parameters, such as the time by which the meeting must begin. Unlike TACOMA, Telescript agents always supported remote connection between agents, which is carried out through a *connect* command, which uses the same parameters as *meet* command.

Security

Telescript's security mechanisms are based on authorities and permits. Authority relates to the individual or organization in the physical world that the agent represents. Authorities let agents and places shape interactions with one another in three different ways. First, a place can request authority of any agent that attempts to enter it, and can arrange to admit only agents of certain authorities. Secondly, an agent can discern authority of any place it visits, and can arrange to visit only places with certain authorities. Finally, an agent can discern authorities of any other agents with which it meets or to which it connects, and can arrange to meet with or connect to only agents of certain authorities.

Authorities limit capabilities of agents and places by assigned *permits*. Permits protect authorities by limiting the effects of malicious agents and places. An agent's permit can specify the maximum lifetime of this agent, its size or computation/execution time allowance. Temporary permits can be voluntarily imposed by agents or places on themselves, which enable the agent to plan and adjust its actions according to the resources available. The agent is notified if it violates one of temporary permits, rather than being destroyed after having exceeded the permanent permit.

D.9 Voyager

Voyager [Voyager 1999a] is an agent-enhanced Object Request Broker (ORB) under development since 1996 by ObjectSpace, Inc [Voyager 1997, Voyager 1999b]. Voyager uses the Java programming language to provide a universal solution for distributed object interaction techniques. The main focus of Voyager is on programming universal clients and servers as well as acting as a gateway for CORBA, Java RMI, DCOM and Java Beans [Sun 1999]. Voyager should really be considered as a Java-based messaging broker, which possesses capabilities from the mobile agent field. This allows programmers to create network applications by choosing between traditional and mobile distribution technologies; because of this advantage, Voyager has been a successful product.

The Voyager ORB facilitates the creation of traditional distributed object-oriented systems using CORBA, Java RMI and DCOM as well as mobile agents. The Voyager security framework offers a lightweight security implementation, support for secure network communications via SSL adapters, and firewall tunneling using the industry standard SOCKS and HTTP protocols. Voyager Transactions delivers full OTS-compliant distributed transactions support, including a two-phase commit protocol and a one-phase commit JDBC adapter. The Voyager Application Server offers an Enterprise JavaBeans development environment that decouples application logic from systems programming logic.

Voyager implements the following basic concepts: agent, application, secretary, and messenger.

Agents

Each agent in Voyager consists of a virtual stationary agent and the mobile agent itself. The virtual agent acts as a reference to the mobile agent and resides with the hosting application. The application then communicates with a mobile agent through its virtual agent, which provides two-way messaging between the application and the mobile agent. The virtual agent is generated automatically at runtime upon creation of the mobile agent.

Each agent receives a globally unique identifier and a defined life span. The virtual agent can be programmed to send a lightweight "ping" to the mobile agent at regular time intervals. Voyager then uses these pings together with the agent's life span to determine when the agent can be garbage-collected.

Servers

Servers in Voyager are called applications. They provide the runtime core and services such as the local service registry and can host various objects that provide services to visiting agents. Applications can store a snapshot of any agent and subsequently act as code servers by using the class loader with Voyager's built-in HTTP support.

Mobility

An agent can move from one application to another by sending itself a *move* message, where not only the destination application address can be specified, but also the name of the member function that should be executed after migration. If the agent moves to the new destination, it can still be located by using the last known address. Before moving, it creates a special secretary agent, which forwards all relevant messages to the new location. The virtual agent at the application host then updates the mobile agent's location every time a response message is originating from a host, which is different to the host of the previous message. Subsequent messages are sent directly to the agent at its new location. Secretaries are automatically destroyed by Voyager's distributed garbage collector when they are no longer used.

Communication

Voyager's synchronous messages are used by default: they block until the message is received and a response message has been returned. One-way messages return to the sender immediately after delivery and do not allow return values. Future messages return to the sender without waiting for the message to be confirmed as delivered; they return a placeholder that can be used to retrieve the returned value later by polling, blocking, or waiting for a callback.

Voyager supports multicast and publish/subscribe messaging models through its scalable architecture for a message and event replication system called Space. Messages in Voyager are delivered by lightweight mobile agents called messengers.

Security

The Voyager security manager extends the standard Java security manager and distinguishes between "native" and "foreign" agents, or objects. Native objects are those, whose classes reside in the application's CLASSPATH. Foreign objects are objects, whose classes were loaded across the network from another host. By default, foreign objects are restricted and have nearly the same permissions as a Java applet, while the Voyager security manager allows native objects to fully access the functionality provided by the JVM.

Appendix E Quantitative Performance Model of the Architecture

E.1 Introduction

In this appendix we present a model for estimation of performance of a collaborative multi-agent consistency check in the proposed software agent architecture.

E.2 Assumptions

N - Number of hosts, containing one or more documents, participating in distributed consistency checking.

K – Number of domains: n_1, n_2, ..., n_k hosts in each domain.

M – Number of participating documents. Two forms of access to the data in this set are used: m_i = { m_1, m_2, ..., m_N }gives the number of documents at each host; m_{ij} = { m_{11}, m_{12}, ..., m_{1n_1}, m_{21}, m_{22}, ..., m_{2n_2}, ..., m_{Kn_k} } gives the number of documents at each host within each domain.

R – Number of consistency rules for consistency checking in the system.

Application of each rule to the participating documents results in selection of a following number (x) of relevant documents at each host:

$$\forall i = 1...N, x = \Re(m_i, R),$$

where function $\Re()$ is defined as a table, for each set of consistency rules and set of documents used.

Sizes of documents are given by an array S = [S_1, S_2, ..., S_N], each element of the array specifies number of elements in all documents at each host. Another form of access to number of document elements by host and by domain is also used: S_{ij} = { S_{11}, S_{12}, ..., S_{1n_1}, S_{21}, S_{22}, ..., S_{2n_2}, ..., S_{Kn_k} }.

E.3 Initial findings

Execution of a consistency rule requires selection of elements from all applicable documents, at each host:

$$t_{\text{pars}} \cdot \Re(m_i, R) \cdot \mathrm{S_i}, \tag{3.1}$$

where t_{pars} is a parsing time for an element, and $\Re(\mathrm{m_i}, \mathrm{R}) * \mathrm{S_i}$ is the number of elements to be parsed.

Queries the domain agent for an itinerary, containing list of documents, relevant to a consistency rule, take the following interval of time:

$$(\varphi + t_{\text{query}}) \sum_{i=1}^{N} \Re(m_i, R), \tag{3.2}$$

where φ is the timing function of network transmission, depending on the number of transmitted elements. We use both notations: $\varphi \cdot Arg$ and $\varphi(Arg)$ to specify, that the function is applied to the argument *Arg*.

E.4 Local consistency check

The timing of a local consistency check consists of parsing time and the time, required to generate consistency links. The latter timing ($t_{generation}$) is determined by an experiment, and is given as a table of values for different consistency rules.

$$T_c = \sum_{v=1}^{N} t_{pars} \cdot \Re(m_v, R) \cdot S_v \,. \tag{4.1}$$

Let $\rho = \{ \mathrm{r_1}, \mathrm{r_2}, \ldots, \mathrm{r_R} \}$ be link generation times for different consistency rules, then

$$\sum_{i=1}^{R} \rho = t_{generation} \tag{4.2}$$

The checking time of a local consistency check will then be the following:

$$T^{c}_{total} = T_{\mathrm{c}} + t_{\text{generation}} \tag{4.3}$$

E.5 Distributed in-domain consistency check

Consider the worst-case scenario for a single-domain consistency check: all hosts in a domain need to be visited by a mobile checking agent.

Single-domain consistency check then takes time T_I . This time consists of the timing for itinerary query to the domain agent, timing of transfer of nodesets and time of a local consistency check:

$$T_I = (\varphi + t_{\text{query}}) \sum_{i=1}^{N} \Re(m_i, R) + \sum_{j=1}^{N-1} \varphi\,(\mathrm{L}_{\text{code}} + \Re(m_j, R) \cdot S_j) + T^c{}_{total} \quad (5.1)$$

where $\mathbf{L}_{\text{code}}$ – length of mobile agent code.

E.6 Single agent distributed check across multiple domains

Inter-domain location discovery timing ($T^{\max}_{idld}$) is a sum of in-domain itinerary queries for each of the participating domains:

$$T^{\max}_{idld} = \sum_{j=1}^{K} (\varphi + t_{query}) \sum_{i=1}^{n_j} \Re(m_{ij}, R). \quad (6.1)$$

Inter-domain location discovery starts from a certain host in domain A, $A = 1 \ldots K$.

$$T^{A}_{idld} = (\varphi + t_{query}) \sum_{i=1}^{n_A} \Re(m_A, R) + \sum_{j=1}^{A-1} (\varphi + t_{query}) \sum_{i=1}^{n_j} \Re(m_{ij}, R) + \sum_{j=A+1}^{K} (\varphi + t_{query}) \sum_{i=1}^{n_j} \Re(m_{ij}, R) \quad (6.2)$$

First component is the location discovery timing within the domain of host A, two remaining components – inter-domain discovery timings outside of domain A.

Worst-case scenario for multi-domain consistency check is when all relevant documents are located at hosts of different domains. The timing of a multiple domain consistency check $T^{\max}_{M}$ is then as follows:

$$T_M^{\max} = T_{idld}^A + \sum_{i=1}^{K}\sum_{j=1}^{n_i} \varphi(L_{code} + \Re(m_{ij}, R)\cdot S_{ij}) + T^c{}_{total} \,. \qquad (6.3)$$

Once again, similarly to (4.1)-(4.3), T_{total} is the timing of a local consistency check:

$$T^c{}_{total} = \sum_{j=1}^{K}\sum_{v=1}^{n_i} t_{pars}\cdot\Re(m_{vj}, R)\cdot S_{vj} + t_{generation} \,.$$

E.7 Multi-agent distributed check across multiple domains

In an optimal scenario, a number of agents (*y* agents) perform *concurrent* consistency checks at different hosts:

$$T_M^{opt} = T_{idld}^A + \frac{1}{y}\Lambda + \Pi \qquad (7.1)$$

$$T^d{}_{\text{total}} = T_M^{opt} + t_{\text{generation}}, \qquad (7.1.2)$$

where $\Lambda = \sum_{i=1}^{K}\sum_{j=1}^{IT(i,n_i)} \varphi(L_{code} + \Re(m_{ij}, R)\cdot S_{ij}) + \sum_{i=1}^{K}\sum_{v=1}^{IT(i,n_i)} t_{pars}\cdot\Re(m_{iv}, R)S_{iv}$,

$$\Pi = \sum_{i=1}^{K}\sum_{j=1}^{IT(i,n_i)} \varphi\left(\Re(m_{ij}, R)\cdot S_{ij}\right). \qquad (7.2)$$

The expression for T_M^{opt} consists of inter-domain document location discovery timing T_{idld}^A and the timing of a distributed multi-agent nodeset collection and nodeset transfer (7.1). In addition to that, total time for checking a particular rule T_{total} includes link generation time (7.1.2). Inter-domain discovery and link generation timings here are the same as in a single-agent scenario, thus any difference in performance of a multi-agent approach can be derived from values of Λ and Π.

Λ constitutes the total time it takes for an agent to migrate between all hosts of all domains, specified in its itinerary, to parse documents at these hosts and to transport the collected nodesets during migrations between hosts. In a multi-agent scenario, this job is carried out concurrently by *y* agents. Consequently, in the formula for T_M^{opt}, Λ is divided by *y*, thus potentially decreasing the total time for a multi-agent check. Function *IT*(*i*, n_i) serves as a summation index and returns number of hosts, specified in agent's itinerary, out of total n_i hosts at each domain *i*.

All collected nodesets need to be transported to a location, where link generation is to be carried out. Π is a corresponding time interval, which is spent on information exchange between collaborating mobile agents. Π is a *penalty* for carrying out a concurrent multi-agent check. The formula for Π sums up the transfers of nodesets, collected from documents of all domains, specified in the itinerary. Regardless of the number of agents, these nodesets are transmitted only once, therefore *y* does not directly participate in the formula for Π.

It is evident, that use of *y* agents (1<*y*<M) for a distributed check is more efficient in terms of distributed nodeset collection time (Λ/*y*), rather than use of a single anent (Λ). The question for consideration is whether efficiency advantage will remain even when a penalty Π is in place. The following Theorem aims to address this question.

E.8 Theorem on multiple-agent distributed consistency checking

Theorem. When carrying out a check of a consistency rule, there exist certain circumstances, where performance advantages can be gained from use of distributed concurrent mobile checking agents, rather than from use of a single mobile agent for sequential processing of distributed documents.

Comment: This theorem establishes the comparison relation between Λ and (Λ /y + Π).

Statement to be proved: There exist such conditions, that sequential transfer of agents and retrieval of nodesets takes longer than concurrent retrieval of nodesets, followed by exchange of retrieved nodesets preceding link generation. It is then an advantage to deploy multiple mobile agents for distributed checking in such conditions.

Prove: $\forall y : 1 < y < M, i = 1 \ldots K,\ j = 1 \ldots IT(i, n_i),$

$$t_{pars} > \frac{1}{y-1} \cdot \varphi(\Xi_{ij}) \quad \Leftrightarrow \Pi < \frac{(y-1)}{y}\Lambda \qquad (8.1)$$

In other words, we have to prove that penalty Π will be such, that Λ / y + Π is less than Λ, for any number of agents greater than 1. In this case, the larger the number of agents (upwards to the number, equal to that of number of documents), the greater the efficiency advantage of multi-agent checking:

$$\forall y : 1 < y < M, y \uparrow \Rightarrow \left(\Lambda - \left(\Lambda/y + \Pi\right)\right) \uparrow \qquad (8.2)$$

Proof:

Let the size of transported nodeset be $\Xi_{ij} = \Re(m_{ij}, R) \cdot S_{ij}$, then:

$$\frac{\Lambda}{y} = \frac{1}{y}\sum_{i=1}^{K}\sum_{j=1}^{IT(i,n_i)}\left(\varphi(L_{code}) + \varphi(\Xi_{ij})\right) + \frac{1}{y}\sum_{i=1}^{K}\sum_{v=1}^{IT(i,n_i)} t_{pars} \cdot \Xi_{ij} =$$

$$= \frac{K}{y}\varphi(L_{code}) + \frac{1}{y}\sum_{i=1}^{K}\sum_{j=1}^{IT(i,n_i)}\left(\left(\varphi + t_{pars}\right)\Xi_{ij}\right), \qquad (8.3)$$

$$\Pi = \sum_{i=1}^{K}\sum_{j=1}^{IT(i,n_i)}\varphi\left(\Xi_{ij}\right). \qquad (8.4)$$

At this point, taking into account (8.3) and (8.4), we need to establish the following relation ((y-1) Λ / y > Π) , 1<y<M:

$$\frac{y-1}{y}\sum_{i=1}^{K}\sum_{j=1}^{IT(i,n_i)}\left(\left(\varphi + t_{pars}\right)\Xi_{ij}\right) > \sum_{i=1}^{K}\sum_{j=1}^{IT(i,n_i)}\varphi\left(\Xi_{ij}\right) \qquad (8.5)$$

We make the relationship stricter in (8.5) by removing from consideration a component of Λ in (8.3), which has to do with transfer of mobile agent's code during migration.

Transforming both parts of the relation, and removing the summation:

$$\forall i = 1\ldots K,\ j = 1\ldots IT(i,n_i) : \left(\varphi + t_{pars}\right)\Xi_{ij} > \frac{y}{y-1} \cdot \varphi(\Xi_{ij})$$

We derive the condition for the parsing time of an individual file, which is:

$$t_{pars} > \frac{y-(y-1)}{y-1} \cdot \varphi(\Xi).$$

After transformation, we get:

$$\forall y : 1 < y < M, i = 1\ldots K,\ j = 1\ldots IT(i,n_i)$$

$$t_{pars} > \frac{1}{y-1} \cdot \varphi(\Xi_{ij}). \qquad (8.6)$$

For (8.5) to hold, *parsing time* for each document has to be *greater* than transmission time of the nodesets, collected from that document. The *transmission time* is taken in the inverse proportion to the number of concurrently deployed mobile checking agents. In practice, for sufficient number of concurrent mobile agents, this inequation will be satisfied.

Relation (8.5) then necessitates somewhat more relaxed inequation (8.1).

End of proof.

With the increase in a number of concurrently used mobile agents, total time of a distributed consistency check decreases: $y \uparrow \Rightarrow T^{opt}{}_{M} \downarrow$.

E.9 Theorem on multi-agent distributed and local centralised consistency checking

In the previous section, we compared single-agent and multi-agent approaches to distributed checking of consistency relations between documents. This section continues the comparison between the mult-agent approach and a centralised checker of local documents. Here, we also extend our check to cover multiple consistency rules in a batch.

Let $\rho = \{ r_1, r_2, \ldots, r_R \}$ be link generation times for different consistency rules, then

$$\sum_{i=1}^{R} \rho = t_{generation}$$

In the multi-agent architecture, the total *concurrent* checking time of all participating consistency rules until consistency links are generated is then:

$$T^{d}{}_{total} = \frac{\sum_{l=1}^{R} T_M^{opt}(l)}{R} + \sum_{l=1}^{R} r_l \qquad (9.1)$$

$T_M{}^{opt}$ in (9.1) is taken as a function of consistency rule number to signify that different rules result in differing itineraries and check times. The sum of $T_M{}^{opt}$ is averaged by the number of rules to signify concurrent checking. This is true with the assumption that all checks can be carried concurrently (i.e. there are sufficient resources to execute checks concurrently at all participating hosts).

Local centralised checking of the same rules:

$$T^{c}_{total} = R\sum_{i=1}^{M} t_{pars} \cdot \Xi i + \sum_{i=1}^{R} r_i \qquad (9.2)$$

This calculation for centralised local check is provided for equal conditions with the distributed multi-agent architecture. Documents are re-parsed for checking of different rules in order to accommodate change, since for each incremental change complete run of all consistency rules occurs in the centralised local checking scenario.

Comparison between T^{d}_{total} and T^{c}_{total}. If we use as many agents as documents (y = M), discard inter-domain discovery process timing and use (8.3-8.4), we transform (9.1) into:

$$T^{d}_{total} = \frac{1}{R}\sum_{i=1}^{R}\left(\frac{K}{M}\varphi(L_{code}) + \frac{M}{M}(\varphi + t_{pars})\Xi + M \cdot \varphi(\Xi)\right) =$$

$$\frac{K}{M}\varphi(L_{code}) + (M+1)\cdot\varphi(\Xi) + t_{pars}\cdot\Xi \qquad (9.3)$$

Theorem. In checking of a set of consistency rules, there exist certain circumstances, where performance advantage can be gained from the use of multiple distributed mobile agents for concurrent checking of a number of consistency rules in an incremental fashion, rather than from using a centralised checker for the same set of rules on the documents available locally.

Statement to prove: There exist circumstances, in which $T^{c}_{total} > T^{d}_{total}$.

Prove:

When $t_{pars} > \frac{\varphi(\Xi)}{R}$, $T^{c}_{total} > T^{d}_{total}$

Proof:

We aim to establish a set of parameters, in which:

$$T^{c}_{total} = R\sum_{i=1}^{M} t_{pars} \cdot \Xi i + \sum_{i=1}^{R} r_i \;>\; T^{d}_{total} = \frac{\sum_{l=1}^{R} T_M^{opt}(l)}{R} + \sum_{l=1}^{R} r_l \qquad (9.4)$$

After transformation:

$$R \cdot M \cdot t_{pars} \cdot \Xi > \frac{K}{M}\varphi(L_{code}) + (M+1)\cdot\varphi(\Xi) + t_{pars}\cdot\Xi,$$

$$t_{pars} \cdot \Xi \cdot (R \cdot M - 1) > \frac{K}{M}\varphi(L_{code}) + (M+1)\cdot\varphi(\Xi) \qquad (9.5)$$

For practical applications, where the number of rules in the set R>1 and the number of documents M>>1: RM >> 1, therefore (RM - 1) ~ RM, and (M + 1) ~ M.

When M >> K, $\frac{K}{M}\varphi(L_{code}) = o(\varphi)$.

Transformation of (9.5) then gives:

$$t_{pars} \cdot \Xi \cdot R \cdot M > M \cdot \varphi(\Xi) + o(\varphi),$$

$$t_{pars} \cdot \Xi \cdot R > \varphi \cdot \Xi + o(\varphi),$$

$$t_{pars} > \frac{\varphi(\Xi)}{R} \qquad (9.6)$$

For a sufficient number of consistency rules checked concurrently, parsing time will exceed the result in the right side of the inequation.

End of proof.

Appendix F Incremental Checking: Technical Challenges

F.1 Selection of Relevant Consistency Rules

Initial relevance of consistency rules to documents is established by an exhaustive check of all consistency rules on all documents when the incremental checker initialises. A consistency rule is relevant to a document when there is at least one rule operator, which contains an XPath expression that selects an element from that document. In addition to initial rule relevance, incremental checking requires selection of consistency rules, relevant to a particular document modification.

The proposed implementation of an incremental consistency checking approach was designed intentionally lightweight. In the software agent architecture, which builds on incremental checking, selection of relevant consistency rules is carried out at each participating network host. Lightweight implementation allows users to perform incremental checking on "thinner" clients, and reduces an amount of resources taken away by the checker from main applications, executed by the user on the client.

Our rule selection approach is based on pair-wise comparison of XPath expressions. First expression is generated by a tree-differencing (TreeDiff) algorithm for each modified document, when current document's content is compared to that of the document's backup copy. Second expression in each compared pair is an XPath expression, specified in the operators of every consistency rule. Rule selection is a part of the incremental checking algorithm (Chapter 5, Fig. 5.7, line 10).

A result of comparison of XPath pairs establishes whether there exists an intersection between the two XPath expressions. In other words, the comparison establishes whether one XPath points to a document element, which is the same as, or is a sub-element of, another element, selected by another XPath. If a TreeDiff XPath and an XPath in a consistency rule intersect, the rule is *relevant* to a document modification and is selected for incremental checking.

Selected rules are executed during the incremental check. As a result, incremental consistency check produces consistency links, which relate changed elements in one or more documents to the current state of all documents in the system, relevant to the selected consistency rules.

Computing Intersection of XPaths

Efficiency of the incremental checking algorithm is ultimately dependent on correctly establishing whether XPath expressions intersect. Semantics of the underlying consistency rule language determines complexity of the comparison algorithm.

Consistency rules in XLinkit make use of XPath language, which includes advanced navigation constructs. These include relative XPath expressions, wildcards, and selection of elements by value of an attribute. Return values of such constructs depend on the contents of a particular document instance. Therefore, when these constructs are used, XPath expression comparison cannot be carried out with string manipulation of two expressions alone.

In general, in order to give a definite answer on whether two XPath expressions are comparable, execution of XPaths on a document instance or on the definition of a document type is required. However, execution of *all* XPath expressions in a rule during rule selection is inefficient, because such approach would require the same resources as execution of the rule on the current document. We have created a lightweight comparison algorithm, which in the case of simpler expressions allows us to avoid unnecessary XPath executions in comparison of XPath expressions.

Consistency Rules: Global Variables and Relative XPaths

XLinkit consistency rule language makes use of global variables. Consistency rules then include relative paths to the "base" paths, stored in global variables. When a rule is checked, a "base" path is concatenated with a relative path.

Relative paths in XPath language can be specified starting from a given element, from the parent of that element, from any ancestor of that element, or from any elements at the current "level" of the DOM tree (specified by a wildcard). These methods can also be combined. String-wise concatenation of the values of global variables and the values of relative XPath expressions in rule classifiers results in complex expressions, which cannot be readily compared to the XPath expression from a TreeDiff. For example, expressions in Fig. F.1 are taken from the set of UML well-formedness rules [Appendix A].

Consistency Rule Identifier: gen1
Value of a Global Variable: //Foundation.Core.StructuralFeature.type/
(I) Example XPath Expression:
//Foundation.Core.StructuralFeature.type/../ancestor::Foundation.Core.Namespace.ownedElement/*
Consistency Rule Identifier: n2
(II) Example XPath Expression:
//Fouxncation.Core.Namespace.ownedElement/../Foundation.Core.Namespace.ownedElement/Foundation.Core.Association
(III) Example XPath Expression:
id(//Foundation.Core.Method/Foundation.Core.Method.specification/Foundation.Core.Operation/@xmi.idref)/Foundation.Core.ModelElement.visibility/@xmi.value

Fig. F.1. Examples of complex XPath expressions from the UML rule base.

F.2 Lightweight Rule Selection Approach

We propose a lightweight approach, which determines intersection between two XPath expressions, and a more heavyweight extension to this approach, where some complex XPath expressions are executed when the lightweight approach fails to compare the expressions.

The lightweight approach is based on string manipulation; execution of XPath expressions on documents is avoided, which results in a "small footprint" implementation. The string manipulation approach, however, relaxes the matching criteria: since complete information about the contents of a document instance is not available in this approach, numerous XPath directives are excluded from processing (Fig. F.2).

Return values of extended XPath constructs (id functions, wildcards, selection by attribute, indices) can only be identified when expressions are executed, and node selection operations are performed on the DOM tree of a document instance. However, in addition to any node selection performed when relevant rules are identified, node selection for all XPath expressions in a selected rule occurs again when the rule is being checked. As we expect documents in the software engineering domain to generate large DOM trees, extra node selection operations should be avoided. The lightweight rule selection algorithm (Fig. F.2) follows this motivation.

The XPath comparison algorithm discards from consideration complex XPath constructs such as id() functions (line 2.1 on Fig. F.2), wildcards (line 2.2), selection of elements by existence of xmi.id attributes (2.3) and ancestry (both ".." and "ancestor::" prefix constructs, 3.1-3.3).

Removal of the mentioned XPath expression elements during selection of relevant rules can only result in selection of *additional* consistency rules, which do not apply to a particular document change. Relevant rules are not be excluded from checking, as exclusion of parts from the XPath expression only makes selection criteria *broader*, not *narrower*. In other words, "stricter" XPath expressions, containing id() functions, selection by xmi.id attributes and wildcards are only "relaxed" when such extended constructs are removed. This relaxation may result in selection of additional rules. The scenarios in Fig. F.3 help to illustrate the rule selection algorithm.

1. *For each* consistency rule: concatenate XPath from global variable and relative XPaths in classifiers; result: composite rule XPath expressions.
2. Processing of the composite consistency rule XPath expression. Within the string representation of the composite rule XPath:
   2.1. discard all occurrences of id() function with any attributes;
   2.2. discard all occurrences of wildcard ("/*") element;
   2.3. discard all occurrences of identifier element ("@xmi.id");
   2.4. discard all indices of elements (substrings between "[" and "]").
3. Starting from the end of the resulting string representation of composite rule XPath, and parsing towards the beginning of the string:
   3.1. select one complete sub-path: a substring of all consecutive *elements* (i.e., substrings between "/" and "/", or after "/"), connected via "/". The sub-path may contain a single element or more;
   3.2. trim subpath of elements with "ancestor::" prefix construct;
   3.3. store the resulting subpath and string representation of composite XPath for use in the future;
   3.4. if no subpaths can been selected in 3.1, then *select this rule for checking* and restart at 1.
4. *For each* XPath expression in the TreeDiff:
   4.1. remove element indices (substrings between "[" and "]");
   4.2. find occurrence of a sub-path (result of 3.3) in the XPath (from 4.1);
   4.3. if occurrence found:
   4.3.1 if XPath (from 4.1) is terminated by the sub-path, then *select current rule for checking*;
   4.3.2 proceed to the next TreeDiff XPath expression – restart at 4.1;
   4.4. else remove ending element from sub-path, remember that the sub-path has been trimmed; if sub-path is not empty then proceed to 4.2.

Fig. F.2. Algorithm for finding intersection between two XPath expressions.

In step 4.4 of the algorithm (Fig. F.2), if a rule sub-path was not found within TreeDiff XPath, the rule sub-path is trimmed of an ending element. Then, a shorter sub-path is attempted for a match with the TreeDiff XPath. If the shorter sub-path terminates the TreeDiff XPath, then the rule, once executed, will select children of the changed element. Consequently, selection of such rule for incremental checking is desired (Scenario 1, Fig. F.3).

Scenario 1: XPath expression in a consistency rule points to a child of the changed element.
XPath expression in the rule: //B/C/D/F
TreeDiff XPath expression: /A/B/C/D
Desired result: It is envisaged, that children of /A/B/C/D may have been affected by change in //D, therefore it is desirable that the rule is selected in this case.
Execution of the rule selection algorithm:
Repetition 1 of the final loop 4.2-4.4:
Sub-path B/C/D/F not found in /A/B/C/D, trimming sub-path to B/C/D
Repetition 2: B/C/D found in /A/B/C/D, and it terminates /A/B/C/D
Result: rule is selected for checking (as desired).
Scenario 2: Rule points to a "neighbour" element of the changed element, "neighbour" element is of a different type.
XPath expression in the rule: //B/C/E
TreeDiff XPath expression: /A/B/C/D

Desired result: Logically, it is not necessary to select this rule for incremental checking (otherwise if a rule points to a model element of certain kind, this rule would be selected for changes in model elements of all different kinds, as normally model elements are "neighbours" in the XMI DOM tree).

Execution of the rule selection algorithm:

Repetition 1: Sub-path B/C/E not found in /A/B/C/D, trimming sub-path to B/C

Repetition 2: B/C found in /A/B/C/D, but does not terminate /A/B/C/D

Result: rule is *not* selected for checking (as desired).

Scenario 3: Rule points to the parent of the changed element.

XPath expression in the rule: //B/C

TreeDiff XPath expression: /A/B/C/D

Desired result: It is not necessary to select this rule for incremental checking. Otherwise, if in such conditions a rule points to a root element, this rule would always be selected for any change in the document.

Execution of the rule selection algorithm:

Repetition 1: B/C is found in /A/B/C/D, but does not terminate /A/B/C/D

Result: rule is *not* selected for checking (as desired).

Fig. F.3. Consistency rule selection scenarios.

Following the algorithm in Fig. F.2 and processing rule XPath expressions from Fig. F.1, we obtain the following results (Fig. F.4).

Consistency rule id: "gen 1"

(I) Rule XPath expression:

//Foundation.Core.StructuralFeature.type/../Foundation.Core.Namespace.
ownedElement

Sub-path selected (3.3, Fig. F.2): Foundation.Core.Namespace.ownedElement

Rule applies to changes in all //Foundation.Core.Namespace.ownedElement model elements

Consistency rule id: "n2"

(II) Rule XPath expression:

//Fouccation.Core.Namespace.ownedElement/../Foundation.Core.Namespace.
ownedElement./Foundation.Core.Association

Sub-path selected (3.3, Fig. F.2):
Foundation.Core.Namespace.ownedElement/Foundation.Core.Association

Rule applies to changes in all //Foundation.Core.Namespace.ownedElement/
Foundation.Core.Association and //Foundation.Core.Namespace.ownedElement model elements

(III) Rule XPath expression:

id(//Foundation.Core.Method/Foundation.Core.Method.specification/Foundation.
Core.Operation/@xmi.idref)/Foundation.Core.ModelElement.visibility/@xmi.value

Sub-path selected (3.3, Fig. F.2): Foundation.Core.ModelElement.visibility

Rule applies to changes in all //Foundation.Core.ModelElement.visibility model elements

Fig. F.4. Results of processing XPath expressions

F.3 Refined Approach: Selective Execution of XPath Expressions

An amendment to the lightweight approach selectively executes XPath expressions and improves comparison accuracy for complex XPaths, where string manipulation is not sufficient to establish intersection.

In the lightweight approach above, accuracy of selection of relevant consistency rules improves, when an XPath in a rule is contains a longer section of the path to a document element, and this section does not contain extended constructs, which are not processed by the lightweight approach. However, if an expression contains id selector operators (Expression III, Fig. F.4) or makes use of ancestry

constructs and only contains short relative paths, the lightweight approach will select this rule for incremental checking for a larger number of different document changes than necessary. Selection of an irrelevant rule for incremental checking results in "unnecessary" execution of XPath expressions in the selected consistency rule.

In the amended approach, if parts of the composite XPath expression in the rule selection algorithm are to be discarded (steps 2.1-2.4, 3.2 in Fig. F.2), the expression is *executed* instead. This results in selection of a node set; for each element of this set, a complete XPath from the root of the document DOM tree is then computed. This simple XPath contains sequence of node names without id function calls, wildcards or ancestry selection operators. The simple path is then compared string-wise to the TreeDiff XPath to the changed element (as in 4.1-4.4, F.2).

In our implementation of this amended approach, identification of a complete XPath expression from the root to each element in the generated node set is efficiently carried out by the underlying XPath implementation. The Xalan processor [Apache 2000], used for implementation of our incremental checker prototype, provides the functionality for generation of XPath expressions for document elements. If a different XPath processor is used, which does not provide such functionality, a simple XPath expression to an element may be computed by traversing the document's DOM tree "upwards" towards the root, and concatenating element names on the way.

The amended rule selection approach allows us to eliminate any irrelevant consistency rules from an incremental check, at the expense of execution of some of more complex rule XPath expressions. In any case, however, selection of irrelevant consistency rules would result in execution of these expressions on a number of documents, including the modified document. The amended approach significantly improves effectiveness of rule selection, although somewhat hinders performance. Our implementation of the incremental checking approach is evaluated in Chapter 10.

F.4 Merging Sets of Consistency Links

We envisage, that users would wish to combine linksets from numerous incremental checks, and update global linksets with results of incremental checks.

Our approach for merger of linksets assumes that the rule identifiers have remained unchanged in the time between check executions. Consistency links refer to respective rule identifiers, which are used to establish matching between different links. If rule identifiers have changed, older linksets need to be updated by mapping old rule identifiers to new ones before linksets can be merged. We expect that in a software engineering project, consistency rules, which express consistency constraints, will change much less often than the documents, so the assumption of uniqueness of rule identifiers is satisfied. For our UML scenario, consistency relations of well-formedness between elements of a UML model are specified in the UML standard, and will change relatively infrequently.

We also assume that globally unique XML element identifiers within the documents remain unchanged between consistency checks. Since XLinkit framework does not require actual *development* of documents to be carried out in XML, the task of assignment of element identifiers within XML representations of participating documents is delegated to a converter. In our UML scenario, representation of a UML model in XMI contains element identifiers, which are assigned to elements by the XMI generator (i.e., Rational Rose XML exporter). The UML standard demands that XMI identifiers are unique throughout the UML model.

When these basic assumptions are satisfied, the algorithm for merger of linksets is as follows (Fig. F.5).

```
1: Sort linksets in a descending order by time stamp of consistency checks;
2: Start merging from the two 'topmost' linksets on the list;
On each next step, consider the next linkset and update the 'topmost' linkset;
3: FOR each link in the 'topmost' linkset DO
3: FOR each link in the older linkset DO BEGIN
4:   IF one link from 'topmost' and the other link from older linkset
refer to the same document element (indicated by the same element identifier) THEN
5:     IF rule identifiers in the links are the same THEN
6:       REM this link has already been updated in the recent linkset.
7:       REM do not update the topmost linkset
8:       Indicate the update status to the user;
```

```
8:      ELSE copy a link from an older linkset into the new one;
9:  END;
9:  NEXT link;
10: NEXT link;
11: NEXT linkset.
```

Fig. F.5. Algorithm for merging linksets.

Problem of XPath Fragility in XLinkit Locators

Simple XPath expressions found in locators of consistency links use element indices to "point" to a particular element within a certain level of the document's tree. Consider the following simple example, where consistency links are created between elements of two documents (Fig. F.6).

```
<Document id="StoreList1">
<list>
<fruit barcode="A" type="Apple"
xmi.id="F.0001"/>
<fruit barcode="B" type="Blueberry"
xmi.id="F.0002"/>
<fruit barcode="C" type="Citron"
xmi.id="F.0003"/>
<bread barcode="Z" type="White"
xmi.id="B.0001"/>
</list>
</Document>
```

```
<Document id="StoreList2">
<list>
<fruit barcode="B" type="Citron"
xmi.id="F.0011"/>
<fruit barcode="C" type="Apple"
xmi.id="F.0012"/>
<fruit barcode="A" type="Blueberry"
xmi.id="F.0013"/>
<bread barcode="Z" type="White"
xmi.id="B.0001"/>
</list>
</Document>
```

Fig. F.6a. Example documents for demonstration of the XPath fragility problem.

```
<xlinkit:LinkBase date="Sept 03 14:21:39 GMT+01:00 2001"
  docSet="file:// XPfra.xml" ruleSet="file://RuleSet.xml"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:xlinkit="http://www.xlinkit.com">
  <!-- Consistency rule "FruitBarcodes": for any barcode, fruit, marked with this barcode, should be
of the same type -->
    <xlinkit:ConsistencyLink ruleid="FruitBarcodes">
      <xlinkit:State>inconsistent</xlinkit:State>
      <xlinkit:Locator xlink:href="doc1.xml#/list/fruit[1]" />
      <xlinkit:Locator xlink:href="doc2.xml#/list/fruit[2]" />
    </xlinkit:ConsistencyLink>
    <xlinkit:ConsistencyLink ruleid="FruitBarcodes">
      <xlinkit:State>inconsistent</xlinkit:State>
      <xlinkit:Locator xlink:href="doc1.xml#/list/fruit[2]" />
      <xlinkit:Locator xlink:href="doc2.xml#/list/fruit[3]" />
    </xlinkit:ConsistencyLink>
    <xlinkit:ConsistencyLink ruleid="FruitBarcodes">
      <xlinkit:State>inconsistent</xlinkit:State>
      <xlinkit:Locator xlink:href="doc1.xml#/list/fruit[3]" />
      <xlinkit:Locator xlink:href="doc2.xml#/list/fruit[1]" />
    </xlinkit:ConsistencyLink>
  <!—Consistency rule "FruitInStock": for each fruit in the base store list, there should be a fruit of
the same type in any subsequent store lists" -->
    <xlinkit:ConsistencyLink ruleid="FruitInStock">
      <xlinkit:State>consistent</xlinkit:State>
      <xlinkit:Locator xlink:href="doc1.xml#/list/fruit[1]"/>
      <xlinkit:Locator xlink:href="doc2.xml#/list/fruit[2]" />
```

```
    </xlinkit:ConsistencyLink>
    <xlinkit:ConsistencyLink ruleid="FruitInStock">
      <xlinkit:State>consistent</xlinkit:State>
      <xlinkit:Locator xlink:href="doc1.xml#/list/fruit[2]" />
      <xlinkit:Locator xlink:href="doc2.xml#/list/fruit[2]" />
    </xlinkit:ConsistencyLink>
    <xlinkit:ConsistencyLink ruleid="FruitInStock">
      <xlinkit:State>consistent</xlinkit:State>
      <xlinkit:Locator xlink:href="doc1.xml#/list/fruit[3]" />
      <xlinkit:Locator xlink:href="doc2.xml#/list/fruit[3]" />
    </xlinkit:ConsistencyLink>
  <xlinkit:LinkBase/>
```

Fig. F.6b. Links between the two documents.

When a change is introduced, the following links become outdated and need to be updated (Fig. F.7).

```
    <Document id="StoreList1">
    <list>
    <fruit barcode="A" type="Apple"
xmi.id="F.0001"/>
    <fruit barcode="B" type="Blueberry"
    xmi.id="F.0002"/>
    <fruit barcode="C" type="Citron"
    xmi.id="F.0003"/>
    <bread barcode="Z" type="White"
xmi.id="B.0001"/>
    </list>
    </Document>
```

```
    <Document id="StoreList2">
    <list>
    <!-- discounted citrons added -->
    <fruit barcode="C0" type="Citron"
xmi.id="F.0010"/>
    <fruit barcode="B" type="Citron"
xmi.id="F.0011"/>
    <fruit barcode="C" type="Apple"
    xmi.id="F.0012"/>
    <fruit barcode="A" type="Blueberry"
    xmi.id="F.0013"/>
    <bread barcode="Z" type="White"
xmi.id="B.0001"/>
    </list>
    </Document>
```

Fig. F.7a. Changed documents.

```
  <xlinkit:LinkBase date="Sept 23 16:21:39 GMT+01:00 2001"
 docSet="file:// XPfra.xml" ruleSet="file://RuleSet.xml"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:xlinkit="http://www.xlinkit.com">
  <!-- Consistency rule "FruitBarcodes": for any barcode, fruit, marked with this barcode, should be
of the same type -->
    <xlinkit:ConsistencyLink ruleid="FruitBarcodes">
      <xlinkit:State>inconsistent</xlinkit:State>
      <xlinkit:Locator xlink:href="doc1.xml#/list/fruit[1]" />
      <xlinkit:Locator xlink:href="doc2.xml#/list/fruit[3]" />
    </xlinkit:ConsistencyLink>
    <xlinkit:ConsistencyLink ruleid="FruitBarcodes">
      <xlinkit:State>inconsistent</xlinkit:State>
      <xlinkit:Locator xlink:href="doc1.xml#/list/fruit[2]" />
      <xlinkit:Locator xlink:href="doc2.xml#/list/fruit[4]" />
    </xlinkit:ConsistencyLink>
    <xlinkit:ConsistencyLink ruleid="FruitBarcodes">
      <xlinkit:State>inconsistent</xlinkit:State>
      <xlinkit:Locator xlink:href="doc1.xml#/list/fruit[3]" />
      <xlinkit:Locator xlink:href="doc2.xml#/list/fruit[2]" />
```

```
    </xlinkit:ConsistencyLink>
  <!—Consistency rule "FruitInStock": for each fruit in the base store list, there should be a fruit of
the same type in any subsequent store lists" -->
    <xlinkit:ConsistencyLink ruleid="FruitInStock">
      <xlinkit:State>consistent</xlinkit:State>
      <xlinkit:Locator xlink:href="doc1.xml#/list/fruit[1]"/>
      <xlinkit:Locator xlink:href="doc2.xml#/list/fruit[3]" />
    </xlinkit:ConsistencyLink>
    <xlinkit:ConsistencyLink ruleid="FruitInStock">
      <xlinkit:State>consistent</xlinkit:State>
      <xlinkit:Locator xlink:href="doc1.xml#/list/fruit[2]" />
      <xlinkit:Locator xlink:href="doc2.xml#/list/fruit[3]" />
    </xlinkit:ConsistencyLink>
    <xlinkit:ConsistencyLink ruleid="FruitInStock">
      <xlinkit:State>consistent</xlinkit:State>
      <xlinkit:Locator xlink:href="doc1.xml#/list/fruit[3]" />
      <xlinkit:Locator xlink:href="doc2.xml#/list/fruit[1]" />
      <xlinkit:Locator xlink:href="doc2.xml#/list/fruit[4]" />
    </xlinkit:ConsistencyLink>
  <xlinkit:LinkBase/>
```

Fig. F.7b. Incremental link base.

The change in document "doc2.xml" was an addition of the a fruit element of type citron with a unique barcode. Both consistency rules (identifiers "Barcodes" and "FruitInStock") are relevant to the change. Both documents doc1 and doc2 are relevant to these rules and incremental check creates an incremental linkbase linking both documents (Fig. F.7b). Any links, which have changed after this check in comparison to previous linkbase (Fig. F.6b), are shown in italics in Fig. F.7b.

The only link we have been expecting to see change after doc2 modification is the last link in Fig. F.7b, which connects all fruits of the same type "Citron". All other links between "fruit" elements are unaffected by the change and, ideally, would appear unchanged in the new linkbase. Yet, all links to "fruit" elements have indeed changed, because indices of XPath expressions pointing to document elements within XLinkit locators have changed after insertion of a fruit element at the beginning of doc2.

XPath fragility problem, demonstrated in this example, is inherent in the XLinkit framework. However, since XLinkit completely re-generates its global linksets on each check, XPath fragility problem does not appear in the current global linksets. At the same time, due to XPath fragility, it is very difficult to compare the current global linkset and a linkset, which was generated before certain document modifications were made.

Because different XLinkit linksets cannot be compared, it impossible to establish which consistency links have *changed* their status after a document has been modified. The incremental checking approach is a step in the direction towards being able to provide such functionality. Below, we present our solution to XPath fragility in XLinkit linksets.

Mapping of XPath Expressions

Our solution for the XPath fragility problem is based on uniqueness of element identifiers. The same requirement applies before link merger algorithm (Fig. F.2) can be used. The XMI standard, in which UML models are represented in our UML scenario, requires that individual model elements are assigned unique identifiers.

The following algorithm matches XPath expressions in the old linkbase to current XPath expressions, pointing to the same document elements (Fig. F.8).

```
1: FOR each consistency link in an older linkset DO BEGIN
2:  FOR each locator DO BEGIN
3:    Get an XLink from the locator, extract an XPath expression (after '#')
4:    IF this XPath expression has not been entered into the hash table THEN BEGIN
5:      Apply XPath to the backup version of the document, an element is selected
```

```
    6:      Take the value of the xmi.id attribute of that element
    7:      IF this value exists in the hash table THEN
    8:        Replace the old XPath hashtable value with current XPath
    9:      ELSE BEGIN
    10:       Scan the new version of the document for appearance of an element with a
11:     matching xmi.id attribute
    12:       IF element found THEN BEGIN
    13:         Generate an XPath expression pointing to that element
    14:         Insert key xmi.id, and values of old XPath and new XPath into the hashtable
    15:       END ELSE BEGIN
    16:         REM element has been deleted in the new version
    17:         Remove old consistency link from the old linkset, proceed to the next link
    18:       END // ELSE BEGIN
    19:     END // ELSE BEGIN
    20:   END // ELSE BEGIN
    21: END // For each locator
    22: END // For each consistency link
```

Fig. F.8. XPath update algorithm.

Xmi.id	Old XLink	New XLink
F.0001	doc1.xml#/list/fruit[1]	doc1.xml#/list/fruit[1]
F.0002	doc1.xml#/list/fruit[2]	doc1.xml#/list/fruit[2]
...	...	...
F.0010	doc2.xml#/list/fruit[1]	doc2.xml#/list/fruit[2]
F.0011	doc2.xml#/list/fruit[2]	doc2.xml#/list/fruit[3]
F.0012	doc2.xml#/list/fruit[3]	doc2.xml#/list/fruit[4]
F.0013	Null	doc2.xml#/list/fruit[1]
B.0001	doc1.xml#/list/bread[1]	doc1.xml#/list/bread[1]

Fig. F.9. XPath expression mapping hashtable.

As a result of the proposed algorithm for updating of XPath expressions, all expressions in older linksets are re-adjusted and correspond to current element locations (Fig. F.9). This allows us to perform string-wise comparison of XLinkit locators between linkbases when linkbases are being merged. The XPath mapping algorithm also removes invalid links from older linkbases, which contain elements no longer existing in the current version of the documents.

Execution of the XPath mapping algorithm has to precede the linkset merger algorithm. Using these two algorithms in conjunction allows us to directly compare consistency links from different link sets, correctly identify most current links and include them into the merged linkset.

Fig. F.10 shows an example of an updated link from the linkbase in Fig. F.6b. After all links in that linkbase have been similarly mapped, the link merger algorithm establishes that one consistency link has changed in the current linkbase (Fig. F.11). The result of update of consistency links and merger of two linkbases makes the sense: insertion of a fruit of type "citron" has resulted in one link, corresponding to that particular element, becoming inconsistent.

```
<xlinkit:ConsistencyLink ruleid="FruitInStock">
    <xlinkit:State>consistent</xlinkit:State>
    <xlinkit:Locator xlink:href="doc1.xml#/list/fruit[3]" />
    <xlinkit:Locator xlink:href="doc2.xml#/list/fruit[4]" />
    <!- - was: xlinkit:Locator xlink:href="doc2.xml#/list/fruit[3]" -->
  </xlinkit:ConsistencyLink>
```

Fig. F.10. Updated link in the older linkbase.

```
<xlinkit:ConsistencyLink ruleid="FruitInStock">
```

```
    <xlinkit:State>consistent</xlinkit:State>
    <xlinkit:Locator xlink:href="doc1.xml#/list/fruit[3]" />
    <xlinkit:Locator xlink:href="doc2.xml#/list/fruit[1]" />
    <xlinkit:Locator xlink:href="doc2.xml#/list/fruit[4]" />
</xlinkit:ConsistencyLink>
```

Fig. F.11. A changed link is detected after linkbases are updated and merged.

F.5 Distribution of UML Models

The motivation behind the XMI standard is that of enabling and making easier the exchange of UML diagrams in XML. Even though the standard mentions a provision for *splitting* large XMI UML models into separate files [OMG Nov. 2000], such provision has not yet been implemented.

In a distributed setting, software engineers often work on parts of a large UML model. For example, a functional analyst may be developing a class diagram, and a number of software engineers are creating collaboration and sequence diagrams for the same project.

Sharing or exchange of a single large model file between numerous distributed developers is inconvenient and unsatisfactory, as a relatively small number of model elements are changed on every incremental update. Our approach for separation of a UML model into several smaller files, containing individual elements or groups of elements, gives developers flexibility to operate on the level of model elements. We have created a tool UMLXMI, which provides this functionality. The tool allows a development team to overcome an inefficiency of deployment of XMI for distributed development of UML models.

By separating the UML model into parts, UMLXMI allows workgroup systems (such as CVS) to function more effectively: naturally, execution of locking policies on a large original UML model file would severely limit flexibility of distributed concurrent development of this model by a number of software engineers.

In a UML model represented as an XML file, the utility selects sub-trees of major model elements and stores them into separate XML files. XPath expressions to model elements coincide with values of global variables in the UML consistency rule set [Appendix A], as shown in Fig. F.12.

Model Element	XPath
Association	//Foundation.Core.Association[@xmi.id]
Association Class	//Foundation.Core.AssociationClass[@xmi.id]
Association End	//Foundation.Core.AssociationEnd[@xmi.id]
Behavioural Feature	//Foundation.Core.BehavioralFeature.parameter/..
Class	//Foundation.Core.Class[@xmi.id]
Classifier	//Foundation.Core.Classifier.feature/..
Component	//Foundation.Auxiliary_Elements.Component
Constraint	//Foundation.Core.Constraint
Data Type	//Foundation.Core.DataType
Generalizable Elements	//Foundation.Core.GeneralizableElement.generalization/..
Generalization	//Foundation.Core.Generalization[@xmi.id]
Interface	//Foundation.Core.Interface
Method	//Foundation.Core.Method
Namespace	//Foundation.Core.Namespace.ownedElement/..
Structural Feature	//Foundation.Core.StructuralFeature.type/..
Type	//Foundation.Core.Class[id(Foundation.Core.ModelElement.stereotype/ Foundation.Extension_Mechanisms.Stereotype/@xmi.idref)/ Foundation.Core.ModelElement.name/text()='Type']

Fig. F.12. Model elements and XPath expressions to them.

The structure of XMI ensures, that information about each model element is contained within sub-trees of the nodes, selected by the XPath expressions (Fig. F.12).

An example class in Fig. F.13 is extracted from the UML model for Building Management application [Rational 1998]. The "Building Management Domain" class refers to a stereotype "domain" with an XMI identifier (xmi.id) "G.35" and to package "Actors" with an xmi.id "S.10147". The example class is persistent and transient.

```
<?xml version="1.0" encoding="UTF-8"?>
<XMI>
   <XMI.content>
     <Model_Management.Model xmi.id="G.1">
         <Foundation.Core.Namespace.ownedElement>
            <Model_Management.Model xmi.id="G.562">
               <Foundation.Core.Namespace.ownedElement>
                  <Model_Management.Package xmi.id="S.10147">
                     <Foundation.Core.Namespace.ownedElement>
                        <Foundation.Core.Class xmi.id="S.10148">
<Foundation.Core.ModelElement.name>
Building Management Domain
</Foundation.Core.ModelElement.name>
<Foundation.Core.ModelElement.visibility xmi.value="public"/>
<Foundation.Core.GeneralizableElement.isRoot xmi.value="true"/>
<Foundation.Core.GeneralizableElement.isLeaf xmi.value="true"/>
<Foundation.Core.GeneralizableElement.isAbstract xmi.value="false"/>
<Foundation.Core.Class.isActive xmi.value="false"/>
<Foundation.Core.ModelElement.stereotype>
<Foundation.Extension_Mechanisms.Stereotype xmi.idref="G.35"/>
<!-- domain-->
</Foundation.Core.ModelElement.stereotype>
<Foundation.Core.ModelElement.namespace>
<Model_Management.Package xmi.idref="S.10147"/>
<!--Actors -->
</Foundation.Core.ModelElement.namespace>
<Foundation.Core.ModelElement.taggedValue>
<Foundation.Extension_Mechanisms.TaggedValue>
<Foundation.Extension_Mechanisms.TaggedValue.tag> persistence
</Foundation.Extension_Mechanisms.TaggedValue.tag>
<Foundation.Extension_Mechanisms.TaggedValue.value> transient
</Foundation.Extension_Mechanisms.TaggedValue.value>
</Foundation.Extension_Mechanisms.TaggedValue>
</Foundation.Core.ModelElement.taggedValue>
                        </Foundation.Core.Class>
                     </Foundation.Core.Namespace.ownedElement>
                  </Model_Management.Package>
               </Foundation.Core.Namespace.ownedElement>
            </Model_Management.Model>
         </Foundation.Core.Namespace.ownedElement>
      </Model_Management.Model>
   </XMI.content>
</XMI>
```

Fig. F.13. Class element, extracted from a UML model.

Uniqueness of XMI element identifiers

Since model elements refer to each other via XMI identifiers, it is important that all XMI identifiers continue to remain unique throughout model development in order for the model to be internally consistent. Initially, uniqueness of XMI identifiers is guaranteed by a generator, when a UML model is converted into its XMI representation (we use Unisys XMI exporter for Rational Rose [Rational 1999]).

In a practical application scenario, after UMLXMI has separated model elements into individual files, developers will "check out" groups of elements from an initial model location onto their workstation. Each developer can edit individual elements, or import groups of them into the UML

development environment (i.e., Rational Rose). When some changes are made, modified elements are marked as "committed". At this point, other developers will be able to "check out" any of the committed elements for local development.

At any time, UMLXMI tool can be executed again in order to collect individual element files into a single UML model. However, the distributed consistency checking architecture, described in this thesis, facilitates distributed development. It allows developers to check relations between distributed representations of model elements, with no requirement for a model to be re-assembled into a single file.

As distributed development of model elements commences, model elements are modified, added and removed from the model. Most modifications are expected to occur within UML development environments or, less conveniently and more error-prone, within the source of XMI files, representing UML model elements. Developers need to ensure that identifiers of model elements are kept *unique* within the model. UMLXMI checks violation of xmi.id uniqueness, when a model is collected from individual elements, and reports any problems found. Once uniqueness is established, consistency of identifiers is checked by checking UML well-formedness consistency rules.

The UMLXMI utility gives developers extra flexibility in exchanging, sharing and editing individual model elements by means of their favourite UML development environment, which imports XMI, or at the level of the source of XMI element files. Having adopted a model of distributed collaborative authoring of separate model elements, we have also delegated the responsibility for keeping existing unique model element identifiers intact.

Deployment Scenario: Initial Distribution

The first deployment scenario concerns initial distribution of a UML model, when the model has been developed centrally and is now being separated for distributed development. The UMLXMI utility is executed every time the XMI representation of the UML model is re-generated at the central location, in order for all element files to be updated if necessary. The proposed sequence of execution is shown in Fig. F.14.

1. UML model is changed within the design environment (i.e., Rational Rose);
2. XMI representation of the UML model is re-generated (Unisys XMI exporter);
3. Watchdog monitor detects changes in the XMI representation;
4. Watchdog runs UMLXMI utility on the new XMI representation of the UML model;
5. UMLXMI utility regenerates representations for model elements;
6. UMLXMI utility creates document universe descriptions for each of network hosts, where groups of representations of model elements are to be copied to;
7. Groups of representations of model elements are copied to respective network hosts, together with relevant document universe descriptions;
8. Consistency framework: at each host, watchdog identifies the change on monitored documents as new versions of model element representations appear on the host;
9. Consistency framework: consistency checking takes place.

Fig. F.14. Distribution of UML models into separate XMI documents.

Distribution configuration of model elements is specified by the system administrator in the UMLXMI configuration file. *Document universe* descriptions, created in line 6 (Fig. F.14), contain lists of all local XMI files, distributed to each participating host. These descriptions serve as a startup configuration for our distributed consistency framework; all files in the description are monitored for change (line 8), and are checked incrementally if a modification is detected.

Deployment Scenario: Regrouping Elements Into a Single Model

This scenario occurs, when incremental development has reached a point release, and a distributed UML model is collected into a single model file for release. UMLXMI reads in current 'document universe' file lists from each participating host, and concatenates all XMI documents, mentioned in the lists. Concatenation respects nesting of XML tags, therefore model elements, which were located at the same level in the model's tree hierarchy before separation, will also appear as neighbours after concatenation. During model distribution, the full path to every element is saved in that element's representation, therefore, after concatenation, positioning of model elements in the DOM tree is restored.

UMLXMI also checks uniqueness of XMI identifiers for different elements, and alerts users when uniqueness is violated. Developers can use this functionality of UMLXMI to check a distributed model for internal consistency of identifiers throughout distributed development.

Summary

UMLXMI provides functionality to distribute groups of UML model elements across a number of network hosts. Developers are then able to engage in distributed development of individual model elements locally, on their workstations. Consequently, the UMLXMI utility provides developers with additional flexibility, and partly overcomes the deficiency of XMI standard – its inability to generate partial views on UML models. UMLXMI makes individual elements and sets of elements easier to exchange and, therefore, facilitates distributed concurrent development.

UMLXMI integrates with the software agent architecture for consistency management, proposed in this thesis. When groups of model element representations, generated by UMLXMI, are propagated to their destination domains, old representations are updated, and the update is then detected by watchdog monitors, which form a part of the architecture. Consistency checking then takes place as usual: tree-wise differences are identified, followed by identification of relevant consistency rules, and finalised by carrying out consistency checks if necessary. Reduction in size of elements' XMI representations in comparison with the representation of a complete model improves performance of the TreeDiff algorithm, which identifies document modifications. In this way, deployment of UMLXMI on large UML models has a significant positive impact on performance of incremental checks in the distributed software agent architecture.