# An Ontology-Based Approach to Concern-Specific Dynamic Software Structure Monitoring

by

**Barry Robert Pekilis**

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2006

©Barry Robert Pekilis, 2006

I hereby declare that I am the sole author of this thesis. This is a true copy of my thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

# Abstract

Software reliability has not kept pace with computing hardware. Despite the use reliability improvement techniques and methods, faults remain that lead to software errors and failures. Runtime monitoring can improve software reliability by detecting certain errors before failures occur. Monitoring is also useful for online and electronic services, where resource management directly impacts reliability and quality. For example, resource ownership errors can accumulate over time (e.g., as resource leaks) and result in software aging. Early detection of errors allows more time for corrective action before failures or service outages occur. In addition, the ability to monitor individual software concerns, such as application resource ownership structure, can help support autonomic computing for self-healing, self-adapting and self-optimizing software.

This thesis introduces *ResOwn* - an application resource ownership ontology for interactive session-oriented services. ResOwn provides software monitoring with enriched concepts of application resource ownership borrowed from real-world legal and ownership ontologies. ResOwn is formally defined in OWL-DL (Web Ontology Language Description Logic), verified using an off-the-shelf reasoner, and tested using the call processing software for a small *private branch exchange (PBX)*. The ResOwn Prime Directive states that every object in an operational software system is a resource, an owner, or both simultaneously. Resources produce benefits. Beneficiary owners may receive resource benefits. Nonbeneficiary owners may only manage resources. This approach distinguishes resource ownership use from management and supports the ability to detect when a resource's role-based runtime capacity has been exceeded.

This thesis also presents a greybox approach to concern-specific, dynamic software structure monitoring including a monitor architecture, greybox interpreter, and algorithms for deriving monitoring model from a monitored target's formal specifications. The target's requirements and design are assumed to be specified in SDL, a formalism based on communicating extended finite state machines. Greybox abstraction, applicable to both behavior and structure, provides direction on what parts, and how much of the target to instrument and what types of resource errors to detect.

The approach was manually evaluated using a number of resource allocation and ownership scenarios. These scenarios were obtained by collecting actual call traces from an instrumented PBX. The results of an analytical evaluation of ResOwn and the monitoring approach are presented in a discussion of key advantages and known limitations. Conclusions and recommended future work are discussed at the end of the thesis.

# Acknowledgements

First and foremost, I would like to thank my two co-supervisors, Professor K. Czarnecki and Professor R.E. Seviora, for their support and guidance during the writing and defense of this Ph.D. thesis. I would also like to thank my Ph.D. examination committee, Professor J.H. Weber-Jahnke, Professor K. Kontogiannis, Professor P. Dasiewicz, Professor J.M. Atlee and Professor P. Ward, whose comments and advice helped to improve the quality of this thesis.

I must also express my deep gratitude to Karl Trygve Kalleberg, Sean Lau and Vlad Ciubotariu for their useful discussions, encouragement and thought provoking comments during the final stages of my research and thesis. I would also like to acknowledge the colleagues I have had the pleasure of knowing over the years from the Software Engineering Group, the Bell Canada Software Reliability Lab, the Generative Software Development Lab and the Waterloo Formal Methods group. In addition, I would like to thank Audrey Heutzenroeder for the formidable task of proofreading the contents of this thesis.

As every graduate student knows, it is the Graduate Secretary that really runs the program. I would like to express a sincere thank you to Wendy Boles, Graduate Studies Coordinator, Department of Electrical and Computer Engineering for her valuable assistance in ensuring that all my paperwork was always done correctly and on time.

While living in Waterloo, I have been fortunate to make new friends who helped keep me going through the peaks and the valleys that inevitably occur during any Ph.D. In no particular order, I wish to thank my friends: Eric, Shawn, Alana, Rob, Cathie, Janelle, Peter, Ethan, Saul, Jason, Sunita, Julia, Kristine, Christine, Mark, Pete, Julie, Chris, Jen, Audrey, Phil, Fernando, Paul, Sanjay, Katherine, Robert, Larry, Heather, Lambrini and Angela. You know who you are and why your name is here.

I would like to extend a very special thank you to Teresa Huegle, co-owner of Angie's Kitchen in Waterloo, for the years of weekend breakfasts and for willingly becoming my "adopted mom" who made sure, repeatedly, that I did in fact finished this PhD degree successfully.

Lastly, and most importantly, I acknowledge that it is impossible for me to express in words alone the full extent of my appreciation and gratitude owed to my family for their unconditional love and support, and their many willing sacrifices made, so that I could successfully complete my Ph.D.

# Dedication

*For my mother Shirley, my sister Deborah,*
*my daughter Megan and my son Brandon.*
*Thank you for always being there.*
*This thesis is for you.*

# Table of Contents

Abstract....iii
Acknowledgements....iv
Dedication....v

Table of Contents....vi
List of Figures....xvi
List of Tables....xix

Chapter 1 Introduction....1
1.1 Motivation....1
1.2 Software Failure, Error, and Fault....2
1.2.1 Software Aging....2
1.3 Runtime Monitoring....2
1.4 Interactive Session-Oriented Services....3
1.4.1 Why Session-Oriented Services?....3
1.4.2 Why Resource Ownership?....4
1.5 Software Concerns....4
1.5.1 Concern-Specific Evolving Structure....4
1.6 Scope....6
1.6.1 Why Application-Level?....8
1.6.2 Application Domain Considered....8
1.6.3 Individual Software Concern Considered....8
1.6.4 Specializing ResOwn....9
1.7 Objective....9
1.8 Contributions....9
1.9 Thesis Organization....10

Chapter 2 Background and Related Work....12
2.1 Introduction....12

2.2 Definitions and Concepts ........ 12
2.2.1 Resource and Resource Management ........ 12
2.2.2 Ownership ........ 13
2.2.3 Roles and Role Contexts ........ 14
2.3 Software Modeling ........ 15
2.3.1 Ontological Modeling ........ 15
2.3.1.1 Semantic Web ........ 17
2.3.1.2 Ontology for Software Engineering ........ 17
2.4 The Web Ontology Language (OWL) ........ 18
2.4.1 OWL Ontology Modeling Tools ........ 20
2.5 OWL-DL Ontologies ........ 21
2.5.1 Components of OWL-DL Ontologies ........ 21
2.5.1.1 Owl:Thing ........ 23
2.5.1.2 Named or Primitive Classes ........ 23
2.5.1.3 Subclasses ........ 23
2.5.1.4 Disjoint Class ........ 24
2.5.2 OWL Properties ........ 24
2.5.3 Inverse Properties (I) ........ 25
2.5.4 OWL Property Characteristics ........ 26
2.5.5 Property Domain and Ranges ........ 28
2.5.6 Describing and Defining Classes ........ 29
2.5.7 Existential Restrictions ........ 29
2.5.8 Reasoners and RACER ........ 29
2.5.9 Necessary and Sufficient Conditions ........ 30
2.5.10 Automatic Classification ........ 31
2.5.11 Universal Restrictions ........ 32
2.5.12 Open World Reasoning ........ 32
2.5.13 Closure Axioms ........ 32
2.5.14 Extending OWL-DL with Value Partitions ........ 33
2.5.15 Covering Axioms ........ 33
2.5.16 Cardinality Restrictions ........ 34
2.5.17 hasValue Restrictions ........ 35

2.6 Meta-modeling with UML ........ 35
2.6.1 UML ........ 37
2.7 SDL ........ 38
2.8 Runtime Monitoring ........ 39
2.8.1 What is Runtime Monitoring? ........ 39
2.8.2 RTM Types ........ 40
2.8.3 Date Structure Monitoring ........ 42
2.8.4 Model-Based Monitoring ........ 43
2.8.5 Other Related Monitoring Work ........ 45
2.9 Interactive Session Oriented Service Domain ........ 46
2.9.1 Real-Time Software Systems ........ 46
2.9.2 RTSS with Soft Deadlines ........ 47
2.9.3 Interactivity ........ 47
2.9.4 Session-Orientation ........ 47
2.9.5 Discrete Event-Driven Software ........ 47
2.9.6 Semi-Stationary Systems ........ 48
2.9.7 Non-Criticality ........ 48
2.10 The PBX ........ 49

Chapter 3 Overview ........ 52
3.1 Introduction ........ 52
3.2 Dynamic Software Structure Monitoring ........ 52

Chapter 4 The ResOwn Ontology ........ 56
4.1 Introduction ........ 56
4.1.1 Suggestion for Reading This Chapter ........ 58
4.1.2 Chapter Organization ........ 58
4.2 Conceptual Overview ........ 59
4.2.1 The ResOwn Prime Directive ........ 59
4.2.2 Top-Level Concepts and Properties ........ 60
4.2.3 Resources Benefits ........ 62
4.2.4 Beneficiary and Nonbeneficiary Owner Roles ........ 63

4.2.5 Direct and Indirect Proof of Ownership ....... 64
4.3 Taxonomy Overview ....... 64
4.3.1 Traversing the ResOwn taxonomy ....... 65
4.3.2 Top-Level Class Origins ....... 66
4.3.3 Baseline and Specialized ResOwn Class Hierarchies ....... 66
4.3.4 ResOwn Completeness ....... 67
4.3.5 Self-Contained and Distributed Subclass Hierarchies ....... 67
4.4 Properties ....... 68
4.4.1 ResOwn Property Extensions ....... 69
4.4.2 ResOwn Property Types ....... 69
4.4.2.1 Asserted Property Chaining ....... 71
4.5 ResOwn Core, Support, and Value Classes ....... 72
4.6 Proof of Ownership Instruments ....... 73
4.6.1 Instrument Class ....... 73
4.6.1.1 Instrument Capacity ....... 74
4.6.1.2 Instrument Persistency ....... 75
4.7 Defined Instrument Classes ....... 75
4.7.1 BaseInstrument Class ....... 75
4.7.2 ExtentInstrument Class ....... 76
4.7.3 TransferableInstrument Class ....... 76
4.7.4 NontransferableInstrument Class ....... 77
4.7.5 ClosedInstrument Class ....... 77
4.7.6 OpenInstrument Class ....... 77
4.7.7 LifeLongInstrument Class ....... 77
4.7.8 LifeLimitedInstrument Class ....... 77
4.8 Named Instrument Classes ....... 77
4.8.1 Titledeed Class ....... 78
4.8.1.1 EmbeddedTitledeed Subclass ....... 78
4.8.1.2 NontransferableTitledeed Subclass ....... 79
4.8.1.3 TransferableTitledeed Subclass ....... 79
4.8.1.3.1 NonreusableTransferableTitledeed Subclass ....... 79
4.8.1.3.2 ReusableTransferableTitledeed Subclass ....... 79

4.8.2 License Class ........ 79
4.8.2.1 SerialLicense Subclass ........ 80
4.8.2.2 ConcurrentLicense Subclass ........ 80
4.8.3 Proxy Class ........ 80
4.8.4 PowerOfAttorney Class ........ 81
4.8.5 PermitToHold Class ........ 81
4.9 Application Resources ........ 82
4.9.1 Resource Class ........ 82
4.9.1.1 Resource Capacity ........ 82
4.9.1.2 Resource Persistency ........ 83
4.10 Defined Resource Classes ........ 84
4.10.1 TransferableResource Class ........ 84
4.10.2 NontransferableResource Class ........ 84
4.10.3 Embedded Resource Class ........ 85
4.10.4 ClosedResource Class ........ 86
4.10.5 OpenResource Class ........ 86
4.10.6 NonconsumableResource Class ........ 86
4.10.7 ConsumableResource Class ........ 86
4.10.8 ExternalGatewayResource Class ........ 86
4.10.9 InternalResource Classes ........ 87
4.10.10 CompoundResource Class ........ 87
4.10.10.1 CompoundResource Class as a Supplier Class ........ 87
4.10.11 DispatchableConsumer (Inferred Resource) Class ........ 87
4.11 Application Resource Owners ........ 88
4.11.1 Owner, OwnerRole and OwnerBase Classes ........ 88
4.12 Defined OwnerBase Classes ........ 89
4.12.1 Passive Base Class ........ 90
4.12.2 Active Base Class ........ 90
4.13 Defined OwnerRole Classes ........ 91
4.13.1 BeneficiaryOwner Class ........ 91
4.13.1.1 CurrentOwner Subclass ........ 91
4.13.2 LicensedOwner Subclass ........ 92

4.13.2.1 ExclusiveLicensedOwner Subclass ........ 92
4.13.2.2 SharedLicensedOwner Subclass ........ 92
4.13.2.3 ProxiedOwner Subclass ........ 93
4.13.3 Nonbeneficiary Owner Class ........ 93
4.13.3.1 ContainmentOwner Subclass ........ 93
4.13.3.2 PermanentOwner Subclass ........ 94
4.13.3.3 DefaultOwner Subclass ........ 94
4.13.3.4 SurrogateOwner Subclass ........ 95
4.13.3.5 TemporaryOwner Subclass ........ 95
4.13.3.6 PreviousOwner Subclass ........ 95
4.13.3.6.1 The Virtual Previous Owner Stack ........ 95
4.14 Defined Owner Subclasses ........ 96
4.14.1 ConsumerOwner Subclass ........ 96
4.14.1.1 Dedicated Consumer Subclass ........ 96
4.14.1.2 DispatchableConsumer Subclass ........ 97
4.14.1.3 Classifying a DispatchableConsumer as a Nonconsumable Resource ........ 97
4.14.2 Supplier Subclass ........ 98
4.14.2.1 ActiveSupplier Subclass ........ 98
4.14.2.2 PassiveSupplier Subclass ........ 98
4.14.2.3 CachedResourceSupplier Subclass ........ 98
4.14.2.4 ManagedResourceSupplier Subclass ........ 99
4.14.2.5 PooledResourceSupplier Subclass ........ 99
4.14.2.6 SurrogateResourceSupplier Subclass ........ 99
4.14.2.7 CompoundResource (Inferred Supplier) Subclass ........ 99
4.14.3 Dispatcher Subclass ........ 99
4.15 OwnershipRight Value Classes ........ 100
4.15.1 AccessRight ........ 100
4.15.1.1 DataAccess ........ 100
4.15.1.2 ControlAccess ........ 100
4.15.2 Consumption Right ........ 100
4.15.3 ExchangeRight ........ 100
4.15.3.1 HoldingRight ........ 100

4.15.3.2 TransferRight ... 101
4.15.3.3 ReleaseRight ... 101
4.15.4 DelegationRight ... 101
4.15.4.1 ProxyingRight ... 101
4.15.4.2 PermittingRight ... 102
4.15.4.3 LicensingRight ... 102
4.15.4.4 AttorneyingRight ... 102

Chapter 5 ResOwn Instance Example and Ownership Scenarios ... 103
5.1 Introduction ... 103
5.2 An Asserted ResOwn Instance Example ... 104
5.2.1 Assumptions ... 104
5.2.2 ResOwn Specialization Methodology ... 104
5.3 A Specialized Inferred ResOwn Instance Example ... 107
5.3.1 ResOwn Specialization Issues ... 110
5.4 Resource Acquisition and Ownership Scenarios ... 111
5.4.1 Monitoring Event-Driven Snapshots ... 112
5.4.2 Transferable Resource Scenario ... 113
5.4.2.1 Scenario Semantics ... 113
5.4.2.2 Examples of Structural Errors ... 114
5.4.3 Nontransferable Resource Scenario ... 115
5.4.3.1 Scenario Semantics ... 115
5.4.3.2 Examples of Structural Errors ... 116
5.4.4 Embedded Resource Scenario ... 117
5.4.4.1 Scenario Semantics ... 117
5.4.4.2 Examples of Structural Errors ... 119
5.4.5 Surrogate Resource Supplier Registration Scenario ... 119
5.4.5.1 Scenario Semantics ... 120
5.4.5.1.1 Orphaned Power Of Attorney Scenario ... 121

Chapter 6 Session-Oriented Model of Computation ... 123
6.1 Introduction ... 123

6.1.1 Hierarchical Abstract Layering ........ 124
6.1.2 Behavioral Partitioning ........ 124
6.1.3 What is Session-Orientation? ........ 124
6.1.4 SOMOC: Internal Structural View ........ 124
6.1.5 SOMOC: External Behavioral View ........ 125
6.1.5.1 Service Provisioning Path ........ 126
6.1.5.2 Service Annulment Paths ........ 126
6.1.5.3 Idle Superstate ........ 127
6.1.5.4 Resynchronization ........ 127
6.1.6 Specification Refinement and Refinement Mapping ........ 127
6.1.7 Epoch of Behavior Models ........ 128
6.1.7.1 EoB Anatomy ........ 129
6.1.7.2 Quiescent States ........ 130
6.1.7.3 EoB Example ........ 130
6.1.7.4 Implementation States ........ 131

Chapter 7 Concern-Specific Dynamic Software Structure Monitor ........ 132
7.1 Introduction ........ 132
7.2 The Monitor ........ 133
7.2.1.1 Behavioral Considerations ........ 134
7.2.2 The Greybox Interpreter ........ 134
7.2.2.1 Interpreter Extensions ........ 135
7.2.3 The Pattern matcher ........ 136
7.2.4 The Dynamic Knowledge Base ........ 137
7.2.4.1 Dynamic Operations ........ 137
7.2.4.2 Tuple Lifespan and Persistency ........ 138
7.2.5 Behavioral versus Structural Considerations ........ 139
7.3 Monitoring Commands and Monitoring Constructs ........ 139
7.3.1 Monitoring Command Types ........ 140
7.3.2 Monitoring Construct Types ........ 141
7.3.3 Sensor Plan ........ 141
7.3.3.1 State- versus Input-Oriented Implementation Structure ........ 141

7.4 Deriving Interpretable Models........142
7.4.1 Structural Reduction........143
7.4.1.1 Classifying CEFSMs and Signals........143
7.4.2 State Evolution Monitoring........144
7.4.2.1 State Evolution Model Derivation........145
7.4.2.2 EoB Entry Point Monitoring Scenario........148
7.4.2.3 EoB Exit Point Monitoring Scenario........149
7.4.3 Structural Transaction Monitoring........149
7.4.3.1 Instance Monitoring Command Types........150
7.4.3.1.1 Object and Association Tuples........151
7.4.3.2 Structural Transaction Monitoring Command Types........151
7.4.3.2.1 ACQUIRE Monitoring Command and Construct Type........152
7.4.3.2.2 RELEASE Monitoring command and Construct Type........152
7.4.3.2.3 REGISTER Monitoring Command and Construct Type........153
7.4.3.2.4 UNREGISTER Monitoring Command and Construct Type........153
7.4.3.3 Structural Transaction Signaling........154
7.4.3.4 EoB Model Derivation Algorithm........156

Chapter 8 Evaluation........160
8.1 Introduction........160
8.2 ResOwn........160
8.2.1 Advantages........160
8.2.2 Known Limitations........164
8.3 Software Structure Monitoring........165
8.3.1 Advantages........165
8.3.2 Known Limitations........167

Chapter 9 Conclusions........168
9.1 Introduction........168
9.2 Conclusions........168
9.3 Research Contributions........169
9.4 Future Work........170

Appendix A Protégé-Owl and Racer Screenshots .......... 172

Appendix B Instrumentation Examples .......... 175

Appendix C ResOwn Class Hierarchy .......... 176

Appendix D Phone Handler Example .......... 179

References .......... 184

# List of Figures

Figure 1-1: The concern-specific evolving software structure. 5
Figure 1-2: Organizational block diagram of greybox software structure monitor. 7
Figure 2-1: Example architecture relevant for the verification [Jia05]. 15
Figure 2-2: Excerpt from host resources ontology. 18
Figure 2-3: OWL in the semantic web architecture [Dju05]. 19
Figure 2-4: Continuum of formal ways to express knowledge [Usc06]. 19
Figure 2-5: The three sublanguages of OWL. 20
Figure 2-6: Representation of individuals [Hor04]. 22
Figure 2-7: Representation of properties [Hor04]. 22
Figure 2-8: Representation of classes containing individuals [Hor04]. 23
Figure 2-9: The meaning of subclass in OWL [Hor04]. 24
Figure 2-10: Different types of OWL properties. 25
Figure 2-11: Example of an inverse property [Hor04]. 26
Figure 2-12: Example of a functional property characteristic [Hor04]. 26
Figure 2-13: Example of an inverse functional property characteristic [Hor04]. 27
Figure 2-14: Example of a transitive property characteristic [Hor04]. 27
Figure 2-15: An example of a symmetrical property characteristic [Hor04]. 28
Figure 2-16: An example domain and range for a property and inverse property [Hor04]. 28
Figure 2-17: Schematic description of a Pizza [Hor04]. 30
Figure 2-18: Description of CheesyPizza. 31
Figure 2-19: Example of closure axiom. 33
Figure 2-20: Example value partition. 34
Figure 2-21: Effects of a covering axiom: (i) uncovered; (ii) covered [Hor04]. 35
Figure 2-22: A general, 4-layer modeling architecture inspired by MDA. 36
Figure 2-23: Support UML constructs [Bel91]. 37
Figure 2-24: Subset of supported SDL constructs. 39
Figure 2-25: Example configuration for runtime monitoring for real-time software system. 41
Figure 2-26: Organizational block diagram of real-time supervision. 43
Figure 2-27: A typical event-driven software code distribution [Mem06]. 48

Figure 2-28: Layered interactive session-oriented service architecture for the PBX. ........................50
Figure 3-1: Internet telephony gateway with embedded PBX and greybox monitor. ........................53
Figure 3-2: Architectural overview of the software structure monitor. ............................................54
Figure 3-3: Block diagram overview of approach. ..........................................................................55
Figure 4-1: Organizational block diagram of the entire ResOwn ontology. .....................................57
Figure 4-2: Top-level ResOwn core classes and ResOwn properties. ..............................................62
Figure 4-3: Top-levels of the ResOwn class hierarchy. ..................................................................65
Figure 4-4: Top-level core classes and key object properties. ........................................................68
Figure 4-5: Asserted (defined and named) Instrument class hierarchy. ...........................................73
Figure 4-6: Inferred (defined and named) Instrument class hierarchy. ............................................74
Figure 4-7: Asserted (defined) Resource class hierarchy. ...............................................................83
Figure 4-8: Inferred (defined) Resource class hierarchy. ................................................................83
Figure 4-9: Asserted (defined) Owner, OwnerRole, and OwnerBase class hierarchies. ....................89
Figure 4-10: Inferred (defined) Owner class hierarchy. ..................................................................90
Figure 4-11: OwnershipRight VP value partition class hierarchy. .................................................101
Figure 5-1: Named Owner classes from the PBX. ........................................................................105
Figure 5-2: Named Resource classes from the PBX. ....................................................................106
Figure 5-3: Inherited Resource properties in an example of a Named Resource class. ....................107
Figure 5-4: Inferred Owner class hierarchy of the PBX's specialized ResOwn instance. ................109
Figure 5-5: Inferred Resource class hierarchy of the PBX's specialized ResOwn instance. ............110
Figure 5-6: Transferable Resource scenario. ................................................................................114
Figure 5-7: Nontransferable Resource scenario. ...........................................................................116
Figure 5-8: Compound and Embedded Resource scenario. ............................................................118
Figure 5-9: Surrogate Resource Supplier scenario. .......................................................................121
Figure 5-10: Orphaned Power Of Attorney scenario. .....................................................................122
Figure 6-1: Dual-view model: internal structural view. .................................................................125
Figure 6-2: Dual-view model: external behavioral view. ...............................................................126
Figure 6-3: Specification refinement and refinement mapping. ......................................................128
Figure 6-4: Example session behavior partitioned into epochs of behavior. ...................................129
Figure 6-5: Example mappings between $S_H$ and $S_{L.}$ .....................................................................131
Figure 7-1: Organizational block diagram of monitoring approach. ...............................................133
Figure 7-2: Internal organization of the greybox interpreter. ..........................................................135

Figure 7-3: One possible internal organization for the structural pattern matcher. .......................... 136
Figure 7-4: Internal organization of the dynamic knowledge base. ............................................. 138
Figure 7-5: Generic: (i) monitoring command; (2) monitoring construct...................................... 140
Figure 7-6: Block diagram of interpretable models derivation process. ........................................ 143
Figure 7-7: Example: (i) message sequence chart; (ii) layered CEFSMs and signals. ..................... 144
Figure 7-8: EoB ENTRY and EoB EXIT monitoring command. .................................................. 145
Figure 7-9: EoB entry point monitoring scenario. ......................................................................... 148
Figure 7-10: EoB exit point monitoring scenario. .......................................................................... 149
Figure 7-11: Structural transaction monitoring commands. .......................................................... 150
Figure 7-12: MSC for Consumer and Supplier. ............................................................................. 155
Figure 7-13: MSC Consumer and Compound Resource. .............................................................. 155

# List of Tables

Table 2-1: Tabular comparison of UML and OWL-DL based on [Usc06]. ....................................38
Table 2-2: PBX class / CEFSM descriptions. ....................................51
Table 4-1: ResOwn properties modeling core classes. ....................................70
Table 4-2: ResOwn properties modeling support classes. ....................................71
Table 4-3: ResOwn properties modeling value classes. ....................................71
Table 4-4: Top-level Instrument class definition. ....................................74
Table 4-5: Persistency versus Time. ....................................75
Table 4-6: Defined Instrument class definitions. ....................................76
Table 4-7: Titledeed class definitions. ....................................78
Table 4-8: License and Proxy class definitions. ....................................80
Table 4-9: Power Of Attorney and Permit To Hold class definitions. ....................................82
Table 4-10: Top-level Resource class definition. ....................................84
Table 4-11: Defined Resource class definitions. ....................................85
Table 4-12: Top-level Owner, Owner Role, and Owner Base class definitions. ....................................88
Table 4-13: Defined OwnerBase class definitions. ....................................91
Table 4-14: Effects of physical versus logical capacity. ....................................92
Table 4-15: Defined Beneficiary Owner Role subclass definitions. ....................................93
Table 4-16: Nonbeneficiary Owner Role subclass definitions. ....................................94
Table 4-17: Defined Owner subclass definitions. ....................................96
Table 4-18: Consumer subclass definitions. ....................................97
Table 4-19: Supplier and Dispatcher subclass definitions. ....................................98
Table 5-1: Named Owner classes for the PBX. ....................................107
Table 5-2: Named Resource class for the PBX. ....................................108
Table 5-3: Transferable Resource scenario. ....................................113
Table 5-4: Nontransferable Resource scenario. ....................................115
Table 5-5: Embedded Resource scenario. ....................................117
Table 5-6: Surrogate Resource Supplier scenario. ....................................120

Table 7-1: INSTANCE monitoring command. .......... 151
Table 7-2: ACQUIRE / RELEASE commands. .......... 153
Table 7-3: REGISTER / UNREGISTER command types. .......... 153
Table 8-1: Comparison of monitoring approaches. .......... 166

# Chapter 1

# Introduction

*"The proliferation of new semantics may be fun for semanticists, but developing a practical method for reasoning about systems is a lot of work."*

- M. Abadi and L. Lamport, 1991

## 1.1 Motivation

Rapid advances in computing hardware have led to more reliable platforms, but *software reliability* has not kept pace [Hang02, Sul91]. Software *failures* range from inconveniences in service applications, to financial loss in mission-critical applications, to loss of human life in safety-critical applications. To date, a substantial amount of research has been devoted to methods and techniques intended to maintain or improve software reliability such as fault avoidance, fault elimination, fault tolerance and formal verification [Ben03, Dod92, Gar98, Hla95, Lee90, Lyu95, Pek97, Sch95, Gao03]. Despite rigorous use of these methods in practice, faults remain in software in the order of one to ten per thousand lines of code [Tas02]. These *hidden* faults often do not surface until a software product has already been released and is operating in its production environment [Cha00, Kuh87, Mol93, Ost02]. Runtime monitoring can play an important part detecting *hidden* faults and ensure that a software system operates as intended in its production environments. This thesis investigates an *ontology*-based, *greybox* approach to dynamic software structure monitoring of interactive session-oriented *services* that are delivered by discrete event-driven, soft real-time software systems. The selected software concern is *application resource ownership*, and the software system used in the examples throughout this thesis is a small *private branch exchange (PBX)*.

## 1.2 Software Failure, Error and Fault

The standard definitions are adopted from [Iee90]:

- A *software failure* is defined as the inability of a system or component to perform its required functions within specified performance requirements.
- A *software error* is defined as the difference between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition.
- A *software fault* is defined as an incorrect step, process or data definition in a computer program.

If encountered during program execution, a *software fault*, under the right activation circumstances or conditions, may manifest as a *software error*. An *error* may propagate and result in other internal *errors* and/or an externally observable *failure*. *Software faults* are characterized as *Heisenbugs* or *Bohrbugs* [Vai01]. *Bohrbugs* are software *design* faults that should have been eliminated during testing, but were missed. *Bohrbugs* manifest an error upon each repeated activation. *Heisenbugs* are usually transient and only manifest errors during specific collusions of events or execution sequences.

### 1.2.1 Software Aging

*Software aging* is a temporal phenomenon that can occur in long-running, shared-resource software systems that must respond to asynchronous events, varying usage profiles and varying loads [Cas01,Vai01]. Software aging errors, such as resource leaks, may lead to service degradation as the usable service capacity monotonically decreases until a failure or outage occurs [Avri97].

## 1.3 Runtime Monitoring

In general, runtime monitoring approaches may be classified according to the amount of intrusion into the target software system they require:

- *Blackbox monitors* are unobtrusive and support failure detection. Blackbox monitors are limited in their ability to detect errors because they only monitor the topmost layer of abstraction; namely, those external processes that communicate directly with the environment. Some reasoning about the state of certain internal processes that communicate with external processes is possible, but typically limited again by space and time cost considerations. These approaches are

ideal for testing and for detecting failures in third-party production software whose requirements or high-level design specifications are available, but whose source code access may be restricted.

- *Whitebox monitors* are highly intrusive and are thus able to support both failure and error detection, but require intimate internal system knowledge and may literally require instrumenting every line of source code. For example, whitebox monitoring is similar to *whitebox testing* where the control structure of the procedural design of a system is used to derive test cases. The tester has knowledge of the internal workings, components and specifications of the product under test [Gao03]. *Whitebox monitoring* approaches are ideal for software debugging and execution tracing where storage, processing and retrieval of large volumes of collected, runtime data are possible.
- *Greybox monitors* offer a compromise. They are partially intrusive, thus allowing some degree of error detection, but they abstract away certain internal implementation details, thus reducing the amount of intrusion and runtime data processing requirements. This thesis presents a greybox approach to dynamic software structure monitoring.

## 1.4 Interactive Session-Oriented Services

This thesis presents an ontology-based approach for application resource ownership structure monitoring for the *interactive session-oriented services* application domain. *Interactivity* is a mode of service operation with an *"input-compute-output"* processing structure in which user commands (i.e., *inputs*) cause service responses (i.e., *outputs*) [Bac99, Bro01]. This thesis assumes that the service's software implementation is a set of *objects* that exchange runtime information via interactions [Amb88]. *Session-orientation* is a service delivery property in which an application executes cyclically through repeated activations [Pek03]. The concept of a *session* is widely used for both an end user and software system perspective [Chr01, Dos05, Haz01, Kol98, Mur98, Par00].

### 1.4.1 Why Session-Oriented Services?

The *session-oriented* concept is becoming increasingly important for *interactive* services [Dos05]. Many of today's network applications are *interactive session-oriented*, and accumulate a unique *session* state [Mur98, Par00, Son02]. For example, an *e-service* may be a *discrete-oriented*, short-running service such as an electronic shopping cart, or a *session-oriented*, long-running service such as collaborative text chatting [Chr01]. Other examples of interactive session-oriented services include

telephony, *voice-over-Internet-protocol (VoIP)*, *video-on-demand (VoD)*, online banking, and online ticketing.

### 1.4.2 Why Resource Ownership?

In *online* and *electronic* interactive services, *resource management* directly impacts the ability of the underlying software system to provide the specified (or contractual) *quality of service (QoS)* expected by the service's end users [Chr01, Dos05, Hau01]. Consistent and efficient resource allocation and management is hard, and often *nonfunctional requirements* depend heavily on the correct and efficient management of resources [Kir04]. For example, resource management problems can go undetected at low service loads causing only smoothly degrading service, while at higher loads, a sudden increase in demand may cause a failure or a major outage [Avr97]. This situation is analogous to a civil or mechanical engineering structure wearing out gradually over time until a sudden stress causes a failure to occur.

## 1.5 Software Concerns

In today's highly competitive online and electronic service environments, vendors and providers are increasingly looking to *autonomic concepts*, such as self-healing, self-adapting and self-optimizing software [Kep03], to help cope with maintaining stringent service dependability requirements and contractual *QoS* obligations. The integration of autonomic concepts into software systems that are responsible for delivering interactive session-oriented services presents an opportunity to software designers, developers and maintainers to manage, control and monitor individual software concerns within an operational software system. Examples of individual software concerns include resource ownership, data security, intrusion detection, performance, maintenance and system adaptation. Further, the ability to monitor the runtime health of a particular software concern would be very valuable; however, a formal way to specify and derive a monitoring model of the state-dependent structure of an individual concern is required.

### 1.5.1 Concern-Specific Evolving Structure

In software engineering, "*separation of concerns"* refers to the ability to identify, encapsulate and manipulate those parts of software that are *relevant* to a particular concept, goal, purpose or issue

[Par72]. Traditionally, software requirements are divided into *functional* and *nonfunctional* concerns. An individual behavioral or structural concern represents an abstraction of the software system's full behavior or structure, respectively. A clean separation of concerns reduces complexity, improves comprehensibility, increases traceability, limits the impact of change and helps to facilitate software evolution, adaptation, customization, integration and reuse [Tar99].

This thesis considers individual software concerns in terms of behavior and structure, as shown in Figure 1-1. Further, at runtime, a software system's concern-specific structure is not *static*; instead it changes or *evolves*[1] over time in a state-dependent manner. This change occurs because there is a correspondence between the concern-specific, *behavioral interactions* that occur between certain internal *objects* in the operational software system and the corresponding *association instances* or *structural links* that are created and/or destroyed as a result of those *behavioral interactions*. Therefore, at any given execution point, only a subset of all the possible concern-specific *structural links* that could occur will be in effect.

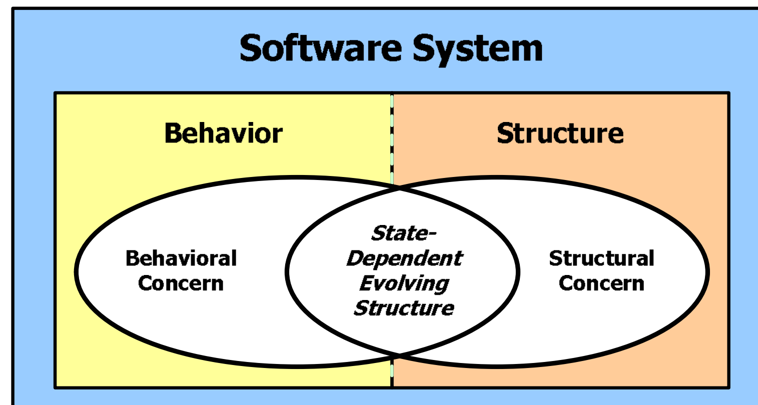



**Figure 1-1: The concern-specific evolving software structure.**

In *object-oriented* software systems, the specification of *evolving software structure* takes place on two levels: the object level and the conceptual class level. In this thesis (Chapter 5):

- An *object diagram* represents a *state-dependent snapshot* (i.e., individual view) of the selected

---

[1] In this thesis, *"the evolving software structure"* is a runtime concept distinct from the notion of software *evolution* that refers to the *maintenance phase* of the *software development life cycle* [Pfl06].

concern-specific software structure.

- A *class diagram* represents a *state-independent family of snapshots* (i.e., compound view) of the selected concern-specific software structure.

In this thesis, a behavior-driven, ordered sequence of individual snapshots is used to represent the evolving software structure of the operational software system for the select software concern. The addition and removal of individual links between pairs of objects represents a micro-step between snapshots and the full transition from object diagram to the next in the state-dependent sequence represents a macro-step. This approach provides a deeper understanding of the evolving structure of software systems beyond the conventional visibility offered by simply observing the operational software system's runtime behavior.

## 1.6 Scope

This thesis proposes a *greybox* approach to concern-specific *dynamic software structure monitoring* that uses a *concern-specific model*, as shown in Figure 1-2. While the monitor executes as a separate unit, the *monitoring interface* is comprised of *software sensors* woven into the target software system's implementation according to a manually derived *sensor plan*. The concern-specific model is derived (i.e., abstracted) from the target's formal behavioral specifications. During the derivation process, the model is extended with *special* model constructs called *monitoring constructs*. Each *monitoring construct* in the model corresponds to a specific *monitoring command* produced by an associated *software sensor* in the instrumented target software system.

The top-level architecture of the monitor is comprised of a *greybox interpreter* and a *tuple*-based *dynamic knowledge base*. The *greybox interpreter* uses the *dynamic knowledge base* to maintain a representation of the operational target software system's concern-specific evolving software structure. The greybox interpreter receives monitoring commands while interpreting the concern-specific model and updates the contents of the dynamic knowledge base to *match* the monitored portion of the target's actual software structure. The main focus of the work in this thesis is on the monitoring architecture and model derivations. A practical implementation of the monitor is outside the thesis scope.

A novel *Application Resource Ownership Ontology*, called ***ResOwn***, was created to provide a vocabulary, along with a set of *concepts*, *properties* and *property restrictions* for modeling the

*application resource ownership structure* for interactive session-oriented services. ResOwn is specified in the W3C's *Ontology Web Language Description Logic (OWL-DL)* [W3c04b]. The *behavior* of the software system is orthogonal to ResOwn and specified using the *Specification and Description Language (SDL)* [Itu91], a formalism based on *communicating extended finite state machine (CEFSM)* [Hie01]. This thesis assumes that the target software system's *source code*, SDL-based software *requirements* and SDL-based *software* design are available. The design is assumed to be a *refinement* of the requirements, and the source code is assumed to be a *refinement* of the design.

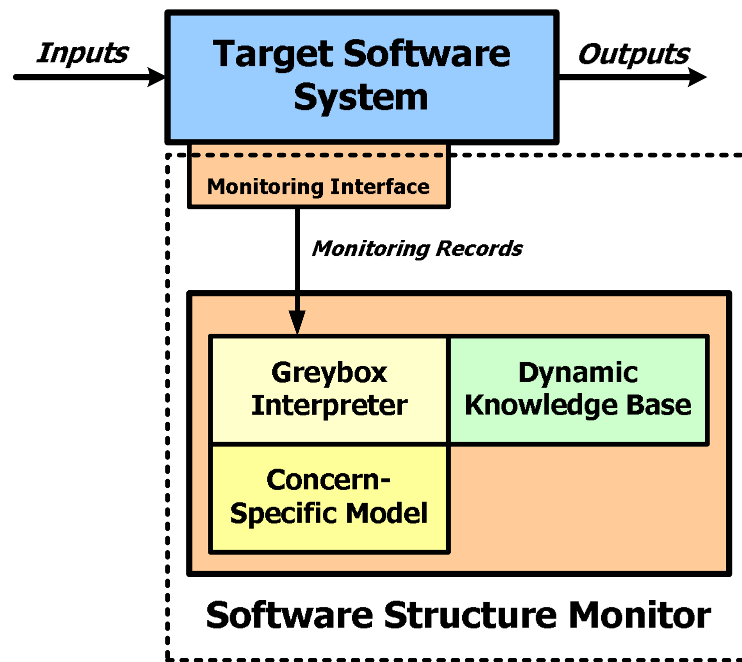



**Figure 1-2: Organizational block diagram of greybox software structure monitor.**

Traditional model-driven monitoring approaches tend to focus on *behavioral correctness* [Dvo91, Gat04, Hla95, Sav97] or *performance tuning* [Gar00, Pek03] rather than tracking evolving software structure. Monitoring concern-specific software structures can be very valuable. This thesis assumes that runtime structural knowledge can enhance blackbox monitoring, detect errors in operational software before they manifest as failures and give time for manual or autonomic *corrective* actions to be invoked. Runtime structural knowledge can also be used to make dynamic decisions with respect to reconfiguration of adaptive systems. In addition, monitoring only an individual structural software concern, such as *application resource ownership*, provides an *abstraction* that reduces the amount of

*intrusion* and *complexity* when compared to a full *whitebox* monitoring approach.

### 1.6.1 Why Application-Level?

The familiar *"end-to-end" argument* states that functions placed at low levels of abstraction in a system may be redundant or of little value when compared with the cost of providing them at that low level [Sal84]. The argument can be extended to *application-level services*, which know better than operating systems or runtime environments what the goal of their resource allocations and management decisions should be [Eng95]. An application-level monitor, like the one presented in this thesis, could provide runtime knowledge useful for autonomic control, management and maintenance and for dealing with generic error recovery from encountered *application faults* [Cha00].

### 1.6.2 Application Domain Considered

This thesis considers software systems that are real-time with soft deadlines, interactive, session-oriented, discrete event-driven, semi-stationary and non-critical, and whose behavior is modeled using CEFSMs. For concreteness, the PBX control program is used in many of the examples presented throughout this thesis. A detailed description of the PBX is given in Chapter 2.

### 1.6.3 Individual Software Concern Considered

This thesis considers the individual concern of *application resource ownership* for interactive session-oriented services whose concern-specific structure is formally described using the *ResOwn* ontology. The ResOwn ontology was constructed using the *Protégé-Owl* ontology development environment and verified and classified using the *RacerPro* reasoner[1] and inference engine[2]. These tools are described in more detail in Chapter 2.

---

[1] A *reasoner* is a software tool that can derive new, formally annotated facts from a set of predefined, formally annotated facts.

[2] An *inference engine* is a computer program that tries to derive answers from a knowledge base.

### 1.6.4 Specializing ResOwn

A methodology to specialize the concern-specific portion of the ResOwn ontology with application-level knowledge for a particular software system is presented in Chapter 5. The methodology is then applied to a concrete example that creates an application-specific instance of ResOwn using the object classes defined in the PBX. A reasoner is then used to automatically classify the inserted application classes, resulting in an inferred class hierarchy of the application-specific ResOwn instance. This inferred ResOwn instance is used in Chapter 6 for the derivation of the concern-specific model, and at runtime by the greybox interpreter when processing incoming monitoring commands from the instrumented target software system.

## 1.7 Objective

One of the primary objectives of the thesis is to investigate the use of a concern-specific ontologies from Knowledge Engineering to model structural concerns for software monitoring. The intended use of the ontology is for a concern-specific, model-based approach to greybox monitoring evolving software structures in operational software systems. The selected structural concern's vocabulary, concepts, properties and restrictions are specified and modeled using the ontology. The selected structural concern is *application resource ownership*. The application domain is *interactive session-oriented services*. The example target software system considered is the call processing software for a small *private branch exchange (PBX)*.

## 1.8 Contributions

The major novel contributions of this thesis are:

- A reusable and extensible, concern-specific ontology called ***ResOwn*** provides enriched concepts of application resource ownership borrowed from real-world legal and ownership ontologies. ResOwn is defined in the *Web Ontology Language Description Logic (OWL-DL)*, verified with a reasoner and tested using the PBX example.
- A methodology to create an application-specific ResOwn instance that specializes the concern-specific portion of the ResOwn ontology with application-level knowledge for a particular

software system.

- A dual-view, *Session-Oriented Model of Computation (SOMOC)* for interactive session-oriented services that relates observable, external service behavior to internal, evolving software structure.
- A greybox, concern-specific dynamic software structure monitoring *approach* and *architecture* devised for tracking the state-dependent *evolving software structure* of an operational software system.
- A pair of algorithms for deriving the *concern-specific monitoring model*:
  - An *algorithm* for deriving a *state evolution model* from the target's *software requirements* specification. The state evolution model allows the greybox interpreter to track the *specification state* (i.e., *macro-steps* in the evolving structure) of the operational target.
  - An *algorithm* for deriving a set of *epoch of behavior (EoB) models* from certain *slices* of the target's *software design* specification. Each EoB model contains the monitoring constructs that allow the greybox interpreter to track certain concern-specific *structural transactions* (i.e., *micro-steps* in the evolving structure) as they are reported by the instrumented target.

# 1.9 Thesis Organization

The remainder of this thesis is organized as follow:

**Chapter 2** introduces fundamental theories used in the thesis and discusses related work. A detailed introduction is given to the OWL-DL as the formalism used to model knowledge in ResOwn. The chapter also includes background concepts and related work in runtime monitoring.

**Chapter 3** provides an overview of the approach presented in this thesis including ResOwn and the greybox concern-specific monitor.

**Chapter 4** forms the core of the thesis and describes, in detail, the top-level concepts, asserted and inferred taxonomies, vocabulary, classes, properties and property restrictions of ResOwn.

**Chapter 5** presents the methodology for mapping ResOwn to a concrete application software example for PBX, resulting in an application-specific specialized ResOwn instance. The second part of chapter presents a number runtime snapshots (i.e., object diagrams) that describe some of the permissible dynamic resource ownership evolution patterns using the example PBX.

**Chapter 6** introduces a session-oriented model of computation (SOMOC) for modeling interactive session-oriented services that is used as the basis for deriving the concern-specific monitoring model described in detail in Chapter 7.

**Chapter 7** describes the actual greybox concern-specific dynamic software structure monitor approach and architecture, the syntax and semantics of monitoring commands and constructs, and the monitoring model derivation algorithms.

**Chapter 8** discusses the analytical evaluation of the ontology-based monitoring approach presented in the thesis, including the conceptual benefits and known limitations.

**Chapter 9** summarizes the presented research and suggests some future research work.

# Chapter 2

# Background and Related Work

*"Eventually, everything will just be knowledge."*

- K. Czarnecki, 2006.

## 2.1 Introduction

This chapter describes several topics which form the background and related material for the work presented in this thesis. Highlights of the chapter include a tutorial on OWL-DL-based ontologies that was compiled from a number of sources for this thesis and a review of related work.

## 2.2 Definitions and Concepts

### 2.2.1 Resource and Resource Management

In (*business) process modeling*, a *resource* is defined as a necessary item, tool or person and may include equipment, time, office space, people and techniques [Pfl06]. In software, a *resource* is defined as everything that is required by an application to provide its required service [Zsc04]. In this thesis, the definition of an *application resource* is adapted from [Kir04] to be an entity (i.e., **Resource** instance) that is available in limited supply such that there exists a *requestor* (i.e., **Consumer** instance) that needs the entity to perform a function, and there exists a *provider* (i.e., **Supplier** instance) that provides the entity upon request. Two important properties of a **Resource** instance are:

(1) it can be allocated and used by an application, and (2) it has a maximum capacity[1] [Zsc04]. *Resource management* is the process of controlling the *availability* of Resource instances for **Consumer** instances to ensure that: **Resource** instances are available when needed, the *resource lifecycle* is deterministic, and **Resource** instances are released in a timely manner to ensure software system *liveliness* [Kir04].

### 2.2.2 Ownership

In the Anglo-American legal system, *ownership* is defined as a *relationship* between a legal person (i.e., individual, group, corporation or government) and an object [Bri06]. The object of concern may be corporeal or completely a creature of the law such as a patent, copyright or annuity. Although ownership can be treated as a single, conceptual object, it is often necessary to view ownership as a *bundle of rights* [Mcca02, Sha05, Yip02, Yip03, Yip04]. The concept of ownership provides several main perspectives:

- The notion of *possessing property*.
- The notion of *legal ownership rights* that may vary according to one's relationship with some property.
- The ability to prove those legal ownership rights have been properly granted via some legally binding *proof of ownership instrument*.
- The concept of an *owner* (i.e., **Owner** instance), where an **Owner** instance can itself be viewed as property and be owned by other **Owner** instances.

In this thesis, *application resource ownership* is defined as a relationship between a physical **Resource** instance and one or more physical **Owner** instances. An **Owner** instances has certain **Ownership Right** instances applicable to a **Resource** instance, along with the ability to prove those rights via a logical *proof of ownership* **Instrument** instance. Lastly, an **Owner** instance itself can be a **Resource** instance, and conversely, a **Resource** instance can be an **Owner** instance.

[1] Resources with *unlimited* availability and *unlimited* capacity are not considered.

### 2.2.3 Roles and Role Contexts

Popular role theory, or the observation that human beings play multiple, context-sensitive roles with corresponding identities, expectations and behaviors, originated in psychology and sociology in the 1960s and 1970s [Bid66, Bid79]. In today's software domain, role theory has been integrated with the object-oriented paradigm to be applied to areas such as role-based security access modeling [Cra03] and role-based, interactive agent systems [Cab03, Woo00]. In most object-oriented paradigms, objects are independent, isolated entities with a unique identity and a uniform singleton set of behavioral capabilities [Kri95]. However, in operational object-oriented software, objects tend not to be isolated entities, but rather they relate to each other through interactions within a number of different settings or *contexts*.

In accordance with [Kri95], an *object* is said to play a *role* when a *role instance* is assigned to that object. In this thesis, the assignment of a role instance to an object is assumed to occur *implicitly* via the *structural context* that the object finds itself in. For example, consider a sequence of interactions between a set of objects, $O_1$, $O_2$, and $O_3$. An internal observer could deduce that $O_2$ plays a certain role in the resulting state-dependent software structure (i.e. *structural context*) that has evolved between $O_1$, $O_2$, and $O_3$. However, if that $O_2$ also participates in a sequence of interactions with $O_4$, and $O_5$, then an internal observer could conclude that $O_2$ now plays a different, second role in the structural context that now exists between $O_2$, $O_4$, and $O_5$. Hence, in this example, the same object ($O_2$) may simultaneously play two distinct, context-sensitive roles. This approach is preferable to viewing role instances as a static property because the assignment of role instances can change in synchronization with evolving structure.

In [Jia05], a networked service composed of structural and behavioral arrangements of service components is presented. Service components execute as nodes representing physical processing units such as servers, routers, switches, phones, laptops and PDAs. A service component, as shown in Figure 2-1, is a generic software component, called an *actor,* whose executable functionality, or *role*, is based on downloadable *Extended Finite State Machines (EFSM)*. A *session role* is a projection of an actor's role with respect to the interaction between pairs of actors. Actors play roles according to *manuscripts* and a *director* manages their operations. In this thesis, objects are modeled as **Owner** or **Resource** instances. In addition, an object may also play one or more **Owner Role** instances. The set of **Owner Role** instances played depends of the different *structural resource ownership contexts* in which the object finds itself.

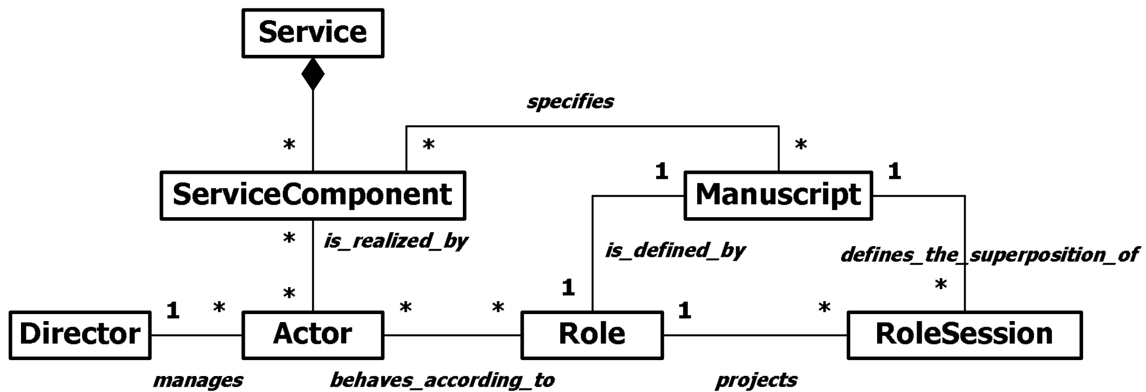



**Figure 2-1: Example verification architecture [Jia05].**

# 2.3 Software Modeling

*Models* are abstractions [Sel03] that eliminate irrelevant details or simplify concepts; that is, a simplified view of a system that increases understandability, predictiveness and/or accuracy and lowers cost. The three subsections that follow introduce three modeling paradigms [Atk03]: *ontology modeling* using the *Web Ontology Language Description Logic (OWL-DL)* [W3c04a, W3c04ab], *meta-modeling* using the *Unified Modeling Language (UML)* [Omg05a, Omg05b] and *behavioral modeling* using the *Specification and Description Language (SDL)* [Itu91]. The subsections provide only a high-level discussion of the key modeling concepts of each paradigm. Interested readers are directed to a suitably detailed reference or tutorial for further investigation.

## 2.3.1 Ontological Modeling

There are numerous definitions of what exactly constitutes an *ontology*, depending on the discipline the applicable ontological concepts originate. In *philosophy*, "Ontology, understood as a branch of *meta-physics*, is the science of being in general, embracing such issues as the nature of existence and the categorical structure of reality. Different systems of ontology propose alternative categorical schemes. A categorical scheme typically exhibits a hierarchical structure, with 'being' or 'entity' as the topmost category, embracing everything that exists [Hon95]." Today's interest in ontologies extends well beyond meta-physics. In the *knowledge representation* domain, an *ontology* may be defined as:

- A specification of a conceptualization, an explicit specification of some topic, or a formal and declarative representation of some subject area [Gru93, Gua98].
- A set of knowledgeable terms (i.e., vocabulary), the semantic interconnections, and some rules of inference and logic for some particular topic [Hen01].
- The basic structure, skeletal knowledge, or armature around which knowledge bases can be built or integrated at the knowledge level, independent of any particular implementations [Dev99, Swa99].

The main ontological components, relevant to a particular domain of discourse, are conceptualizations or concepts (i.e., *classes*) organized into a hierarchical taxonomy, with relations among the concept, and constraints, restrictions or axioms (i.e., *properties*) distinguishing concepts and refining definitions and relations [Noy97]. A sample of the number of different kinds of ontologies discussed in the literature includes:

- An *Upper Ontology* [Noy97], *Top-Level Ontology* [Sha05], *Foundation Ontology* defines very general base concepts that that are the same across all domains and thus support ontology development and facilitate common-sense, human-like understanding and reasoning. The aim is to have a large number on ontologies accessible under this upper ontology.
- A *Domain Ontology* [Noy97] defines the terminology and concepts relevant to a particular topic or area of interest. As systems that rely on domain ontologies expand, they often need to merge domain ontologies into a more general representation. Different ontologies in the same domain can also arise due to different perceptions of the domain based on cultural background, education, ideology or because a different representation language was chosen.
- A *Business Process Ontology* [Noy97] defines the inputs, outputs, constraints, relations, terms and sequencing information relevant to a business process. A business process ontology serves two distinct purposes. Firstly, it makes knowledge explicit and allows for knowledge sharing among domain experts and information technology people engaged in software design and development. Secondly, since it includes machine-readable definitions of concepts, it serves as a requirements specification from which a number of software artifacts can be generated.
- An *Interface Ontology* [Noy97] defines the structure, content, messaging and other restrictions for a particular interface (e.g., Application Programming Interface).

- A *Service Ontology* [Noy97] defines a core set of constructs for describing the vocabularies and capabilities of services such as the *World Wide Web Consortium (W3C) Web Service Modeling Ontology*[1].
- A *Role Ontology* defines terminology and concepts relevant for a particular end-user.

#### 2.3.1.1 Semantic Web

The role of ontologies in the Semantic Web is to establish additional levels of syntactic and semantic interoperability on the Web. *Syntactic interoperability* pertains to reusability in parsing data. *Semantic interoperability* pertains to mappings between terms within the data using some form of content analysis. Ontologies serve to standardize and provide interpretations for Web content using a *semantic markup* to make that Web content *machine-understandable*.

#### 2.3.1.2 Ontology for Software Engineering

In software engineering, ontologies may be used to model *domain-specific concepts* and *software structure*. A domain-specific ontology provides a vocabulary of concepts from a selected application domain, along with a set of logical statements that describe what concepts are, and how concepts can or cannot be related to each other. The philosophy of constructing ontologies free from implementation bias, as studied in the ontology field [Gru93], is very attractive when modeling the *structure* of an individual *software concern*. This was one of the key reasons this thesis chose to use a *concern-specific ontology* to model *application resource ownership structure*.

Ontologies for modeling software philosophically and practically support the use of *platform independent knowledge* at the domain-level. In addition, these same domain-specific ontologies can be extended and transformed into *platform-* or *application-specific* models via a process known as *specialization*. In this way, ontological languages have *declarative* power that can improve object-oriented models, plus support *automatic reasoning* and *inferencing*. Inference may be used to determine *multiple inheritance* for *specialized conceptualizations* from the *declared* or *asserted class hierarchy* of an ontology automatically, rather than manually. For example, this thesis uses automatic inference with the concern-specific, *Application Resource Ownership Ontology (ResOwn)* (described in Chapter 4) to *automatically* determine the possible **Resource** or **Owner** classes an *object class* may

---

[1] http://www.w3.org/Submission/2005/SUBM-WSMO-20050603/

belong to from a target software system.

In [Yu5], an OWL-based ontological approach is presented to monitor grid resources. The approach focuses on integrating and sharing *real-time status information* of voluntary nodes for *quality of service (QoS) management*. The designed ontology originates from the *Management Information Base (MIB)* in network management for open network devices monitoring and management. The taxonomy of classes in the *host resources ontology* is displayed in Figure 2-2. The presented ontology is similar to ResOwn in that it specified using OWL and focuses on resources. The ResOwn ontology differs in that it provides a conceptually rich and concern-specific domain of discourse for application resource ownership structure that integrates knowledge from a different domain of discourse including the software domain, the client-server domain, the legal and real property domains and the interactive service domain.

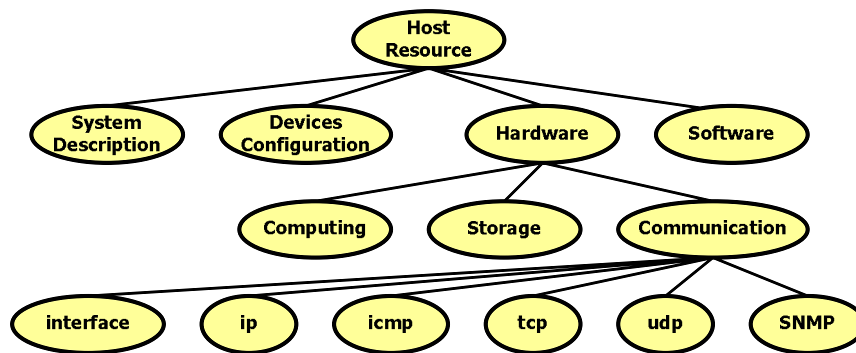



**Figure 2-2: Excerpt from host resources ontology.**

## 2.4 The Web Ontology Language (OWL)

The *Web Ontology Language (OWL)* is a set of *eXtensible Markup Language (XML)* elements and attributes, with well defined meaning, that are used to define terms and their relationships [Hor04, W3c04a, W3c04b]. As shown in Figure 2-3, OWL is actually an extension of the *Resource Description Framework (RDF)* and *RDF Schema*, which in turn extends *XML* and *XML Schema* [Dju05]. OWL is one entry on the continuum of ways to express knowledge, as shown in Figure 2-4.

There are three *species* of OWL, as shown in Figure 2-5 [W3c04a, W3c04b]:

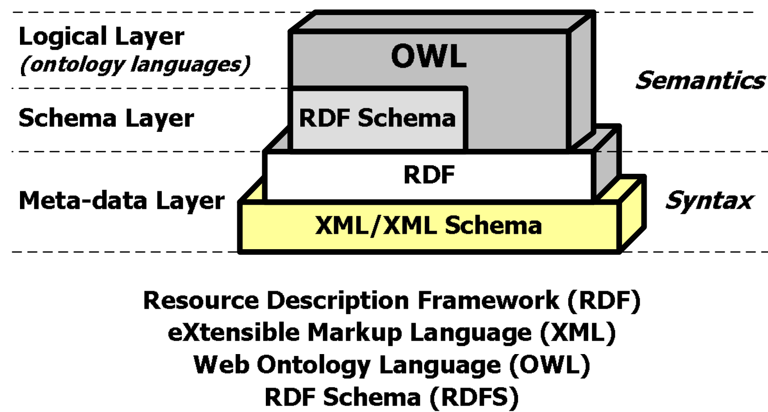



**Figure 2-3: OWL in the semantic web architecture [Dju05].**

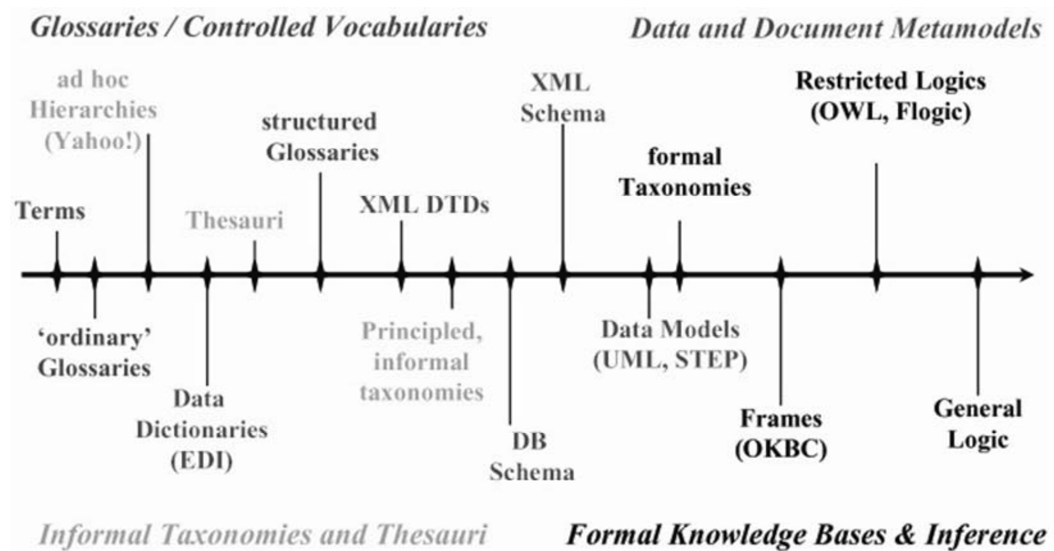



**Figure 2-4: Continuum of formal ways to express knowledge [Usc06].**

- *OWL-Lite* has limited expressiveness and is suitable for simple class hierarchies and constraints. *Cardinality* is restricted to values of 0 or 1.

- *OWL-DL* supports *description logics* [Baa03] and *automated reasoning* and is the OWL species used throughout this thesis. OWL-DL has maximum expressiveness while maintaining computational completeness (i.e., all conclusions are guaranteed) and decidability (i.e., all conclusions finish in a finite time). OWL-DL includes all language constructs but certain constructs can only be used under certain restrictions (e.g., a class cannot be an instance of

another class).

- *OWL-Full* has the most expressiveness and syntactic freedom (e.g., a class may be treated simultaneously as a collection of individuals and as an individual itself), but offers no computational guarantees and therefore, does not support automated reasoning.

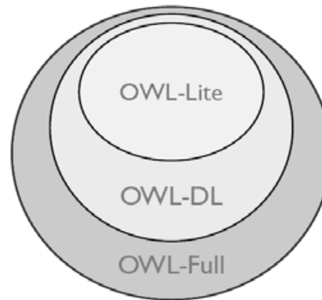



**Figure 2-5: The three sublanguages of OWL.**

## 2.4.1 OWL Ontology Modeling Tools

The ResOwn ontology was constructed, tested, debugged, and visualized using the following tools:

- *Protégé-OWL*[1] is a tool and development environment for ontologies and knowledge-based systems comprised of *Protégé* (v3.1.1) and the *OWL-Plug-in* (v2.1). The tool provides an interactive facility to iteratively devise, construct, test, debug and classify the ResOwn ontology. The tool runs on the Sun *JVM*[2] (v1.5.0_06) for *Windows XP Professional* (v2002, SP2). In this thesis, Protégé-OWL provided the author with an interactive facility to iteratively devise, construct, test, debug and classify the *ResOwn* ontology presented in Chapter 4.

- The *RacerPro*[3] (v1.9.0) is a reasoner and inference engine used in conjunction with the *Protégé-OWL* tool for OWL-DL-based ontologies. RacerPro allowed the author to automatically check and manually debug the consistency of the ResOwn asserted class hierarchy. RacerPro also

[1] http://protege.stanford.edu/overview/protege-owl.html

[2] http://www.sun.com/java/

[3] http://www.racer-systems.com/

automatically found implicit subclass relationships when generating the ResOwn inferred class hierarchy and computed equivalent ResOwn classes. The RacerPro *academic research license* was provided to the author free of charge upon request by *Racer Systems GmbH and Co. KG*. This thesis used the *RacerPro* reasoner.

- *OWLViz*[1] (v14) is a plug-in that allows *Protégé-OWL* users to visually display selected portions of both an OWL-DL ontology's asserted and inferred class hierarchies. The author used *OWLViz* to create the various ResOwn screenshots presented throughout this thesis.
- *Graphviz*[2] is an open source graph visualization program required specifically by *OWLViz* and is used to represent structural ontological information as diagrams of abstract graphs and networks.

# 2.5 OWL-DL Ontologies

Much of the material information presented in the subsections that follow was condensed, adapted and/or modified by the author from several sources [Hor04, Knu04a, Knu04b]. Many of the diagrams were redrawn from those originally appearing in the detailed *Protégé-OWL* tutorial [Hor04].

## 2.5.1 Components of OWL-DL Ontologies

An OWL ontology consists of these components.

- An *individual* (i.e., *Instance*) is a *constant* in *Description Logic* and represents an *object* in the *domain of discourse* in OWL. OWL-DL, being based on Description Logic, does imposed the *Unique Name Assumption*. This means that just because two names are *different* in OWL-DL does not mean they refer to *different* individuals. Two different OWL-DL class names, for example, may refer to the *same* individual. Therefore, it must be explicitly stated in OWL-DL whether individuals are the same as each other or different from each other. A representation of some individuals (represented by diamonds) is shown in Figure 2-6.

---

[1] http://www.co-ode.org/downloads/owlviz/co-ode-index.php

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

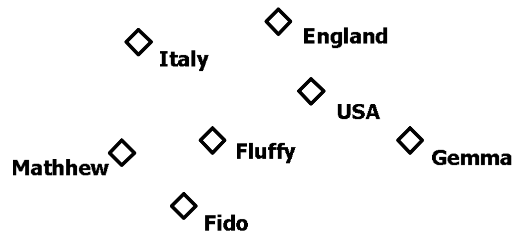



**Figure 2-6: Representation of individuals [Hor04].**

- A *Property* (i.e., Slot) is *role* or *binary predicate* in *Description Logic* and represents a binary relation in OWL that links two individuals together. In Figure 2-7, the property ***hasSibling*** links the individual **Matthew** to the individual **Gemma**. Properties can have an *inverse*. For example, the *inverse* of ***hasOwner*** is ***IsOwnerOf***. *Functional* properties are limited to having a single value. Properties can also be *symmetrical* or *transitive*. These property characteristics are explained in Section 2.5.4.

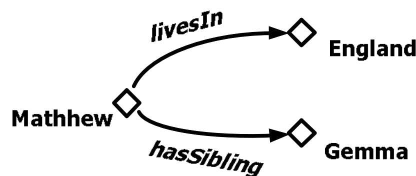



**Figure 2-7: Representation of properties [Hor04].**

- *A Class* is a *concept* or *unary predicate* in *Description Logic* and represent a *set of individuals* in OWL that are defined using formal, mathematical descriptions that state precisely the requirements for membership in the class. For example, in Figure 2-8, the class **Person** contains the individuals **Matthew** and **Gemma**, the class **Pet** contains **Fluffy** and **Fido**, and the class **Country** contains **Italy**, **England** and **USA**. Classes are shown as circles or ovals similar to *Venn diagrams*. Classes are concrete representations of *concepts* and may be organized in a superclass-subclass hierarchy called taxonomy. Subclasses specialize, or are subsumed by, their superclasses. Consider, for example, the classes **Animal** and **Cat** where **Cat** may be a subclass of **Animal** so that **Animal** is the superclass of **Cat**. This says that: (1) all cats are animals, (2) all members of the class **Cat** are also members of the class **Animal**, and (3) being a **Cat** implies being an **Animal**. One of the key features of OWL-DL is that this subsumption relationship can be automatically computed by a reasoner. OWL classes are essentially descriptions that specify

the conditions that must be satisfied by an individual for that individual to be a member of a particular defined or primitive class.

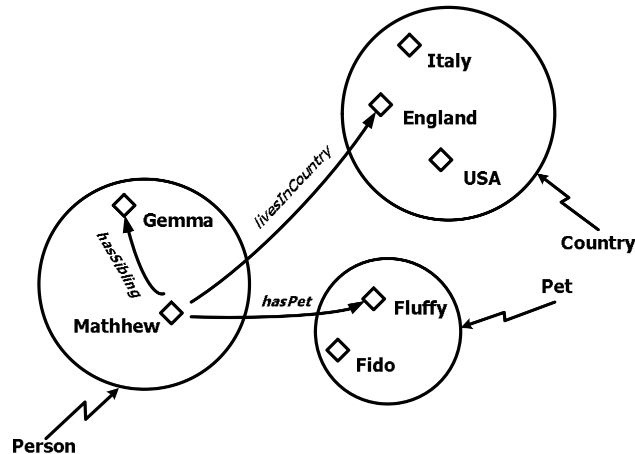



**Figure 2-8: Representation of classes containing individuals [Hor04].**

### 2.5.1.1 Owl:Thing

Every empty OWL ontology contains one *root* class called **owl:Thing**. **owl:Thing** is part of the OWL vocabulary and represents the set containing all possible individuals. Therefore, all classes in an OWL ontology are subsumed by **owl:Thing**.

### 2.5.1.2 Named or Primitive Classes

Although there is no mandatory naming convention, OWL classes are normally named using the **CamelBack** notation; that is, starting with a capital letter and without any spaces. Examples of named classes include **Pizza**, **PizzaTopping**, and **MargheritaPizza**.

### 2.5.1.3 Subclasses

In OWL, belonging to a subclass is a necessary condition for that individual to also belong to the subclass's superclass. For example, if a **VegetableTopping** is a subclass of **PizzaTopping**, then being an instance of **VegetableTopping** implies being instances of **PizzaTopping**, without exception. This means that if an individual is a **VegetableTopping** then it *necessarily implies* that the same individual is also a **PizzaTopping**, as shown in Figure 2-9.

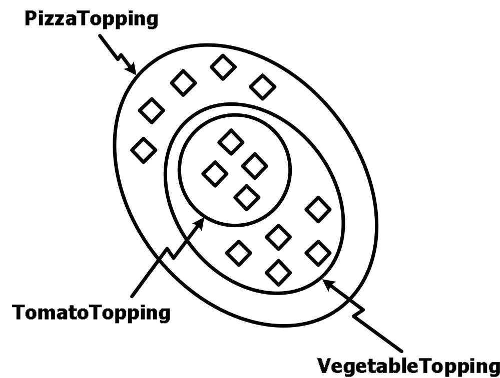



**Figure 2-9: The meaning of subclass in OWL [Hor04].**

#### 2.5.1.4 Disjoint Class

If two or more classes are *disjoint*, then an individual cannot be an instance of more than one of those disjoint classes. This must be stated explicitly because OWL classes are assumed to overlap. Therefore, one cannot assume that an individual is not a member of a particular class simply because the individual has not been *asserted* to be a member of that class. Therefore, by making a group of classes disjoint, conceptually separates them and ensures that an individual asserted as a member of one class in the group cannot be a member of another class in the group. This means that it is not possible for an individual to be a member of a combination of these classes. For example, if the classes **Pizza**, **PizzaTopping**, and **PizzaBase** are specified as disjoint, then an individual could not be both a pizza and a pizza base.

### 2.5.2 OWL Properties

OWL properties represent relationships between pairs of individuals. There are types:

- *Object properties* link an individual to an individual. For example, Figure 2-10(i) shows an object property ***hasSister*** linking the individual **Barry** to the individual **Debbie**. The ResOwn ontology, presented in Chapter 4, only uses object properties.
- *Datatype properties* link an individual to an XMLS datatype value or an RDF literal. For example, Figure 2-10(ii) shows a datatype property ***hasAge*** linking the individual **Barry** to the data literal '**46**' which has type **xml:Integer**.
- *Annotation properties* meta-data on the model which may be added to classes, individuals or

properties and are not instantiated with individuals[1]. For example, Figure 2-10(iii) shows an annotation property ***dc:creator*** linking the individual '**Thesis**' to the data literal (string) "**Barry Pekilis**".

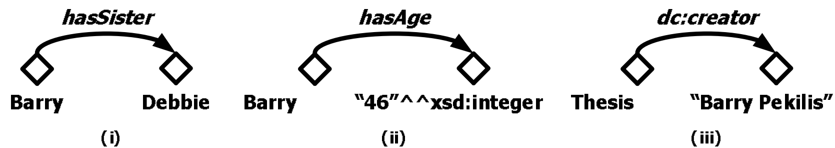



**Figure 2-10: Different types of OWL properties.**

Although there is no mandatory naming convention, OWL properties are normally named starting with a lower case letter, no spaces and the remaining words capitalized. Normally, a property name is prefixed with the word ***has*** or ***is*** such as ***hasPart***, ***isPartOf***, ***hasManufacturer*** or ***isProducerof***. Further, in OWL, properties may have *sub-properties* that specialize a *super-property* in a hierarchy of properties. For example, in a famous *Pizza Ontology* [Hor04, Rec04] the properties ***hasTopping*** and ***hasBase*** are created as sub-properties of ***hasIngredient***. This implies that pairs of individuals linked by either the ***hasTopping*** or ***hasBase*** property are also related to each other via the ***hasIngredient*** property.

### 2.5.3 Inverse Properties (I)

Each object property may have a corresponding *inverse property (**I**)*. If some property links individual **a** to individual **b** then its inverse property will link individual **b** to individual **a**. For example, as shown in Figure 2-11, shows the property ***hasParent*** and its inverse property ***hasChild***. This implies that if **Matthew** ***hasParent*** **Jean** then because of the inverse property, it can be *inferred* that **Jean** ***hasChild*** **Matthew**.

---

[1] This is similar to a Java comment.

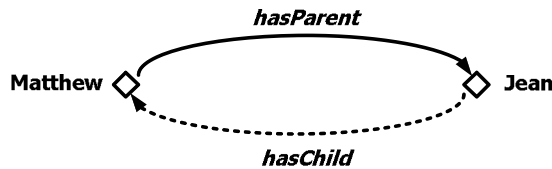



**Figure 2-11: Example of an inverse property [Hor04].**

### 2.5.4 OWL Property Characteristics

In OWL, the meaning of properties is enriched through the use of *property characteristic*:

- If a property is a *functional property*, then for a given individual, there can be at most one individual that is related to the individual via the functional property. For example, Figure 2-12 shows the functional property ***hasBirthMother*** which means that an individual can only have one birth mother. If the individual **Jean** ***hasBirthMother*** **Peggy** and the individual **Jean** ***hasBirthMother*** **Margaret**, then because ***hasBirthMother*** is a functional property, it can be inferred that **Peggy** and **Margaret** are the *same* individual. If **Peggy** and **Margaret** were explicitly stated as two different individuals, then the above statements would be inconsistent.
- If a property is an *inverse functional property*, then it means that the *inverse* property is *functional*. Figure 2-13 shows an example of the inverse functional property ***isBirthMotherOf***, the inverse property of ***hasBirthMother***. In this example, since ***hasBirthMother*** is functional, ***isBirthMotherOf*** is also defined as functional. If **Peggy** is specified as the birth mother of **Jean**, and **Margaret** is also specified as the birth mother of **Jean**, then it can be *inferred* that **Peggy** and **Margaret** are the same individual.

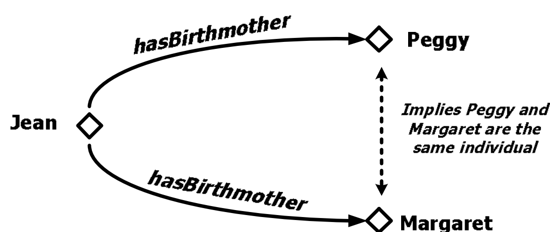



**Figure 2-12: Example of a functional property characteristic [Hor04].**

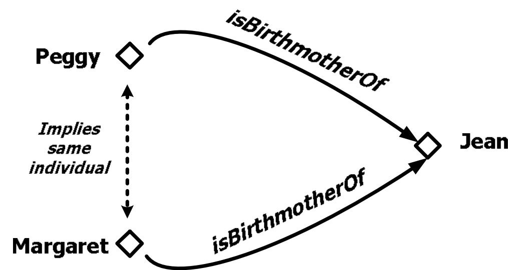



**Figure 2-13: Example of an inverse functional property characteristic [Hor04].**

- If a property is a *transitive property*, and the transitive property ***P*** relates an individual **a** to an individual **b**, and also an individual **b** to an individual **c**, then it can be inferred that individual **a** is related to individual **c** via ***P***. For example, the example transitive property ***hasAncestor*** is given in Figure 2-14. If an individual **Matthew** has an ancestor **Peter**, and **Peter** has an ancestor **William**, then it can be inferred that **Matthew** has an ancestor **William**. If a property is transitive, then its inverse should be transitive too. Transitive properties cannot be functional.
- If a property is a *symmetrical property*, and the symmetrical property ***P*** relates an individual **a** to an individual **b**, then individual **b** is also related to individual **a** via property ***P***. Figure 2-15 shows an example in which the individual **Matthew** is related to the individual **Gemma** via the symmetrical property ***hasSibling***. Therefore, it can be inferred that **Gemma** must also be related to **Matthew** via the ***hasSibling*** property. In other words, ***hasSibling*** is the inverse of itself and, so, if **Matthew** is a sibling of **Gemma**, then **Gemma** must be a sibling of **Matthew**.

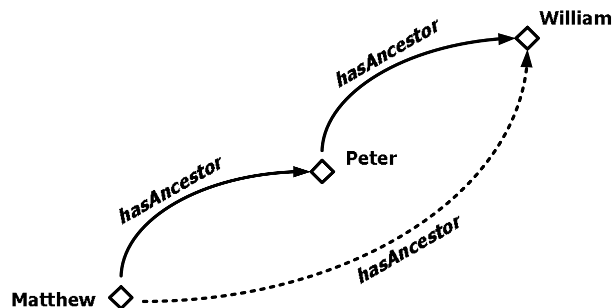



**Figure 2-14: Example of a transitive property characteristic [Hor04].**

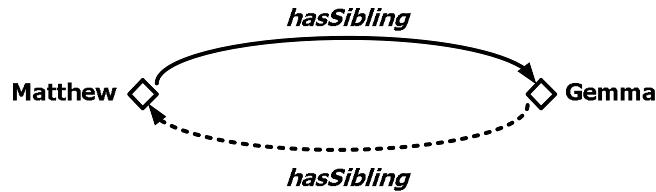



**Figure 2-15: An example of a symmetrical property characteristic [Hor04].**

## 2.5.5 Property Domain and Ranges

Every OWL property has a specified *domain* and a *range*, where a property links an individual from its domain to an individual in its range. In the example excerpt from the Pizza ontology, as shown in Figure 2-16, the property ***hasTopping*** links individuals from the class **Pizza** in the property's domain to individuals belonging to the class **PizzaTopping** in the property's range. Similarly, the inverse property ***isToppingOf*** links individuals from **PizzaTopping** in the domain to individuals belonging to **Pizza** in range. OWL property domains and ranges are *axioms* in reasoning. For example, consider individual **a** and individual **b** along with the assertion that **a** ***hasTopping*** **b**. Then it can be inferred that **a** is a member of class **Pizza** and **b** is a member of class **PizzaTopping**. Further, if the property ***hasTopping*** has the domain **Pizza** and the property is applied to a class ***IceCream*** (i.e., individuals that are members of the class **IceCream**), and if **Pizza** is not explicitly specified as *disjoint* from **IceCream**, then it could be inferred that the class **IceCream** is a subclass of **Pizza**. It is, however, possible to specify multiple classes as the range for a property. This is interpreted as the *union* of the classes. For example, if the range of a property has the classes **Man** and **Woman**, the range of the property is interpreted as **Man** *union* **Woman**.

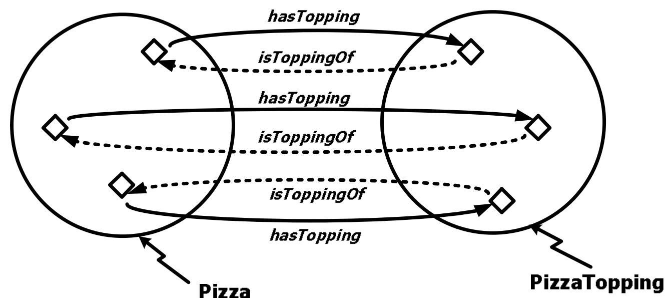



**Figure 2-16: An example domain and range for a property and inverse property [Hor04].**

### 2.5.6 Describing and Defining Classes

In OWL, properties are used to create restrictions that describe and define classes. Restrictions restrict the individuals that may belong to a particular class. Restrictions describe an anonymous or unnamed class of individuals that can satisfy the restriction. Restrictions fall into three categories:

- *Quantifier restrictions* are composed of a *quantifier*, a *property* and a *filler*. In OWL-DL, quantifiers produce an anonymous class of individuals. The two types of quantifiers are:
    - The *existential quantifier* (∃), which is interpreted in OWL as *"some values from."*
    - The *universal quantifier* (∀), which can be interpreted in OWL as *"all Values From."*
- Cardinality restrictions (i.e., minimum, maximum, exact).
- hasValue restrictions (϶).

### 2.5.7 Existential Restrictions

*Existential restrictions* (∃) are the most common type of restriction used in OWL ontologies. For a set of individuals, an existential restriction specifies the existence of *at least one* relationship along a property to an individual that is a member of a specific class. For example (∃ ***hasBase*** **PizzaBase**) describes all the individuals that have at least one relationship along the ***hasBase*** property to an individual that is a member of the class **PizzaBase**. Notice these are *necessary conditions*. As shown in Figure 2-17, for something to be a **Pizza**, it is *necessary* for it to have (*at least one*) **PizzaBase**. Therefore, a **Pizza** is a *subclass* of the things that have *at least one* **PizzaBase**.

### 2.5.8 Reasoners and RacerPro

One of the key features of ontologies described using OWL-DL is that they can be processed by a *reasoner*. One of the main services offered by a reasoner is *subsumption testing* which determines whether or not one class is a *subclass* of another class. By performing these tests on the ontology's classes, it is possible for a reasoner to compute the *inferred* ontology class hierarchy. Further, a reasoner can perform *consistency checking*. A reasoner can check on whether or not it is possible for a class to have any *instances* based on the *descriptions* or *conditions* of that class. A class is deemed *inconsistent* if it cannot possibly have any *instances*. To *reason* over ontologies constructed under

Protégé-OWL, a *Description Logic Implementers Group (DIG)* compliant reasoner may be used. This thesis uses the *RacerPro* reasoner allowing the *manually constructed ontology* in Protégé-OWL, called the *asserted class hierarchy*, to be sent to RacerPro reasoner to compute the *classification hierarchy* and to check the *logical consistency* of the ontology. The reasoner's *automatically computed ontology* is called the *inferred class hierarchy*. The task of computing the *inferred* hierarchy is called *classifying the ontology*.

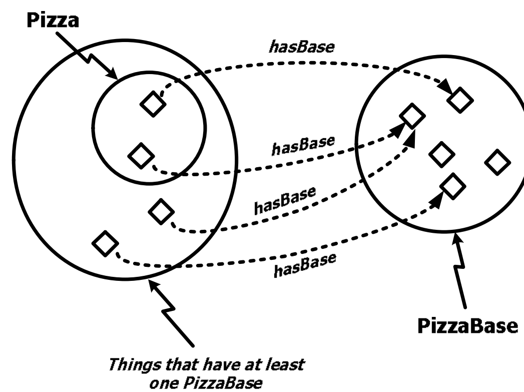



**Figure 2-17: Schematic description of a Pizza [Hor04].**

## 2.5.9 Necessary and Sufficient Conditions

The OWL classes discussed so far have been described using only *necessary conditions*. A class specified with only a *necessary condition* is known as a *primitive class*. A *necessary condition* reads as:

*"If something is a member of this class, then it is necessary to fulfill these conditions."*

However, with necessary conditions alone, it is not possible to infer whether something that fulfills these conditions is actually a member of the class. For example, consider the subclass of **Pizza** called **CheesyPizza** which is a **Pizza** that has at least one kind of **CheesyTopping**. Consider now the *primitive class* description of CheesyPizza shown in Figure 2-18(i), which states that, if something is a member of the class **CheesyPizza**, it is *necessary* for it to be a member of the class **Pizza** and it is *necessary* for it to have *at least one* topping that is a member of the class **CheesyTopping**. Suppose that it is known that a particular individual is a member of the class **Pizza** and that this same

individual has *at least one* kind of **CheesyTopping**. Given the current *primitive class* description of **CheesyPizza**, the knowledge about this particular individual is *not sufficient* to determine whether or not the individual is a member of the class **CheesyTopping**.

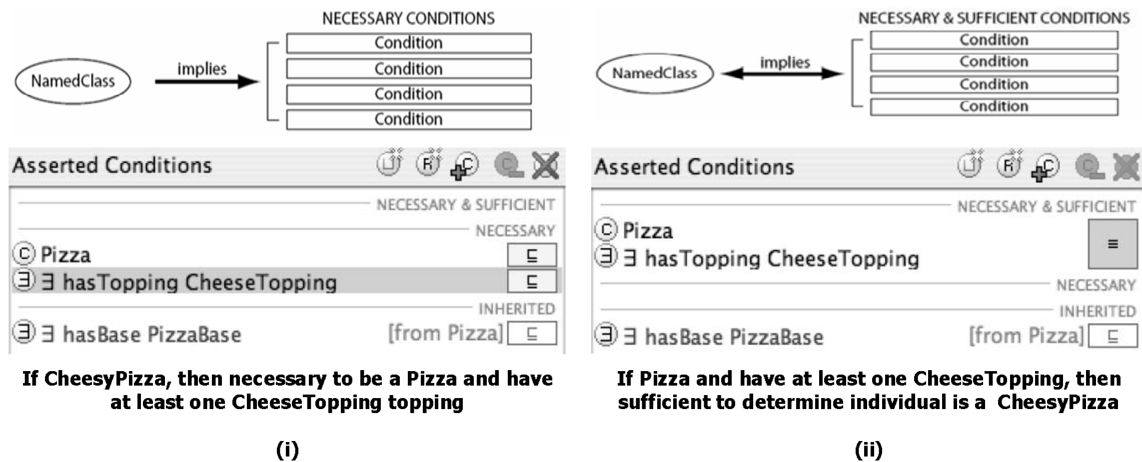



**Figure 2-18: Description of CheesyPizza.**

To be able to *classify* the individual, the conditions for the **CheesyPizza** need to be changed from *necessary conditions* to *necessary and sufficient conditions*, as shown in Figure 2-18(ii). This means that not only are the conditions *necessary* for membership in the class **CheesyPizza**, they are also *sufficient* to determine whether any (random) individual that satisfies these conditions must be a member of the class **CheesyPizza**. In OWL, *primitive class* description have only necessary conditions, whereas *define class* definition have at least one *necessary and sufficient condition*. Any individual that satisfies the *definition* will belong to the *defined class*.

## 2.5.10 Automatic Classification

Being able to use a reasoner to automatically compute the inferred class hierarchy is one of the major benefits of building an ontology with OWL-DL. When constructing very large ontologies, which may contain several thousand classes, the use of a reasoner to compute subclass-superclass relationships becomes vital to maintaining the ontology and keeping it logically correct. Further, in multiple inheritance cases where ontologies have classes that have one or more superclasses, it is often better to construct the asserted class hierarchy as a simple tree. Then the reasoner is used for computing and maintaining multiple inheritance hierarchy coordination. Doing this helps keep the ontology

maintainable, extensible, reusable and modular, and helps to minimize human error.

### 2.5.11 Universal Restrictions

*Universal restrictions* (∀) mandate that the only relationships that exist for a given property must be to individuals that are members of the specified *filler* class; that is, *universal restrictions* constrain the relationship along the specified property to individuals of a specified class. For example, ∀ ***hasTopping*** **MozzarellaTopping** describes all the individuals all of whose ***hasTopping*** relationship are to member of **MozzarellaTopping**. In other words, individuals do not have a ***hasTopping*** relationships to individuals that are not members of **MozzarellaTopping**. Note that the *universal restriction* ∀ ***hasTopping*** **MozzarellaTopping** also describes the individuals that *do not participate* in any ***hasTopping*** relationships. An individual that does not participate in any ***hasTopping*** relationships by definition does not have any ***hasTopping*** relationships to individuals that are not members of **MozzarellaTopping**, and the restriction is therefore satisfied. Therefore, for a given property, *universal restrictions* do not specify the *existence* of a relationship, but merely state that if a relationship exists for the property then it must be to individuals that are members of a specified class.

### 2.5.12 Open World Reasoning

Reasoning in OWL-DL is based on the *Open World Assumption (OWA)*, and is often referred to as *Open World Reasoning (OWR)*. OWA means that it cannot be assumed that something does not exist until it has been explicitly stated that it does not exist. In other words, just because something has not been stated as *true*, it cannot be assumed to be *false*. Instead, it must be assumed that the knowledge has just not yet been added to the knowledge base.

### 2.5.13 Closure Axioms

A *closure axiom* on a property consists of a *universal restriction* that acts along the property to specify that it can only be *filled* by the list of specified *fillers*. The restriction has a filler that is the *union* of the fillers that occur in the *existential restrictions* for the property. For example, Figure 2-19 shows the *closure axiom* on the ***hasTopping*** property for **MargheritaPizza** is a *universal restriction* that acts along the ***hasTopping*** property, with a filler that is the *union* of **MozzarellaTopping** and **TomatoTopping**; that is, ∀ ***hasTopping*** (**MozzarellaTopping** ⊔ **TomatoTopping**). This restriction

says that if an individual is a member of **MargheritaPizza**, then the individual must be a member of **NamedPizza**, and it must have at least one topping of the kind **MozzarellaTopping**, and it must have at least one topping that is a member of **TomatoTopping**, and the toppings must only be of the kinds **MozzarellaTopping** or **TomatoTopping.**

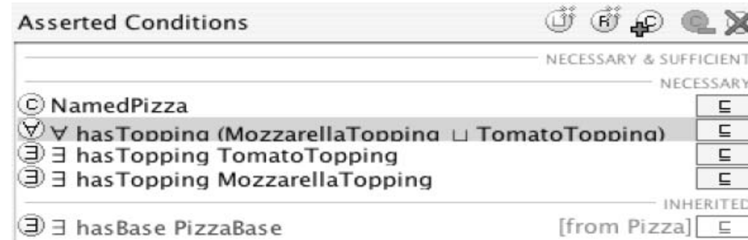



**Figure 2-19: Example of closure axiom.**

## 2.5.14 Extending OWL-DL with Value Partitions

A *Value Partition (VP)* [Hor04, W3c05] is not part of OWL, nor any other ontology language. VPs are used to refine the descriptions of OWL classes. A VP is essentially an ontological design pattern, analogous to a design pattern in meta-modeling, whose value classes, as shown in the example in Figure 2-20(i), are *disjoint* and their union makes up a set that is *covered*, as shown in the example in Figure 2-20(ii). VPs are solutions to modeling problems that occur repeatedly across a number of different ontologies or domains and model descriptive features such as qualities or attributes as OWL properties whose range specifies the constraints on the values the property can assume. Consider the example VP called **SpicinessValuePartition**, as shown in Figure 2-20, that describes the *spiciness* of **PizzaTopping**s. The VP *restricts* the range of possible values to an *exhaustive list*. For example, **SpicinessValuePartition**, modeled as the *functional object property* ***hasSpiciness***, restricts the range of ***hasSpiciness*** to the *disjoint subclasses* **Mild**, **Medium** and **Hot**, as shown in Figure 2-20(i), represent degrees of spiciness. In Figure 2-20(ii), **SpicinessValuePartition** is defined with a *covering axiom* to make the list of spiciness exhaustive.

## 2.5.15 Covering Axioms

A VP partition uses a *covering axiom*. A *covering axiom* consists of two parts:

- The class *being* covered.

- The classes that *form* the covering.

For example, consider three classes: class **A**, class **B** and class **C**, where **B** and **C** are *subclasses* of **A**, as shown in Figure 2-21(i). Suppose that a *covering axiom* specifies that **A** is *covered* by **B** and **C**. This means that a member of A *must* be a member of **B** and/or **C**. If **B** and **C** are specified to be *disjoint*, then a member of **A** *must* be a member of *either* **B** *or* **C**. Recall that normally, if **B** and **C** are *subclasses* of **A**, then an individual may be a member of **A** without being a member of either **B** or **C**. However, a *covering axiom* manifests itself as a class that is the *union* of the classes being covered and conceptually forms a superclass of the classes that is being covered. In this case, **A** would have a superclass of **B** ⊔ **C**, as shown in Figure 2-21(ii).

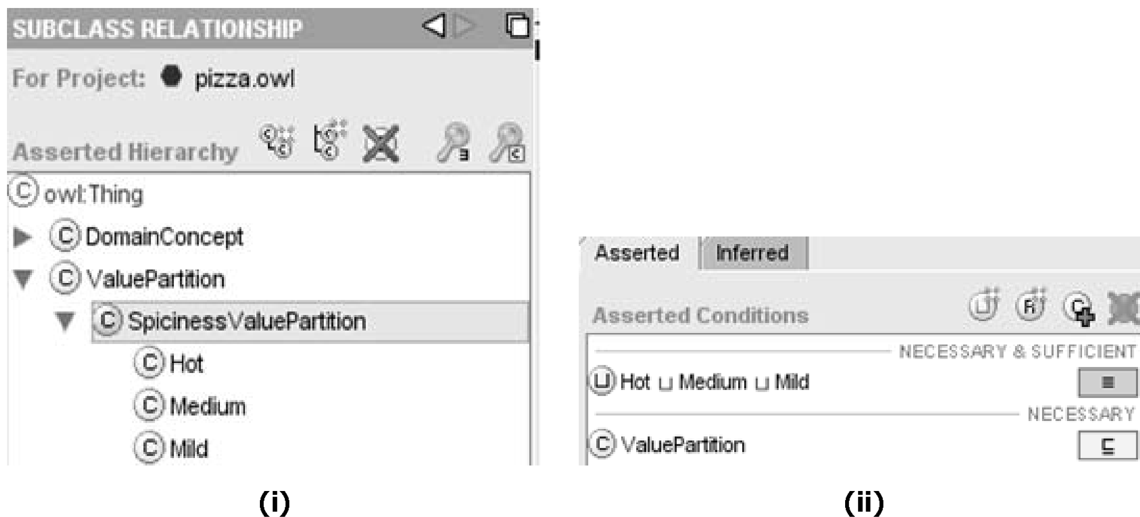



**Figure 2-20: Example value partition.**

## 2.5.16 Cardinality Restrictions

In OWL, a class of individuals can be described by *cardinality restrictions* to have *at least*, *at most* or *exactly* some specified number of relationships with other individuals or datatype values. For a given property **P**:

- A *minimum cardinality restriction* specifies the *minimum* number of **P** relationships that an individual *must* participate in.

- A *maximum cardinality restriction* specifies the *maximum* number of **P** relationships that an individual *can* participate in.
- A *cardinality restriction* specifies the *exact* number of **P** relationships that an individual *must* participate in.

Relationships between pairs of individuals are only counted as separate if it can be determined that the individuals are in fact *different* from each other.

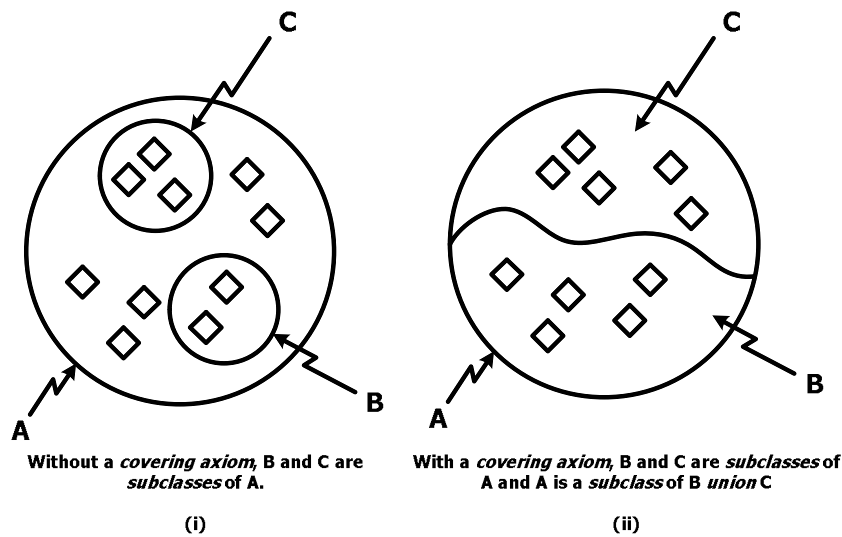



**Figure 2-21: Effects of a covering axiom: (i) uncovered; (ii) covered [Hor04].**

## 2.5.17 hasValue Restrictions

A *hasValue restriction* (϶) describes the set of individuals that have *at least one* relationship along a specified property to a specific individual. For example, the *hasValue* restriction ***hasCountryOfOrigin*** ϶ **Italy**, where **Italy** is an individual, describes the set of individuals of the anonymous class that have at least one relationship along the ***hasCountryOfOrigin*** property to the specific individual **Italy**. This thesis does not use *hasValue restrictions*.

# 2.6 Meta-modeling with UML

A *meta-model* makes statements about what can be expressed in the valid models of a certain

modeling language [Sei03]. A Meta-model is a *model* of a *modeling language* that defines the correspondence between a model and a system. A meta-model makes statements about what can be expressed in the valid models of a certain modeling language. For example, the common understanding of the 4-layer *Model Driven Architecture (MDA)* is given in the general modeling architecture shown in Figure 2-22. The MDA architecture provides a standard meta-modeling framework for model and meta-data driven systems as follows:

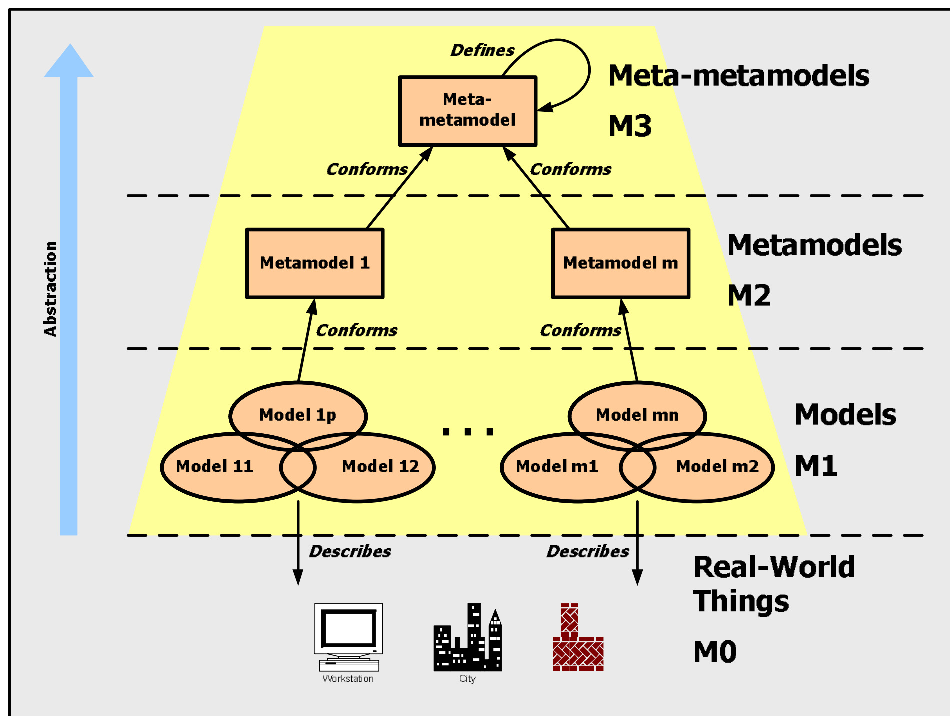



**Figure 2-22: A general, 4-layer modeling architecture inspired by MDA.**

- The ***M3*** *Meta-Object Facility (MOF) Layer* is the self-defining, MDA meta-meta-model layer that provides a standard framework for model-driven and meta-data-driven systems. MOF is the basic mechanism for defining modeling languages. MOF is essentially a subset of *Unified Modeling Language (UML)* class diagrams plus *Object Constraint Language (OCL)*.

- The ***M2*** *Meta-Model Layer* is the layer where modeling languages are defined. Examples of applicable modeling language include: the *Unified Modeling Language (UML)*, the *Common Warehouse Metamodel (CWM)* and the *Ontology Definition Metamodel (ODM)*. MOF uses a UML graphical notation.

- The ***M1*** *Model Layer* is where real-world models (i.e., domain models) are developed using concepts such as classes, relations and states. UML constructs such as classes, relations and states are instance of meta-model concepts.
- The ***M0*** *Instance Layer* contains concrete instances such as object, data and executions that model real world things.

## 2.6.1 UML

The *Unified Modeling Language (UML)* is an object-oriented modeling paradigm and graphical notation that has become a *de facto* academic and industrial standard for modeling software systems [Lar98, Omg05a, Omg05b]. UML has recently been extended by the semi-formal *Object Constraint Language (OCL)* which allows UML models to express constraints on object semantics [War98]. However, there are still many aspects of UML that rely heavily on natural language descriptions. OWL-DL-based ontologies rely on description logic [Baa03] to provide a well understood, semantic basis. Object models for software engineering are a hierarchy of classes and their attributes and methods with *overridable inheritance*. This thesis assumes UML is generally better-known and well understood by most software engineering practitioners compared to OWL-DL. As a result, this thesis uses only a subset UML, as shown in Figure 2-23 and used predominantly in Chapter 5. The reader is directed to [Fow04] for a condensed reference guide of UML 2.0. Table 2-1 presents a comparison of UML and OWL-DL based on the tables presented in [Usc06].

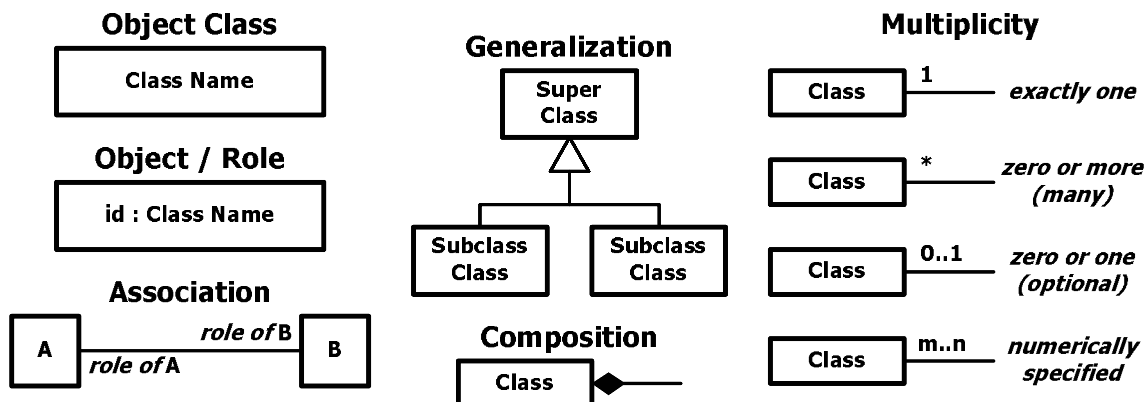



**Figure 2-23: Support UML constructs [Bel91].**

| UML | OWL-DL |
|---|---|
| Classes are templates for creating objects | Classes are sets of individuals with common characteristics |
| Every instance has a unique class (not necessarily) | An individual can be in multiple classes |
| Instances always in the same class | Individuals can change class at runtime |
| Set of classes fixed compile time | Can add new classes and edit classes at runtime |
| Classes are primitive | Class expressions are possible (not in Protégé-OWL) |
| Attributes are defined locally to a class | Properties are defined independently from any particular class and associated to multiple classes via domain and range constraints and inheritance |
| Can have different attributes with same name in different classes | Property names are unique |
| Associations may be n-ary where binary association ends are analogous to OWL domain and range | Binary properties only |
| No property hierarchies (but association refinement) | Property hierarchies are supported |
| Classes are primitive, not defined using properties | Properties are used to define classes |
| Instances can have values only for attached attributes | Individuals can have arbitrary values for any property |
| Inheritance is like an overridable template for creating instances for programming convenience not inference | Inheritance is strictly logic-based, no overridable defaults – inference is key |
| Computation-based | Inference-based |
| Closed-Word where unknown = FALSE | Open World where unknown = UNKNOWN |
| Compiler problems indicate errors at build time only | Reasoners check consistency at build time and at run time |
| Mature with range of commercial tools and products | Still maturing with commercial tools and products still emerging |

**Table 2-1: Tabular comparison of UML and OWL-DL based on [Usc06].**

# 2.7 SDL

The *International Telecommunications Union (ITU)* developed the *Specification and Description Language (SDL)* to give telecom administrations and manufacturers a common language precisely and unambiguous specifying the behavior of telecom systems [Bel91]. SDL has become a general language for specifying the discrete stimuli-response behavior of most interactive, real-time system [Mit99]. The theoretical foundation of SDL is that of *communicating extended finite state machines (CEFSM)* [Hie01]. Examples of other CEFSM-based languages include Estelle and Lotos [Tur93]. The structure of the language is *hierarchal* and the use of *abstract data types* is supported. SDL has a graphical syntax called *SDL/GR* and a linear (textual) form called *SDL/PR* that share a common abstract syntax. Communication between SDL processes is performed asynchronously via *channels*

and infinite, but not unbounded *first-in-first-out (FIFO) queues*. A detailed description of the formal static and dynamic semantics of SDL is given in [Bel91, Itu91]. In this thesis, only the *SDL/GR* notation is used. The subset of supported SDL constructs for his thesis is given in Figure 2-24. The reader is directed to [Bræ93] for a tutorial paper on the basic concepts, ideas and features of SDL.

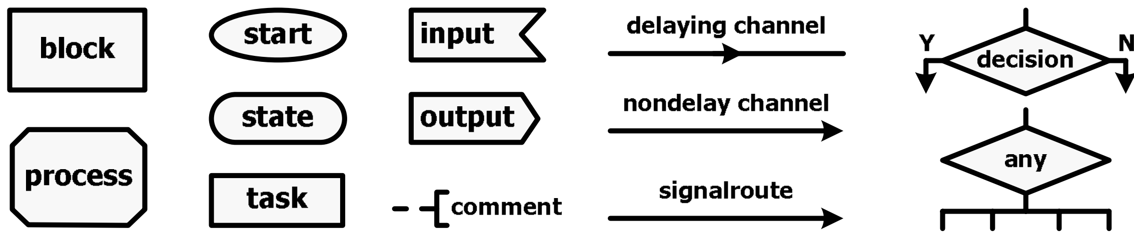



**Figure 2-24: Subset of supported SDL constructs.**

# 2.8 Runtime Monitoring

This section introduces the concepts behind *runtime monitoring (RTM)* and includes several *model-based runtime monitoring* approaches from the literature related to the work presented in this thesis. The reader is directed to [Sch95] for a tutorial on *runtime monitoring* and to [Gat04] for a comprehensive *taxonomy* and *catalog* of *runtime* software-*fault monitoring tools.*

## 2.8.1 What is Runtime Monitoring?

In software engineering, the concept of *runtime monitoring (RTM)* is normally associated with software dependability[1] [Pla84]. RTM is applicable to software testing during development and maintenance, and to ensure that software systems execute as intended in their production environments. RTM allows the observation and tracking of genuine runtime data in the laboratory or the production environment. The monitored data collected may be used for statistical purposes or to detect when a program has entered an illegal or unexpected state. Many of the RTM tools described in the literature require a software engineer to have intimate knowledge of the target. In other words,

most RTM tools take a whitebox monitoring approach.

Traditionally, RTM has been used to observe the behavior of operational, real-time applications for statistical data collection or to detect behavioral failures. It is paramount that RTM should not appreciably alter the internal timing constraints of the target system and therefore, monitored data is usually logged and analyzed off-line. This is still true for most resource-limited, embedded system environments. However, with the recent significant advances in raw processing power present, especially for heavyweight servers in online application environments, it is not unreasonable to assign some percentage of CPU processing time (e.g., $< 10\%$), specifically for RTM purposes.

Figure 2-25 provides an overview of the traditional approaches to RTM. Early approaches used stand-alone hardware, software embedded into the target or a combination of both external hardware and internal software. More recently, monitoring software has migrated out of the target software system and runs on a separate, stand-alone host. With the advent of aspect-oriented approaches, there is new interest in using automated aspect-weaving technology to facilitate software instrumentation. These aspect-based software sensors range in capability from those that report raw to those that provide local processing and report value-added monitoring data [Det01, Tha01a, Tha01b].

### 2.8.2 RTM Types

In general, these are the categories of RTM systems.

- A *hardware monitor* is a class of monitoring tool that can be attached to a target via test probes, keeping the hardware monitor completely separate from the target. Attaching test probes requires detailed knowledge of the target's hardware architecture and implementation as well as a clear specification of which signals are to be monitored. The degree of sophistication of these tools is based on the level of self-control they possess. For example, simple, unintelligent hardware monitors are manually controlled and often require human intervention. On the other hand, intelligent hardware monitors use programmable logic running under microcomputer control. Examples include sophisticated test equipment such *in-circuit emulators*, *logic analyzers* that attach directly to a system's internal data bus and *communications analyzers* that can monitor

---

[1] *Software dependability* is the trustworthiness of a software system such that *reliance* can be justifiably placed on the service the software delivers [Lyu96].

data and control traffic flowing across complex networks. One major drawback to hardware monitors is that it is very difficult to identify interesting events in a problem-oriented manner or deal with certain dynamic aspects, such as process creation and destruction. In general, hardware monitors are excellent for use in debugging system software and firmware, but fall short in monitoring and measuring application programs.

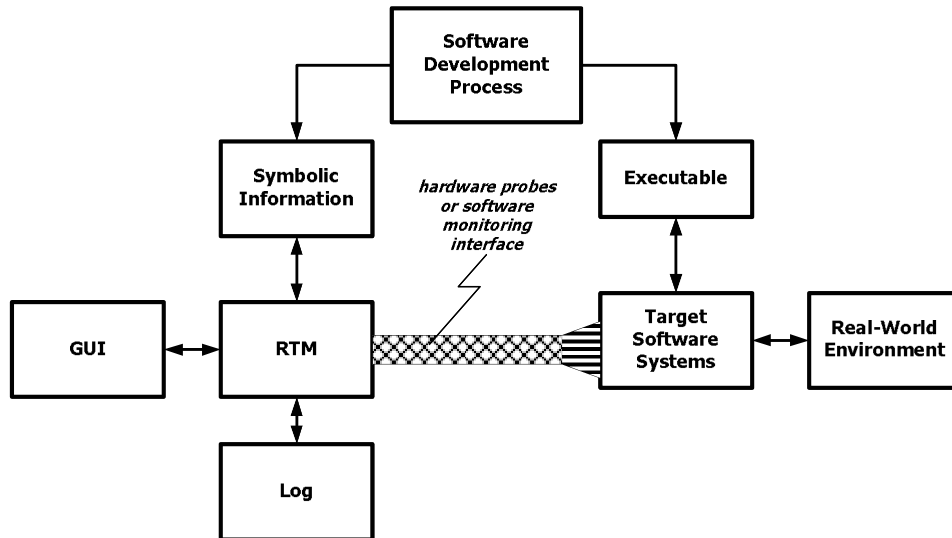



**Figure 2-25: Example configuration for runtime monitoring for real-time software system.**

- A *software monitor* is a program, or program fragments, that executes on the monitored target, presenting the opportunity to employ an application-oriented approach to monitoring. However, the execution of additional instructions (i.e., software sensors) embedded in a target can cause serious time and space interference in time-critical, real-time or embedded software systems. Unlike hardware monitors that tend to be sample-driven, software monitors are typically event-driven. There are three general classes of software monitors: (1) those that measure and gather accounting statistics, (2) those that detect and report internal error and failures, and (3) those that measure and/or command performance. *Statistical monitors* are commonly used to measure and command values from a predefined data set, such as the number of times a particular resource is accessed, or the time of day a resource was used. *Error* and *failure detection* monitors typically do not perform any measurements, but instead carry out an *internal audit function* such as counting the number of times a particular error, once detected, occurs. *Response performance monitors* are designed to collect performance data for statistical purposes. Software monitors are

often not suitable for continuous monitoring and measurement activities in a full production environment because of the excessive amount of data reduction that may be necessary before collected event traces can be made meaningful. Software monitors are often suitable only for a particular *application domain* or *implementation.*

- A *hybrid monitor* uses a combination of specialized hardware devices and interfaces in conjunction with a software monitor. The software monitor may reside within the target system or may actually run on a physically separate host processor. The hybrid monitor approach offers a reasonably good trade-off between the hardware and software approaches. A hybrid monitor can observe and track the behavior of target as well as collect trace data and detect illegal or unexpected process states.

### 2.8.3 Date Structure Monitoring

*Software audit programs* monitor the evolving data structures of real-time software systems to detect, and possibly correct, software *data errors* before they manifest as failures [Con72, Pen80]. A software audit program typically consists of additional software that periodically accesses and checks some or all of a target software system's data structures for errors. Audits are designed to use three main error checks: (1) *direct comparison error* checks using *duplicate data structures* for comparison purposes to check *data integrity*, (2) *association comparison error* checks using *data structure redundancy* such as doubly-linked lists to check the *structural integrity* of the *data structures*, and (3) *format comparison error* checks to perform commonsense checks to identify *out of range* or *bound* errors. While audits are able to detect software errors before they manifest as failures, they are limited in scope and may themselves contain faults which could potentially reduce the reliability of the target software system. The monitoring approach proposed in this thesis uses additional software, called software sensors, and differs in that sensors produce commands that are sent to an external monitor and used to construct a representation of the instrumented target's evolving software structure, rather than for comparison purposes with internal data structures.

In [Rod97], the *Nanites* approach is described that simplifies the task of monitoring complex data structures to address the difficult problem of monitoring system component interactions in data-centric systems. The Nanites is an object-oriented approach that incorporates a Monitor object attachment; that is, a watcher module attaches a Monitor object to each data object in its scope-of-interest. When a method is invoked on the data object, the same method is invoked on each Monitor

object attached to it, thus giving each attached Monitor object an opportunity to respond to the method invocation. Monitor objects can be given pre- and post-opportunities to respond by calling the Monitor's method both before and after the change is made. This approach requires each data object to store a list of references to Monitor objects that are attached to it. This approach is similar to the monitoring approach in [Gha03] in which a local monitor attached to a process reports state changes to a central monitoring point that performs the required state consistency checks.

### 2.8.4 Model-Based Monitoring

*Real-time supervision (RTS)* is a blackbox behavioral monitoring approach that passively observes the external inputs and outputs of an operational software system while interpreting a supervisor-model derived from the target's software requirements specification [Hay91, Hla95, Ior94, Sav97]. The organization block diagram for RTS is shown in Figure 2-26. The RTS reports a discrepancy between the observed and expected behavior of the operational software systems as a failure. One of the chief disadvantages of RTS is that detection is complicated by the state explosion problem [Hie01] due to the occurrence of specification nondeterminism in the SDL-based supervisor-model as the RTS attempts to account for all possible observed behaviors.

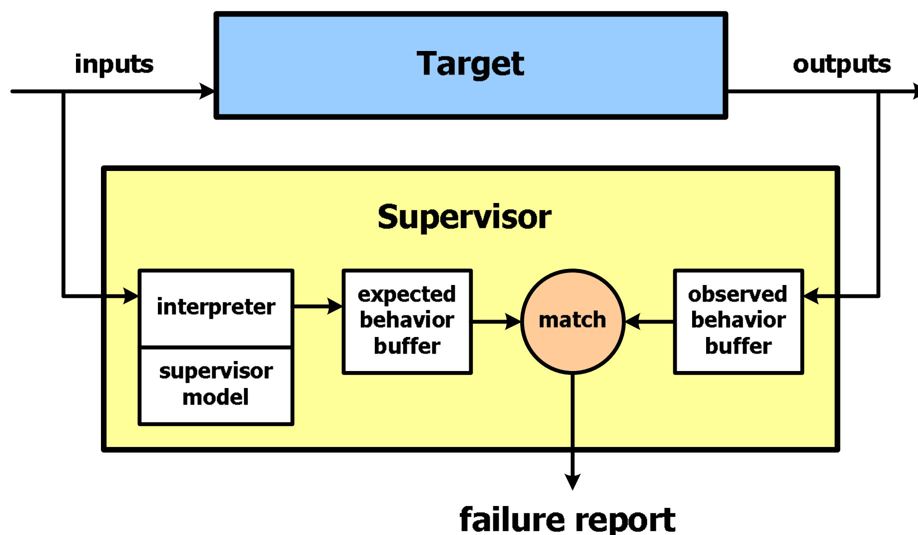



**Figure 2-26: Organizational block diagram of real-time supervision.**

The monitoring approach in this thesis also uses an interpretable specification-based model, but

differs from RTS in a number of ways:

- The model is concern-specific; that is, derived using both the target's SDL-based software requirements and software design specification for a selected structural software concern.
- The interpreter is greybox; that is both external and internal instrumented knowledge is used to interpret the concern-specific.
- The monitor maintains a representation of the evolving software structure of the operational target for selected software concern.

*Response performance monitoring* is a blackbox response time monitoring approach that observes a target's external inputs and outputs, interprets a timepost-model derived from a combination of the target's SDL-based software requirements specification and tabular response objectives specifications [Pek97, Pek03]. The timepost-model is extended using a new SDL construct called an Interval Timing Directive. The interpreter tracks the specification state of the target, interprets the timepost-model, and directs response time interval measurements using the interval timing directives for guidance. The monitor can detect response time failures during the current observation interval and response performance failures that have occurred over a number of previous observation periods. The monitoring approach in this thesis also extends the concern-specific model with several new model constructs. These new constructs are used to direct the interpreter on how to match and process incoming monitoring commands that have been produced by the instrumented operational target software system and delivered to the monitor's greybox interpreter via the monitoring interface.

In [Zul04], a *pseudo-greybox monitoring* approach is presented where information about the certain internal behavioral conditions is sent from the target to the monitor whenever the target enters a *stable state*. The approach is pseudo-greybox because only *limited behavioral reporting* is possible at *high loads* because the target may not enter a *stable state* for long periods of time. The monitoring model consists of a set of concurrently executing nodes, where the state of each node is modeled such that a state transition is only fired upon receipt of a message from another node or from the environment. Rather than sessions, the system is assumed composed of a number of communicating processes where the global state consists of the current state of all the processes in conjunction with the state of the communication channels connecting each node. The monitoring approach in this thesis is similar because it considers *quiescent states*, which are similar to the idea of *stable states*. However, while a stable state is a *global property* of the target, a quiescent state is a *session-oriented*

*property*. The notion of *session-orientation* and quiescent states are explained in Chapter 6. For now it is sufficient to know that every *macro-step* begins and ends in a quiescent state.

### 2.8.5 Other Related Monitoring Work

In [San93], a methodology for continuously monitoring a program for *specification consistency* at runtime is described. The target program is annotated with formal specification constructs from a formal specification language. The constructs are transformed into checking code, which is inserted into the target program. Further, calls to this checking code are inserted into the target program wherever at locations where the program may potentially become inconsistent with its specification. If an inconsistency is detected, diagnostic information is provided. According to the authors, significant effort is required each time the monitoring approach is applied to a different product line family. The related monitoring approach may not detect all failure types. For example, lost or superfluous signals would not activate checking code. In other cases, checking code may be inadvertently activated while the program is in the wrong state. The monitoring approach in this thesis uses model constructs and embedded software sensors that produce monitoring commands rather than formal specification constructs and checking code.

In [Det01], the advantages and costs of using *aspects* to instrument a target to *harvest* runtime information for offline (dynamic) analysis is examined. Information is harvested concerning program state, memory state, and real-time requirements. *Analysis aspects* are inserted where execution-flow join points are *advised* to collect on analysis data. *Reference-probing aspects* keep track of checking legal, Java scoping hierarchies. *Death-probing aspects* determine when objects become unreachable. Other analysis aspects include instantiation and collection behavior of live objects and threads, inter-class behavior including references and call behavior, dead code removal for code that is never executed and class preloading. This related approach also uses software sensors to collect runtime information. However, location of software sensors is determined when the formal concern-specific monitoring model is derived; that is, to facilitate application resource ownership monitoring.

- *State evolution sensors* are inserted into the program to command when the target executes a *macro-step* and enters (or exits) a specification state according to the software requirements. Recall from Chapter 1 that a macro-step indicates that a new state-dependent snapshot of the evolving concern-specific structure has come into effect.
- *Structural transaction sensors* are inserted into the program to command when the target executes

a *micro-step* and completes a structural transaction according to the software design. Recall from Chapter 1 that a micro-step only partially transforms the existing concern-specific structure of the target to a transient state between macro-steps (i.e., between snapshots).

*Software health monitoring* is another approach to error detection that promotes the continuous monitoring of an operational target to provide an indication of well-being or health [Gha02, Lau05, Tha01a, Tha01b]. Software health monitoring strives for *early detection* of *state* and/or *data inconsistencies* through the use of intelligent software sensors called *health indicators*. Software health, as a *statistical measure*, is intended to be a comprehensive and diverse notion, rather than a simple error or a failure indicator. Each software indicator monitors one or more specific *facets* of a software system's *execution*. From the perspective of this thesis, a *facet* may be viewed as some concern-specific piece of runtime knowledge for an individual software concern, whose runtime health is to being monitored. In taking the analogy further, the statistical measure produced by health monitoring could be used to provide the well-being (or health) of either an individual software concern or a collective set or basket of software concerns. Instrumentation for software health monitoring tends to be more ad hoc and implementation-driven, whereas in the approach presented in this thesis, instrumentation is driven by the need to provide runtime knowledge according to an interpretable concern-specific monitoring model.

# 2.9 Interactive Session Oriented Service Domain

The application domain considered in this thesis is the interactive session-oriented service domain. Interactive session-oriented services are typically delivered by software systems that are real-time with soft deadlines, interactive, session-oriented, discrete event-driven, semi-stationary and non-critical. It is assumed that the software system's behavior requirements and design are specified using SDL, a formalism based on communicating extended finite state machines.

## 2.9.1 Real-Time Software Systems

*Real-time software systems (RTSS)* are tightly coupled to the external world and must respond to the real-world in a suitable timeframe or by a specified deadline. In general, an RTSS is comprised of two key component sub-systems: a controlling sub-system and a controlled sub-system. For the PBX, the controlling sub-system is the call processor and control software and the controlled sub-system is the

switching network and subscriber circuits. It is the controlling sub-system that interacts with the service environment.

### 2.9.2 RTSS with Soft Deadlines

A soft RTSS depends on time constraints or deadlines where the inability of the RTSS to meet a prescribed deadline does not necessary mean that a failure has occurred. In a soft RTSS, there is a trade-off between minimizing the number of late transactions and minimizing the mean lateness. The trade-off occurs because any attempt to minimize lateness usually results in even more deadlines being missed. Instead, resources remain unavailable for new tasks because the resources are being utilized by tasks whose deadlines have still not yet been met. Therefore, a soft RTSS is usually only concerned with achieving good average performance. The control program in the PBX used in this thesis is a soft, real-time software system.

### 2.9.3 Interactivity

*Interactivity* is a mode of *service operation* with an *"input-compute-output"* processing structure in which end user commands (i.e., *inputs*) cause service responses (i.e., *outputs*) [Bac99, Bro01]. This interactivity could be with humans, external hardware or other external software system. The control program of the PBX is an interactive software system.

### 2.9.4 Session-Orientation

Session-orientation is an application-level attribute reflected in the behavior of the overall software system. An application may consist of one or more non-terminating processes that operate through repeated activations, possibly for an infinite number of times, resulting in the concept of cyclic application. In general, these types of applications are referred to as session-oriented because they always return to a predetermined idle state upon completion of a current session before the commencement of a new session. The control program of the PBX is session-oriented.

### 2.9.5 Discrete Event-Driven Software

Discrete *event-driven software (EDS)* is a particular class of software that is fast becoming ubiquitous [Mem06]. All EDS systems share a common event-driven model in that they take sequences of events

(e.g., messages, signals, mouse-clicks) as inputs, change their state, and (sometimes) produce an event sequence as an output. A typical EDS code distribution is given in Figure 2-27. The control program of the PBX in this thesis is a discrete EDS system.

### 2.9.6 Semi-Stationary Systems

A *semi-stationary* software system has specification states that may be categorized according to each state's average holding times as either stable or transient [Gha03]. Semi-stationariness is a CFSM-based process level property where:

- A *stable state* has a relatively long holding time compared to the sum of the maximum communication channel delay and the local clock drift, such that a CFSM-based process spends most of its time in stable states.
- A *transient state* has a comparatively short average holding time. Periodically, an external input is received by a process that then leaves a stable state, passes through a sequence of transient states, and eventually settles again in a stable state.

The probability of observing a particular process in a stable state is much higher than the probability of observing the same process in a transient state. Interactive, session-oriented services exhibit a similar process or system-wide semi-stationary property, but on a session-by-session basis.

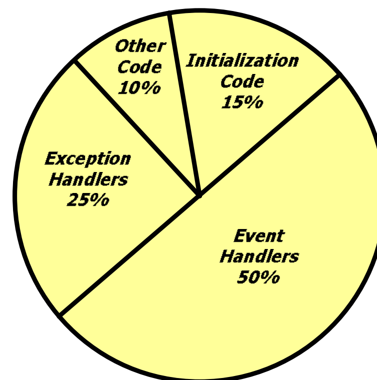



**Figure 2-27: A typical event-driven software code distribution [Mem06].**

### 2.9.7 Non-Criticality

The monitoring approach presented in this thesis may not be suitable for software systems that are classified as mission- or safety-critical. Critical software systems are typically RTSS with hard deadlines whose catastrophic failure could have an impact on human safety, loss of life or major financial or social losses. The PBX is a soft RTSS and, therefore, non-critical.

## 2.10 The PBX

For concreteness, the control program of a small *private branch exchange (PBX)* was selected for a running example of an interactive session-oriented service and provides the reader with concrete illustrations of the modeling concepts in this thesis. The PBX makes an ideal example target software system service because it exhibits most of the characteristic and properties of the application domain. Further, the control program's SDL requirements specification, SDL design specification and Java source code were available. An operational Java implementation from a number of previous research efforts was also available that could have been instrumented and used to collect operational state evolution and ownership transaction traces. The layered architecture for the PBX in [Gha02, Lau05, Tha01a] is adapted for example PBX used in this thesis and is shown in Figure 2-28. The PBX control program is organized into a number of functional layers, as shown by the labels on the right hand side of Figure 2-28. Each layer also has a corresponding ResOwn classification, as shown on the left hand side of Figure 2-28. These classifications will be explained in the ResOwn description of Chapter 4 and ResOwn specialization example in Chapter 5. The different classes implemented in the PBX control program are described in Table 2-2. A description of the PBX hardware can be found in [Ece00]. To simplify the examples presented throughout this thesis, the functionality of the PBX has been limited to those basic telephony services referred to as *Plain Old Telephone Services (POTS)* and each line card is dedicated to an associated phone handler at PBX initialization.

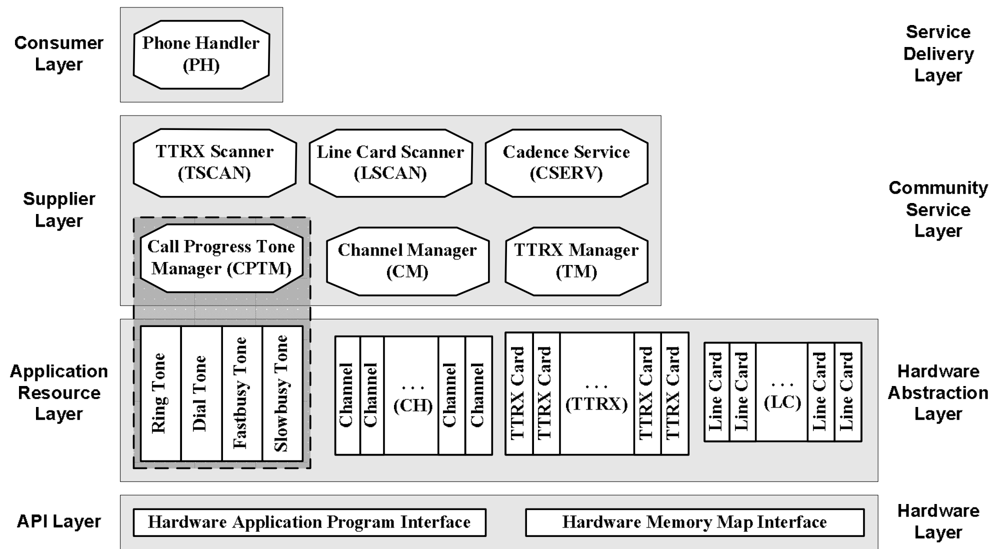



**Figure 2-28: Layered interactive session-oriented service architecture for the PBX.**

| PBX Class / CEFSM | ResOwn Class | Id | Max Num | Thread | Description |
|---|---|---|---|---|---|
| Wait For Call Service | Dispatcher | WFC | 1 | yes | Manages idle line cards and dispatches phone handlers to call request when offhook detected |
| Phone Handler | Consumer | PH | 60 | yes | May be statically or dynamically assigned to a line card to handle call requests for both originating and terminating phones |
| Line Card Scanner | Supplier | LSCAN | 1 | yes | Periodically scans line card registered with LSCAN for switch hook change in switch hook status (i.e., offhook and onhook) |
| TTRX (Digit) Scanner | Supplier | TSCAN | 1 | yes | Periodically scans touch tone receiver card registered with TSCAN for incoming dialed digits |
| Cadence Service | Supplier | CSERV | 2 | yes | (1) Ringer Cadence: periodically cycles ringer relay on a registered line card registered to make phone start and stop ringing<br>(2) Tone Cadence: periodically cycles idle relay on a registered line card so tones will turn on and off |
| Call Progress Tone Manager | Supplier | CPTM | 1 | no | Provides managed access by connected line card and requested dial, ring, idle, slow busy, or fast busy tone generator card to switching network. |
| Switching Network | Supplier | SN | 1 | no | Conceptually connects phones to resources or phones - actually connects line cards to cards via uni-directional channels |
| Phone Handler Manager | Supplier | PM | 1 | no | Manages a pool of dispatchable phone handlers - not required if phone handlers are dedicated to line cards. |
| Line Card Manager | Supplier | LM | 1 | no | Manages a pool of line cards - not required if phone handlers are dedicated to line cards. |
| Channel Manager | Supplier | CM | 1 | no | Manages a pool of uni-directional space channels |
| Touch Tone Receiver Card Manager | Supplier | TM | 1 | no | Manages a pool of touch tone receiver cards |
| Line Card | Resource | LC | 60 | n/a | External gateway (portal) between real-world phone and phone handler instance - one per phone - contains relay devices |
| Idle Relay Device | Resource | IR | 60 | n/a | Produces idle tone that logically disconnects enclosing line card from internal switching network - one per line card |
| Ringer Relay Device | Resource | RR | 60 | n/a | Causes phone to start and stop ringing - one per line card |
| Space Channel | Resource | CH | 30 | n/a | Used to connect two cards through the switching network - one per uni-directional path through the switching network |
| Touch Tone Receiver Card | Resource | TTRX | 7 | n/a | Used to detect status change keys on a phone's keypad, which are converted to a dialed digit |
| Idle Tone Generator Card | Resource | IDLE | 1 | n/a | Connected to a line card by call progress tone manager to produce an idle tone in a phone's handset |
| Dial Tone Generator Card | Resource | DIAL | 1 | n/a | May be connected to a line card by call progress tone manager to produce a dial tone in a phone's handset |
| Ring Tone Generator Card | Resource | RING | 1 | n/a | May be connected to a line card by call progress tone manager to produce a ring tone in a phone's handset |
| Slow Busy Tone Generator Card | Resource | SBUSY | 1 | n/a | May be connected to a line card by call progress tone manager to produce a slow busy tone in a phone's handset |
| Fast Busy Tone Generator Card | Resource | FBUSY | 1 | n/a | May be connected to a line card by call progress tone manager to produce a fast busy tone in a phone's handset |

**Table 2-2: PBX class / CEFSM descriptions.**

# Chapter 3
# Overview

## 3.1 Introduction

This chapter provides an overview of the greybox approach to concern-specific, dynamic software structure monitoring presented in this thesis. The intent of the chapter is to introduce the reader to some of the important concepts used throughout this thesis.

## 3.2 Dynamic Software Structure Monitoring

This thesis describes a novel greybox approach of monitoring the evolving resource ownership structure for interactive session-oriented services. Figure 3-1 shows the concern-specific software structure monitor executing as a separate program. The monitor has limited access to the operational target's internal implementation via a *monitoring interface*. The monitoring interface is comprised of *software sensors* that are embedded into the operational target's source code implementation. In this thesis, the monitored *target* software system is the *call processing program* of a small *private branch exchange (PBX)*. The PBX (Section 2.10) is assumed to be embedded into an *Internet Telephony Gateway (ITG)* [Rose98], as shown in Figure 3-1. The ITG operates as an application-level proxy and provides protocol translation services for the I*nternet Protocol (IP)* network and the PBX. When a call from an IP host arrives, the ITG ensures it reaches the intended destination. Similarly, a subscriber must connect through the ITG to place a call through an IP host. Monitoring is performed at the application-level; that is, that layer responsible for implementing network appliances and applications [Kur01]. The monitor is capable of tracking the evolving resource ownership structure for calls originating on both Internet trunks and subscriber lines.

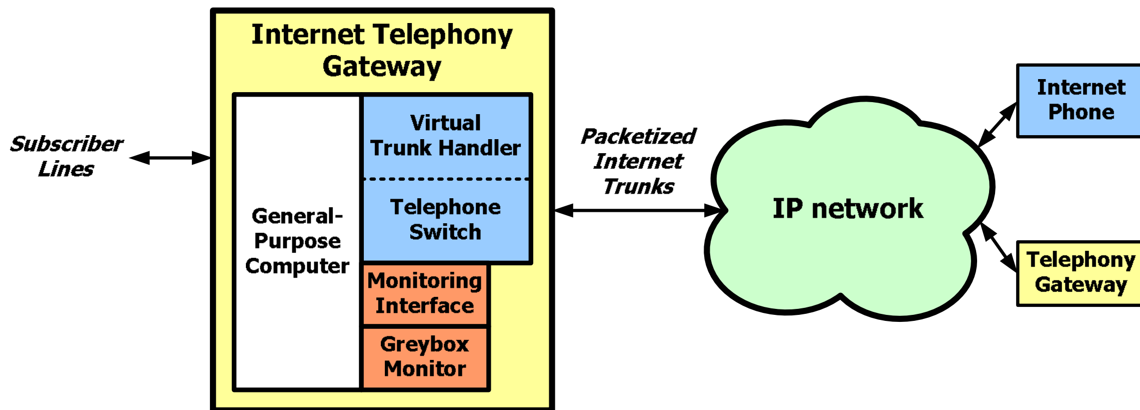



**Figure 3-1: Internet Telephony Gateway with embedded PBX and greybox monitor.**

An architectural overview of the monitor's typical working environment is given in Figure 3-2. The manually derived sensor plan specifies the location of embedded software sensors in the PBX source code implementation. The dynamic knowledge base contains a representation, implemented in tuples, of the evolving resource ownership structure of the operational PBX. The monitor interprets the concern-specific model, which is actually comprised of two derived models:

- The *state evolution model* is automatically derived from the target's SDL-based *software requirement specification* and allows the greybox interpreter to track the *specification state* (i.e., a *macro-step* in the evolving structure) of the operational PBX on a session-by-session basis.

- The *EoB model library* is automatically derived from certain *slices* of the target's SDL-based *software design specification*. Each EoB model contains constructs that direct the interpreter so that it can process the incoming *structural resource ownership transactions* (i.e., *micro-steps* in the evolving structure) as they are reported by the instrumented target.

The monitor's greybox interpreter receives monitoring commands from the operational target via the monitoring interface. Two types of monitoring commands are produced:

- A *state evolution monitoring command* indicates the beginning or end of a *macro-step* or *requirements level state transition* in the instrumented target. At the beginning of a macro-step, the greybox interpreter loads the appropriate *EoB model* from the EoB library and begins interpreting the model. As explained in detail in Chapter 6, a *macro-step* occurs between two *quiescent states* in the *state evolution model.*

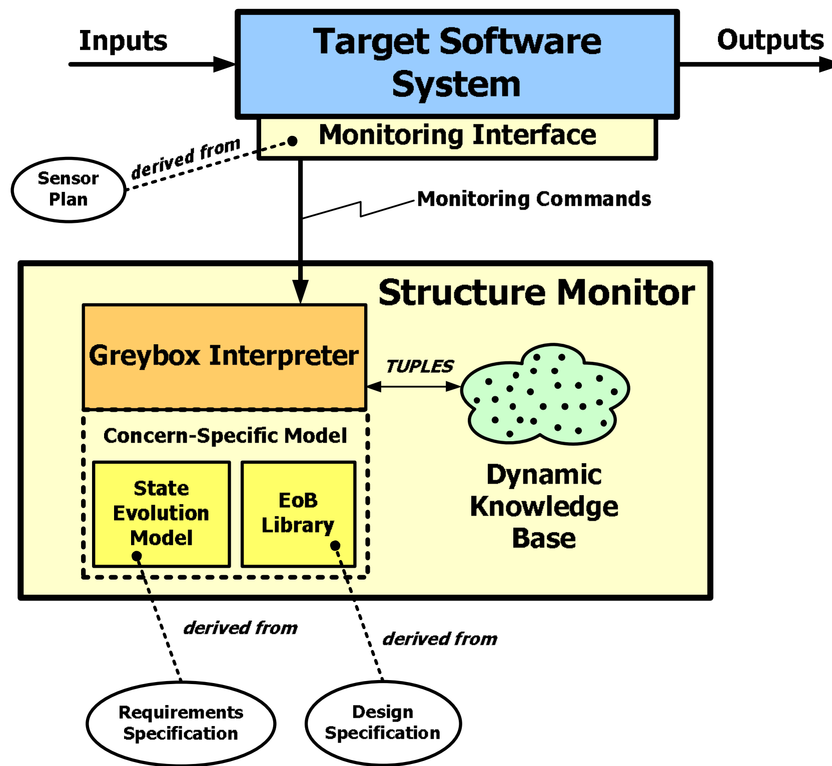



**Figure 3-2: Architectural overview of the software structure monitor.**

- A *structural transaction monitoring command* indicates a *micro-step* or *design-level state transition* has occurred in the instrumented target. Each micro-step corresponds to the successful completion of an individual structural resource ownership transaction and causes the interpreter to add, remove or update an appropriate number of tuples in the *dynamic knowledge base*. Each *structural transaction* is essentially a graph transformation rule that synchronizes the evolving resource ownership structure stored in the *dynamic knowledge base* with the actual evolving resource ownership structure inside the instrumented target.

An overview of ResOwn in the concern-specific software structure monitor approach is given in Figure 3-3. As shown, the baseline ResOwn ontology (Chapter 4) can be specialized with application-specific ontological classes derived from the PBX (Chapter 5). The specialized ResOwn ontology is then automatically classified using a reasoner, resulting in a specialized ResOwn instance (Chapter 5). The resulting specialized ResOwn instance is then used to manually construct tables (Chapter 7) that the interpreter uses to convert structural transaction monitoring commands to tuples for insertion into, or removal from, the dynamic knowledge base during runtime monitoring.

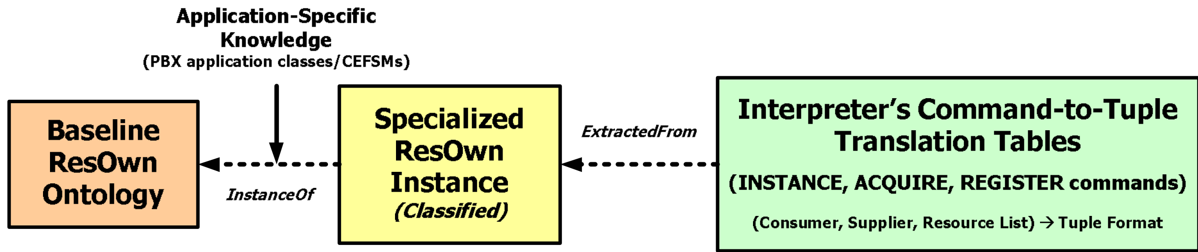



**Figure 3-3: Block diagram overview of approach.**

Although the greybox approach to concern-specific dynamic software structure monitoring presented in this thesis uses a number of projections of the target's behavioral specifications to derive the concern-specific model, the monitor's objective is not to detect behavioral errors or failures. Rather, its prime objective is to track the runtime evolution of the target's resource ownership structure. The objective is very specific and narrow in scope. If the monitor does detect any behavior errors due to one or more incorrectly ordered structural transactions, then this is a side-benefit of the approach, and not the monitor's true intent.

# Chapter 4
# The ResOwn Ontology

*"The first step towards wisdom is calling things by there right names."*
- Chinese Proverb

## 4.1 Introduction

This chapter introduces ***ResOwn***, a novel ontology for *Application Resource Ownership Ontology*. ResOwn provides a vocabulary along with a set of concepts and properties for modeling the application resource ownership structure of operational software systems. ResOwn is:

- *Concern-specific* because its domain of discourse is the individual *software concern* for application resource ownership *structure* in object-oriented software systems.
- *Application domain-specific* because its intended *application domain* is interactive session-oriented services that are delivered by discrete, event-driven, soft real-time software systems.
- *Role-based* because it supports the notion that objects in the operational software system are capable of playing different, context-sensitive resource ownership roles.
- *Flexible* because the resource ownership structure it models is not hard-coded into the resource and owner model concepts, but instead built upon a dedicated concept of *proof of ownership instruments*. These *instruments* support a *rich* notion of *resource ownership* that allows different *owners* to play different *ownership roles*, each with different *ownership rights*, even with the same *resource*.
- *Modular*, as shown in Figure 4-1, because its structure has been intentionally constructed as two main *subontologies*, where a *subontology* is defined as a *top-level subclass* of the domain of

discourse which subsumes either:

- All concern-*specific* knowledge is encapsulated as *core concepts* or *classes* and manually placed (i.e., asserted) under the *ResOwn core subontology*. In this thesis, core concepts pertain to the selected application resource ownership concern.
- Any related knowledge not directly part of the selected concern is encapsulated as *support concepts* or *classes* and manually placed (i.e., asserted) under the *ResOwn support subontology*. *Support concepts* are used to *support* property restrictions of defined classes in the *ResOwn core subontology*. In this thesis, support concepts pertain to the *interactive session-oriented service application domain*, *resource capacity*, and *object persistency*.

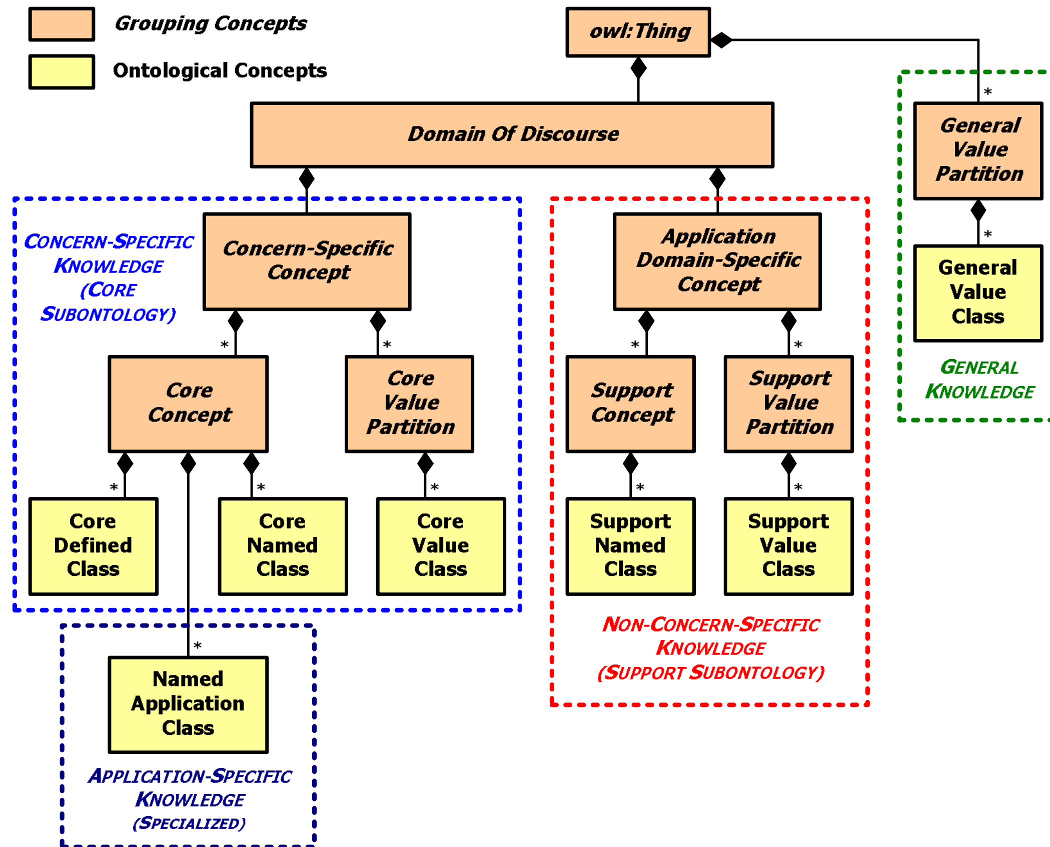



**Figure 4-1: Organizational block diagram of the entire ResOwn ontology.**

- *Extensible* because its concern-specific, core classes in the ResOwn core subontology can be extended with application-level knowledge using process known as *application-specific*

*specialization* (Chapter 5) and new support classes may be added to support subontology.

- *Automatically classifiable* by a reasoner because it is defined using the standard Web Ontology Language Description Logic (OWL-DL).

The organizational block diagram given in Figure 4-1 is intended to help readers organize their thoughts as they read through the major sections in this chapter. This diagram is a *schema* or *template* to be interpreted as follows: an instance of a *UML class diagram* will be a *containment hierarchy* of objects which is then reinterpreted as an *inheritance hierarchy* within the context of this thesis. This thesis considers only a single, concern-specific ontology for *application resource ownership structure* (i.e., ResOwn). The applicability of Figure 4-1 as a *concern-specific ontology template* to organize knowledge for other software concerns is *possible*. However, a definitive statement of applicability would require a number of test cases, and is therefore a subject of future work.

It will become apparent later in this chapter that the separation of core and support knowledge in ResOwn is used to impose an important *invariant* on ResOwn object properties. This invariant constrains how ResOwn property restrictions may be defined from an originating *core* class, in the *ResOwn core subontology*, to a terminating *support* or *value* class (or classes) in the *ResOwn support subontology* or *value partitions*, respectively.

### 4.1.1 Suggestion for Reading This Chapter

This chapter serves as a reference guide for readers already familiar with ResOwn. Unfortunately, first-time readers will find reading this chapter from start to finish impractical and difficult without the visual understanding of the ResOwn dynamic semantics obtained from Chapter 5. As a result, it is suggested that a first-time reader of this thesis read this chapter up to the end of Section 4.4, then read all of Chapter 5, and then return to Section 4.5 to complete reading the remaining chapter.

### 4.1.2 Chapter Organization

The remainder of this chapter is organized as follows. First, an overview of the main principles and conceptualizations, including the *ResOwn Prime Directive*, *resource benefits*, the difference between *beneficiary* and *nonbeneficiary* owners and the notion of *proof of ownership* is given. Second, a step-by-step discussion is given to introduce the ResOwn top-level *core*, *support* and *value partition* concepts and class hierarchies. Third, the ResOwn *foundation*, *self-contained*, *bridging* and *inferred*

properties types are defined, along with an annotated table containing all of the ResOwn object properties. Lastly, the chapter concludes with a complete reference guide to all ResOwn classes complete with individual *natural language descriptions* and the corresponding *description logic (DL) definitions* that are presented in a hierarchical, tabular form.

# 4.2 Conceptual Overview

ResOwn is a modular, reusable, extensible and specializable concern-specific ontology that provides the required vocabulary, concepts and properties, from the *knowledge representation domain*, for modeling application resource ownership structure, in the *object-oriented software domain*. ResOwn's underlying application *resource allocation and management scheme* relies on an ownership role- and ownership rights-based *proof of ownership* scheme that is presently applicable for interactive session-oriented services. The modularization of knowledge encoded in ResOwn implies that the ontology, in its current form, could be adapted to other *shared-resource* domains. This might include *low-level* runtime environments that use *lightweight* resources such as memory allocations, file handles and CPU execution time. In this thesis, the scope of ResOwn was intentionally limited for the *heavyweight application resources*. Therefore, from this point forward, the terms *resource* and *resource ownership* should be interpreted by the reader to mean *application resource* and *application resource ownership*, respectively.

## 4.2.1 The ResOwn Prime Directive

ResOwn has been devised and constructed as a *role-based* resource ownership ontology under a guiding philosophy or principle called the *ResOwn Prime Directive*[1] which states:

> *Every object that participates in the evolving resource ownership structure of an operational software system is assumed, under ResOwn, to be a* ***Resource****, an* ***Owner****, or both a* ***Resource*** *and an* ***Owner*** *simultaneously.*

---

[1] In the fictional universe of the 1960s television show *Star Trek*, the *Prime Directive* is Starfleet's *General Order #1* and is the most prominent guiding principle of the *United Federation of Planets* [Wik06].

From a dynamic modeling or monitoring perspective, each object in the runtime system may be represented as either a **Resource** and/or **Owner** in ResOwn depending on the particular structural ownership context the objects finds itself in[1]. For example, in the PBX, a *phone handler* is an **Owner**, a *line card* is a **Resource** to the *phone handler* and also an **Owner** to the card's *idle relay*, which is a **Resource** *embedded* into the *line card*. As will be seen later in this chapter, the *Prime Directive* impacts ResOwn's static and dynamic semantics and this permits **Resource** subclasses to be also be classified as an **Owner** subclass (i.e., *multiple inheritance*), and vice versa, in ResOwn's automatically generated *inferred class hierarchy*.

### 4.2.2 Top-Level Concepts and Properties

Consider the top-level ResOwn classes and ResOwn properties, as shown in Figure 4-2.

- The concern-specific, core **Resource** class and core **Owner** class are used to represent the existing *physical objects* of an operational software system.
- The **Owner** class subsumes the **Consumer** subclass, which essentially requests resources, the **Supplier** subclass, which essentially provides resources, and **Dispatcher** subclass, which may request and provide resources simultaneously. In the PBX:
    - A *phone handler* is a **Consumer** instance because it may own and consume the benefits[2] provided by a **Resource** instance.
    - A *channel manager* is a **Supplier** instance because it can provide *channels* to *phone handlers* upon request.
    - A *wait for call service* is a **Dispatcher** instance because it can assign a *phone handler* to process a *call request* when an originating *phone* goes *off hook*.
- The **Resource** class subsumes the **Transferable**, **Nontransferable**, **Embedded** and **Compound Resource** subclasses. In the PBX:

---

[1] This is similar to the notion that a *mobile* software agent's *role* depends on the agent's local *context* [Ken99].

[2] The definition for a resource benefit is given in Section 4.2.3.

- A *touch tone receiver card* is a **Transferable Resource** instance because its ownership may be directly *transferred* to a **Consumer** instance for exclusive use.
- A *dial tone generator card* is a **Nontransferable Resource** instance because its ownership is *not* directly transferable, but only be *indirectly* acquired via the *call progress tone manager*.
- An *idle relay device* is an **Embedded Resource** instance because it is *contained* in a *line card*. The *line card* is a **Transferable Compound Resource** instance because it *contains* an *idle relay device* and a *ringer relay device*. Ownership of these *embedded devices* can only be *indirectly* acquired via the *line card*.

- The concern-specific, core *proof of ownership* **Instrument** class is used to represent the *logical association instances* that are created or destroyed between **Resource** and **Owner** instances in the evolving resource ownership structure of an operational software system.
  - Every **Resource** instance is logically bound to a unique **Instrument** instance via the ***isBoundTo*** property.
  - Every **Instrument** instance is logically bound to a *holding* **Owner** instance via the ***hasHolder*** property and to an *issuing* **Owner** instance via the ***hasIssuer*** property.
  - The **Instrument** class subsumes the **Base Instrument** and the **Extent Instrument** subclasses. An **Extent Instrument** instance logically extends the *ownership scope* of an associated **Base Instrument** instance via the ***hasExtent*** property (not shown). Looking ahead, an example of an **Extent Instrument** instance (i.e., **Serial License** instance) extending a **Base Instrument** instance (i.e., **Nontransferable Titledeed** instance) is given in Figure 5-7 of Chapter 5.
  - The *ownership scope* of an **Instrument** instance is the n-ary association that includes at least the **Instrument** instance and its *holding* **Owner** instance, *issuing* **Owner** instance and associated **Resource** instance.
- The concern-specific, core *value partition* containing the **Ownership Right** value class is used to represent predefined sets of ownership rights that are designated to a particular **Instrument** instance via the ***hasRight*** property for the associated **Resource** instance and are normally assigned to the holder and/or the issuer of the **Instrument** instance.

While the **Instrument** class is specified as *disjoint* from the **Resource** and **Owner** classes, the

**Resource** and **Owner** classes are *not* specified as *disjoint* specifically because of the *ResOwn Prime Directive*. Further, in Figure 4-2, inferred links are shown between the **Owner** and **Resource** classes via the *inferred inverse* properties ***hasOwner*** and ***isOwnerOf***. A detailed description of the ResOwn properties is given in a tabular format in Section 4.4. The use of *asserted* and *inferred* properties will be discussed in Section 4.4.1.

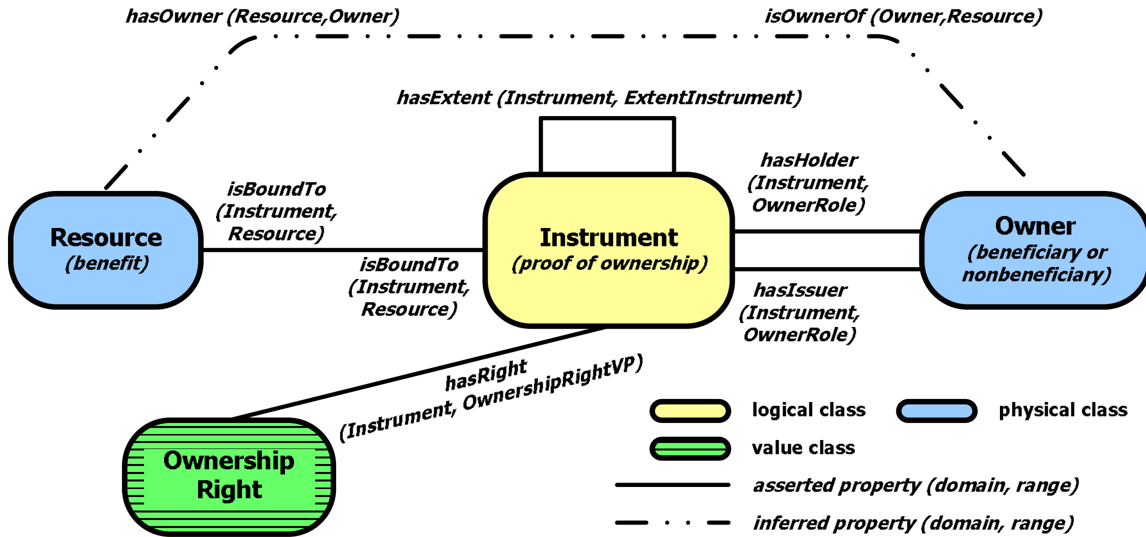



**Figure 4-2: Top-level ResOwn core classes and ResOwn properties.**

### 4.2.3 Resources Benefits

In the legal domain, a benefit is any profit or acquired right or privilege, primarily through a contract [Far06b]. In ResOwn, every **Resource** instance is assumed to be capable of producing a *benefit.* A *benefit* occurs in ResOwn whenever a **Resource** instance *produces* or *receives* application-level *data*, or produces an application-related *control action*, to the *benefit* of one (or more) **Owner** instances. A *benefit* normally propagates to, or from, the environment. A valid benefit recipient must hold an **Instrument** instance with the necessary **Data** or **Control Access Ownership Right** corresponding to the associated **Resource** instance. In the PBX, a *touch tone receiver card* (i.e., **Resource** instance) delivers *dialed digit* (i.e., a *benefit*) from the environment on behalf of a *phone handler* (i.e., **Consumer** instance). Similarly, a *dial tone generator* card (i.e., **Resource** instance) delivers a *dial tone* (i.e., a *benefit*) for the *phone handler* (i.e., **Consumer** instance) to the environment. This thesis considers the exact nature of a *benefit* to be behavioral in scope and specified orthogonal to structure.

### 4.2.4 Beneficiary and Nonbeneficiary Owner Roles

In the legal domain, an *owner* is one who has legal title or right to something and a *beneficiary* is any person or entity who receives assets or profits from any instrument in which there is distribution. In ResOwn, the **Consumer**, **Supplier** and **Dispatcher** subclasses are specialized *types* (i.e. *subclasses*) of **Owner** instances. However, each **Owner** *type* may also play one or more **Owner Role** instances. While an **Owner** *type* is static, **Owner Role** instances may change dynamically according to the resource ownership *context(s)* an **Owner** instance finds itself in. Therefore, *types* and *roles* are specified separately in ResOwn.

An **Owner** instance may play a beneficiary owner role and/or nonbeneficiary owner role:

- A *beneficiary owner* holds the necessary **Instrument** instance that allows it to own a **Resource** instance and receive the **Resource** instance's *benefit*. Normally, only a **Consumer** instance may be a *beneficiary owner.*
- A *nonbeneficiary owner* holds the necessary **Instrument** instance that allows it to own a **Resource** instance but *not* to receive the **Resource** instance's *benefit*. Normally, all **Owner** instance may be a nonbeneficiary *owner.*

Contrary to the adage *"Possession is nine-tenths of the law,"* possession in ResOwn does not necessarily make one a *beneficiary owner*. In the PBX, the same *line card* (i.e., **Resource** instance) may have a single *phone handler* (i.e., **Consumer** instance) as its *beneficiary owner*, and *line scanner* (i.e., **Supplier** instance) as a *nonbeneficiary owner*. Yet only the *phone handler* is entitled to receive the *switch hook status* (i.e., the *benefit*) from the *line card*.

Distinguishing **Owner Role** instances in ResOwn provides an important means for dealing with role-based resource capacity and capacity-based cardinality restrictions at runtime. In general, a **Resource** instance may have multiple **Owner** instances, but not all of them are entitled to receive a benefit. For example, consider a *touch tone receiver (ttrx) card* from the PBX with a specified *user capacity* of 1 *beneficiary owner*:

- If the *ttrx card* is owned by a *phone handler* and, at the same time, owned by a *ttrx scanner*, then the *card* has one *beneficiary owner* and one *nonbeneficiary owner*; the *capacity* is not violated.
- If the *ttrx card* is owned by two *phone handlers* at the same time, then the *card* has two *beneficiary owners*; the capacity is violated.

Therefore, ResOwn's use of role-based resource ownership provides a means to distinguish between **Owner** instances that can be monitored and checked at runtime.

### 4.2.5 Direct and Indirect Proof of Ownership

In ResOwn, *ownership* is defined between an **Owner** instance and a **Resource** instance, and *proof of ownership* is provided to an **Owner** instance that is the *holder* of an appropriate **Instrument** instance that is (*directly* or *indirectly*) bound to the associated **Resource** instance.

- *Direct* proof of ownership occurs when an **Owner** instance *holds* a **Base Instrument** instance that is *directly* bound to the owned **Resource** instance via an ***isBoundTo*** property restriction.
- *Indirect* proof of ownership occurs when an **Owner** instance *holds* an **Extent Instrument** instance that is bound to the **Base Instrument** instance of the owned **Resource** instance via an ***isExtentOf*** property restriction. The **Extent Instrument** instance facilitates the evolution of resource ownership structure at runtime by *extending* a predefined set of **Ownership Right** instances from the extended **Base Instrument** instance to the holding **Owner** instance of the **Extent Instrument** instance. Looking ahead again, two examples of how **Extent Instrument** instances facilitate the evolution of resource ownership structure at runtime are given in Figure 5-7 and Figure 5-8 of Chapter 5.

## 4.3 Taxonomy Overview

The top-level of the ResOwn asserted class hierarchy is shown in Figure 4-3. ResOwn encodes and integrates support and core knowledge into a single, unified *application resource ownership domain of discourse*. The organizational block structure of ResOwn follows the diagram presented in Figure 4-1. Labels, shown in *SMALL CAPS ITALICS* with the ResOwn taxonomy in Figure 4-3, relate back to corresponding blocks in Figure 4-1. These systematic, enforced modularizations of the overall ResOwn ontology structure itself promotes extensibility, reusability, comprehensibility and maintainability, and provides a path for creating a *specialized ResOwn instance* that *extends* the *baseline* ResOwn ontology with *application-specific* knowledge. An example of a *specialized ResOwn instance* for the PBX is presented in Chapter 5.

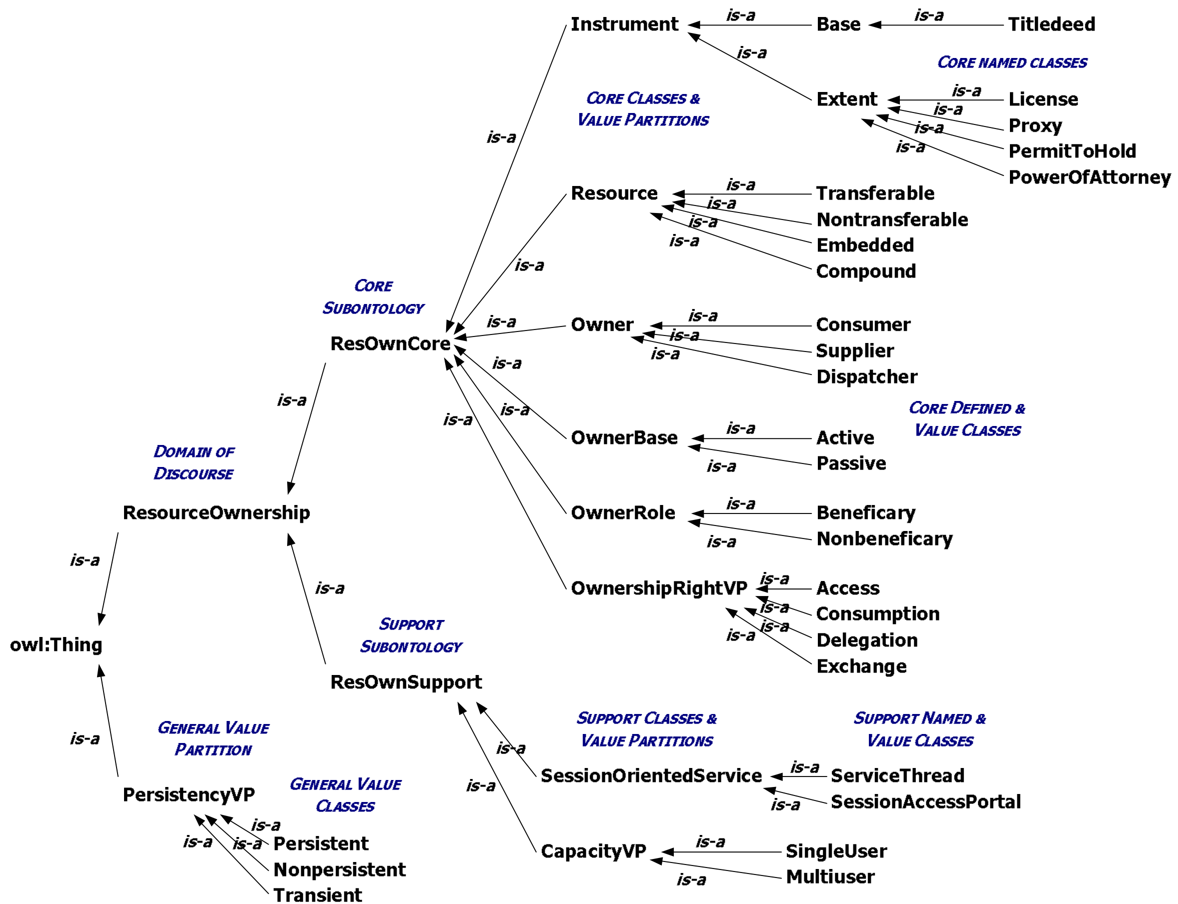



**Figure 4-3: Top-levels of the ResOwn class hierarchy.**

### 4.3.1 Traversing the ResOwn taxonomy

Traversing down the ResOwn taxonomy from the **owl:Thing** root, as shown in Figure 4-3.

- The top-level of the asserted class hierarchy consists of the **ResourceOwnership** *domain of discourse* and the **Persistency VP** *general (knowledge) value partition.*
- **ResourceOwnership** subsumes the **ResOwnCore** *subontology* and the **ResOwnupport** subontology.
- The **ResOwnCore** subontology is disjoint from the **ResOwnSupport** *subontology* and is comprised of concern-specific concepts:
  - o *Core subclasses* subsumed by the core **Resource**, **Owner**, **Instrument** classes.

- *Core value classes* subsumed by the **Ownership Right VP** *core value partition.*

- The **ResOwnSupport** subontology is disjoint from the **ResOwnCore** subontology and is comprised of:
  - *Support* subclasses subsumed by **Session Oriented Service** class.
  - *Support value classes* subsumed by the **Capacity VP** value partition.

### 4.3.2 Top-Level Class Origins

The conceptualizations that make up the top-level classes in ResOwn, as shown in Figure 4-3, have originated from a number of related or orthogonal domains of interest.

- The **Resource** class originated from the *client-server software* domain [Cou01, Lew98] and the DAML scheduling (resource) ontology [Dam00b].
- The **Owner** class originated from a blend of the *real property* and *legal* domains [Far06b, Mcc02, Yip04] and the *client-server software* domain [Cou01, Lew98].
- The **Instrument** class originated from the *real property* and *legal* domains [Far06b, Mcc02, Yip04].
- The **Session Oriented Service** class originated the *interactive service* domain [Bro01, Kur01] and from previous work [Pek97, Pek03].
- The **Access** and **Consumption Ownership Right** value classes originated from the *software* domain [Mcca02, Pfl06].
- The **Exchange** and **Delegation Ownership Right** value classes originated from the *real property* and *legal* domains [Far06b, Mcc02, Yip04].
- The **Capacity** value class originated from the DAML scheduling (capacity) ontology [Dam00a].
- The **Persistency** value class originated from the object-oriented software domain [Cou01].

### 4.3.3 Baseline and Specialized ResOwn Class Hierarchies

The *baseline* ResOwn ontology is *application-general* and, therefore, pertinent to a wide range of software systems from the interactive session-oriented service application domain. A *specialized*

ResOwn ontology is *application-specific*; that is, the *baseline* ResOwn ontology is *specialized* or *extended* with application-specific ontological classes that have been derived directly from a software system's actual object classes, resulting in a *specialized ResOwn instance*. An example to illustrate how the *baseline* ResOwn class hierarchy is specialized using object classes from the PBX is presented in Chapter 5.

### 4.3.4 ResOwn Completeness

This thesis does not claim that ResOwn is a complete ontology. The thesis aims to consolidate core and support knowledge about resource ownership into a practical, workable and, most importantly, extensible resource ownership ontology for the selected interaction session-oriented service domain. These factors make ResOwn useful in its current form, as well as adaptable to other new applications or concerns, even if ResOwn is not complete.

### 4.3.5 Self-Contained and Distributed Subclass Hierarchies

In ResOwn, the top-level classes subsume either *self-contained* or *distributed* subclass hierarchies.

- Most of the *top-level* classes in ResOwn are *self-contained*; that is, all knowledge pertaining to the particular top-level core class (e.g., **Resource** or **Instrument**) is subsumed by a *single* subclass hierarchy originating from the top-level core class itself. The s*elf-contained* approach is used when knowledge is to be encoded into a single *top-level* class hierarchy.
- The *top-level* **Owner** class is *distributed*; that is, all knowledge pertaining to the particular top-level core class is contained in a *single* subclass hierarchy that is subsumed by a single, top-level core class (e.g., **Owner**) and called the *main class* plus *one or more* subclass hierarchies, each subsumed by its own top-level core class (**Owner Base**, **Owner Role**) called a *library* class[1]. Each library class is disjoint from its main class and from the other library classes. Library classes are normally modeled as ResOwn properties. For example, **Owner Role** and **Owner Base** are modeled as ***hasRole*** and ***hasBase***, respectively, as shown in Figure 4-4. The *distributed* approach promotes modularization and extensibility and uses *property restrictions* to related knowledge

[1] Analogous to a software library.

between the *main* class and each of the *library* classes.

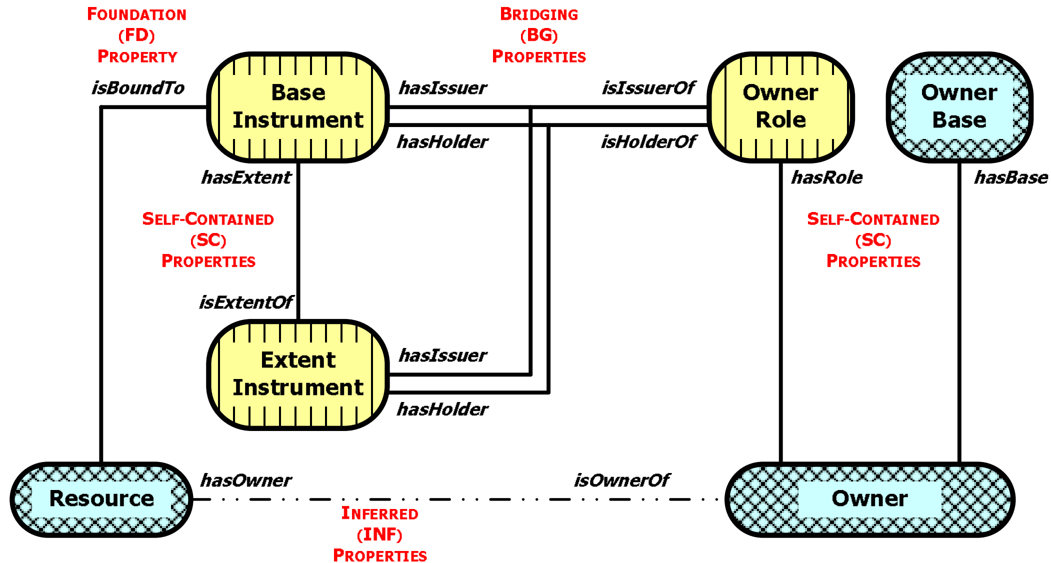



**Figure 4-4: Top-level core classes and key object properties.**

# 4.4 Properties

In addition to an asserted class hierarchy, ResOwn also defines a number of *properties*[1]. In OWL, *object properties* are used to formally specify *links* (i.e., *property restrictions*) between pairs of *individuals* like those shown in Figure 4-4. For example, a **Resource** instance is *linked* to a corresponding **Base Instrument** instance via the ***isBoundTo*** property. For presentation purposes, ResOwn properties are grouped in a tabular format according to whether a property models a *core*, *support* or *value* class in Table 4-1, Table 4-2, and Table 4-3. Each property in the tables is specified along with its applicable *property characteristics* (i.e., *Functional (F)*, *Inverse (I)*, *Inverse Functional (IF)* and/or *Symmetrical (S)*, the corresponding *inverse property* (if applicable) and the property's associated *domain* and *range*. The definition of these property terms was given in the OWL-DL tutorial in Section 2.5 of Chapter 2. While mostly self-explanatory, the meanings of each property

[1] ResOwn only uses OWL object properties.

will become more apparent once the reader sees how these properties are used in various property restrictions to specify the ResOwn define classes shown later in this chapter.

### 4.4.1 ResOwn Property Extensions

In ResOwn, OWL object properties are extended in two ways.

- A property's *domain* and *range* in ResOwn is constrained depending on whether the class the property models is a core ResOwn concept, or merely providing a ResOwn *support* or *value* concept, in accordance with the organizational block diagram of Figure 4-1.
  - A *core* property models a *core* class from the *core* subontology and may only contain individuals from other *core* classes in its *domain* and *range*. The ResOwn *core* properties are shown in Table 4-1.
  - A *support* property models a *support* class from the *support* subontology and may only contain individuals from other *core* classes in its *domain* and individuals from other *support* classes in its *range*. The ResOwn *support* properties are shown in Table 4-2.
  - A *value* property models a *value* class from a *value partition* and may only contain individuals from other core classes in its *domain* and individuals from value partition *value* classes in its *range*. The ResOwn *value* properties are shown in Table 4-3.

### 4.4.2 ResOwn Property Types

Each ResOwn property has a property type, as shown in Table 4-1, Table 4-2 and Table 4-3. These property types enforce the invariants imposed on the *property restriction* used to define the core ResOwn classes as follows:

- A *foundation (FD) property* in ResOwn is used to create a symmetrically, *context-free*[1] property restriction that asserts a *bidirectional link* between pairs of individuals from two *disjoint*, top-level, *core* classes in the core subontology. An example is the ***isBoundTo*** property between the **Resource** and **Base Instrument** classes, as shown in Figure 4-4 and Table 4-1. Individuals from

---

[1] In ResOwn, context-free means individuals are acceptable in a property's domain or range, interchangeably.

these two *core* classes may appear separately in either the FD property's domain or range, but not both.

| Property | Property Characteristic | Property Type | Inverse Property | Property Domain | Property Range |
|---|---|---|---|---|---|
| hasContainer | F, I | SC | isContainerOf | Resource | Resource |
| hasDispatcher | F, I | SC | isDispatcherOf | DispatchableConsumer | Dispatcher |
| hasExtension | F, IF, I | SC | isExtensionOf | Instrument | Instrument |
| hasHolder | F, I | BG | isHolderOf | Instrument | OwnerRole |
| hasIssuer | F, I | BG | isIssuerOf | Instrument | OwnerRole |
| hasOwner | I | INF | isOwnerOf | Resource | Owner |
| hasBase | F | BG | - | Owner | OwnerBase |
| hasRole | | BG | - | Owner | OwnerRole |
| isBoundTo | F, IF, S, I | FD | isBoundTo | Resource ⊔ Instrument | Instrument ⊔ Resource |
| isContainerOf | I | SC | hasContainer | Resource | Resource |
| isDispatcherOf | I | SC | hasDispatcher | Dispatcher | DispatchableConsumer |
| isExtensionOf | F, IF, I | SC | hasExtension | Instrument | Instrument |
| isHolderOf | I | BG | hasHolder | OwnerRole | Instrument |
| isIssuerOf | I | BG | hasIssuer | OwnerRole | Instrument |
| isOwnerOf | I | INF | hasOwner | Owner | Resource |

**Table 4-1: ResOwn properties modeling core classes.**

- A *self-contained (SC) property* in ResOwn is used to create a *context-sensitive*[1] property restriction that asserts a *unidirectional link* between pairs of individuals from the same top-level core class in the core subontology. An example is the ***hasExtent*** property between the **Base Instrument** and **Extent Instrument** classes, as shown in Figure 4-4 and Table 4-1. Only individuals from subclasses subsumed by a single core class may appear in the SC property's domain and range. A SC property with an inverse (e.g., ***hasExtent*** and ***isExtentOf***) results in a pair of *unidirectional links* between a pair of individuals from the two core classes, respectively.
- A *bridging (BG) property* in ResOwn is used to create a *context-sensitive* property restriction that asserts a *unidirectional link* between individuals from a single, core class to individuals

from a disjoint *core*, *support* or *value* class. An example is the ***hasRight*** property between the **Instrument** and **Ownership Right VP** classes, as shown in Figure 4-2 and Table 4-3. Another example is the ***hasHolder*** and inverse ***isHolderOf*** properties between the **Instrument** and **Owner Role** classes, as shown in Figure 4-4 and Table 4-1.

- An *inferred (INF) property* in ResOwn is used to create an inferred property restriction that implicitly deduces the existence of a virtual link between pairs of individuals from one or more core classes. This is contrasted by an asserted (i.e., normal) property restriction, which is explicitly stated in the ontology and does not rely on any implicit constraints. An example is the hasOwner and inverse isOwnerOf properties between the Resource and Owner classes, as shown in Figure 4-2, Figure 4-4, and Table 4-1. An INF property restriction can be derived using a process known as *asserted property chaining*.

| Property | Property Characteristic | Property Type | Inverse Property | Property Domain | Property Range |
|---|---|---|---|---|---|
| hasPortal | F | BG | - | Resource | SessionAccessPortal |
| hasServiceThread | F | BG | - | OwnerBase | ServiceThread |

**Table 4-2: ResOwn properties modeling support classes.**

| Property | Property Characteristic | Property Type | Property Domain | Property Range |
|---|---|---|---|---|
| hasCapacity | F | BG | Resource ⊔ Instrument | CapacityVP |
| hasPersistency | F | BG | Resource ⊔ Instrument ⊔ Owner | PersistencyVP |
| hasRight | | BG | Instrument | OwnershipRightVP |

**Table 4-3: ResOwn properties modeling value classes.**

#### 4.4.2.1 Asserted Property Chaining

*Asserted property chaining* uses Functional Composition to formally specify or define INF properties in ResOwn by logically *chaining* the *domain* of one asserted property in a chain to the *range* of

[1] In ResOwn, context-sensitive means different individuals are acceptable in the domain or range, but not both.

another property, in an appropriate way, until the required INF property is defined[1]. The resultant INF property will have the *domain* of the first asserted property in the chain, and the *range* of the last asserted property in the chain. The use of INF properties, in conjunction with the concept of *proof of ownership*, as shown in Figure 4-2 and Figure 4-4, gives ResOwn the power to dynamically model an operational software system's evolving resource ownership structure. Consider the INF property isOwnerOf, as shown in Figure 4-2, Figure 4-4, and Table 4-1, which may be defined using the following asserted property chain:

***isOwnerOf*** (**Owner**, **Resource**) ← ***hasRole***(**Owner**, **OwnerRole**) o
***isHolderOf*** (**OwnerRole**, **Instrument**) o
***isBoundTo*** (**Instrument**, **Resource**)

This chain specifies that an *inferred ownership relationship* exists between an **Owner** instance and a **Resource** instance, if and only if the **Owner** instance has the appropriate **Owner Role**, and that **Owner Role** is permitted to be the *holder* of the necessary **Instrument** instance associated with the **Resource** instance. Since ***isOwnerOf*** and ***hasOwner*** are inverse INF properties (i.e., ***isOwnerOf*** implies ***hasOwner***, and vice versa), the same chain specifies either INF property in the inverse pair.

# 4.5 ResOwn Core, Support, and Value Classes

The sections which follow provide the reader with a detailed presentation of the subclass hierarchies, natural language class descriptions, and OWL-DL-based definitions for the core, support, and value classes of the ResOwn ontology. The sections are essentially a reference guide for those readers who wish to acquire an indepth, detailed understanding of ResOwn. The figures containing the various asserted subclass hierarchies were manually drawn based on the Protégé-Owl tool version of ResOwn. The figures containing the various *inferred subclass hierarchies* are actually *screenshots* from the Protégé-Owl tool's OWLViz plugin. The OWL-DL-based class definitions are presented to the reader in tabular format. To simplify the description logic definitions themselves, *closure axioms*,

---

[1] In mathematics, a *composite function*, formed by the composition of one function on another, represents the application of the former to the result of the application of the latter to the argument of the composite. The functions $\boldsymbol{f}: \boldsymbol{X} \rightarrow \boldsymbol{Y}$ and $\boldsymbol{g}: \boldsymbol{Y} \rightarrow \boldsymbol{Z}$ can be composed by first applying $\boldsymbol{f}$ to an argument $\boldsymbol{x}$ and then applying $\boldsymbol{g}$ to

present in the Protégé-Owl version of ResOwn, have been intentionally left out of the tables.

## 4.6 Proof of Ownership Instruments

The self-contained, *asserted class hierarchy* of the top-level **Instrument** class is given in Figure 4-5 and includes both the *defined* and *named* (i.e., *primitive*) **Instrument** subclasses. The more interesting *inferred class hierarchy*, shown in Figure 4-6, was obtained by classifying ResOwn automatically using the RacerPro reasoner and inference engine.

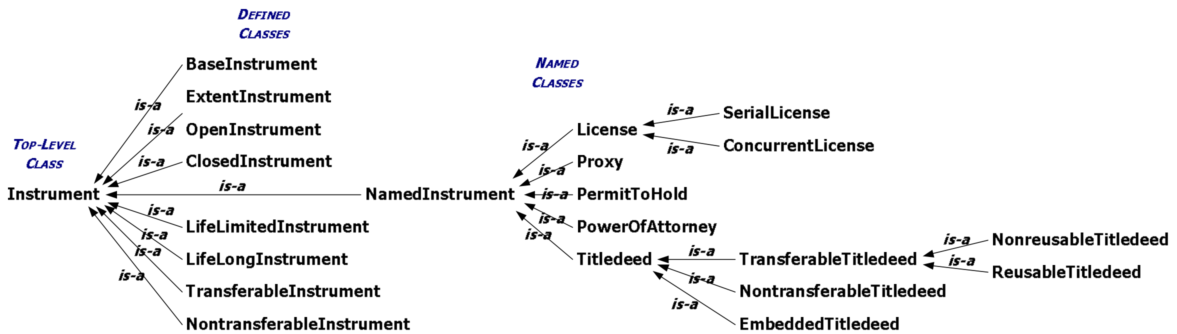



**Figure 4-5: Asserted (defined and named) Instrument class hierarchy.**

### 4.6.1 Instrument Class

An *instrument* is a written, legal document, such as a contract, lease, deed, will or bond, that lays out the parties involved, triggering events and terms of the contract, communicates the intended purpose and scope and represents a share of a liability or ownership [Far06b, Far06c]. In ResOwn, an **Instrument** instance conveys a set of **Ownership Right** instances upon a holding **Owner** instance and provides that **Owner** instance with *proof of ownership* of an associated Resource instance. The OWL-DL for the **Instrument** class is given in Table 4-4.

---

the result. Thus one obtains a function $\boldsymbol{g} \circ \boldsymbol{f}: \boldsymbol{X} \rightarrow \boldsymbol{Z}$ defined by $(\boldsymbol{g} \circ \boldsymbol{f})(x) = \boldsymbol{g}(\boldsymbol{f}(\boldsymbol{x}))$ for all $\boldsymbol{x}$ in $\boldsymbol{X}$.

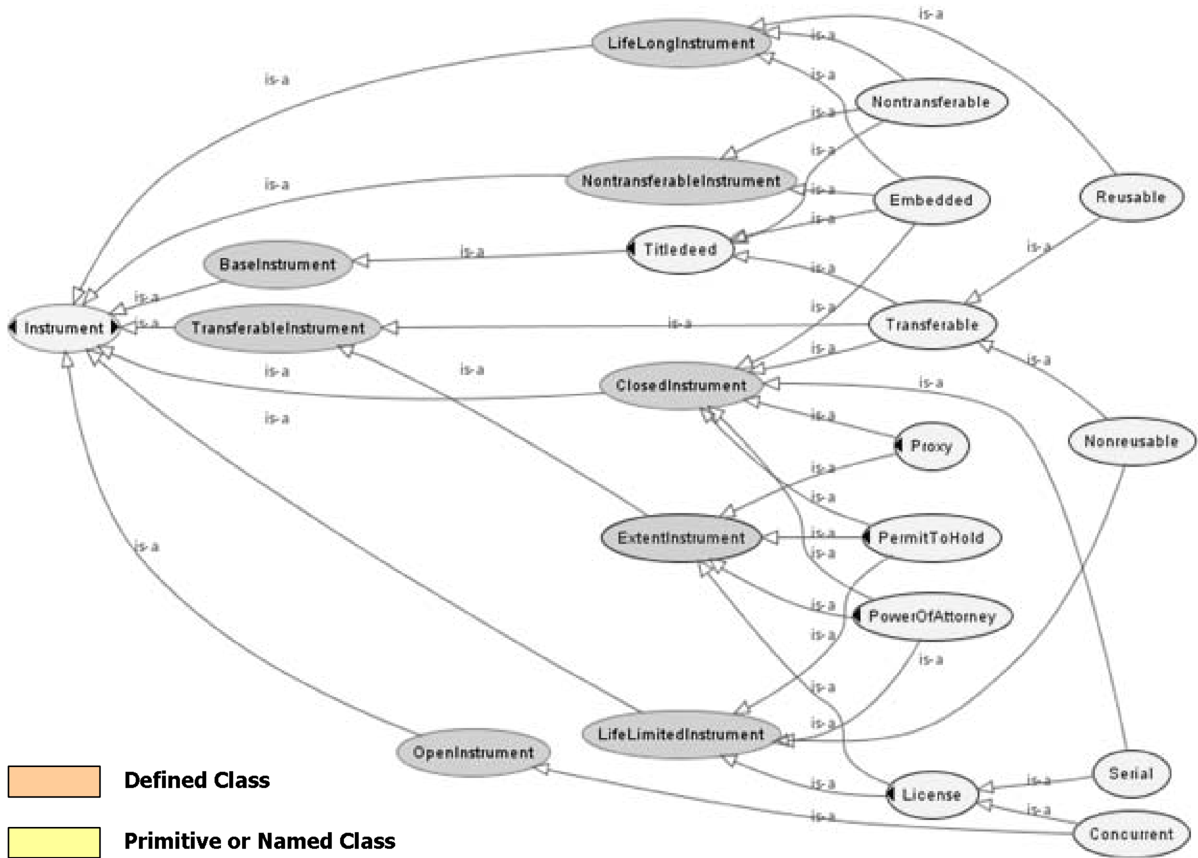



**Figure 4-6: Inferred (defined and named) Instrument class hierarchy.**

| Core Class | Asserted and Inferred Conditions | Property Name and Range [N = necessary] [S = necessary and sufficient] | Disjoint |
|---|---|---|---|
| Instrument | (N) ResOwnCore<br>(N) ∀ isBoundTo Resource | isBoundTo (single Instrument ⊔ Resource)<br>isExtentOf (single Instrument)<br>hasHolder (multiple OwnerRole)<br>hasIssuer (multiple OwnerRole)<br>hasExtent (multiple Instrument)<br>hasCapacity (single CapacityVP)<br>hasPersistency (single PersistencyVP)<br>hasRight (multiple OwnershipRightVP) | Resource<br>Owner<br>OwnerRole<br>OwnerBase<br>OwnershipRightVP<br>ResOwnSupport |

**Table 4-4: Top-level Instrument class definition.**

### 4.6.1.1 Instrument Capacity

*Capacity* is defined as the ability to receive, hold or absorb [Far06a]. In ResOwn, The *openness* of an **Instrument** instance is based on its associated **Capacity** *value* instance. The **Capacity VP** *value*

*partition* is modeled as the property ***hasCapacity***, as shown in Table 4-3, whose range may hold an individual from either the disjoint **Single User Capacity** or **Multi User Capacity** *value class*. A **Single User Capacity** *value* instance represents a *cardinality restriction* of ***1***and a **Multi User Capacity** *value* instance represents a cardinality *restriction* of ***N***.

#### 4.6.1.2 Instrument Persistency

*Persistence* is the property of a continuous and connected period of time [Far06a]. An object that is guaranteed to live between activations is called a *persistent* object [Cou01]. *Lifespan* is the average or maximum length of time an organism, material or object can be expected to survive or last [Far06a]. In ResOwn, the lifespan of an **Instrument** instance is based on its associated **Persistency** *value* instance. The **Persistency VP** *value partition* is modeled as the property ***hasPersistency***, as shown in Table 4-3, whose range may hold an individual from the disjoint **Persistent**, **Nonpersistent** or **Transient** *value class*. An explanation of these general value classes is given in Table 4-5. The table column headings specify *temporal boundaries* (i.e., windows of time) that relate to the *natural* runtime life of an object or a runtime binary association instance (i.e., link) between a pair of objects. The table row headings specify **Persistency**.

| | **Persistent** | **Nonpersistent** | **Transient** |
|---|---|---|---|
| **System** | Exists for runtime life of system | Exists from system start until event before end of system runtime life | Exists from some intermediate period during the system runtime life |
| **Session** | Exists for runtime life of session | Exists from session start until event before end of session | Exists from some intermediate period during a session |
| **Transaction** | Exists for runtime life of transaction | Exists from transaction start until event before end of transaction | Exists from some intermediate period during a transaction |

**Table 4-5: Persistency versus Time.**

## 4.7 Defined Instrument Classes

The subsections that follow describe and define the *defined* **Instrument** classes in ResOwn.

### 4.7.1 BaseInstrument Class

A *base* is defined as the fundamental principle or underlying concept of a system or theory [Far06a].

In ResOwn, each **Base Instrument** instance is *bi-directionally* linked to a particular **Resource** instance via an ***isBoundTo*** property restriction. The OWL-DL definition is given in Table 4-6. The bi-directional link is *permanent*, unless the **Resource** is *consumable*.

| Defined Instrument Class | Subsumed By Class | | Active Asserted Conditions [N = necessary] [S = necessary & sufficient] |
|---|---|---|---|
| | Asserted | Inferred | |
| BaseInstrument | (S) Instrument | (S) Instrument | (S) ∃ isBoundTo Resource<br>(S) isExtentOf = 0 |
| ExtentInstrument | (S) Instrument | (S) Instrument<br>(N) TransferableInstrument | (S) isBoundTo = 0<br>(S) ∃ isExtentOf Instrument |
| OpenInstrument | (S) Instrument | (S) Instrument | (S) ∃ hasCapacity MultipleUser |
| ClosedInstrument | (S) Instrument | (S) Instrument | (S) ∃ hasCapacity SingleUser |
| LifeLongInstrument | (S) Instrument | (S) Instrument | (S) ∃ hasPersistency Persistent |
| LifeLimitedInstrument | (S) Instrument | (S) Instrument | (S) ∃ hasPersistency Nonpersistent |
| TransferableInstrument | (S) Instrument | (S) Instrument | (S) ∃ isBoundTo TransferableResource ⊔ isBoundTo = 0<br>(S) isBoundTo = 0 |
| NontransferableInstrument | (S) Instrument | (S) Instrument | (S) ¬ TransferableInstrument |

**Table 4-6: Defined Instrument class definitions.**

## 4.7.2 ExtentInstrument Class

An *extent* is a writ allowing a creditor to assume temporary ownership of a debtor's property [Far06b]. In ResOwn, an **Extend Instrument** instance, when in effect, forms pairs of *temporarily, unidirectional links* between itself and either a **Base Instrument** instance or another **Extent Instrument** instance, via the ***isExtentOf*** and ***hasExtent*** property restrictions. The OWL-DL definition is given in Table 4-6.

## 4.7.3 TransferableInstrument Class

In ResOwn, a **Transferable Instrument** instance may be issued by the **Instrument** instance's current holding **Owner** instance to a new holding **Owner** instance. Therefore, a **Transferable Instrument** instance may, over its lifespan, potentially have many different holding **Owner** instances. An **Extent Instrument** is inherently a **Transferable Instrument**. The OWL-DL definition is given in Table 4-6.

### 4.7.4 NontransferableInstrument Class

In ResOwn, a **Nontransferable Instrument** instance remains with its initial holding **Owner** instance for the **Instrument** instance's lifespan. Therefore, a **Nontransferable Instrument** instance may only be issued indirectly to other **Owner** instances via an **Extent Instrument** instance. The OWL-DL definition is given in Table 4-6.

### 4.7.5 ClosedInstrument Class

A **Closed Instrument** instance is one that has been specified with a **Single User Capacity** *value* instance. This creates a *cardinality restriction* of "1" between a *holding* **Owner** instance and the **Closed Instrument** instance. The OWL-DL definition is given in Table 4-6.

### 4.7.6 OpenInstrument Class

An **Open Instrument** instance is one that has been specified with a **Multi User Capacity** value instance. This creates a *cardinality restriction* of "N" between a set of *holding* **Owner** instances and the **Open Instrument** instance. The OWL-DL definition is given in Table 4-6.

### 4.7.7 LifeLongInstrument Class

A **Life Long Instrument** instance is one that has been specified with a **Persistent** *value* instance, meaning the **Instrument** instance remains in *effect* for the *runtime life* of the software system. The OWL-DL definition is given in Table 4-6.

### 4.7.8 LifeLimitedInstrument Class

A **Life Limited Instrument** instance is one that has been specified with a **Nonpersistent** *value* instance, meaning the **Instrument** instance remains in *effect* for some period of time *less* than the *runtime life* of the software system. The OWL-DL definition is given in Table 4-6.

## 4.8 Named Instrument Classes

The subsections that follow describe and define the *named* **Instrument** classes in ResOwn. Note that

in Figure 4-6, the identifier names of the **Named Instrument** classes were shortened so that the inferred class hierarchy would fit on the page. The sections that follow use the full class names.

## 4.8.1 Titledeed Class

A *deed* is a written document that transfers ownership or an interest in real property from a transferring party (i.e., *issuer*) to a receiving party (i.e., *holder*) [Far06b]. In ResOwn, the **Titledeed** class is a **Base Instrument**. ResOwn defines three main **Titledeed** subclasses: **Embedded**, **Nontransferable**, and **Transferable**, described below and each corresponding to a **Resource** subclass with the same prefix. The **Titledeed** class is disjoint from all other **Named Instrument** classes. The OWL-DL definition is given in Table 4-7.

| Named Instrument Class | Subsumed By Class | | Active Asserted Conditions [N = necessary] [S = necessary & sufficient] |
|---|---|---|---|
| | Asserted | Inferred | |
| Titledeed | (N) NamedInstrument | (N) NamedInstrument<br>(N) BaseInstrument | (N) ∃ isBoundTo Resource<br>(N) isExtentOf = 0 |
| EmbeddedTitledeed | (N) Titledeed | (N) Titledeed<br>(N) LifeLongInstrument<br>(N) NontransferableInstrument<br>(N) ClosedInstrument | (N) ∃ isBoundTo EmbeddedResource<br>(N) ∃ hasCapacity SingleUser<br>(N) ∃ hasPersistency SystemPersistent<br>(N) ∃ hasRight DataAccessRight<br>(N) ∃ hasRight ControlAccessRight<br>(N) ∃ hasRight ProxingRight |
| NontransferableTitledeed | (N) Titledeed | (N) Titledeed<br>(N) LifeLongInstrument<br>(N) NontransferableInstrument | (N) ∃ isBoundTo NontransferableResource<br>(N) ∃ hasPersistency SystemPersistent<br>(N) ∃ hasRight DataAccessRight<br>(N) ∃ hasRight ControlAccessRight<br>(N) ∃ hasRight LicensingRight |
| TransferableTitledeed | (N) Titledeed | (N) Titledeed<br>(N) ClosedInstrument<br>(N) TransferableInstrument | (N) ∃ isBoundTo TransferableResource<br>(N) ∃ hasCapacity SingleUser<br>(N) ∃ hasRight DataAccessRight<br>(N) ∃ hasRight ControlAccessRight<br>(N) ∃ hasRight ReleaseRight<br>(N) ∃ hasRight PermittingRight<br>(N) ∃ hasRight TransferRight |
| NonreusableTitledeed | (N) TransferableTitledeed | (N) TransferableTitledeed<br>(N) LifeLimitedInstrument | (N) ∃ hasPersistency SystemNonpersistent<br>(N) ∃ hasRight ConsumptionRight |
| ReusableTitledeed | (N) TransferableTitledeed | (N) TransferableTitledeed<br>(N) LifeLongInstrument | (N) ∃ hasPersistency SystemPersistent<br>(N) ∃ hasRight AttorneyingRight |

**Table 4-7: Titledeed class definitions.**

### 4.8.1.1 EmbeddedTitledeed Subclass

An Embedded Titledeed instance is a **Nontransferable**, **Life Long**, **Closed Titledeed** that is directly bound to an **Embedded Resource** instance and may only be held by a **Compound Resource** instance

acting as a nonbeneficiary containment owner. The OWL-DL definition is given Table 4-7. An **Embedded Titledeed** instance may be extended by a **Proxy** instance.

#### 4.8.1.2 NontransferableTitledeed Subclass

A **Nontransferable Titledeed** instance is a **Nontransferable**, **Life Long Titledeed** that is *directly* bound to a **Nontransferable Resource** instance and may only be *held* by a **Managed Resource Supplier** instance acting as a *nonbeneficiary permanent owner*. The OWL-DL definition is given in Table 4-7. A **Nontransferable Titledeed** instance may be *extended* by a **License** instance.

#### 4.8.1.3 TransferableTitledeed Subclass

A **Transferable Titledeed** instance is a **Transferable**, **Closed Titledeed** that is *directly* bound to a **Transferable Resource** instance and may only be *held* by a **Consumer** instance acting as a *beneficiary current owner* or by a **Pooled Resource Supplier** instance acting as a *nonbeneficiary default owner*. The OWL-DL definition is given in Table 4-7. A **Transferable Titledeed** instance may be *extended* by a **Permit To Hold** instance.

##### 4.8.1.3.1 NonreusableTransferableTitledeed Subclass

A **Nonreusable Transferable Titledeed** instance is a **Life Limited**, **Transferable Titledeed** *directly* bound to a **Consumable Resource** instance. The OWL-DL definition is given in Table 4-7.

##### 4.8.1.3.2 ReusableTransferableTitledeed Subclass

A **Reusable Transferable Titledeed** instance is a **Life Long**, **Transferable Titledeed** *directly* bound to a **Nonconsumable Resource** instance. The OWL-DL definition is given in Table 4-7. A **Reusable Transferable Titledeed** instance may be *extended* by a **Power Of Attorney** instance.

### 4.8.2 License Class

A *license* is a certificate that proves that one has official or legal permission to do or own a specified thing [Far06b]. In ResOwn, a **License** instance is a **Life Limited**, **Extent Instrument** that is indirectly bound to a **Nontransferable Resource** instance via a **Nontransferable Titledeed** instance. The **License** instance extends *beneficiary ownership* from the **Nontransferable Titledeed** instance to the holding **Consumer** instance acting as a *beneficiary licensed owner*. The **License** instance is issued by the **Managed Resource Supplier** instance, which acts as the *nonbeneficiary permanent*

*owner* of the **Nontransferable Resource** instance. The **License** class is disjoint from all other **Named Instrument** classes. The OWL-DL definition is given in Table 4-8.

### 4.8.2.1 SerialLicense Subclass

In ResOwn, a **Serial License** instance is a **Closed License**. The **Serial License** subclass is *disjoint* from all other **License** subclasses. The OWL-DL definition is given in Table 4-8. A **Serial License** instance may be *extended* by a **Power Of Attorney** instance or **Permit To Hold** instance.

### 4.8.2.2 ConcurrentLicense Subclass

In ResOwn, a **Concurrent License** instance is an **Open**, **Extent License**. The **Concurrent License** subclass is *disjoint* from all other **License** subclasses. The OWL-DL definition is given in Table 4-8. A **Concurrent License** instance may *not* be *extended*.

| Named Instrument Class | Subsumed By Class | | Active Asserted Conditions [N = necessary] [S = necessary & sufficient] |
|---|---|---|---|
| | Asserted | Inferred | |
| License | (N) NamedInstrument | (N) NamedInstrument<br>(N) ExtentInstrument<br>(N) LifeLimitedInstrument | (N) ∃ isExtentOf NontransferableTitledeed<br>(N) isBoundTo = 0<br>(N) ∃ hasPersistency SessionNonpersistent<br>(N) ∃ hasRight DataAccessRight |
| Concurrent License | (N) License | (N) License<br>(N) OpenInstrument | (N) ∃ hasCapacity MultipleUser |
| Serial License | (N) License | (N) License<br>(N) ClosedInstrument | (N) ∃ hasCapacity SingleUser<br>(N) ∃ hasRight ControlAccessRight<br>(N) ∃ hasRight AttorneyingRight<br>(N) ∃ hasRight PermittingRight |
| Proxy | (N) NamedInstrument | (N) NamedInstrument<br>(N) ExtentInstrument<br>(N) ClosedInstrument | (N) ∃ isExtentOf EmbeddedTitledeed<br>(N) isBoundTo = 0<br>(N) ∃ hasCapacity SingleUser<br>(N) ∃ hasPersistency (SystemPersistent ⊔ SessionNonpersistent)<br>(N) ∃ hasRight ControlAccessRight<br>(N) ∃ hasRight DataAccessRight<br>(N) ∃ hasRight AttorneyingRight<br>(N) ∃ hasRight PermittingRight |

**Table 4-8: License and Proxy class definitions.**

## 4.8.3 Proxy Class

A *proxy* is a written authorization to act in place of another person as an agent or substitute [Far06b]. In ResOwn, a **Proxy** instance is a **Closed**, **Life Limited**, **Extent Instrument** that is indirectly bound to an **Embedded Resource** instance via an **Embedded Titledeed** instance. The **Proxy** instance

extends *beneficiary ownership* from the **Embedded Titledeed** instance to holding **Consumer** instance acting as a *beneficiary proxied owner*. The **Proxy** instance is issued by the **Compound Resource** instance, which acts as the *nonbeneficiary containment owner* of the **Embedded Resource** instance. The **Proxy** class is disjoint from all other **Named Instrument** classes. The OWL-DL definition is given in Table 4-8. A **Proxy** instance may be extended by a **Power Of Attorney** instance or **Permit To Hold** instance.

### 4.8.4 PowerOfAttorney Class

A *power of attorney* is a written document, signed by a person, giving another person the power to act in conducting the signer's business activities in the name of the signer, where an *attorney* is an agent or someone authorized to act for another [Far06b]. In ResOwn, a **Power Of Attorney** instance is a **Closed**, **Life Limited**, **Extent Instrument** that extends *nonbeneficiary ownership* from a **Transferable Titledeed**, a **Serial License** or a **Proxy** instance to a *holding nonbeneficiary owning* **Surrogate Resource Supplier** instance. The **Power Of Attorney** instance is *indirectly* bound to the **Resource** instance via the **Instrument** instance it extends. The **Power Of Attorney** instance is *issued* by a *beneficiary owning* **Consumer** instance that holds the **Transferable Titledeed**, **Serial License**, or **Proxy** instance. The **Power Of Attorney** class is disjoint from all other **Named Instrument** classes. The OWL-DL definition is given in Table 4-9. A **Power Of Attorney** instance may not be extended.

### 4.8.5 PermitToHold Class

A *permit* is a document given by an authorized public official or agency to allow a person or business to perform certain acts [Far06b]. In ResOwn, a **Permit To Hold** instance is a **Closed**, **Life Limited**, **Extent Instrument** that extends *nonbeneficiary ownership* from a **Transferable Titledeed**, a **Serial License**, or a **Proxy** instance to a *holding nonbeneficiary owning* **Cached Resource Supplier** instance. The **Permit To Hold** instance is *indirectly* bound to the **Resource** instance via the **Instrument** instance it extends. The **Permit To Hold** instance is *issued* by a *beneficiary owning* **Consumer** instance that holds of the **Transferable Titledeed**, **Serial License** or **Proxy** instance. The **Permit To Hold** class is disjoint from all other **Named Instrument** classes. The OWL-DL definition is given in Table 4-9. A **Permit To Hold** instance may not be extended.

| Named Instrument Class | Subsumed By Class | | Active Asserted Conditions [N = necessary] [S = necessary & sufficient] |
|---|---|---|---|
| | Asserted | Inferred | |
| PermitToHold | (N) NamedInstrument | (N) NamedInstrument<br>(N) ExtentInstrument<br>(N) ClosedInstrument<br>(N) LifeLimitedInstrument | (N) ∃ isExtentOf (SerialLicenseProxy ⊔ TransferableTitledeed)<br>(N) isBoundTo = 0<br>(N) hasExtent = 0<br>(N) ∃ hasCapacity SingleUser<br>(N) ∃ hasPersistency SessionNonpersistent<br>(N) ∃ hasRight HoldingRight |
| PowerOfAttorney | (N) NamedInstrument | (N) NamedInstrument<br>(N) ExtentInstrument<br>(N) ClosedInstrument<br>(N) LifeLimitedInstrument | (N) ∃ isExtentOf (SerialLicense ⊔ Proxy ⊔ TransferableTitledeed)<br>(N) isBoundTo = 0<br>(N) hasExtent = 0<br>(N) ∃ hasCapacity SingleUser<br>(N) ∃ hasPersistency SessionNonpersistent<br>(N) ∃ hasRight ControlAccessRight |

**Table 4-9: Power Of Attorney and Permit To Hold class definitions.**

# 4.9 Application Resources

The self-contained, *asserted class hierarchy* of the top-level **Resource** class is given in Figure 4-7 and includes the *defined* **Resource** subclasses. The more interesting *inferred class hierarchy*, shown in Figure 4-8, was obtained by classifying ResOwn automatically using RacerPro.

## 4.9.1 Resource Class

In ResOwn, a **Resource** instance represents a physical object that provides a benefit to a beneficiary owner in the evolving resource ownership structure of an operational software system. The OWL-DL for the **Resource** class is given in Table 4-10.

### 4.9.1.1 Resource Capacity

Once acquired, a **Resource** instance may be used concurrently by *multiple users* or by a *single user* [Kir04]. In ResOwn, *closeness* or *openness* of a **Resource** instance is based on the physical and/or logical *capacity* to provide its benefit to one or more *beneficiary owners* either *serially* or *concurrently*, respectively. A **Resource** instance may have either a **Single User Capacity** or a **Multi User Capacity** *value class*. See Section 4.6.1.1 for more details.

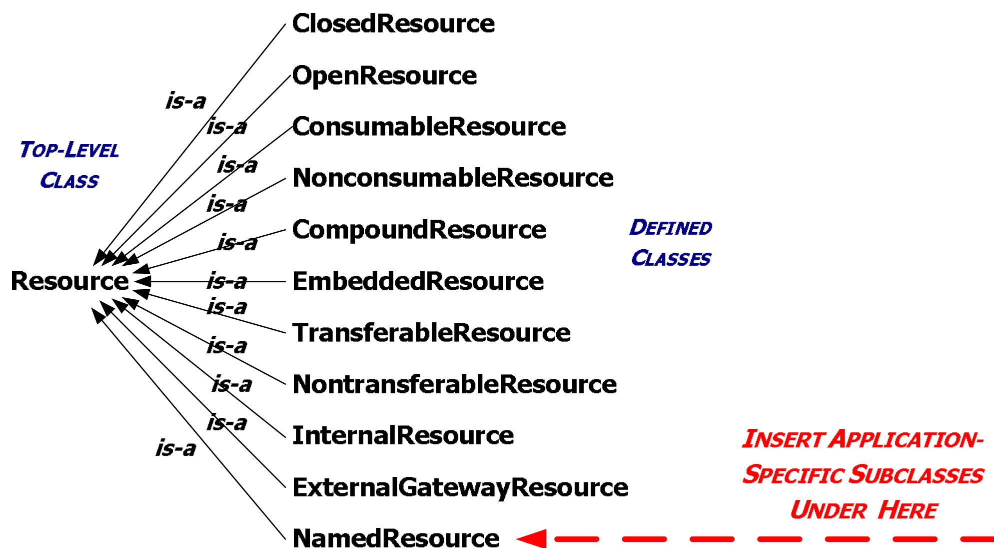



**Figure 4-7: Asserted (defined) Resource class hierarchy.**

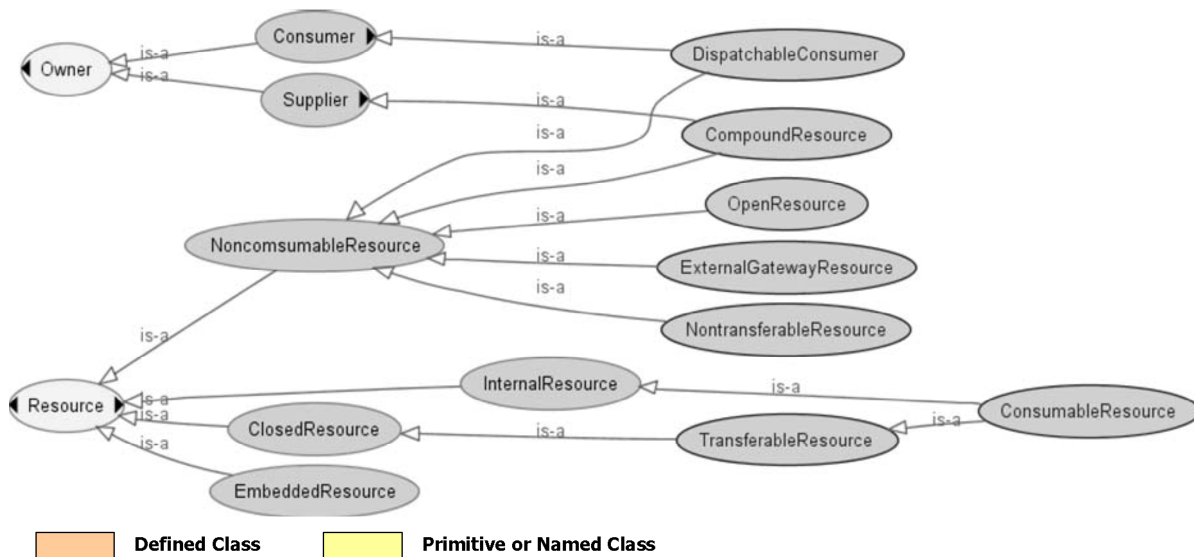



**Figure 4-8: Inferred (defined) Resource class hierarchy.**

### 4.9.1.2 Resource Persistency

In ResOwn, a Resource instance may have a **Persistent**, a **Nonpersistent** or a **Transient** *value class*. See Section 4.6.1.2 for more details. The *consumability* of a **Resource** instance is based on the *persistency* of the **Resource** instance to provide its benefit to a designated beneficiary owner.

| Core Class | Asserted and Inferred Conditions | Property Name and Range<br>[N = necessary]<br>[S = necessary and sufficient] | Disjoint |
|---|---|---|---|
| Resource | (N) ResOwnCore<br>(N) ∀ isBoundTo Instrument | isBoundTo (single Instrument ⊔ Resource)<br>hasContainer (single Resource)<br>isContainerOf (multiple Resource)<br>hasPortal (single SessionAccessPortal)<br>hasCapacity (single CapacityVP)<br>hasPersistency (single PersistencyVP)<br>hasOwner (multiple Owner) | OwnerRole<br>OwnerBase<br>Instrument<br>OwnershipRightVP<br>ResOwnSupport |

**Table 4-10: Top-level Resource class definition.**

## 4.10 Defined Resource Classes

The subsections that follow describe and define the *defined* **Resource** class in ResOwn.

### 4.10.1 TransferableResource Class

A **Transferable Resource** instance is directly bound to a **Transferable Titledeed** instance via an ***isBoundTo*** property restriction. A **Consumer** instance may request *beneficiary ownership* of an unallocated **Transferable Resource** instance either from another **Consumer** instance (i.e., the **Transferable Resource** instance's *current owner*) or from an appropriate **Pooled Resource Supplier** instance (i.e., the **Transferable Resource** instance's *default owner*). The OWL-DL definition is given in Table 4-11. In the PBX, a *space channel* is an example of a **Transferable Resource** instance as it can be *transferred* between *beneficiary owners* (i.e., caller and callee).

### 4.10.2 NontransferableResource Class

A **Nontransferable Resource** instance is directly bound to a **Nontransferable Titledeed** instance via an ***isBoundTo*** property restriction. To obtain *beneficiary ownership* of a **Nontransferable Resource** instance, a **Consumer** instance may *request* a **License** instance from the appropriate **Managed Resource Supplier** instance (i.e., the **Nontransferable Resource** instance's *permanent owner*). The OWL-DL definition is given in Table 4-11. In the PBX, a *dial tone generator card* is an example of a **Nontransferable Resource** instance as it resides *permanently* with the *call progress tone manager*. This means that a *dial tone generator card* can only ever be accessed by a *beneficiary owner* via the *call progress tone manager.*

| Defined Resource Class | Subsumed By Class | | Active Asserted Conditions [N = necessary] [S = necessary & sufficient] |
|---|---|---|---|
| | Asserted | Inferred | |
| EmbeddedResource | (S) Resource | (S) Resource | (S) ∃ isBoundTo EmbeddedTitledeed<br>(S) ∃ hasContainer CompoundResource |
| NontransferableResource | (S) Resource | (S) Resource<br>(N) NonconsumableResource | (S) ∃ isBoundTo NontransferableTitledeed<br>(S) ∃ hasPersistency SystemPersistent |
| TransferableResource | (S) Resource | (S) Resource<br>(N) ClosedResource | (S) ∃ isBoundTo TransferableTitledeed<br>(S) hasContainer = 0<br>(S) ∃ hasCapacity SingleUser |
| OpenResource | (S) Resource | (S) Resource<br>(N) NonconsumableResource | (S) ∃ hasCapacity MultipleUser<br>(S) ∃ hasPersistency SystemPersistent |
| ClosedResource | (S) Resource | (S) Resource | (S) ∃ hasCapacity SingleUser |
| ConsumableResource | (S) Resource | (S) Resource<br>(N) InternalResource<br>(N) DistributableResource | (S) ∃ isBoundTo NonreusableTransferableTitledeed<br>(S) = hasPortal = 0<br>(S) ∃ hasCapacity SingleUser<br>(S) ∃ hasPersistency SystemNonpersistent |
| NonconsumableResource | (S) Resource | (S) Resource | (S) ∃hasPersistency SystemPersistent |
| InternalResource | (S) Resource | (S) Resource | (S) hasPortal = 0 |
| ExternalGatewayResource | (S) Resource | (S) Resource<br>(S) NonconsumableResource | (S) ∃hasPortal SessionAccessPortal<br>(S) ∃hasPersistency SystemPersistent |
| CompoundResource | (S) Resource | (S) Resource<br>(N) NonconsumableResource<br>(N) Supplier | (S) isContainerOf ≥ 1<br>(S) ∃hasOwnerRole ContainmentOwner |

**Table 4-11: Defined Resource class definitions.**

## 4.10.3 Embedded Resource Class

An **Embedded Resource** instance is directly bound to an **Embedded Titledeed** instance via an ***isBoundTo*** property restriction. Further, an **Embedded Resource** instance is bound to a **Compound Resource** instance via a ***hasContainer*** property restriction. To obtain beneficiary ownership of an **Embedded Resource** instance, a **Consumer** instance may *request* a **Proxy** instance from the **Compound Resource** instance (i.e., the **Embedded Resource** instance's *containment owner*). Under normal circumstances, it is assumed that when a **Consumer** instance obtains *beneficiary ownership* of a **Compound Resource** instance, it also obtains *beneficiary ownership* of any *contained* **Embedded Resource** instances as well. The OWL-DL definition is given in Table 4-11. In the PBX, a *ringer relay* is an example of an **Embedded Resource** instance as it is *contained* by a *line card*. This means that a *ringer relay* can only ever be accessed by a *beneficiary owner* via a *line card*.

### 4.10.4 ClosedResource Class

A **Closed Resource** instance is one that has been specified with a **Single User Capacity** *value* instance and thus provides its benefit *serially*. This creates a *cardinality restriction* of "1" between a *beneficiary* **Owner** instance and the **Closed Resource** instance. The OWL-DL definition is given in Table 4-11. In the PBX, a touch tone receiver card is an example of a **Closed Resource** instance as it is capable of supporting only *one* beneficiary owner at a time.

### 4.10.5 OpenResource Class

An **Open Resource** instance is one that has been specified with a **Multi User Capacity** *value* instance and thus provides its benefit *serially*. This creates a *cardinality restriction* of "N" between a set of *beneficiary* **Owner** instances and the **Open Resource** instance. The OWL-DL definition is given in Table 4-11. In the PBX, a dial tone generator card is an example of an **Open Resource** instance as it is capable of supporting *multiple* beneficiary owners *simultaneous*.

### 4.10.6 NonconsumableResource Class

A **Nonconsumable Resource** instance is one that has been specified with a **System Persistent** *value* and therefore is capable of providing its *benefit* for the runtime life of the system. The OWL-DL definition is given in Table 4-11. All PBX resources are **Nonconsumable Resource** instances.

### 4.10.7 ConsumableResource Class

A **Consumable Resource** instance is one that has been specified with a **System Nonpersistent** *value* and therefore is only capable of providing its *benefit* for some limited number of times or length of time. A **Consumable Resource** instance is *directly* bound to a **Nonreusable Transferable Titledeed** and is always specified with a **Single User Capacity** *value class*. The OWL-DL definition is given in Table 4-11.

### 4.10.8 ExternalGatewayResource Class

An **External Gateway Resource** instance is one that has been specified as being *directly* bound to a **Session Access Portal** instance via a ***hasPortal*** property restriction. The **Session Access Portal**

instance allows the **Resource** instance to *communicate* directly with the service environment and is described in more detail in Section 6.1.4. An **External Gateway Resource** instance is always specified with a **System Persistent** *value class*. The OWL-DL definition is given in Table 4-11. In the PBX, a *line card* is an example of an **External Gateway Resource** instance as all external communication with the service environment must pass through *line cards*.

### 4.10.9 InternalResource Classes

An **Internal Resource** instance is one that has been specified with a ***hasPortal*** = **0** *cardinality* restriction. The OWL-DL definition is given in Table 4-11. All PBX resources, *except* the *line card*, are examples of **Internal Resource** instances

### 4.10.10 CompoundResource Class

A **Compound Resource** instance is one that has been specified as *containing* at least one **Embedded Resource** instance via an ***isContainerOf*** property restriction. The *beneficiary current* or *licensed owner* of a **Compound Resource** instance is also the *beneficiary proxied owner* of any *contained* **Embedded Resource** instances.

#### 4.10.10.1 CompoundResource Class as a Supplier Class

Consider the inferred defined Resource class hierarchy, as shown in Figure 4-8 and the OWL-DL definition given in Table 4-11. For the **Compound Resource** *class* row and the *Subsumed By Class: Inferred* column of the table, an individual belonging to the **Compound Resource** class *automatically* belongs to the **Supplier** class. As a direct result of the *invariant* imposed by the *ResOwn Prime Directive* that requires the top-level **Resource** and **Owner** classes not to be *disjoint*, a **Compound Resource** instance is both a *resource* itself and a *supplier of resources*. In the PBX, a *line card* is an example of a **Compound Resource** instance as it *contains* an *idle relay* and a *ringer relay*.

### 4.10.11 DispatchableConsumer (Inferred Resource) Class

Another significant result of the ResOwn Prime Directive is the classification of the **Dispatchable Consumer** class, as shown in the inferred Resource class hierarchy shown in Figure 4-8. The **Dispatchable Consumer** class is detailed in Section 4.14.1.3.

# 4.11 Application Resource Owners

The distributed *asserted class hierarchy* of the top-level **Owner**, **Owner Role** and **Owner Base** classes and their defined subclasses are given in Figure 4-9. The *inferred class hierarchy* for the top-level Owner class is shown in Figure 4-10 and was obtained by classifying ResOwn.

## 4.11.1 Owner, OwnerRole and OwnerBase Classes

In ResOwn, an **Owner** instance represents a *physical* object in the evolving resource ownership structure of an operational software system that may, or may not, receive *benefits*. Each **Owner** instance is capable of playing one or more logical **OwnerRole** instances. The OWL-DL for the **Owner**, **OwnerRole** and **OwnerBase** classes is given in Table 4-12. The *main* and *library* parts of the *distributed* **Owner** class are *disjoint* from each other as well as from all other top-level classes, except the **Resource** class.

| Core Class | Asserted and Inferred Conditions | Property Name and Range [N = necessary] [S = necessary and sufficient] | Disjoint |
|---|---|---|---|
| Owner | (S) ResOwnCore<br>(N) $\exists$ hasBase OwnerBase<br>(N) $\exists$ hasRole OwnerRole<br>(N) $\exists$ hasPersistency SystemPersistent | hasBase (single OwnerBase)<br>hasRole (multiple OwnerRole)<br>hasPersistency (single PersistencyVP)<br>isOwnerOf (multiple Resources) | OwnerRole<br>OwnerBase<br>Instrument<br>OwnershipRightVP<br>ResOwnSupport |
| OwnerBase | (N) ResOwnCore<br>(S) ActiveBase $\sqcup$ PassiveBase | hasServiceThread (single OwnerBase) | OwnerRole<br>Owner<br>Instrument<br>OwnershipRightVP<br>ResOwnSupport |
| OwnerRole | (N) ResOwnCore<br>(S) BeneficiaryOwner $\sqcup$ NonbeneficiaryOwner | isHolderOf (single Instrument)<br>isIssuerOf (single Instrument) | Owner<br>OwnerBase<br>Instrument<br>OwnershipRightVP<br>ResOwnSupport |

**Table 4-12: Top-level Owner, Owner Role and Owner Base class definitions.**

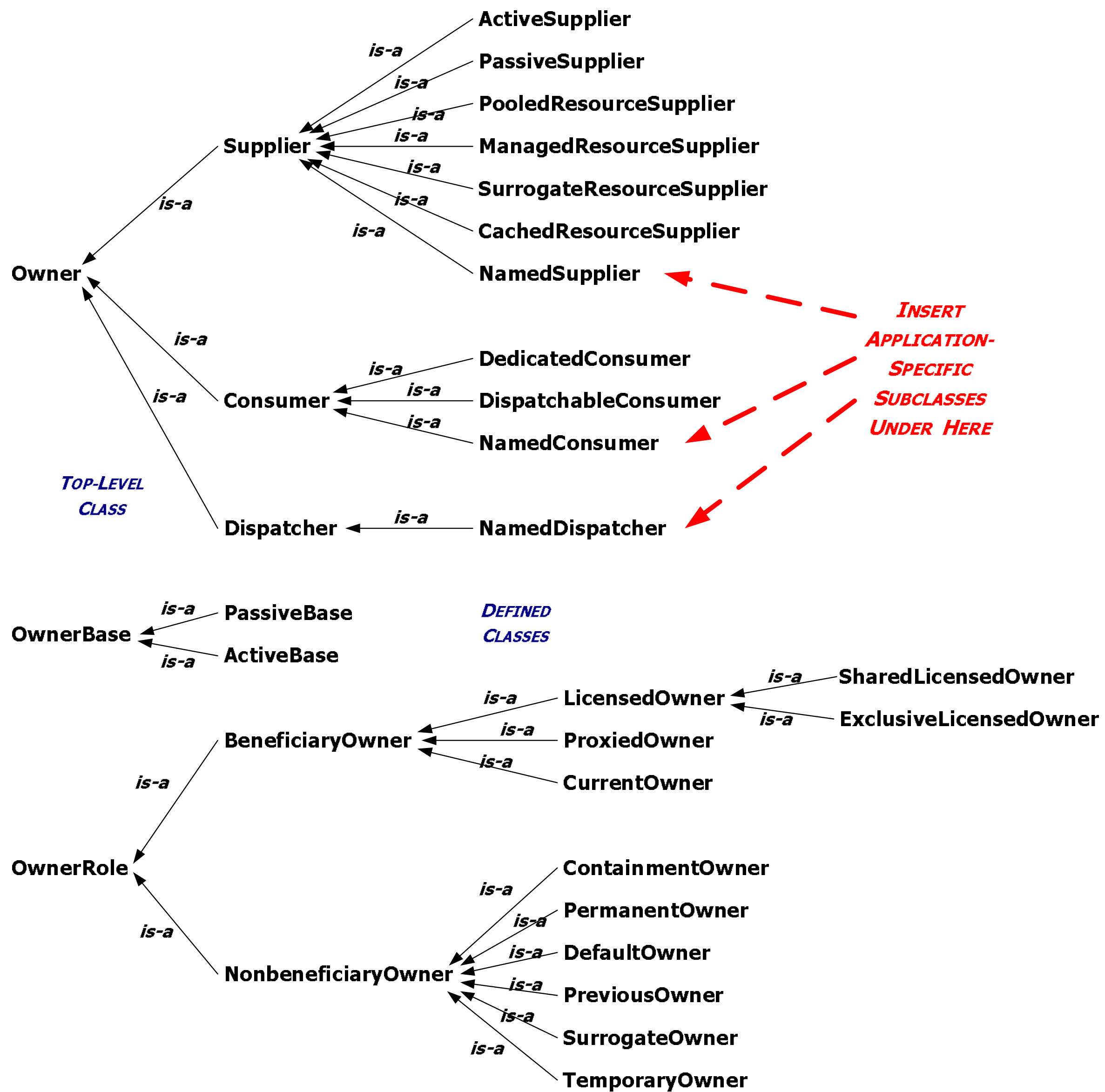



**Figure 4-9: Asserted (defined) Owner, OwnerRole and OwnerBase class hierarchies.**

# 4.12 Defined OwnerBase Classes

The subsections that follow describe and define the *defined* **Owner Base** class in ResOwn. The **Owner Base** class is modeled as the ResOwn property ***hasBase***.

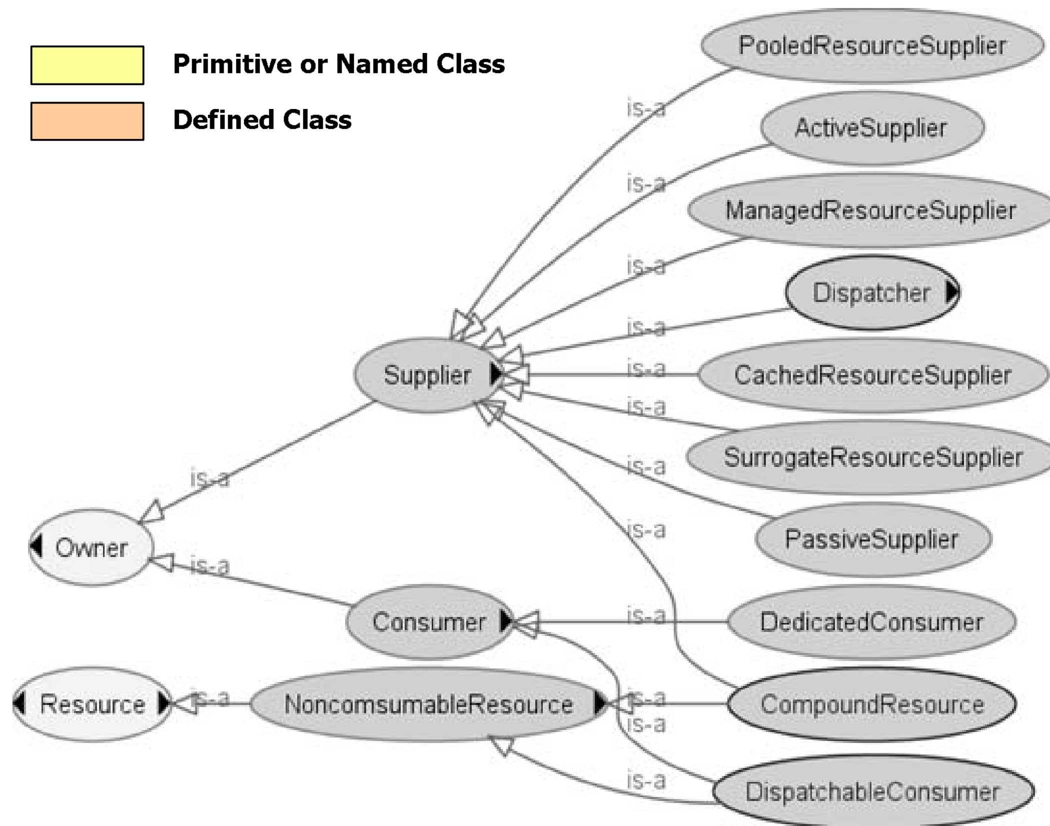



**Figure 4-10: Inferred (defined) Owner class hierarchy.**

### 4.12.1 Passive Base Class

A **Passive Base** instance represents an *unthreaded* object in an operational software system and is assumed to be comprised of a set of *attributes* and a *functional interface* that supports a synchronous *method invocation* protocol. The OWL-DL definition is given in Table 4-13.

### 4.12.2 Active Base Class

An **Active Base** instance represents a *threaded* object in an operational software system and is assumed to be comprised of a set of *attributes*, an application-level *thread* of execution and a *functional interface* that supports an *asynchronous message passing* protocol and/or a *synchronous method invocation* protocol. The OWL-DL definition is given in Table 4-13.

| Defined OwnerBase Class | Subsumed By Class | | Active Asserted Conditions [N = necessary] [S = necessary & sufficient] | Disjoint |
|---|---|---|---|---|
| | Asserted | Inferred | | |
| ActiveOwnerBase | (S) OwnerBase | (S) OwnerBase | (S) ∃ hasServiceThread ActiveBase | PassiveBase |
| PassiveOwnerBase | (S) OwnerBase | (S) OwnerBase | (S) hasServiceThread = 0 | ActiveBase |

**Table 4-13: Defined OwnerBase class definitions.**

# 4.13 Defined OwnerRole Classes

The subsections that follow describe and define the defined **Owner Role** classes in ResOwn. The **Owner Role** class is modeled as the ResOwn property hasRole.

## 4.13.1 BeneficiaryOwner Class

The **Beneficiary Owner** class covers a *group* of *beneficiary* **Owner Role** subclasses. The OWL-DL definition is given in Table 4-15.

### 4.13.1.1 CurrentOwner Subclass

A **Current Owner** instance acts as a *beneficiary* and is permitted to:

- *Hold* a **Transferable Titledeed** instance.
- Receive *benefits* from an owned **Transferable Resource** instance.
- *Consume* a **Consumable Resource** instance.
- *Transfer* a **Transferable Titledeed** instance to a new current owner.
- *Issue* a **Power Of Attorney** instance against a **Transferable Titledeed** instance to a *nonbeneficiary surrogate owner*.
- *Issue* a **Permit To Hold** instance against a **Transferable Titledeed** instance to a *nonbeneficiary temporary owner*.

The OWL-DL definition is given in Table 4-15.

### 4.13.2 LicensedOwner Subclass

The **Licensed Owner** class covers the Exclusive and Shared License subclasses. The effects of *logical licensed capacity* versus *physical resource capacity* are shown in Table 4-14. The OWL-DL definition is given in Table 4-15.

#### 4.13.2.1 ExclusiveLicensedOwner Subclass

An **Exclusive Licensed Owner** instance acts as a *beneficiary* and is permitted to:

- *Hold* a **Serial License** instance.
- Receive *benefits* from a **Closed Nontransferable Resource** instance.
- *Issue* a **Power Of Attorney** instance against a **Serial License** instance to a *nonbeneficiary surrogate owner*.
- *Issue* a **Permit To Hold** instance against a **Serial License** instance to a *nonbeneficiary temporary owner*.

The OWL-DL definition is given in Table 4-15.

#### 4.13.2.2 SharedLicensedOwner Subclass

A **Shared Licensed Owner** instance acts as a *beneficiary* and is permitted to:

- *Hold* a **Concurrent License** instance.
- Receive *benefits* from an **Open Nontransferable Resource** instance.

The OWL-DL definition is given in Table 4-15.

| **Physical Capacity Versus Logical Capacity** | **Open Resource** | **Closed Resource** |
|---|---|---|
| **Serial License (Closed Titledeed)** | Access logically restricted to a single (Exclusive) Licensed Owner | Access physically restricted to a single (Exclusive) Licensed Owner |
| **Concurrent License (Open Titledeed)** | Access unrestricted to multiple (Shared) Licensed Owners | Access physically restricted to a single (Exclusive) Licensed Owner |

**Table 4-14: Effects of physical versus logical capacity.**

| Defined OwnerRole Class | Subsumed By Class | Active Asserted Conditions [N = necessary] [S = necessary & sufficient] |
|---|---|---|
| | Asserted & Inferred | |
| BeneficiaryOwner (covering class) | (N) OwnerRole | (S) LicensedOwner ⊔ CurrentOwner ⊔ ProxyOwner |
| CurrentOwner | (S) BeneficiaryOwner | (S) ∃ isHolderOf TransferableTitledeed<br>(S) ∀ isIssuerOf (TransferableTitledeed ⊔ PermitToHold ⊔ PowerOfAttorney) |
| ProxyOwner | (S) BeneficiaryOwner | (S) ∃ isHolderOf Proxy<br>(S) ∀ isIssuerOf (PermitToHold ⊔ PowerOfAttorney) |
| LicensedOwner (covering class) | (S) BeneficiaryOwner | SharedLicensedOwner ⊔ ExclusiveLicensedOwner |
| SharedLicensedOwner | (S) LicensedOwner | (S) ∃ isHolderOf ConcurrentLicense<br>(S) isIssuerOf = 0 |
| ExclusiveLicensedOwner | (S) LicensedOwner | (S) ∃ isHolderOf SerialLicense<br>(S) ∀ isIssuerOf (PermitToHold ⊔ PowerOfAttorney) |

**Table 4-15: Defined Beneficiary Owner Role subclass definitions.**

### 4.13.2.3 ProxiedOwner Subclass

A **Proxied Owner** instance acts as a *beneficiary* and is permitted to:

- *Hold* a **Proxy** instance.
- Receive *benefits* from an **Embedded Resource** instance.
- *Issue* a **Power Of Attorney** instance against a **Proxy** instance to a *nonbeneficiary surrogate owner*.
- *Issue* a **Permit To Hold** instance against a **Proxy** instance to a *nonbeneficiary temporary owner*.

The OWL-DL definition is given in Table 4-15.

## 4.13.3 Nonbeneficiary Owner Class

The **Nonbeneficiary Owner** class covers a *group* of *nonbeneficiary* **Owner Role** subclasses. The OWL-DL definition is given in Table 4-16.

### 4.13.3.1 ContainmentOwner Subclass

A **Containment Owner** instance acts as a *nonbeneficiary* and is permitted to:

- *Hold* an **Embedded Titledeed** instance.

- *Issue* a **Proxy** instance against the **Embedded Titledeed** instance to a *beneficiary proxied owner*.

The OWL-DL definition is given in Table 4-16.

#### 4.13.3.2 PermanentOwner Subclass

A **Permanent Owner** instance acts as a *nonbeneficiary* and is permitted to:

- *Hold* a **Nontransferable Titledeed** instance.
- *Issue* a (**Serial** or **Concurrent**) **License** instance against the (**Closed** or **Open**) **Nontransferable Titledeed** instance to a *beneficiary* (*exclusive* or *shared*) *licensed owner*.

The OWL-DL definition is given in Table 4-16.

#### 4.13.3.3 DefaultOwner Subclass

A **Default Owner** instance acts as a *nonbeneficiary* and is permitted to:

- *Hold* a **Transferable Titledeed** instance.
- *Issue* the **Transferable Titledeed** instance itself to a *beneficiary current licensed owner*.

The OWL-DL definition is given in Table 4-16.

| Defined OwnerRole Class | Subsumed By Class<br>Asserted & Inferred | Active Asserted Conditions<br>[N = necessary]<br>[S = necessary & sufficient] |
|---|---|---|
| NonbeneficiaryOwner (covering class) | (N) OwnerRole | (S) ContainmentOwner ⊔ PermanentOwner ⊔ DefaultOwner ⊔ SurrogateOwner ⊔ TemporaryOwner ⊔ PreviousOwner |
| PreviousOwner | (S) NonbeneficiaryOwner | (S) isHolderOf = 0<br>(S) ∀ isIssuerOf TransferableTitledeed |
| ContainmentOwner | (S) NonbeneficiaryOwner | (S) ∃ isHolderOf EmbeddedTitledeed<br>(S) ∀ isIssuerOf Proxy |
| PermanentOwner | (S) NonbeneficiaryOwner | (S) ∃ isHolderOf NontransferableTitledeed<br>(S) ∀ isIssuerOf License |
| DefaultOwner | (S) NonbeneficiaryOwner | (S) ∀ isHolderOf TransferableTitledeed<br>(S) ∀ isIssuerOf TransferableTitledeed |
| SurrogateOwner | (S) NonbeneficiaryOwner | (S) ∀ isHolderOf PowerOfAttorney<br>(S) isIssuerOf = 0 |
| TemporaryOwner | (S) NonbeneficiaryOwner | (S) ∀ isHolderOf PermitToHold<br>(S) isIssuerOf = 0 |

**Table 4-16: Nonbeneficiary Owner Role subclass definitions.**

#### 4.13.3.4 SurrogateOwner Subclass

A **Surrogate Owner** instance acts as a *nonbeneficiary* and is permitted to *hold* a **Power Of Attorney** instance *issued* against an associated **Transferable Titledeed**, **Exclusive License**, or **Proxy** instance. A **Surrogate Owner** instance is afforded only a **Control Access Ownership Right** instance via the **Power Of Attorney** instance. The OWL-DL definition is given in Table 4-16.

#### 4.13.3.5 TemporaryOwner Subclass

A **Temporary Owner** instance acts as a *nonbeneficiary* and is permitted to *hold* a **Permit To Hold** instance *issued* against an associated **Transferable Titledeed**, **Exclusive License**, or **Proxy** instance. A **Temporary Owner** instance is afforded only a **Holding Ownership Right** instance via the **Permit To Hold** instance. The OWL-DL definition is given in Table 4-16.

#### 4.13.3.6 PreviousOwner Subclass

A **Previous Owner** instance is a special *nonbeneficiary* **Owner Role** subclass that was created to deal with *traceability* of **Transferable Resource** instances and is permitted to issue a **Transferable Titledeed** instance it holds as a **Current Owner** instance to a *beneficiary current licensed owner*. The OWL-DL definition is given in Table 4-16.

##### 4.13.3.6.1 The Virtual Previous Owner Stack

Recall that a **Current Owner** instance may *transfer* a **Transferable Resource** instance to another *requesting* **Current Owner** instance without first returning the **Resource** instance back to the originating **Pooled Resource Supplier** instance. In this scenario, the **Previous Owner** instance is a *command* or *log* of the last (i.e., previous) *beneficiary owner* that held the **Transferable Resource** instance. Note that a **Previous Owner** instance is a non*beneficiary owner* and no longer receives a benefit from the **Transferable Resource** instance. Every **Transferable Resource** instance not residing with its **Default Owner** instance has a single **Current Owner** plus a *stack* of zero of more **Previous Owners** instances. This virtual *"previous owner stack"* dynamically *grows* (or *shrinks*) each time the associated **Transferable Titledeed** instance is *acquired* by (or *released* to) a **Current** (or **Previous**) **Owner** instance. The *stack* is reset whenever the **Transferable Titledeed** instance is returned to its **Default Owner** instance.

# 4.14 Defined Owner Subclasses

The subsections that follow describe and define the defined **Owner** subclass in ResOwn. These **Owner** subclasses may be thought of as **Owner** types; that is, while an **Owner** instance's **Owner** Role may dynamically change according to the current *structural resource ownership context* an **Owner** instance finds itself in at runtime, the **Owner** type is static. The OWL-DL for the **Consumer**, **Supplier**, and **Dispatcher Owner** subclasses is given in Table 4-17.

| Defined Owner Class | Subsumed By Class | | Property Name & Range | Active Asserted Conditions [N = necessary] [S = necessary & sufficient] |
|---|---|---|---|---|
| | Asserted | Inferred | | |
| Consumer | (S) Owner | (S) Owner | hasDispatcher (single Dispatcher) | (S) ∃ hasBase ActiveOwnerBase<br>(S) ∃ hasRole BeneficiaryOwnerRole |
| Supplier | (S) Owner | (S) Owner | isDispatcherOf (multiple Consumer) | (S) ∃ hasRole NonbeneficiaryOwnerRole |
| Dispatcher | (S) Owner | (S) Owner<br>(N) Supplier | | (S) ∃ hasRole BeneficiaryOwnerRole<br>(S) ∃ hasRole NonbeneficiaryOwnerRole |

**Table 4-17: Defined Owner subclass definitions.**

## 4.14.1 ConsumerOwner Subclass

A **Consumer** instance is *bound* to an **Active Base** instance via a ***hasBase*** property restriction and is capable of acting as a *nonbeneficiary owner* or as a *beneficiary owner* that receives *benefits*.

### 4.14.1.1 Dedicated Consumer Subclass

A **Dedicated Consumer** instance is *statically assigned* (or dedicated) to a unique **Session Access Portal** instance for the runtime life of the operational software system. The assignment remains in place even when the **Session Access Portal** instance is **idle**. The OWL-DL for the **Dedicated Consumer** subclass is given in Table 4-18. In the PBX, a *phone handle* that is *permanently* assigned to a *line card* is an example of a **Dedicated Consumer** instance.

| Defined Consumer Class | Subsumed By Class | | Owl: Equivalent Properties | Active Asserted Conditions [N = necessary] [S = necessary & sufficient] |
|---|---|---|---|---|
| | Asserted | Inferred | | |
| DedicatedConsumer | (S) Consumer | (S) Consumer | - | (S) hasDispatcher = 0 |
| Dispatchable Consumer | (S) Consumer | (S) Consumer<br>(N) NonconsumableResource | Resource:hasOwner ≡ Consumer:hasDispatcher | (S) hasDispatcher = 1<br>(S) ∃ hasPersistency SystemPersistent |

**Table 4-18: Consumer subclass definitions.**

#### 4.14.1.2 DispatchableConsumer Subclass

A **Dispatchable Consumer** instance may be *dynamically assigned* (i.e., dispatched) from a pool **Dispatchable Consumer** instances and *temporarily* assigned to a unique **Session Access Portal** instance for the duration of a *current session* (i.e., in-progress service request). The **Dispatchable Consumer** instance is returned to the pool once the **Session Access Portal** instance becomes *idle* again. The OWL-DL for the **Dispatchable Consumer** subclass is given in Table 4-18. In the PBX, a *phone handle* that is *dynamically assigned* to a *line card* when the associated *phone* goes *off hook* and then *unassigned* when the same *phone* goes *on hook* is an example of a **Dedicated Consumer** instance.

#### 4.14.1.3 Classifying a DispatchableConsumer as a Nonconsumable Resource

A **Dispatchable Consumer** instance is also are *classified* under ResOwn as a **Nonconsumable Resource** instance, as shown in Figure 4-8, Figure 4-10 and Table 4-18. For the **Dispatchable Consumer** *subclass* row and the *Subsumed By Class: Inferred* column of the table, an individual belonging to the **Dispatchable Consumer** class *automatically* belongs to the **Nonconsumable Resource** class. As stated previously, this dual inheritance is a direct result of the *ResOwn Prime Directive* that requires the **Resource** and **Owner** classes not to be *disjoint* from each other.

Now consider the ***hasOwner*** and ***hasDispatcher*** properties specified in Table 4-18. Notice *owl:equivalentProperty construct* column in the **Dispatchable Consumer** class row, which contains:

$$\textbf{Resource}:\boldsymbol{hasOwner} \equiv \textbf{Consumer}:\boldsymbol{hasDispatcher}$$

The *owl:equivalentProperty construct* states that the two specified properties have the same values (i.e., the same property extension), but may have different intensional meaning (i.e., denote different concepts). When automatically classifying the ResOwn class hierarchy, RacerPro treats ***hasOwner*** and ***hasDispatcher*** as *equivalent*. Since a **Dispatchable Consumer** instance belongs to the **Owner** class, it must therefore also belong to the **Resource** class. Further, since a **Dispatchable Consumer** instance is specified with a **System Persistent** *value* instance, the **Dispatchable Consumer** instance is classified correctly as belonging to the more specialized **Nonconsumable Resource** class.

## 4.14.2 Supplier Subclass

A **Supplier** instance is *bound* to a either an **Active** or **Passive Base** instance via a ***hasBase*** property restriction and is only capable of acting as a *nonbeneficiary owner*.

### 4.14.2.1 ActiveSupplier Subclass

An **Active Supplier** instance is one that has been specified with an **Active Base** *value* instance via a ***hasBase*** property restriction. The OWL-DL definition is given in Table 4-19. In the PBX, a *line card scanner* is an example of an **Active Supplier** instance since it has a service thread.

### 4.14.2.2 PassiveSupplier Subclass

A **Passive Supplier** instance is one that has been specified with a **Passive Base** instance via a ***hasBase*** property restriction. The OWL-DL definition is given in Table 4-19. In the PBX, a *touch tone receiver manager* is an example of a **Passive Supplier** instance since it is unthreaded.

| Defined Supplier Class | Subsumed By Class | | Active Asserted Conditions [N = necessary] [S = necessary & sufficient] |
|---|---|---|---|
| | Asserted | Inferred | |
| ActiveSupplier | (S) Supplier | (S) Supplier | (S) ∃ hasBase ActiveOwnerBase |
| PassiveSupplier | (S) Supplier | (S) Supplier | (S) ∃ hasBase PassiveOwnerBase |
| CachedResourceSupplier | (S) Supplier | (S) Supplier | (S) ∃ hasRole TemporaryOwnerRole |
| ManagedResourceSupplier | (S) Supplier | (S) Supplier | (S) ∃ hasRole PermanentOwnerRole |
| PooledResourceSupplier | (S) Supplier | (S) Supplier | (S) ∃ hasRole DefaultOwnerRole |
| SurrogateResourceSupplier | (S) Supplier | (S) Supplier | (S) ( hasRole SurrogateOwnerRole |
| Dispatcher | (S) Owner | (S) Owner<br>(N) Supplier | (S) ∃ hasRole BeneficiaryOwnerRole<br>(S) ∃ hasRole NonbeneficiaryOwnerRole |
| CompoundResource | (S) Resource | (S) Resource<br>(N) Supplier<br>(N) NonconsumableResource | (S) isContainerOf ≥ 1<br>(S) ∃ hasOwnerType ContainmentOwnerType |

**Table 4-19: Supplier and Dispatcher subclass definitions.**

### 4.14.2.3 CachedResourceSupplier Subclass

A **Cached Resource Supplier** instance is one that has been specified with a **Temporary Owner Role** instance via a ***hasRole*** property restriction. The OWL-DL definition is given in Table 4-19. The **Cached Resource Supplier** subclass is included in ResOwn specifically to support the *Architectural Resource Caching Pattern* introduced and described in [Kir04]. Consider a variation of the PBX that

stores all local *phone extension information* (i.e., phone number to line card mappings) in a *database*. In this case, it might be possible for a *phone handler* to *cache* phone extension information that has recently been looked up in a *previous session* using a **Cached Resource Supplier** instance.

#### 4.14.2.4 ManagedResourceSupplier Subclass

A **Managed Resource Supplier** instance is one that has been specified with a **Permanent Owner Role** instance via a ***hasRole*** property restriction. The OWL-DL definition is given in Table 4-19. In the PBX, a *call progress tone manager* is an example of a **Managed Resource Supplier** instance.

#### 4.14.2.5 PooledResourceSupplier Subclass

A **Pooled Resource Supplier** instance is one that has been specified with a **Default Owner Role** instance via a ***hasRole*** property restriction. The OWL-DL definition is given in Table 4-19. In the PBX, a *space channel manager* is an example of a **Pooled Resource Supplier** instance.

#### 4.14.2.6 SurrogateResourceSupplier Subclass

A **Surrogate Resource Supplier** instance is one that has been specified with a **Surrogate Owner Role** instance via a ***hasRole*** property restriction. The OWL-DL definition is given in Table 4-19. In the PBX, a space channel manager is an example of a **Pooled Resource Supplier** instance.

#### 4.14.2.7 CompoundResource (Inferred Supplier) Subclass

A **Compound Resource** instance is one that has been specified with a **Containment Owner Role** instance via a ***hasRole*** property restriction. The OWL-DL definition is given in Table 4-19. See Section 4.10.10 and Table 4-11 for more details on the Compound Resource class.

### 4.14.3 Dispatcher Subclass

A **Dispatcher** instance is one that has been specified with both a **Beneficiary** and **Nonbeneficiary Owner Role** instance via two ***hasRole*** property restrictions. This makes a **Dispatcher** instance capable of receiving *benefits* and supplying **Resources** instances. A **Dispatcher** instance is capable of dispatching a **Dispatchable Consumer** instance (which is classified as both a **Resource** and a **Consumer** instance), from a pool of **Dispatchable Consumer** instances, to temporarily service a **Session Access Portal** instance. The **Dispatchable Consumer** class definition specifies that a **Consumer** instance is *owned* by a **Dispatcher** instance via the ***hasDispatcher*** property restriction. The OWL-DL definition is given in Table 4-19. In the PBX, a *call manager* that *monitors idle line*

*cards* and *dispatches* a *phone handler* when an *off hook* is detected is an example of a **Dispatcher** instance.

# 4.15 OwnershipRight Value Classes

Recall that in ResOwn, every *proof of ownership* **Instrument** class is specified with a predefined set of **Ownership Right** *value* instances. The **Ownership Right VP** *value partition* is modeled as the property ***hasRight***, as shown in Table 4-3. The class hierarchy of the **Ownership Value VP** value *partition* is given in Figure 4-11.

## 4.15.1 AccessRight

The **Access Right** *value* instance permits either **Data** or **Control Access**.

### 4.15.1.1 DataAccess

The **Data Access Right** *value* instance permits a *beneficiary owner* to receive benefits.

### 4.15.1.2 ControlAccess

The **Control Access Right** *value* instance only permits an *owner* to control a resource.

## 4.15.2 Consumption Right

The **Consumption Right** *value* instance permits a *beneficiary owner* to consume a resource.

## 4.15.3 ExchangeRight

The **Exchange Right** *value* instance permits the swapping of a *proof of ownership* **Instrument** instance between an *issuer* and a *holder*.

### 4.15.3.1 HoldingRight

The **Holding Right** *value* instance permits a *nonbeneficiary owner* to hold an **Instrument** instance on behalf of a *beneficiary owner*.

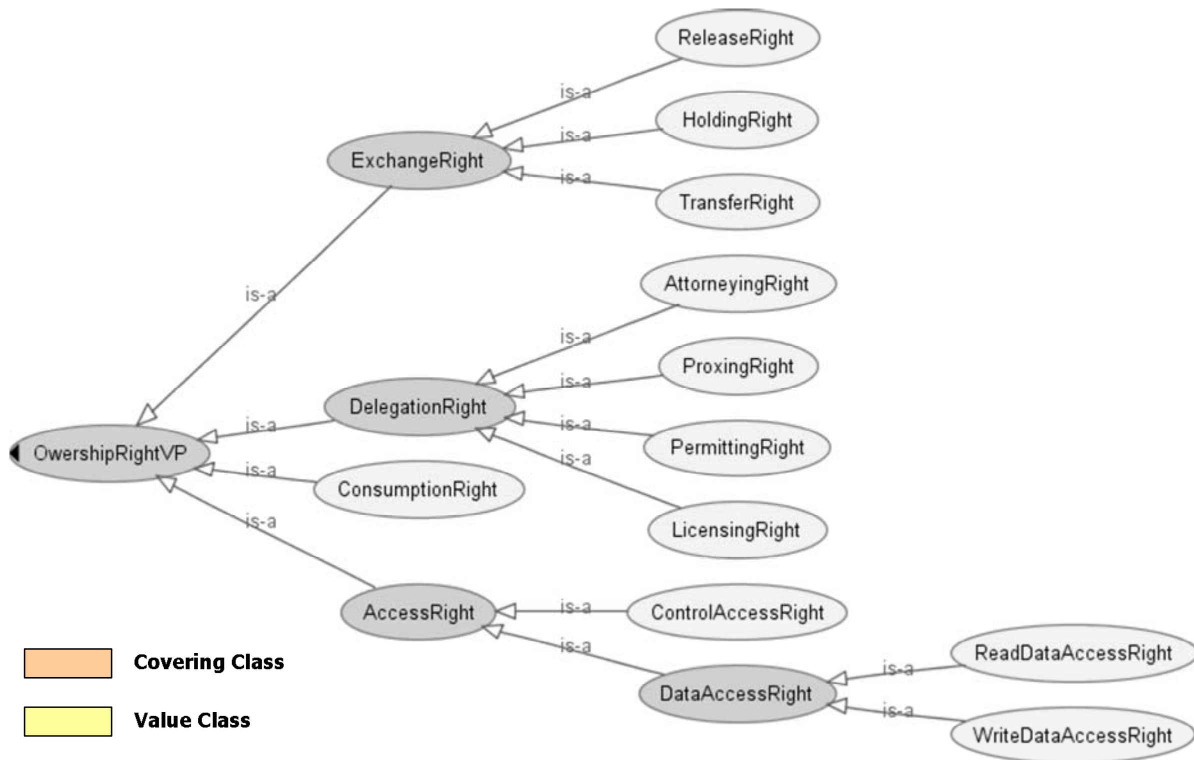



**Figure 4-11: OwnershipRight VP value partition class hierarchy.**

#### 4.15.3.2 TransferRight

The **Transfer Right** *value* instance permits a *beneficiary owner* to transfer an **Instrument** instance to another *beneficiary owner*.

#### 4.15.3.3 ReleaseRight

The **Release Right** *value* instance permits a *beneficiary owner* to return an **Instrument** instance to the original issuing *nonbeneficiary owner*.

### 4.15.4 DelegationRight

The **Delegation Right** *value* instance permits an *owner* to delegate (i.e., issue) some *subset of ownership rights* to another *owner* via an **Extent Instrument** instance.

#### 4.15.4.1 ProxyingRight

The **Proxying Right** *value* instance permits a *nonbeneficiary owner* (i.e., **Compound Resource**

instance) to *issue* a **Proxy** instance to a *beneficiary owner*.

### 4.15.4.2 PermittingRight

The **Permitting Right** *value* instance permits a *beneficiary owner* (i.e., **Consumer** instance) to *issue* a **Permit To Hold** instance to a *nonbeneficiary owner* (i.e., **Cached Resource Supplier** instance).

### 4.15.4.3 LicensingRight

The **Licensing Right** *value* instance permits a *nonbeneficiary owner* (i.e., **Managed Resource Supplier** instance) to *issue* a **License** instance to a *beneficiary owner* (i.e., **Consumer** instance).

### 4.15.4.4 AttorneyingRight

The **Attorneying Right** *value* instance permits a *beneficiary owner* (i.e., **Consumer** instance) to *issue* a **Power Of Attorney** instance to a *nonbeneficiary owner* (i.e., **Surrogate Resource Supplier** instance).

# Chapter 5
# ResOwn Instance Example and Ownership Scenarios

*"Managing resources is hard; managing them efficiently is even harder."*
- M. Kircher and P. Jain, 2004

## 5.1 Introduction

This chapter presents the reader with a detailed, three-stage example intended to illustrate the main modeling concepts of the ResOwn ontology presented in Chapter 4, and to provide an example of the practical use of ResOwn in the software engineering domain.

- **The first stage** of the example shows the reader how ResOwn can be *specialized* with *application-specific* ontological classes derived directly using object classes from the example PBX. The resulting ontology is called a *specialized ResOwn instance*.
- **The second stage** provides the reader with a visual illustration of how the automatically generated *inferred class hierarchy* of the *specialized ResOwn instance* can be used to determine the *multiple inheritance characteristics* of the ontological classes originating from the PBX.
- **The third stage** provides the reader with a number of visual, *role-based resource ownership scenarios* (i.e., resource ownership patterns) that show how the PBX's application resource ownership structure evolves in an ordered or state-dependent manner in synchronization with the operational PBX. The software sensor plan is used to guide the manual PBX instrumentation process, as discussed in Chapter 7.

## 5.2 An Asserted ResOwn Instance Example

Recall from Chapter 4 that a ResOwn instance is a *specialized* version of the *baseline* ResOwn ontology that has been *extended* with application-specific ontological classes. This section presents a ResOwn *specialization methodology* along with a specific example using the actual *object classes* from the example PBX (Section 2.10).

### 5.2.1 Assumptions

The following assumptions are applicable for the *ResOwn specialization methodology* for the PBX:

- The CEFSM-based software requirements specification and software design specification are available. In this thesis, the PBX is specified using SDL.
- The object-oriented source code is available. In this thesis, the PBX is implemented in Java.
- Because the design is assumed to be a refinement of the requirements, and the source code a refinement of the design (Section 6.1.6), externally observable states in the software requirements map to equivalent states in the software design and source code.
- Each CEFSM specified in the design specification maps to an equivalent object class (of the same name) in the source code.

### 5.2.2 ResOwn Specialization Methodology

The *ResOwn specialization methodology* for constructing a *specialized ResOwn instance* is comprised of these steps:

1. From the source code, construct an object class table such that each class in the table corresponds to an equivalent CEFSM in the software design specification. The list of identified PBX classes is given in Table 2.2 of Chapter 2.
2. Place each CEFSM identified from Step 1 into an appropriate level of the layered interactive service architecture like the one for the PBX shown in Figure 2.28 of Chapter 2. If a particular CEFSM could potentially reside in more than one level, place the CEFSM in the primary or dominant level. When the resulting specialized ResOwn instance is classified, the reasoner will use the ontological class definition to automatically deal with cases of multiple inheritance.

3. Assign each CEFSM in the object class table created in Step 1 to a top-level core ResOwn class (i.e., **Resource**, **Consumer** or **Resource**) as shown in the *ResOwn Class column* for the PBX in Table 2.2 of Chapter 2. This assignment determines which **Named** class in the baseline ResOwn ontology that the new application-specific ontological class will be inserted under, as follows:

   i. If the CEFSM resides in the *service delivery layer*, then an ontological class with the same identifier name as the CEFSM is inserted into the *baseline ResOwn ontology* under the **Named Consumer** or **Named Dispatcher** class. For the PBX, the CEFSMs *wait for call service* and *phone handler* are inserted under the **Named Dispatcher** class and the **Named Consumer** class, respectively, as shown in Figure 5-1.

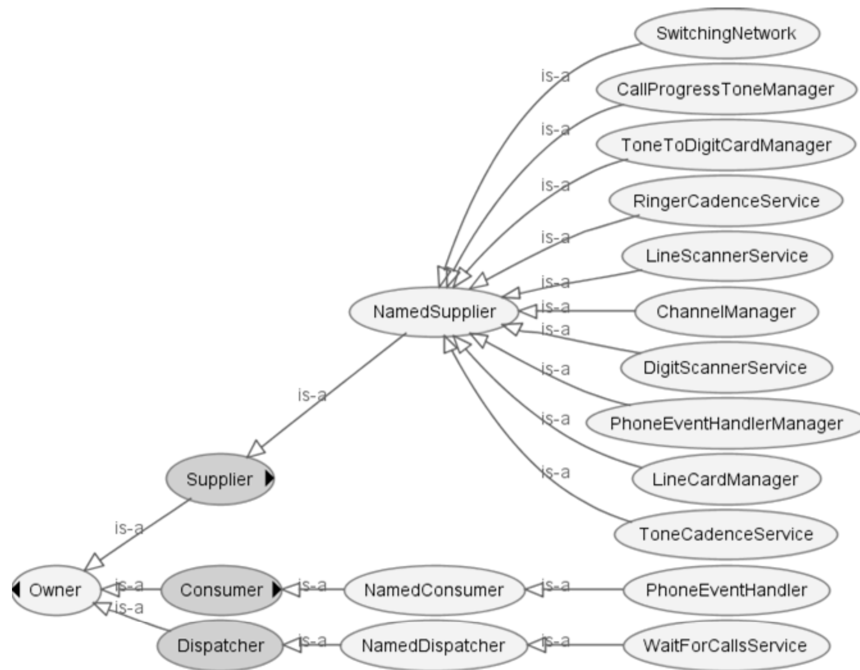



**Figure 5-1: Named Owner classes from the PBX.**

   ii. If the CEFSM resides in the *community service layer*, then an ontological class with the same identifier name as the CEFSM is inserted into the *baseline ResOwn ontology* under the **Named Supplier** class. For the PBX, the CEFSMs *call progress tone manager* and *line card scanner* are inserted under the **Named Supplier** class, as shown in Figure 5-1.

iii. If the CEFSM resides in the *hardware abstraction layer*, then an ontological class with the same identifier name as the CEFSM is inserted into the *baseline ResOwn ontology* under the **Named Resource** class. For the PBX, the CEFSMs *line card* and *idle relay device* are inserted under the **Named Resource** class, as shown in Figure 5-2.

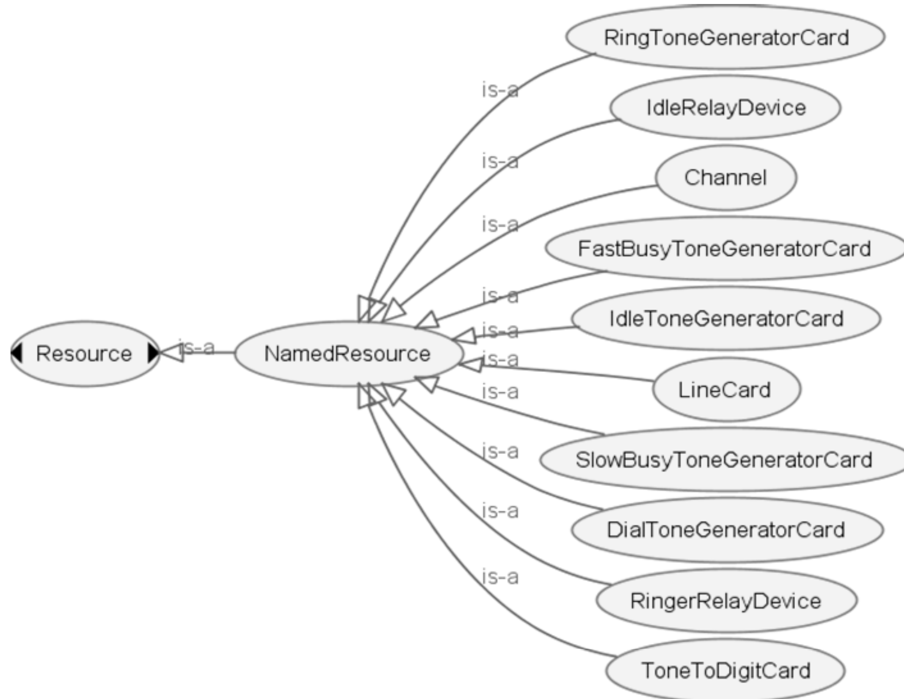



**Figure 5-2: Named Resource classes from the PBX.**

iv. For each **Named** class inserted into the *baseline ResOwn ontology*, any *relevant* or *required property restrictions* must be instantiated according to the class *properties* or *characteristics* of the actual PBX *class* for which the ontological **Named** class was created. An example set of properties that would need to be considered for a **Named Resource** class is shown in the Protégé-OWL screenshot of Figure 5-3. The **Named Resource** class properties are inherited from the *top-level* **Resource** class, as shown in the class hierarchy in Figure 5-2.

After the necessary class property restrictions are instantiated, the *reasoner* is run to check consistency. The resulting **Named Owner** and **Resource** classes for the PBX are shown in Table 5-1 and Table 5-2, respectively. The **Subsumed By Class: Inferred** column is explained in Section 5.3

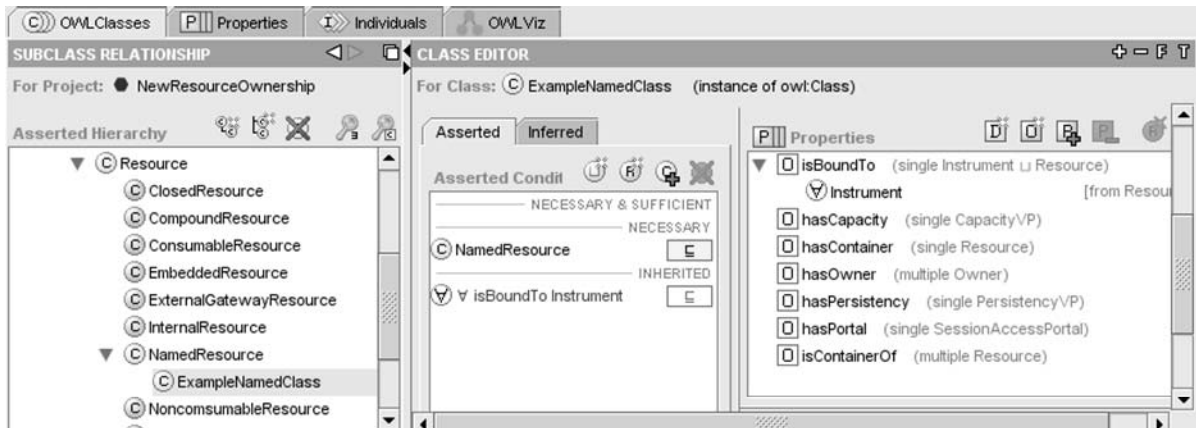



**Figure 5-3: Inherited Resource properties in an example of a Named Resource class.**

| Named Owner Class | Subsumed By Class | | Active Asserted Conditions |
|---|---|---|---|
| | Asserted | Inferred | |
| WaitForCallsService | NamedDispatcher | NamedDispatcher | ∃ isDispatcherOf PhoneEventHandler |
| PhoneHandler | NamedConsumer | NamedConsumer<br>DispatchableConsumer | ∃ hasDispatcher WaitForCallsService |
| LineScannerService<br>DigitScannerService<br>RingerCadenceService<br>ToneCadenceService | NamedSupplier | NamedSupplier<br>ActiveSupplier<br>SurrogateResourceSupplier | ∃ hasOwnerBase ActiveOwnerBase<br>∃ hasOwnerRole SurrogateOwnerRole |
| CallProgressToneManager | NamedSupplier | NamedSupplier<br>PassiveSupplier<br>ManagedResourceSupplier | ∃ hasOwnerBase PassiveOwnerBase<br>∃ hasOwnerRole PermanentOwnerRole |
| SwitchingNetwork | NamedSupplier | NamedSupplier<br>PassiveSupplier<br>SurrogateResourceSupplier | ∃ hasOwnerBase PassiveOwnerBase<br>∃ hasOwnerRole SurrogateOwnerRole |
| PhoneHandlerManager<br>LineCardManager<br>ChannelManager<br>ToneToDigitCardManager | NamedSupplier | NamedSupplier<br>PassiveSupplier<br>PooledResourceSupplier | ∃ hasOwnerBase PassiveOwnerBase<br>∃ hasOwnerRole DefaultOwnerRole |

**Table 5-1: Named Owner classes for the PBX.**

# 5.3 A Specialized Inferred ResOwn Instance Example

Recall from Chapter 4 that the *baseline ResOwn ontology* actually consists of a manually created, asserted class hierarchy from which a *inferred class hierarchy* can be generated automatically using a reasoner. It is interesting to observe that the *specialized ResOwn instance* is merely just the *asserted*

*class hierarchy* of the baseline ResOwn ontology extended with application-specific ontological classes from the PBX. Therefore, the *asserted class hierarchy* for the *specialized ResOwn instance* can also be classified, in the exact same way, using the reasoner. The result is an automatically generated, application-specific *inferred class hierarchy* for the *specialized ResOwn instance*. This result is significant because it means all ontological classes originating from the PBX can be tested and any multiple inheritance automatically determined.

| Resource Class | Subsumed By Class | | Active Asserted Conditions |
|---|---|---|---|
| | Asserted | Inferred | |
| IdleRelayDevice<br>RingerRelayDevice | NamedResource | EmbeddedResource<br>InternalResource<br>ClosedResource<br>NonconsumableResource | ∃ isBoundTo EmbeddedTitledeed<br>∃ hasContainer LineCard<br>isContainerOf = 0<br>hasPortal = 0<br>∃ hasCapacity SingleUser<br>∃ hasPersistency SystemPersistent |
| LineCard | NamedResource | DistributableResource<br>CompoundResource<br>ExternalGatewayResource | ∃ isBoundTo ReusableTransferableTitledeed<br>∃ isContainerOf IdleRelayDevice<br>∃ isContainerOf RingerRelayDevice<br>∃ hasPortal SessionAccessPortal<br>∃ hasCapacity SingleUser<br>∃ hasPersistency SystemPersistent<br>hasContainer = 0 |
| Channel<br>ToneToDigitCard | NamedResource | DistributableResource<br>InternalResource<br>NonconsumableResource | ∃ isBoundTo ReusableTransferableTitledeed<br>∃ hasCapacity SingleUser<br>∃ hasPersistency SystemPersistent<br>hasContainer = 0<br>isContainerOf = 0<br>hasPortal = 0 |
| IdleToneGeneratorCard<br>DialToneGeneratorCard<br>RingToneGeneratorCard<br>SlowBusyToneGeneratorCard<br>FastBusyToneGeneratorCard | NamedResource | StationaryResource<br>OpenResource<br>InternalResource | ∃ isBoundTo NontransferableTitledeed<br>∃ hasCapacity MultipleUser<br>∃ hasPersistency SystemPersistent<br>hasContainer = 0<br>isContainerOf = 0<br>hasPortal = 0 |

**Table 5-2: Named Resource class for the PBX.**

For convenience and readability of the diagrams, the **Owner** and **Resource** class hierarchies of the specialized ResOwn instance for the PBX are considered in these two separate sub-examples.

- Consider the *asserted* **Owner** subclass hierarchy presented in Figure 5-1 of the *specialized ResOwn instance* for the PBX. The definitions for the defined Owner subclasses are given in Table 4-15 of Chapter 4. The definitions of the **Named Owner** classes for the PBX are given in Table 5-1. The resulting *inferred* **Owner** class hierarchy of the *specialized ResOwn instance* for

the PBX is shown in Figure 5-4. Any resulting multiple inheritance now apparent in the specialized *inferred Owner* class hierarchy has been commanded in the **Subsumed By Class: Inferred** column of Table 5-1.

- Consider the *asserted* **Resource** subclass hierarchy presented in Figure 5-2 of the *specialized ResOwn instance* for the PBX. The definitions for the defined Resource subclasses are given in Table 4-10 of Chapter 4. The definitions of the **Named Resource** classes for the PBX are given in Table 5-2. The resulting *inferred* **Resource** class hierarchy of the *specialized ResOwn instance* for the PBX is shown in Figure 5-5. Any resulting multiple inheritance now apparent in the specialized *inferred* **Resource** class hierarchy has been commanded in the **Subsumed By Class: Inferred** column of Table 5-2.

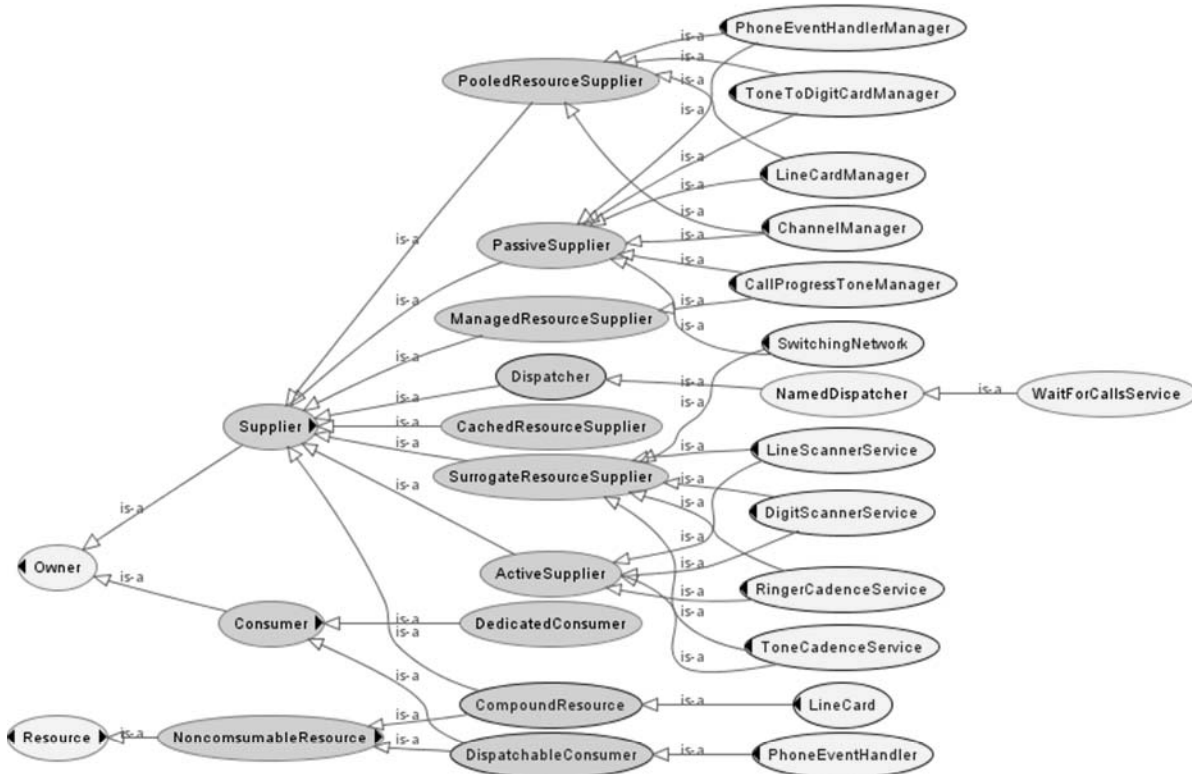



**Figure 5-4: Inferred Owner class hierarchy of the PBX's specialized ResOwn instance.**

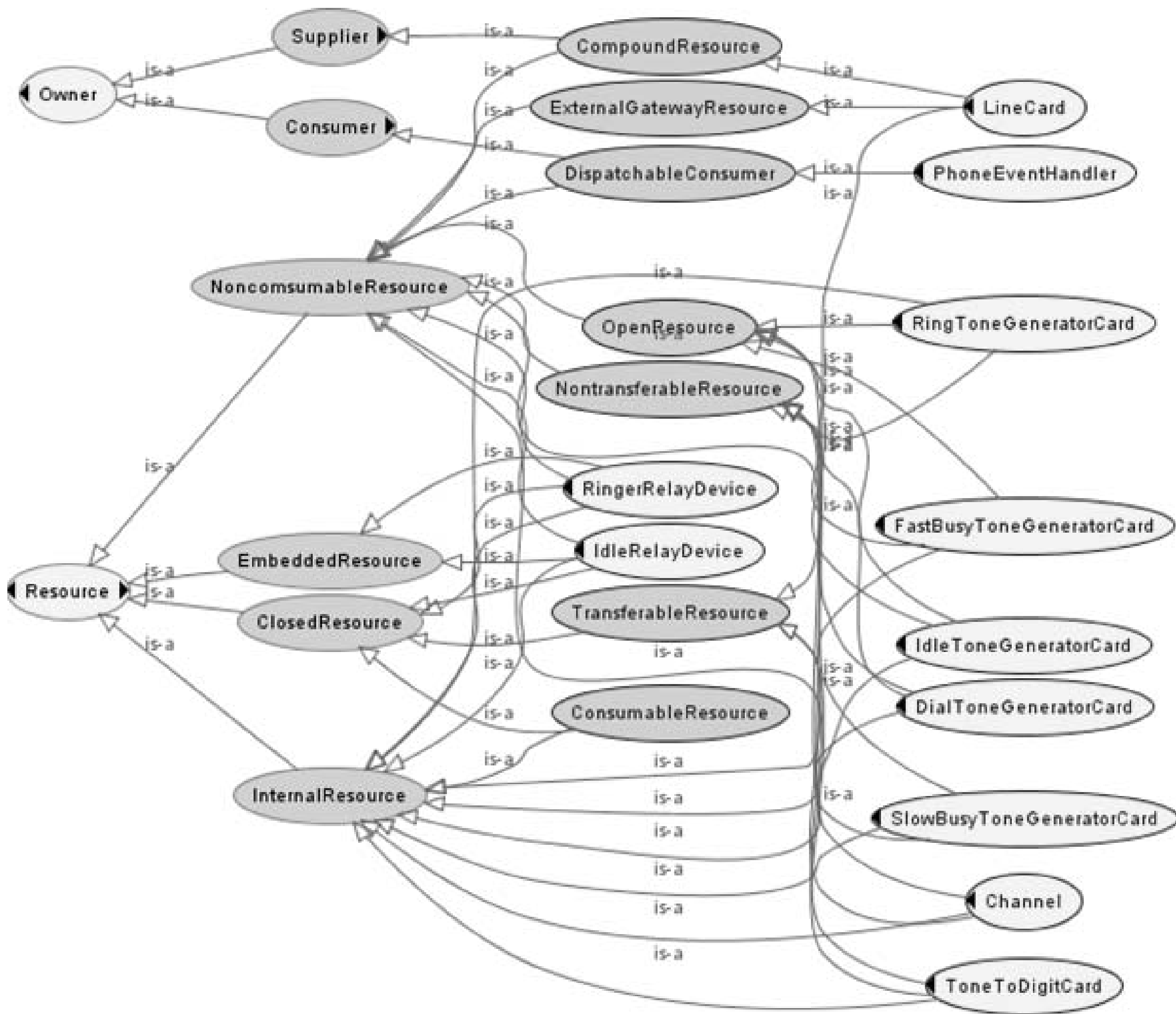



**Figure 5-5: Inferred Resource class hierarchy of the PBX's specialized ResOwn instance.**

## 5.3.1 ResOwn Specialization Issues

One other aspect to consider in the specialization process is the case where an ontological class originating from an application (e.g., PBX) is inconsistent and therefore cannot be automatically classified by a reasoner. There are two main reasons why this might occur. First, the new class should have been consistent, but the property restrictions associated with the inconsistent ontological class were instantiated incorrectly (i.e., human error). Second, the range of a property (or possibly several properties) in the ontology might not be specified to include a necessary value or object class needed to define the inconsistent ontological class. This would mean that the property range would need to be

extended to accommodate the new application-specific class definition. Another potential specialization issue might occur when an application-specific ontological class is consistent, but is not classified under any of the existing defined classes in the baseline ResOwn ontology. Assuming the new ontological class definition was correct, this result would imply that the new ontology class itself specifies a new type of defined class, which extends the baseline ResOwn ontology, rather than simply specializing it. As a result, ResOwn could be extended by adding a new defined class to the appropriate top-level ResOwn class taxonomy (e.g., Resource or Owner).

# 5.4 Resource Acquisition and Ownership Scenarios

This section presents a number of *role-based*, *resource acquisition* and *ownership scenarios* for the **Transferable**, **Nontransferable**, **Embedded** and **Compound Resource** classes. The various scenarios described below use every **Named Instrument** class (Section 4.8), *except* where noted below in Section 5.4.5. The following procedure was employed to obtain more realistic scenarios for testing the monitoring approach described in Chapter 7 using a Java implementation of the PBX:

- The source code for the call processing software of the PBX was manually instrumented, compiled to bytecode and run on a stand-alone Unix workstation.
- Several resource ownership call traces were generated for both single calls and multiple concurrent calls. All call traces were verified to ensure the call processing software of the PBX did adhere to its behavioral design specifications.
- The resulting resource ownership call traces were analyzed. Any observed structural resource ownership transactions recorded. This recorded runtime knowledge was used by the author during the manually derivation of the role-based resource acquisition and ownership scenarios presented further on in this section.

Recall from Chapter 1 that the concern-specific, evolving software structure of an operational software system can be represented or modeled by an ordered sequence of object diagrams. An *object diagram* represents a *state-dependent snapshot* (i.e., individual view) of the selected concern-specific software structure. A *class diagram* represents a *state-independent family of snapshots* (i.e., compound view) of the selected concern-specific software structure. In this example, a scenario is a behavior-driven, ordered sequence of individual snapshots of the evolving resource ownership

structure of the PBX. Each snapshot is comprised of one or more proof of ownership **Instrument** instances, plus one or more related **Owner** and **Resource** instances.

Each role-based resource allocation and ownership scenarios is presented with:

- A natural language description that explains the addition and removal of individual links (i.e., association instances) between **Instrument**, **Resource**, **Supplier** and **Consumer** instances that occur as the resource ownership structure of a current snapshot evolves into the next snapshot in the ordered sequence.
- A UML-like object transformation rules, loosely based on *Attributed Graph Grammar (AGG) notation* from [Tae03], used to visually describe the **Instrument**-based resource ownership structure of each snapshot in a particular scenario.

The reader should note that some nonstandard UML notations are used in the object diagrams for the below-noted scenarios. Specifically, the *stereotypes* shown on the objects in the snapshots are just for information purposes in the object diagrams so that the reader can easily see from where the application-specific object was derived in the inferred class hierarchy of the specialized ResOwn instance described in Section 5.3. Finally, it should be noted that one could take the specialized ResOwn instance and translate it to a simpler UML model by mapping ontological concepts to UML classes and ontological properties to UML attributes.

### 5.4.1 Monitoring Event-Driven Snapshots

In the monitoring approach described in Chapter 7, the sequencing of resource allocation and ownership snapshots is state-dependent and normally requires runtime knowledge from the operation target in the form of monitoring commands. For now, it is sufficient to only introduce these monitoring commands, without parameters, that will be used in the scenarios that follow.

- **ACQUIRE** is reported to indicate when a **Consumer** instance has acquired *direct* or *indirect beneficiary ownership* of a particular **Transferable, Nontransferable** or **Embedded Resource** instance from a granting **Pooled Resource Supplier, Managed Resource Supplier** or **Compound Resource** instance, respectively.
- **RELEASE** is reported to indicate when a **Consumer** instance has returned *beneficiary ownership* of a particular **Resource** instance back to its originator**.**

- **REGISTER** is reported to indicate when a **Consumer** instance has given *nonbeneficiary ownership* of a **Transferable, Nontransferable** or **Embedded Resource** instance to a **Surrogate Resource Supplier** or **Cached Resource Supplier** instance.
- **UNREGISTER** is reported to indicate when a **Consumer** instance has relinquished *nonbeneficiary ownership*.

## 5.4.2 Transferable Resource Scenario

Consider the snapshots for the **Transferable Resource** scenario, as shown in Figure 5-6. The applicable objects are defined in Table 5-3.

| PBX Class | Identifier | ID | ResOwn Superclass |
|---|---|---|---|
| Touch Tone Receiver Card | :TTRX | ttrx5 | Transferable Resource |
| Touch Tone Receiver Card Manager | :TM | tm1 | Pooled Resource Supplier |
| Phone Handler | :PH | ph1 | Dedicated Consumer |
| Transferable Titledeed | :TTD | ttd1 | Base Instrument |

**Table 5-3: Transferable Resource scenario.**

### 5.4.2.1 Scenario Semantics

- At time t: Figure 5-6(i) shows *ttd1* bound to *ttrx5*, as indicated by the ***isBoundTo*** link between the **Transferable Titledeed** instance and the **Transferable Resource** instance. Further, *ttrx1* has only a *default owner* as indicated by the fact that the ***hasHolder*** and ***hasIssuer*** links from *ttd1* are both connected to *tm1*.
- At time t+1: Figure 5-6(ii) shows what happens after the instrumented target produces an **ACQUIRE** indicating that *ph1* has successfully acquired *beneficiary ownership* of *ttrx5* from *tm1*. Therefore, the ***hasHolder*** link of *ttd1* must be disconnected from *tm1* and connected to *ph1* so that the snapshot reflects the new ownership structure. Because the ***hasHolder*** and ***hasIssuer*** links now connect to different objects, *ph1* may be inferred as the *current owner* of *ttrx5* and *tm1* remains the *default owner* of *ttrx5*.
- At time t+2: Figure 5-6(i) shows what happens after the instrumented target produces a **RELEASE** indicating that *ph1* has relinquished *beneficiary ownership* of *ttrx5* back to *tm1*. To keep the snapshot synchronized with the target's evolving ownership structure, the ***hasHolder***

link of *ttd1* must be disconnected from *ph1* and subsequently reconnected to *tm1*, indicating again that *ttrx5* has no *beneficiary owner*.

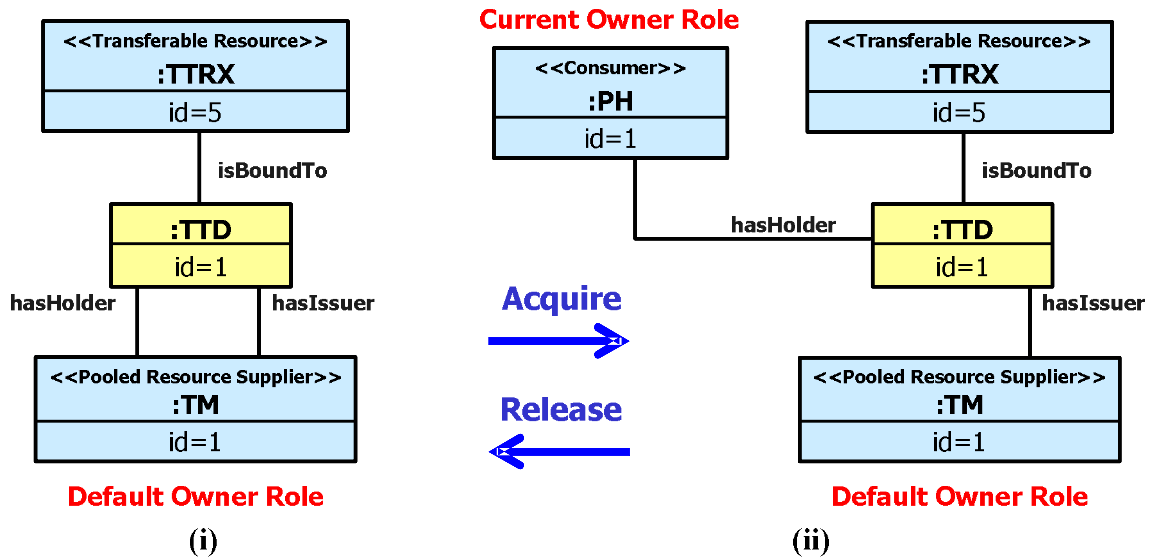



**Figure 5-6: Transferable Resource scenario.**

### 5.4.2.2 Examples of Structural Errors

Here are some samples of the possible structural errors that could be detected during a **Transferable Resource** acquisition and release.

- The **Transferable Resource** instance is made an orphan because the **Transferable Titledeed** instance is not released back to the original *default owner* before the end of the current, in-progress service request (i.e., telephone call).

- The same **Transferable Titledeed** instance, with a *single user capacity*, is issued to more than one *current owner* creating a multiplicity violation.

- The target reports that the acquired **Transferable Resource** instance is actually bound to a **Nontransferable** or **Embedded Titledeed** instance at initialization time. Therefore the *resource ownership acquisition* violates the *invariant* that states that a **Consumer** instance playing a *current owner* role may only directly hold a **Transferable Resource** instance that is bound to **Transferable Titledeed** instance and issued from **Pooled Resource Supplier** instance.

- The acquired **Transferable Resource** instance is release back to a different **Pooled Resource Supplier** instance than from which it was originally acquired. Therefore, the *invariant* that states that if the holder of a **Transferable Titledeed** instance is not a *current owner*, it must be the *default owner* is violated:

$$error \rightarrow (\neg CurrentOwner \wedge (hasHolder(DefaultOwner) \neq hasIssuer(DefaultOwner)))$$

### 5.4.3 Nontransferable Resource Scenario

Consider the snapshots for the Nontransferable Resource scenario shown in Figure 5-7 where the applicable objects are defined in Table 5-4.

| PBX Class | Identifier | ID | ResOwn Superclass |
|---|---|---|---|
| Dial Tone Generator Card | :DIAL | dial2 | Nontransferable Resource |
| Call Progress Tone Manager | :CPTM | cptm1 | Managed Resource Supplier |
| Phone Handler | :PH | ph1 | Dedicated Consumer |
| Nontransferable Titledeed | :NTD | ntd2 | Base Instrument |
| Serial License | :LIC | lic1 | Extent Instrument |

**Table 5-4: Nontransferable Resource scenario.**

#### 5.4.3.1 Scenario Semantics

- At time t: Figure 5-7(i) shows *ntd2* bound to *dial2*, as indicated by the ***isBoundTo*** link between the **Nontransferable Titledeed** instance and the **Nontransferable Resource** instance. Further, *dial2* has only a *permanent owner* as indicated by the fact that the ***hasHolder*** and ***hasIssuer*** links from *ntd2* are both connected to *cptm1*.

- At time t+1: Figure 5-7(ii) shows what happens after the instrumented target produces an **ACQUIRED** indicating that *ph1* has successfully acquired *beneficiary ownership* of *dial2* from *cptm1*. Therefore, *ntd2* is extended by *lic1*, as indicated by the ***hasExtent*** link between the **Serial License** instance and the **Nontransferable Titledeed** instance. Further, *lic1* ***hasIssuer*** link connected to *cptm1* and ***hasHolder*** link connected to *ph1* so that the snapshot will accurately reflect the evolving ownership structure of the operational target. Further, because the ***hasExtent***

now connects to **Extent Instrument** instance held by the *ph1*. It may be inferred that *ph1* is an *exclusive licensed owner* of *dial2,* while *cptm1* remains the *permanent owner* of *dial2*.

- At time t+2: Figure 5-7(i) shows what happens after the instrumented target produces a **RELEASE** indicating that *ph1* has relinquished *beneficiary ownership* of *dial2* back to *cptm1*. Therefore, to keep the snapshot ownership structure synchronized with the evolving target, the ***hasHolder*** and ***hasIssuer*** links of *lic1* must be disconnected from *ph1* and *cptm1*, respectively, and the ***hasExtent*** link must subsequently be disconnected from *lic1*.

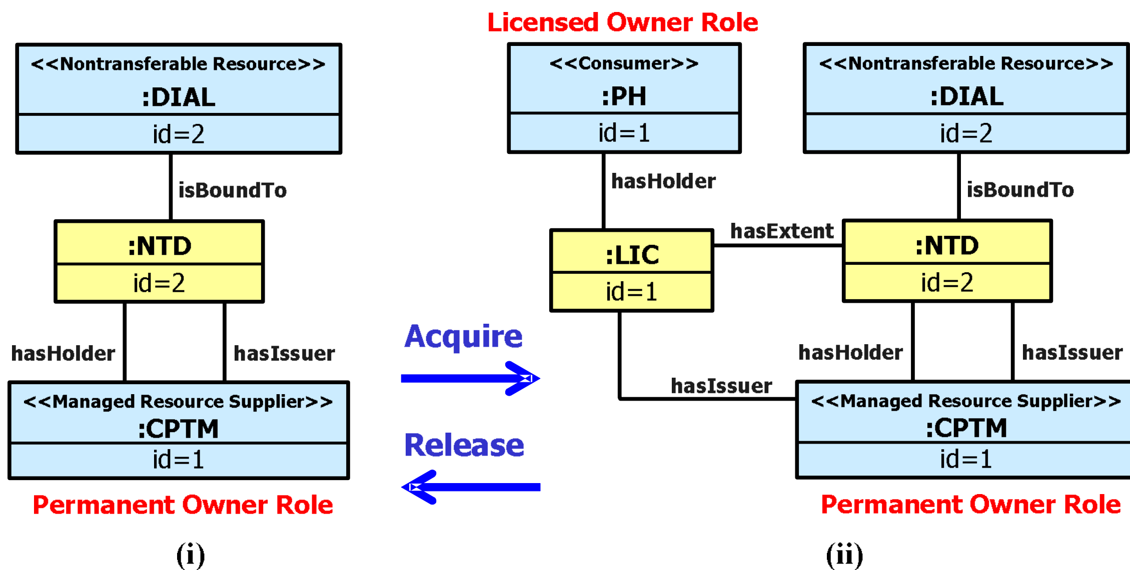



**Figure 5-7: Nontransferable Resource scenario.**

### 5.4.3.2 Examples of Structural Errors

Here are some samples of the possible structural errors that could be detected during a **Nontransferable Resource** acquisition and release.

- The **Nontransferable Resource** instance is made an orphan because the **Serial License** instance is not released back to the original *permanent owner* before the end of the current, in-progress service request.

- The same **Serial License** instance, with a *single user capacity*, is issued to more than one *exclusive licensed owner* creating a multiplicity violation.

- The target reports that the acquired **Nontransferable Resource** instance was one which is actually bound to a **Transferable** or **Embedded Titledeed** instance at initialization time. Therefore the resource ownership acquisition violates the *invariant* that states that a **Consumer** instance playing a *licensed owner role* may only own **Nontransferable Resource** instance through *licensing* from a **Managed Resource Supplier** instance.
- The acquired **Nontransferable Resource** instance is release back to a different **Managed Resource Supplier** instance than from which the **Serial License** instance was originally acquired.

## 5.4.4 Embedded Resource Scenario

Consider the snapshots for the Embedded Resource scenario, as shown in Figure 5-8 where the applicable objects are defined in Table 5-5.

| PBX Class | Identifier | ID | ResOwn Superclass |
|---|---|---|---|
| Ringer Relay Device | :RR | rr8 | Embedded Resource |
| Line Card | :LC | lc1 | Transferable Compound Resource |
| Line Card Manager | :LM | lm1 | Pooled Resource Supplier |
| Phone Handler | :PH | ph1 | Dedicated Consumer |
| Transferable Titledeed | :TTD | ttd2 | Base Instrument |
| Embedded Titledeed | :ETD | etd3 | Base Instrument |
| Proxy | :PRX | prx1 | Extent Instrument |

**Table 5-5: Embedded Resource scenario.**

### 5.4.4.1 Scenario Semantics

- At time t: Figure 5-8(i) shows *etd2* bound to *rr8*, as indicated by the ***isBoundTo*** link between the **Embedded Titledeed** instance and the **Embedded Resource** instance. Further, *rr8* has only a *containment owner* as indicated by the fact that the ***hasHolder*** and ***hasIssuer*** links from *etd2* are both connected to *lc8*. In addition, *ttd3* is bound to *lc8*, as indicated by the ***isBoundTo*** link between the **Transferable Titledeed** instance and the **Transferable Compound Resource** instance. Further, *lc8* has only a *default owner* as indicated by the fact that the ***hasHolder*** and ***hasIssuer*** links from *ttd3* are both connected to *lm1*.

- At time t+1: Figure 5-8(ii) shows what happens after the target produces an **ACQUIRE** indicating that *ph1* has successfully acquired *beneficiary ownership* of both *lc8* from *lm1* and *rr8* from *lc8*. Therefore, ***hasHolder*** link of *ttd3* must be disconnected from *lm1* and connected to *ph1* and *etd2* is extended by *prx1*, as indicated by the ***hasExtent*** link between the **Proxy** instance and the **Embedded Titledeed** instance. Further, *prx1* ***hasIssuer*** link connected to *lc8* and ***hasHolder*** link connected to *ph1*. Note that *ph1* may be *inferred* as both the *current owner* of *lc8* and the *proxied owner* of *rr8*, while *lc8* remain the *containment owner* of *rr8* and *lm1* remains the *default owner* of *lc8*.

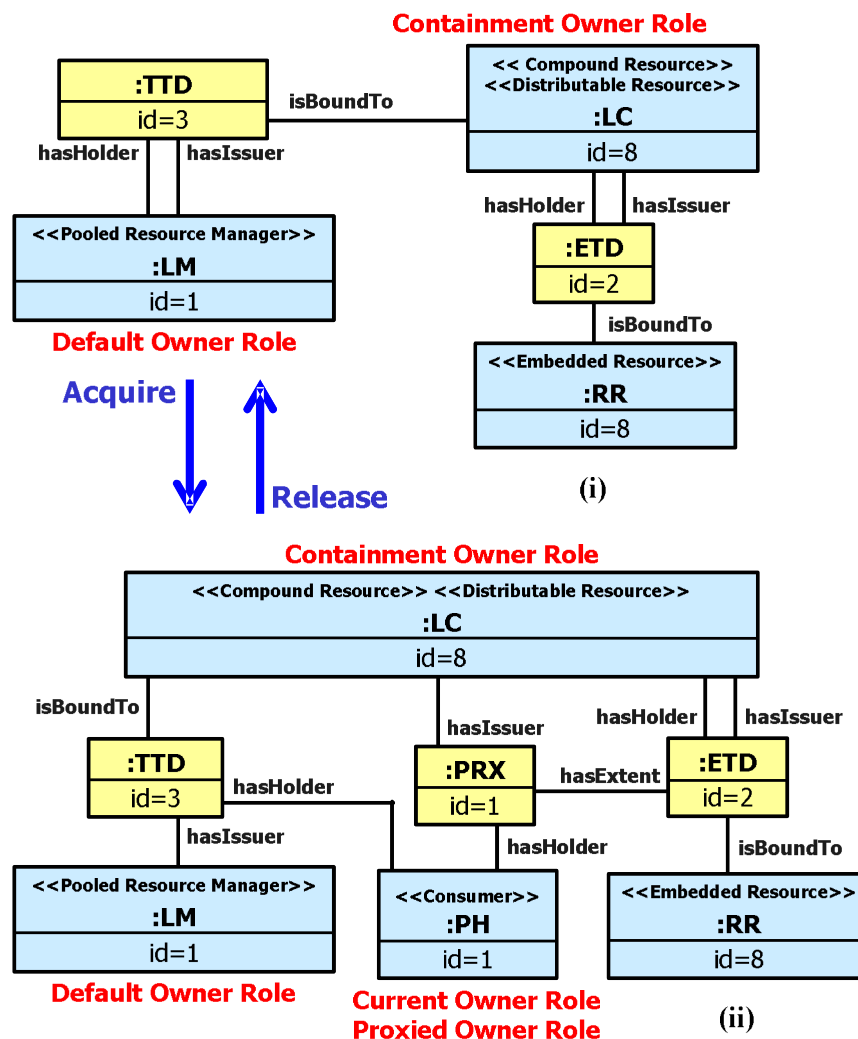



**Figure 5-8: Compound and Embedded Resource scenario.**

- At time t+2: Figure 5-8(i) shows what happens after the target produces a **RELEASE** indicating that *ph1* has relinquished *beneficiary ownership* both *rr8* and *lc8* back to *lc8* and *lm1*, respectively. Therefore, to keep the snapshot ownership structure synchronized with the evolving target, the ***hasHolder*** and ***hasIssuer*** links of *prx1* must be disconnected, the ***hasExtent*** link must disconnected from *prx1* and the ***hasHolder*** link of *ttd2* must be disconnected from *ph1* and reconnected to *lm1*.

#### 5.4.4.2 Examples of Structural Errors

Here are some samples of the possible structural errors that could be detected during Embedded Resource acquisition and release.

- The **Embedded Resource** instance is made an orphan because the **Transferable Titledeed** instance for the **Transferable Compound Resource** instance containing the **Embedded Resource** instance is not released back to the original *default owner* before the end of the current, in-progress service request.
- The **Embedded Resource** instance is made an orphan because the **Proxy** instance for the **Embedded Resource** instance is not released back to the original *containment owner* before the end of the current, in-progress service request.
- The same **Proxy** instance, with a *single user capacity*, is issued to more than one *proxied owner* creating a multiplicity violation.
- The target reports that the acquired **Embedded Resource** instance is one which is actually not bound to a **Compound Resource** instance at initialization time.
- The acquired **Embedded Resource** instance is release back to a different **Compound Resource** instance than the one that acted as the *containment owner* of the **Embedded Resource** instance.

### 5.4.5 Surrogate Resource Supplier Registration Scenario

Consider the snapshots for the **Surrogate Resource Supplier** registration scenario[1], as shown in

---

[1] The **Cached Resource Supplier** registration scenario is essential the same except a *temporary owner role* would replace the *surrogate owner role* and a **Permit To Hold** replace the **Power Of Attorney**.

Figure 5-9 where the applicable objects are defined in Table 5-6.

| PBX Class | Identifier | ID | ResOwn Superclass |
|---|---|---|---|
| Touch Tone Receiver Card | :TTRX | ttrx5 | Transferable Resource |
| Touch Tone Receiver Card Manager | :TM | tm1 | Pooled Resource Supplier |
| TTRX Scanner | :TSCAN | tscan1 | Surrogate Resource Supplier |
| Phone Handler | :PH | ph1 | Dedicated Consumer |
| Transferable Titledeed | :TTD | ttd1 | Base Instrument |
| Power Of Attorney | :POA | poa1 | Extent Instrument |

**Table 5-6: Surrogate Resource Supplier scenario.**

### 5.4.5.1 Scenario Semantics

- At time t+1: Figure 5-9(i) shows the scenario from 5.4.2, after the **ACQUIRE**, where *ttd1* is bound to *ttrx5*, the ***hasHolder*** link is connected to *ph1*, the *current owner* of *ttrx5* and the ***hasIssuer*** link is connected to *tm1*, the default owner of *ttrx5*.
- At time t+2: Figure 5-9(ii) shows what happens after the target produces a **REGISTER** to indicate that *ph1* has successfully registered *nonbeneficiary ownership* of *ttrx5* with *tscan1*. The *ttd1* is extended by the *poa1*, as indicated by the ***hasExtent*** link between the **Power Of Attorney** instance and the **Transferable Titledeed** instance. Further, *poa1* has the ***hasIssuer*** link connected to *ph1* and the ***hasHolder*** link connected to *tscan1*. Since *tscan1* holds *poa1*, *tscan1* is the inferred *nonbeneficiary owner* of *ttrx5* and only controls when *ttrx5* will scan for digits. Further, since *ph1* still holds *ttd1*, it remains the inferred *current owner* and *beneficiary owner* of *ttrx5* and will ultimately be the receiver of any dialed digits.
- At time t+2: Figure 5-9(i) shows what happens again after the target produces the **UNREGISTER**. The **UNREGISTER** disconnects ***hasIssuer*** link between *ph1* and *poa1*, the ***hasHolder*** link between *tscan1* and *poa1* and the *hasExtent* link between *poa1* and *ttd1*, thus returning to the *current owner* snapshot.

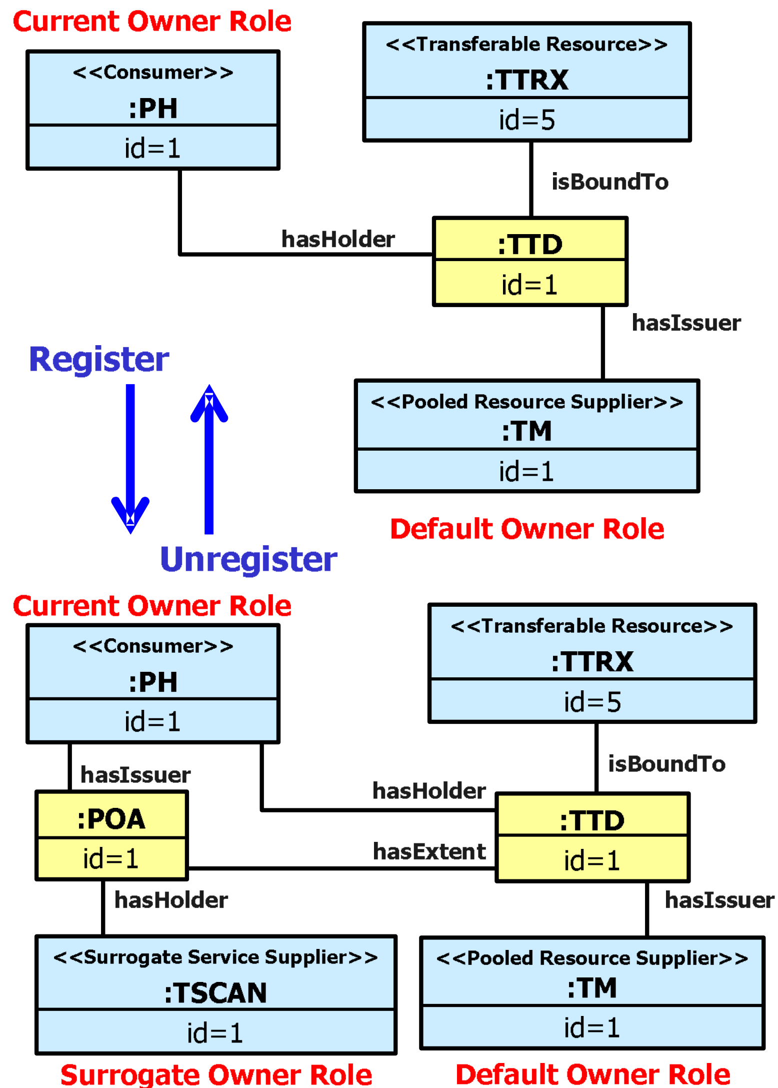



**Figure 5-9: Surrogate Resource Supplier scenario.**

### 5.4.5.1.1 Orphaned Power Of Attorney Scenario

Consider now the snapshots for the *extended* **Surrogate Resource Supplier** unregistration scenario, as shown in Figure 5-10 using the same objects as defined in Table 5-6. In this case, the target reports that a **RELEASE** indicating that beneficiary ownership of ttrx5 has been relinquished by ph1 back to tm1. However, this result is erroneous because the target did not first report an **UNREGISTER** indicating that *ph1* has relinquished *nonbeneficiary ownership* of *ttrx5* from *tscan1*. As a result, a *resource leak* occurs as indicated by the orphaned **Power Of Attorney** instance. The **Power Of Attorney** instance is said to be *orphaned* because *ttrx5* is still *registered* with *tscan1* on behalf of

*ph1*, but *ph1* no longer holds the **Transferable Titledeed** instance for *ttrx5*. Therefore, *ttrx5* no longer has a *beneficiary owner* for which the *surrogate owner*, *tscan1*, acts on behalf of. This error was actually detected in the PBX. The fault occurred in a certain execution path of *ph1* where the *Phone Hander* did not inform the *TTRX Scanner* to stop scanning for *digits* before the *Phone Handler* released the *TTRX* back to the *TTRX M*anager.

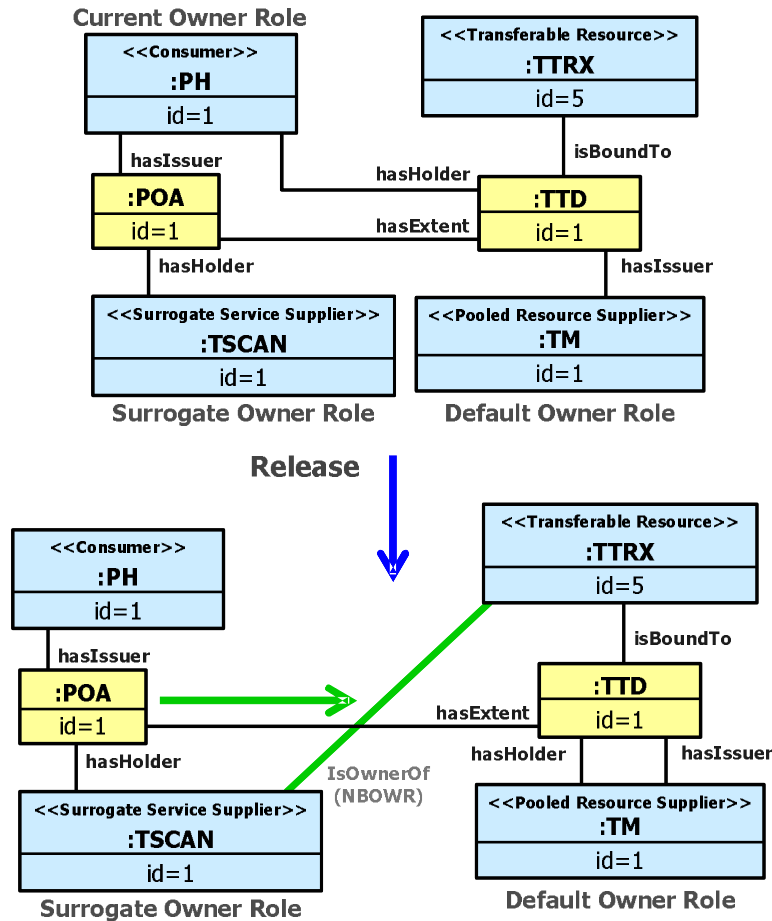



**Figure 5-10: Orphaned Power Of Attorney scenario.**

# Chapter 6

# Session-Oriented Model of Computation

## 6.1 Introduction

This chapter presents a detailed description of a *session-oriented model of computation (SOMOC)* that is used as a basis for monitoring interaction applications. The SOMOC is a *dual-view model* that bridges the semantic gap between an interactive session-oriented service's observable behavior and its underlying software system's evolving structure. The SOMOC decomposes an interactive session-oriented service into two levels of abstraction:

- A requirements[1]-level that provides an external behavioral view or modeling perspective of the interactive session oriented service.
- A design-level that provides an internal structural view or modeling perspective of the service's underlying software system.

Concepts from the SOMOC described in this chapter are used, as described in detail in Chapter 7, for deriving a concern-specific monitoring model from the target software system's formal specifications. To the best of the author's knowledge, a dual-view, *session-oriented model of computation* has not

[1] In this thesis, *requirements-level* mean a high-level SDL design specification, and *design-level* means low-level, detailed SDL design specification.

been described previously in the literature.

### 6.1.1 Hierarchical Abstract Layering

Hierarchical abstract layering formulates general concepts organized into abstraction levels from common properties, where higher-level concepts are dependent on lower-level ones [Szp02]. The SOMOC organizes object classes into a non-strict, hierarchical abstract layered architecture. Non-strict layering allows objects instantiated from classes on one layer of the architecture to interact with objects instantiated from other classes on any lower layer. A class's position in the architecture is position-sensitive. For the example PBX, as shown in Figure 2.28 of Chapter 2, **Resource** instances reside in the lowermost layer, **Consumer** instances in the topmost layer, and **Supplier** instances in the middle layer. Control only flows down. Data may flow up from the service environment or down to the service environment.

### 6.1.2 Behavioral Partitioning

Behavioral partitioning separates a whole into its parts [Iee90]. The SOMOC uses partitioning to logically separate an interactive service's observable behavior into a set of functionally equivalent, concurrent sessions[1]. Doing this partitions the service's input state-space and reduces the number of behavioral alternatives that must be simultaneously considered during monitoring.

### 6.1.3 What is Session-Orientation?

The SOMOC is session-oriented. The SOMOC adopts an object-based definition of a session as sets of externally observable interactions that bind sets of objects to users, for a given activity, over a bounded time period [Tex03].

### 6.1.4 SOMOC: Internal Structural View

The internal view models the structure of an interactive service as a set of ***n*** *logical sessions* $<S_1, ..., S_i, ..., S_n>$. Consider the internal structural view given in Figure 6-1. For the selected structural

---

[1] The SOMOC is also capable of supporting sessions with different functional (i.e., behavioral) requirements.

software concern, each $\boldsymbol{S_i}$ (i.e., dashed oval) is logically comprised of an *end user*, $\boldsymbol{U_i}$, a *session access portal*, $\boldsymbol{P_i}$ , a computational thread of execution, $\boldsymbol{T_i}$ (i.e., **Consumer** instance) and a *local software structure*, $\boldsymbol{L_i}$ (i.e., set of owned **Resource** instances). Each $\boldsymbol{T_i}$ executes concurrently with other portions of the software and delivers the service's functionality to each $\boldsymbol{U_i}$ via a $\boldsymbol{P_i}$. Each $\boldsymbol{P_i}$ represents an *ingress* between the service environment and the software system implementation. The *global software structure* is comprised of a set of *regional software structures*, $R_i$, (i.e., dashed trapezoid). An $R_i$ has an associated Supplier instance and its owned Resource instances. The monitor is capable of tracking the evolving software structure on a local, regional and global basis.

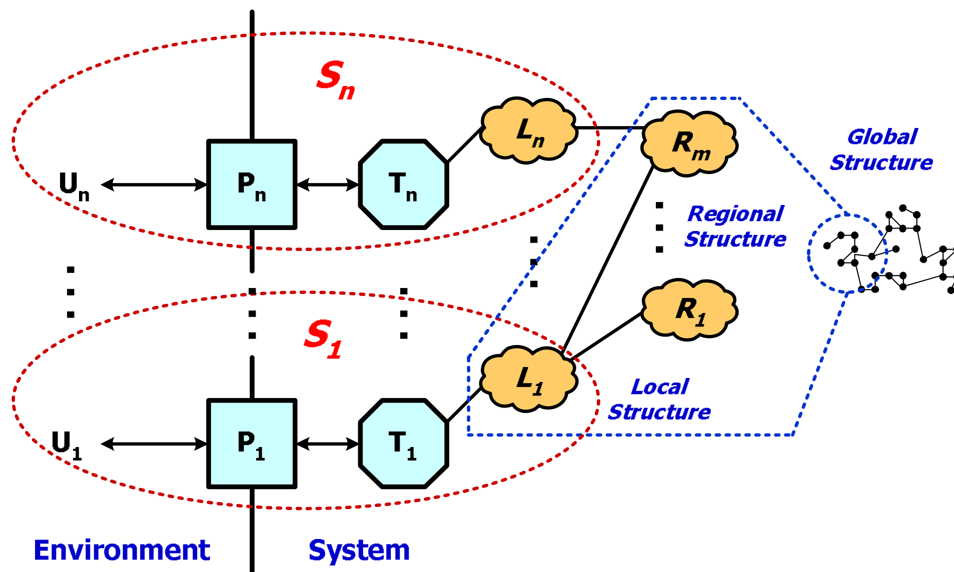



**Figure 6-1: Dual-view model: internal structural view.**

## 6.1.5 SOMOC: External Behavioral View

This external view models the observable session behavior as a *finite state machine (FSM)* of *superstates*. Consider the external behavioral view given in Figure 6-2. The SOMOC distinguishes between those service execution phases that differ in their resource usage, making the use of finite state machines suitable for formalizing this model. An inactive $\boldsymbol{S_i}$ is *idle* and has no pending service requests on $\boldsymbol{P_i}$. An active $\boldsymbol{S_i}$ is processing an *in-progress service request* on $\boldsymbol{P_i}$ and transitions through some or all of the shown model *superstates*. The monitor is capable of detecting when an interactive service enters or leaves a model superstate on a session-by-session basis.

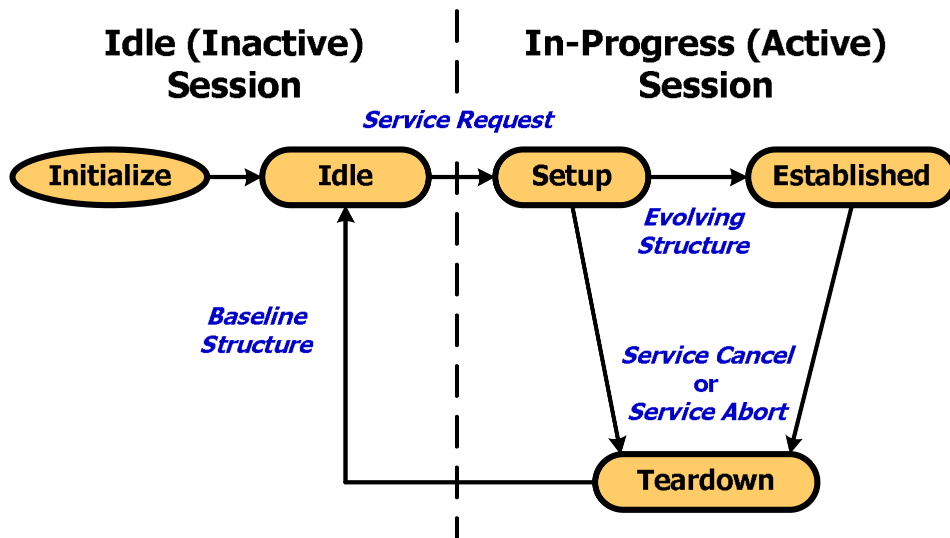



**Figure 6-2: Dual-view model: external behavioral view.**

### 6.1.5.1 Service Provisioning Path

The model's *service provisioning path* is the *trajectory* followed by the service during *normal* operation. In Figure 6-2, the service provisioning path of a session is the *superstate sequence*: **Idle** / **Setup** / **Established** / **Teardown** / **Idle**, where the **Established** / **Teardown** transition only occurs due to a *service cancel* initiated by a user. If a session is on its service provisioning path, then it's evolving software structure must adhere to certain state-dependent invariants.

### 6.1.5.2 Service Annulment Paths

The model's *service annulment path* is the *trajectory* followed by the service during *exceptional* operation. An operational exception occurs when a user initiates a *service cancel* before the service is established, or anytime the service itself initiates a *service abort*. A *service abort* indicates exceptional runtime circumstances such as detection of an invalid service parameter or violation of some computational constraint such as depletion of a shared resource. In Figure 6-2, two possible service annulment paths are the *superstate sequences*: **Idle** / **Setup** / **Teardown** / **Idle**, or **Idle** / **Setup** / **Established** / **Teardown** / **Idle**, where **Established** / **Teardown** only occurs after a service abort. If a session is on a service annulment path, then its evolving software structure may no longer adhere to certain state-dependent invariants in the same way a provisioning session would.

#### 6.1.5.3 Idle Superstate

An inactive session in the SOMOC normally resides in its *Idle* superstate. The *Idle* superstate has a memoryless property that states that any side effects[1] from a previous active session are never carried forward through the *Idle* superstate to a new session. When an inactive session is in its *Idle* superstate, it is the underlying software structure for that session that is said to be at a its *minimal* or *baseline*.

#### 6.1.5.4 Resynchronization

*Resynchronization* allows an automated observer to continue tracking the state of an operational target software system after a behavioral failure has been detected [Ior94, Hua99, Pek03, Sav97]. If a structural error occurs and is detected during a particular session, then the *Idle* superstate is an ideal resynchronization point because a monitor can reset the session's software structure back to its baseline once the associated session becomes inactive.

### 6.1.6 Specification Refinement and Refinement Mapping

The notion of specification refinement, as shown in Figure 6-3, consists of iteratively creating a number of increasingly detailed projections to transform an abstract specification into a more concrete one, until source code is produced [Gez03, Pfl06, Liu02, Sha96]. The process of specification refinement must preserve the externally visible behavior of the software system. On the other hand, a refinement mapping is a function that maps the state-space of a less abstract projection, $\boldsymbol{S_L}$, onto the state-space of a more abstract projection, $\boldsymbol{S_H}$, while preserving *safety* and *liveness* properties [Aba91, Lam84, Lam83, Pav01]. If a refinement mapping holds between $\boldsymbol{S_L}$ and $\boldsymbol{S_H}$, then the external signals specified in $\boldsymbol{S_L}$ must map to an equivalent set of external signals in $\boldsymbol{S_H}$. For example, to guarantee a requesting client without implementation knowledge access to a server through a published interface, a software developer must ensure that any possible behavior of $\boldsymbol{S_L}$ can be mapped to at least one expected behavior of $\boldsymbol{S_H}$.

The monitoring approach in Chapter 7 assumes that the software design and source code have been refined from the requirements and a refinement mapping holds between each pair of adjacent specification projections. Therefore, the observable specification states and signals specified in the

---

[1] Excluding auxiliary variables and history logs as they are orthogonal to the service's functional requirements.

software requirements must be preserved (and identifiable) in the software design and implementation. These assumptions are essential for defining an *epoch of behavior* (Section 0) and a *quiescent state* (Section 6.1.7.2). It is assumed that:

- $S_L$ is *machine-closed* meaning that the *communicating extended finite state machine (CEFSM)* representing $S_L$ defines its complete behavior.
- $S_H$ has *finite nondeterminism*; that is, given any finite number of externally observable behaviors allowed by $S_H$, there are only a finite number of possible choices for the set of all possible corresponding internal state changes.
- $S_H$ is *internally continuous* meaning that any complete behavior (i.e., interactive transaction) allowed can be determined by observing and examining some finite portion of the service's externally observable behavior.

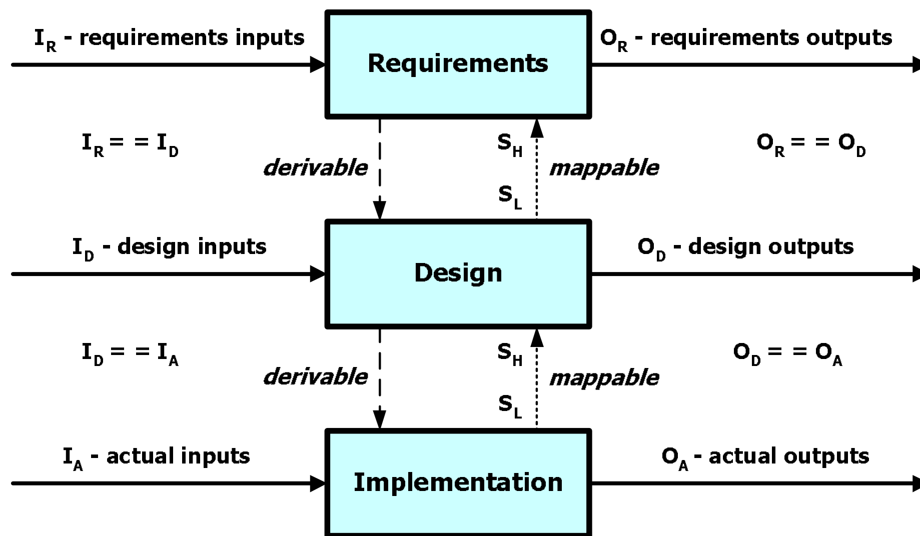



**Figure 6-3: Specification refinement and refinement mapping.**

## 6.1.7 Epoch of Behavior Models

Each active superstate in the SOMOC, as shown in Figure 6-1, is comprised of a set of finite state machines that specify "chunks" of application-specific logic, such as the example shown in Figure 6-4. An *epoch of behavior (EoB)* is a *specification slice* or *fragment* from the software design specification, demarcated by a set of *quiescent states* from the software requirements specification.

Further, an *EoB model* is comprised of a set of one or more structural transactions pertaining to a selected software concern formally derived (Section 7.4) from a target software system's software design specification. Consider an *internal observer* using an EoB model to track the evolving software structure of an operational software system for a selected software concern. When the operational target executes that portion of the code corresponding to the software design from which the EoB model was derived, a series of concern-specific transaction-like interactions are produced that indicate that the dynamic software structure of the target has evolved. For example, in this thesis, each EoB model specifies one or more successful structural resource ownership transactions.

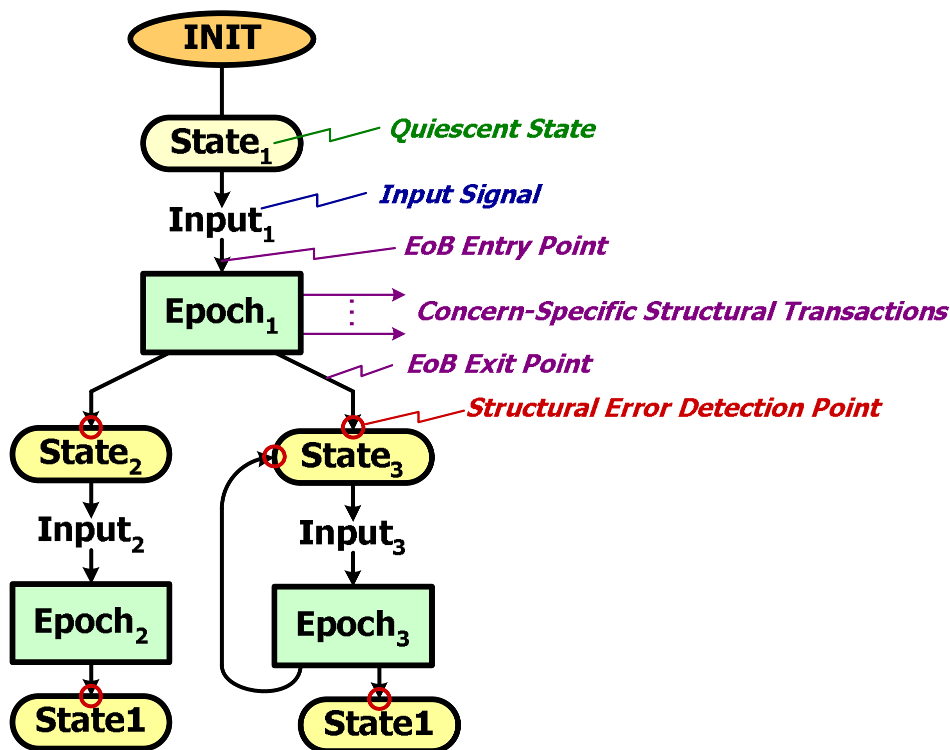



**Figure 6-4: Example session behavior partitioned into epochs of behavior.**

### 6.1.7.1 EoB Anatomy

As depicted in Figure 6-4, an *EoB model* consists of three main parts:

- A single *EoB Entry point* consisting of a unique *quiescent state* and *input signal combination* taken directly from the software requirements.
- An *EoB body* derived from the software system's software design specification and consisting of

a set of concern-specific *structural transactions* called *EoB detection points* that indicate when and how the software structure of the operational software system evolves. For example, ***Epoch$_1$***, as shown in Figure 6-4, produces a number of concern-specific structural transactions.

- A set of one or more *EoB Exit points*, each consisting of a transition-ending quiescent state.

Each EoB entry point, EoB exit point and EoB detection point matches an identifiable location in the software system's source code where an appropriate software sensor must be embedded. When the instrumented software encounters an embedded software sensor, an appropriate *monitoring command* (Section 7.3) is produced to mark the internal runtime event. In general, for the selected resource ownership concern, a monitoring command will be produced by an appropriate software sensor every time the operational target enters an EoB model, exits an EoB model or successfully completes a resource ownership transaction.

#### 6.1.7.2 Quiescent States

A *quiescent state* in the SOMOC is an externally observable specification state, specified in the software requirements, and preserved through refinement as an equivalent state in the software design and implementation. In this thesis, a session that is in a *quiescent state* is said to have exited the current EoB model, but not yet entered a new EoB model. From a runtime monitoring perspective, a session waits an indeterminate amount of time in a quiescent state until the operational software system reports, via a monitoring command, that the next EoB entry has been entered. Under normal operating conditions, if every session is quiescent, then it can be said that the service itself is in a *system-wide quiescent state*. The notion of a system-wide quiescent state is similar to that of a *stable state* which occurs when no processes are executing or messages are in transit [Zul04].

#### 6.1.7.3 EoB Example

Consider the example software requirements, design and source code excerpts given in Figure 6-5. A number of two-way correlations or abstraction pairs exist between adjacent projections between *quiescent states and input signal combinations* representing EoB entry points. It can be observed that every identified EoB always ends in a *mapped* quiescent state as well. In addition, any number of *transient states* may exist inside a demarcated EoB that are not mapped to higher level projections. *Transient states* are short-term states that are not normally visible to an external observer. Transient states normally occur within an EOB model, between pairs of concern-specific structural transactions.

### 6.1.7.4 Implementation States

Consider a refinement mapped software source code implementation, $S_L$, that satisfies a software design, $S_H$, which in turn was refined from a software system's software requirements. In this thesis, the equivalence between externally observable signals is assumed to hold between all three levels of abstraction; that is, the source code, the design and the requirements. Generally, the number of source code implementation states is far greater than the number of observable requirements states. Therefore, the number of quiescent states (i.e., EoB entry and exit points) is always less than the total number of possible source code implementation states through which an operational software system actually transitions. This fact is one of the general benefits of the model-based greybox monitoring over conventional whitebox approaches (Section 2.8.4).

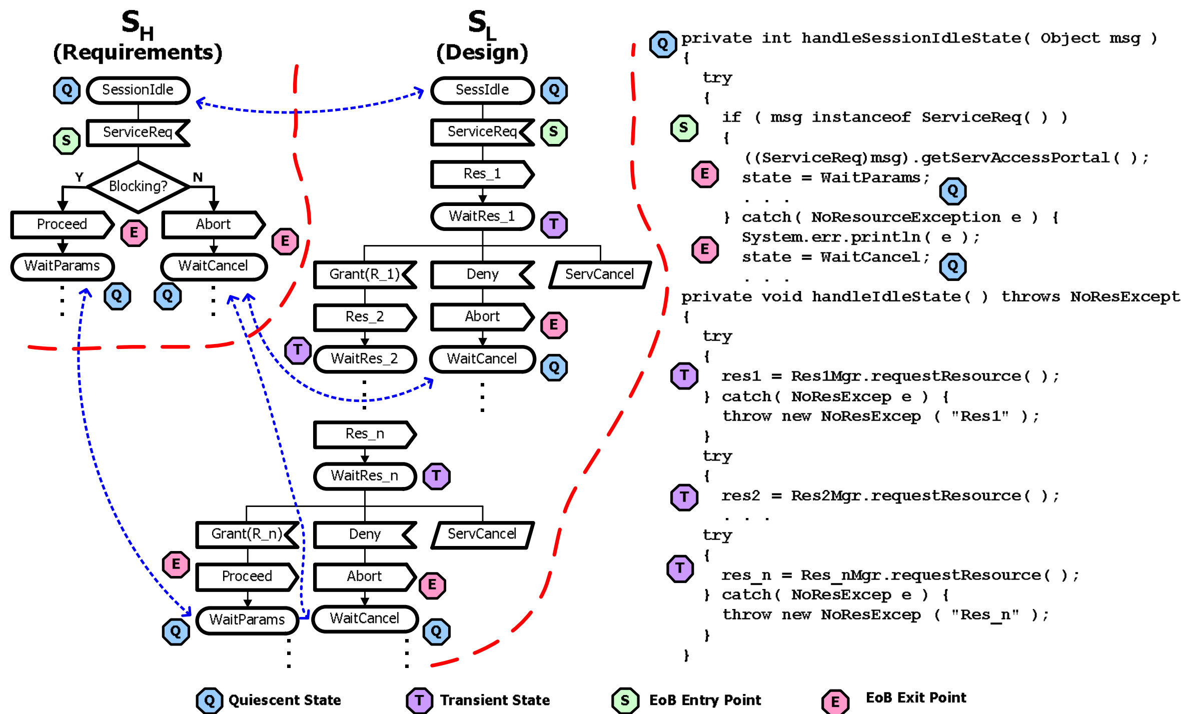



**Figure 6-5: Example mappings between $S_H$ and $S_L$.**

# Chapter 7

# Concern-Specific, Dynamic Software Structure Monitor

*"Good engineering is about ruthlessly eliminating known ways of causing failures."*
- L. Hatton, 2001.

## 7.1 Introduction

This chapter presents a detailed description of the proposed *greybox* approach to *concern-specific*, *dynamic software structure monitoring* for interactive session-oriented services. The monitor provides readers with an illustration of the use of ontologies for runtime monitoring, in general and the specific practical use of ResOwn for monitoring the evolving resource ownership structure of interactive session-oriented services. The first section of this chapter describes the monitor's architecture and internal organization. The next section defines the syntax and semantics of the monitoring commands produced by the instrumented target and the model constructs added to the monitor's specification-based monitoring model. These model constructs direct the monitor's interpreter on how to process the incoming monitoring commands. Finally, two algorithms are presented for deriving the specification-based monitoring model from the target's SDL software requirements and design specifications. In this thesis, the selected software concern is *application resource ownership structure* and the selected interaction session-oriented service is the call processing software of a *small private branch exchange (PBX)*. A *session-oriented model of computation (SOMOC)* for interactive services is presented in Chapter 6. A detailed description of the PBX is provided in Chapter 2.

## 7.2 The Monitor

The presented monitoring approach is intended to detect and report certain concern-specific structural errors in interactive session-oriented services that are delivered by real-time software systems. An organizational block diagram of the monitor is presented in Figure 7-1. The monitor executes as a separate unit and is comprised of a *greybox inte*rpreter, a *dynamic knowledge base* and a *pattern matcher*. In addition, monitoring relies on a number of derived, application-specific models including a *state evolution model*, an *EoB models library* and a *software sensor instrumentation plan (sensor plan)*. An application-specific *monitoring interface* of software sensors is woven into the target source code implementation in accordance with the sensor plan. The monitoring interface produces and transports monitoring commands at runtime from the instrumented target to the monitor's greybox interpreter.

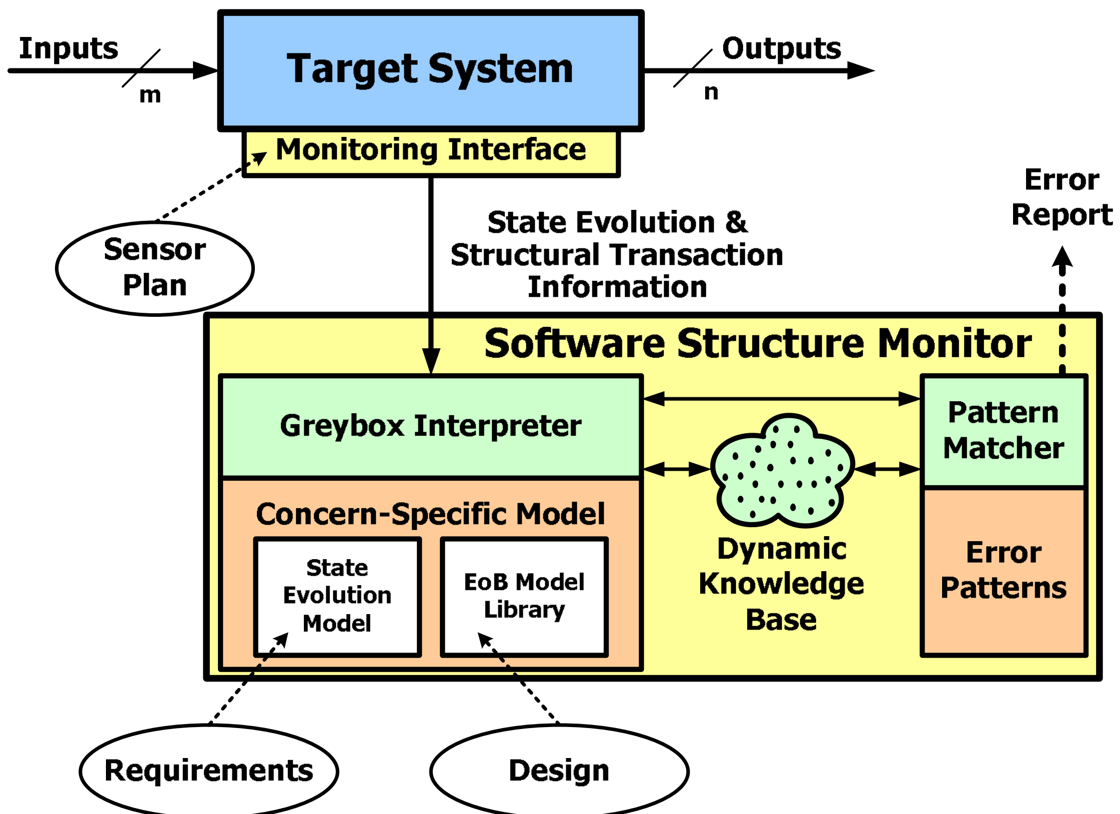



**Figure 7-1: Organizational block diagram of monitoring approach.**

Monitoring is divided into two main stages.

- During the *tracking stage*, the greybox interpreter receives, processes and stores timestamped monitoring commands that are produced and delivered from the operational target via the monitoring interface. The runtime knowledge is used to track both the specification state and the evolving software structure of the operational target on a session-by-session basis. The interpreter uses the runtime knowledge from the monitoring commands, in conjunction with the concern-specific monitoring models, to maintain the contents of the dynamic knowledge base.
- During the *detection phase*, the pattern matcher checks the consistency of the dynamic knowledge base as well as matches known error patterns against all or some of the contents of the dynamic knowledge base. These error patterns provide a declarative way to specify precise structural configurations and constraints.

The two-phase monitoring approach allows the pattern matcher to perform consistency checks offline on a representation of the operational target's evolving software structure separately from the interpretation process. Detected inconsistencies are reported as structural errors. For the selected application resource ownership concern, the tuples stored in the dynamic knowledge base represent:

- **Consumer**, **Supplier** and **Resource** instances that correspond to actual objects in the operational target software system.
- Proof of ownership **Instrument** instances that correspond to sets of active resource ownership links currently in effect in the dynamic software structure of the operational target.

#### 7.2.1.1 Behavioral Considerations

Although the monitor requires some behavioral knowledge from the operational target, the approach does not consider behavioral correctness nor directly detect behavioral errors. The *state evolution model* and *EoB models* are only intended to act as roadmaps for tracking the evolving software structure of the operational target. Although the dynamic software structure monitor may complement behavioral monitoring, it is not intended to replace existing behavioral monitoring approaches.

### 7.2.2 The Greybox Interpreter

The organization of the *greybox interpreter* shown in Figure 7-2 is loosely based on the ITU-T's *Abstract SDL Machine (ASDLM)* [Bel91, Itu91] and consists of a number of synchronously

communicating meta-processes. The system meta-process handles the creation and termination of CEFSM-instances. The global-time meta-process acts like a wall clock. Each interpretable meta-process CEFSM-instance represents the logical state of a corresponding session so that the number of CEFSM-instances is equal to the number of session access portals. The interpreter notifies the pattern matcher whenever a session, or a set of sessions, is in a quiescent state (Chapter 6).

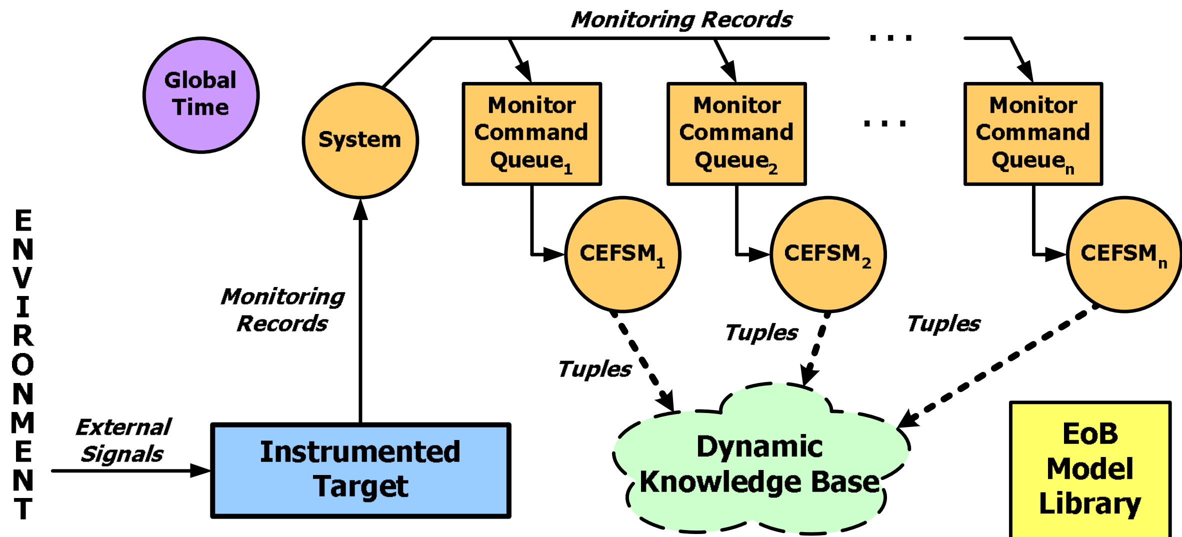



**Figure 7-2: Internal organization of the greybox interpreter.**

### 7.2.2.1 Interpreter Extensions

The greybox interpreter extends and/or adapts the basic ASDLM in a number of ways.

- The system meta-process handles the routing of monitoring commands (rather than signals) from the monitoring interface to the appropriate CEFSM-instance's monitoring command queue.
- The *delaying path process*, used for signals traversing channels, is removed to eliminate signal ordering issues due to nondeterministic channel delay.
- All the CEFSM signal input queues are replaced with CEFSM *monitor command queues*. A CEFSM-instance may only consume a monitoring command from the queue if and only if the command satisfies an enabling condition, effectively disabling implicit transitions.
- The *EoB library* is added to store the derived epoch of behavior specifications. When the enabling condition is satisfied, a monitoring command is consumed and the next *EoB entry point*

is deemed to have been reached. If the next EoB differs from the current EoB, the interpreter automatically unloads the current EoB and loads the next EoB. If the next EoB is the same as the current EoB, the current EoB remains unchanged.

- A *tuple-based dynamic knowledge base* is added to store runtime knowledge. When a monitoring command is consumed, a monitoring construct embedded into the associated fired transition of the monitoring model consumes the monitoring command. The interpreting the monitoring construct directs the interpreter on how to encode the runtime knowledge from the consumed monitoring command into a tuple. The resulting tuple is then delivered or removed from the dynamic knowledge base, depending on the type of monitoring command.

### 7.2.3 The Pattern matcher

This section describes only some *initial ideas* on the architecture and organization of the *pattern matcher*. One possible architecture for the pattern matcher is presented in Figure 7-3.

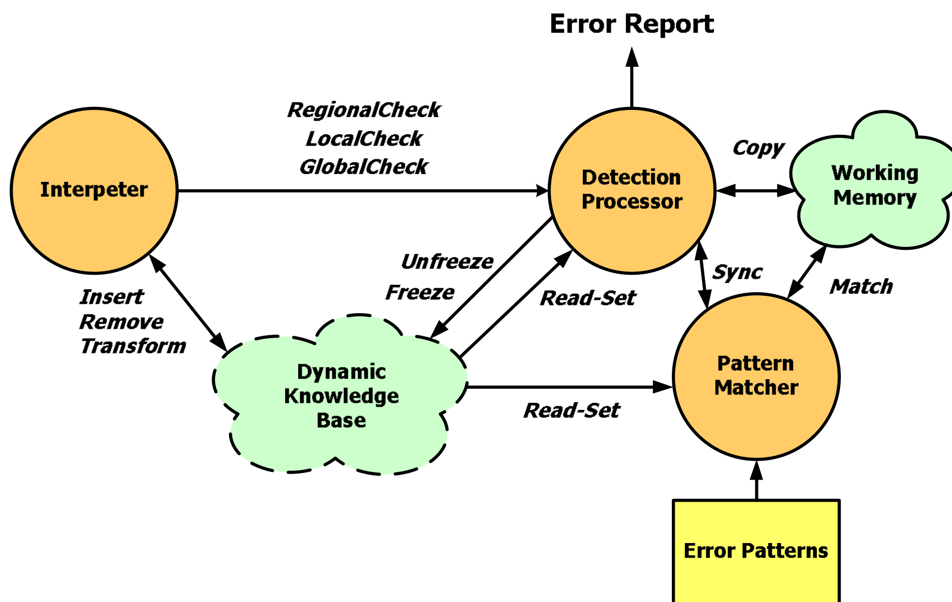



**Figure 7-3: One possible internal organization for the structural pattern matcher.**

The focus of this thesis is on *tracking* the evolving software structure of an operational software system. Some possible manually-derived error detection scenarios were previously described (Chapter 5). However, a *complete* set of error detection and reporting algorithms is beyond the scope

of this thesis and left as a matter for future work. The matcher consists of a number of synchronously communicating meta-processes whose collective purpose is to match state-dependent error patterns and detect inconsistencies in the contents of the dynamic knowledge base. The components are a reusable *detection processor*, a reusable *pattern matcher*, a random access *working memory* and an *error pattern library*. Detected inconsistencies are reported as structural errors. An *invariant* is a declarative, formal constraint that precisely states some condition(s) that must always be obeyed by any object configuration [War98]. The use of the word "always" with regard to invariants is not necessarily suitable for runtime monitoring as invariants may be violated during intermediate computational stages. Therefore, the presented approach assumes that the contents of the dynamic knowledge base can only be verified at specific points in time to ensure invariants are being enforced correctly. An example of error detection points was given in Figure 6-4.

### 7.2.4 The Dynamic Knowledge Base

The *dynamic knowledge base* is at the heart of the monitor and acts as a *dual-ported*, random-access, *tuple space*[1] for both the interpreter and pattern matcher. As shown in Figure 7-4, the dynamic knowledge base is conceptually viewed as three functional components: an *input queue*, a *processor* and a *tuple space*. The interpreter sends tuples to the input queue via the synchronous ***Deliver*** signal in the form of a dynamic operation (Section 7.2.4.1). The pattern matcher is capable of controlling when the dynamic knowledge base is updated using the synchronous ***Freeze*** and ***Unfreeze*** signals. If the dynamic knowledge base is frozen, the interpreter's delivered dynamic operations simple queue up in the FIFO input queue until the dynamic knowledge base is unfrozen by the pattern matcher.

#### 7.2.4.1 Dynamic Operations

The processor and tuple space support these simple, non-blocking operations.

- The **OUT(tuple)** operation adds the specified tuple to the tuple space. If a matching tuple already exists, the existing tuple is overwritten and a warning delivered to the pattern matcher, which analyzes the old and new tuple contents and then decides whether or not to report an error.

---

[1] An implementation of an associative memory paradigm originally proposed for parallel and distributed computing that provides a concurrently accessible repository for collections of logically-ordered data sets called tuples [Gel85, Sto05].

- The **IN(tuple)** operation removes the specified tuple from the tuple space, if it exits. The operation searches the tuple space. If a match is found, it is removed and returned. If a matching tuple is not found, a *NULL_TUPLE* is returned and a warning delivered to the pattern matcher to analyze and decide whether or not to report an error.
- The **IN(tuple-set)** operation works the same as the **IN(tuple)** operation except that a wildcard is used during the search to remove a matched tuple-set from the tuple space. If a *NULL_TUPLE* is returned, no warning is generated.
- The **READ(tuple**) and **READ(tuple-set)** operation are nondestructive **IN** operations.

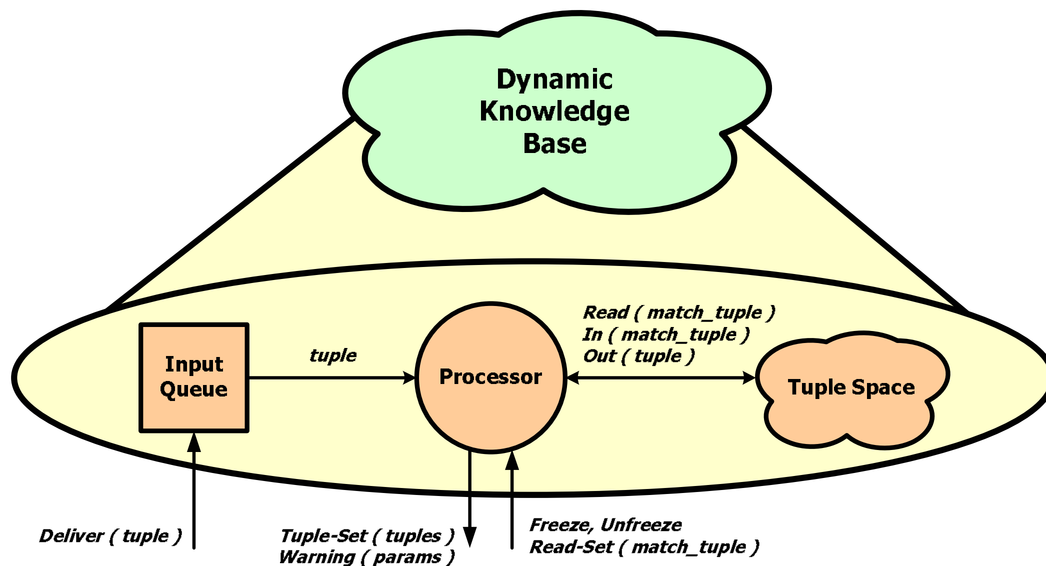



**Figure 7-4: Internal organization of the dynamic knowledge base.**

### 7.2.4.2 Tuple Lifespan and Persistency

Every tuple has a *lifespan* determined by a *timestamp* and predefined *persistency*. The timestamp provides the tuple with a *time of birth*. The persistency provides the tuple with an anticipated *time of death*. The use of a *maximum lifespan* is necessary both to detect tuples that are removed from the state space prematurely, and to prevent orphaned tuples from filling up in the dynamic knowledge base. For the application resource ownership concern, tuples representing **Consumer**, **Supplier**, **Resource** and **Base Instruments** instance are *system persistent*. A *system persistent tuple*, once created and delivered to the dynamic knowledge base, remains in the tuple space for the runtime life

of the operational target[1]. Tuples representing **Extent Instruments** instances are session-persistent and are only supposed to remain in effect for the maximum duration of the current session.

### 7.2.5 Behavioral versus Structural Considerations

Although the greybox monitoring approach tracks the specification state of the operational target, the approach does not attempt to validate the target's trajectory against the software requirements or design specifications to ensure behavioral correctness. This approach is analogous to automatically monitoring the structural integrity of a supersonic jet without regard as to whether the plane is actually on a correct heading, or if the jet will arrive at the prescribed destination on time. With regard to the dynamic software structure monitoring, the monitor tracks the changing specification state of the operational target only as it pertains to providing a state-dependent context for checking the resource ownership structure of the target, without consideration as to whether the resources involved provide their associated benefits correctly, which is considered to be a behavioral monitoring issue.

## 7.3 Monitoring Commands and Monitoring Constructs

The proposed monitoring approach is greybox and relies on embedded software sensors that, when encountered by the operational target, produce one of two types of monitoring:

- An *EoB entry or exit monitoring* command that allows the monitor to track the trajectory of the operational target's quiescent states on a session-by-session basis.
- An *structural transaction monitoring command* that allows the monitor to track the micro-steps of evolving software structure of the operational target for some selected structural concern where one structural transaction equals one micro-step.

The monitoring approach is also model-driven and relies on monitoring constructs that is a model construct embedded into certain transitions in both the state evolution model and the EoB models. Each monitoring construct type corresponds to a specific monitoring command type and each type of

---

[1] As stated previously, all resource roles in this research work are assumed to be nonconsumable.

monitoring command is consumable only by its specific type of monitoring construct. A monitoring construct guides the greybox interpreter in the consumption of the specific, corresponding monitoring command, the creation of a tuple for delivery to the dynamic knowledge base, and the unloading or loading of the current and next EoB model from the EoB library, respectively.

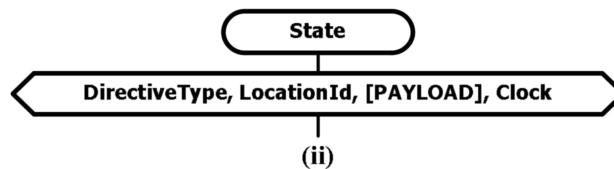
## 7.3.1 Monitoring Command Types

Monitoring commands are of the form ( $\boldsymbol{A_i}$, $\boldsymbol{D_i}$, $TS_i$ ) where: $\boldsymbol{A_i}$ is the command's addressing information, $\boldsymbol{D_i}$ is the command's payload and is comprised of data arguments ( $\boldsymbol{d_1}$, ..., $\boldsymbol{d_n}$ ), and $\boldsymbol{TS_i}$ is the command's logical timestamp. The logical timestamp is issued by the monitoring interface and is used to both preserve order inside the interpreter and to advance the logical clock in the consuming CEFSM-instance. Consider the generic monitoring command, as shown in Figure 7-5(i).

- **CommandType** determines the format and semantics of the command payload.
- **ObjectId** indicates the unique object from the operational target in which the producing software sensor is embedded and tell greybox interpreter's system meta-process to which CEFSM-instance's monitor command queue the monitoring command is to be delivered.
- **LocationId** indicates the unique location of the specific monitoring construct within the CEFSM-instance specified by the ObjectId.
- An optional **Payload** contains the additional runtime knowledge required by the monitoring construct to create a corresponding tuple.

**(CommandType, LocationId, ObjectId, [PAYLOAD], Timestamp)**

**(i)**




**(ii)**

**Figure 7-5: Generic: (i) monitoring command; (2) monitoring construct.**

## 7.3.2 Monitoring Construct Types

Consider the generic monitoring construct presented in Figure 7-5(ii). Each monitoring construct acts as an enabling condition on the transition leading from the preceding state. As a result, an enclosing CEFSM-instance can only fire the transition if and only if an appropriate monitoring command, as specified by the monitoring construct, is present in the CEFSM-instance's monitor command queue. Consider the contents of a generic monitoring construct, as shown in Figure 7-5(ii):

- **ConstructType** determines the monitoring command type that the monitoring construct consumes as well as the format and semantics of the construct's payload.
- **LocationId** indicates the unique location of the monitoring construct within the enclosing CEFSM-instance.
- An optional **Payload** contains the additional runtime knowledge required by the monitoring construct to create a corresponding tuple.
- Clock is the logical clock sort of the enclosing CEFSM-instance.

## 7.3.3 Sensor Plan

As shown in Figure 7-6, the same state and signal details (i.e., EoB entry and Exit points) that are used to derive the state evolution model and the EoB models are also used to derive the sensor plan. The application-specific sensor plan specifies the location and monitoring command format for the set of embedded software sensors that is to be woven into the target implementation. An entry in the sensor plan contains (1) a sensor's unique location in the implementation, (2) the sensor's corresponding monitoring construct's unique location in the interpretable model, and (3) the sensor's monitoring command format (i.e., type and payload). Implementing the resulting sensor plan allows the monitoring interface to capture (1) whenever the operational target enters a new (or reenters the current) EoB model, (2) whenever the operational target exits the current EoB model, and (3) whenever the operational target successfully completes a concern-specific structural transaction. For this research work, target instrumentation was accomplished using a manual procedure. An automated approach to embedding software sensors using weaving technology is the subject of future work.

### 7.3.3.1 State- versus Input-Oriented Implementation Structure

For new software, instrumentation can be built into the development process. However, for legacy

code, the assumption that the software implementation is refined from the CEFSM-based software design has these implications on the source code structure of the software implementation:

- A *state-oriented implementation* is implemented using a **STATE-MESSAGE-ACTION** source code structure such that for each specification state, the operational software system may only consume a predefined set of possible inputs. The fired implementation state transition depends on the current specification state and the particular consumed input, implying that the source code structure is organized according to the same actual structure of the corresponding CEFSM in the software design. An example of state-oriented instrumentation is provided in Appendix C.
- An *input-oriented implementation* is implemented using a **MESSAGE-STATE-ACTION** source code structure such that a given message may only be consumed if the operational software system is in one of a predefined set of specification states. The current state transition depends on both the input and current specification state. In general, the state transition and input combination does not typically follow the structure of the corresponding *communicating extended finite state machines (CEFSM)* specified in the software design make the embedding process less straightforward. An example of input-oriented instrumentation is provided in Appendix C.

# 7.4 Deriving Interpretable Models

The greybox monitoring approach relies on a number of interpretable models that are formally derived from the target software system's formal specifications, as given in Figure 7-6. The interpretable state evolution model is derived from the target's software requirements. The target's software design is assumed to be a refinement of the software requirements. The quiescent states and input signals used to identify each EoB entry and exit point in the software requirements will also demarcate the required EoB models in the software design. Each EoB model contains the required structural transaction for the selected structural concern. The derivation of the state evolution model and EoB model from a common frame of specification reference allows the greybox interpreter to simultaneously monitor the operational target at two distinct, but related levels of abstraction (i.e., requirements and design). Therefore, the greybox interpreter employs a *novel* bi-level, monitoring approach in which the evolving specification state of the operational target is tracked using a requirements-based model while the evolving software structure of the operational target is tracked for a selected software concern using a design-based model.

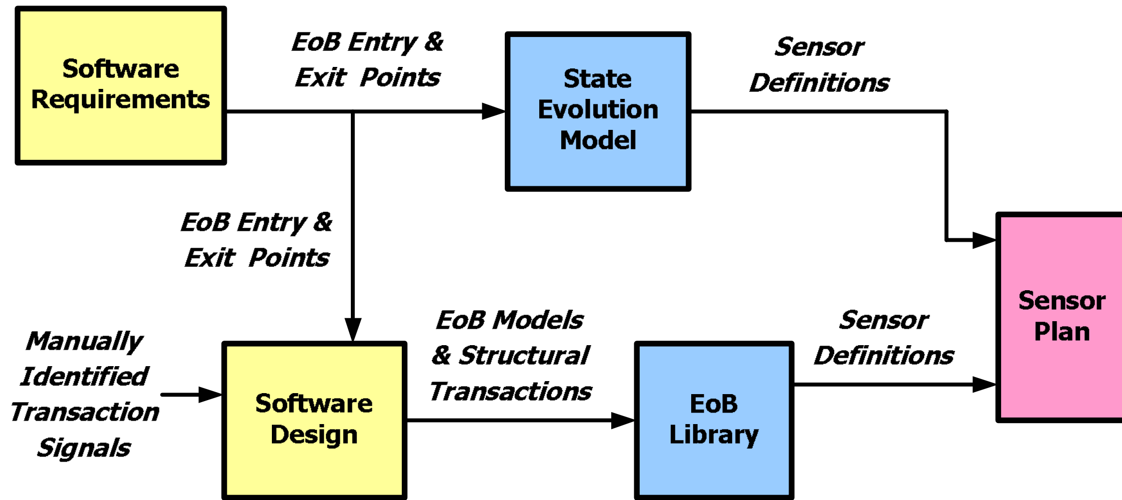



**Figure 7-6: Block diagram of interpretable models derivation process.**

## 7.4.1 Structural Reduction

The concept of structural model reduction, where a model with a smaller state-space footprint is derived from an existing model, has been previously used in protocols, interface design, and model derivation to reduce model complexity and the likelihood of encountering nondeterminism [Floc03, Peh83, Pek03]. In this work, the interpretable model derivation process includes steps to reduce complexity through a structural reduction process called pruning. Pruning strategically removes those specification details deemed unnecessary to the intended monitoring process. Before pruning, model details must be categorized as either external, significant internal and insignificant internal based on the notion of signal observability.

### 7.4.1.1 Classifying CEFSMs and Signals

Observability pertains to the amount of implementation details visible behind a software interface [Szp02]. Blackbox and whitebox testing validate observable functionality and unobservable structures and interactions, respectively [Gao03]. Consider the example message sequence chart and block excerpt in Figure 7-7:

- Signals *ServiceRequest* and *ServiceProceed* are observable at the environment and classified as *external signals*. $\boldsymbol{CEFSM_1}$ and $\boldsymbol{CEFSM_2}$ are classified as external because each has external signals that travel to and from the environment over channels ***C1/C3*** and ***C2/C4***, respectively.

External signals may be used to demarcate EoB models.

- *Dominant internal signals* travel between external ***CEFSM₁*** and ***CEFSM₂*** over signalroutes ***R1***/***R2***. Dominant internal signals may be used to demarcate EoB models.
- ***CEFSM₃*** is classified as *internal* because the instance does not directly communicate with the environment. Recessive internal signals *ResourceRequest* and *ResourceGrant* travel between external ***CEFSM₃*** and internal ***CEFSM₁*** over signalroutes ***R3***/***R4***. *Recessive internal signals* are pruned from the state evolution model, but used in EoB models to demarcate the successful completion of structural transactions for the selected structural concern.
- *Insignificant internal signals SetQueue* and *Ack* travel between internal ***CEFSM₃*** and internal ***CEFSM₄*** over signalroutes ***R5***/***R6***. *Insignificant internal signals* are pruned from both the state evolution model and EoB models.

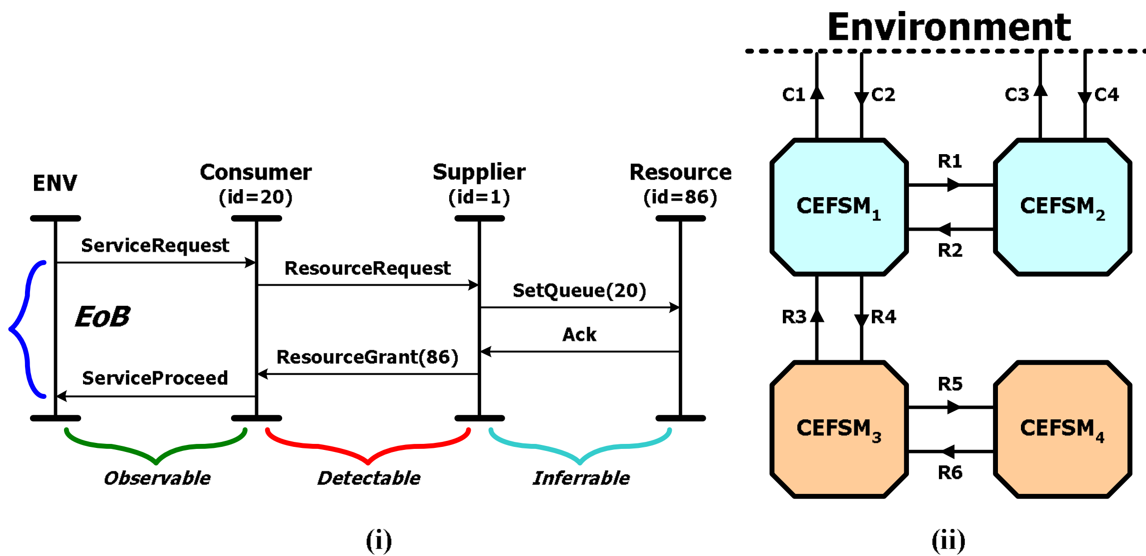



**Figure 7-7: Example: (i) message sequence chart; (ii) layered CEFSMs and signals.**

## 7.4.2 State Evolution Monitoring

There are two state evolution model monitoring command types, as shown in Figure 7-8: (1) an EoB entry monitoring command type, and (2) an EoB exit monitoring command type. *State evolution model* monitoring command types do not have a *payload*. The **ENTER** (**EXIT**) *monitoring command type* is used to inform the interpreter whenever the operational target **ENTER** (**EXIT**) a particular

EoB model. The **ObjectId** is used by the interpreter to route the monitoring command to the appropriate monitor command queue. The **LocationId** is used by the interpreter to match the monitoring command to the appropriate monitoring construct. If the interpreter consumes an EoB entry monitoring command that does not match the current EoB model, then the interpreter automatically loads the required EoB model from the EoB library. If the next EoB model is the same as the current EoB model (i.e., *looping*), loading the new EoB model is not required.

**( ENTRY, LocationId, ObjectId, Timestamp )**

**( EXIT, LocationId, ObjectId, Timestamp)**

**Figure 7-8: EoB ENTRY and EoB EXIT monitoring command.**

#### 7.4.2.1 State Evolution Model Derivation

The derivation of the state evolution model is general in that the same algorithm is applicable regardless of the selected structural concern. This section presents the algorithm for deriving the structurally reduced state evolution model from the target software system's software requirements specification. The derivation process may be automated using the state evolution model derivation algorithm. A line-by-line description follows the algorithm.

```
1.  ALGORITHM: Derive-State-Evolution-Model
2.  INPUT: Requirements-Specification
3.  OUTPUT: State-Evolution-Model
4.  COPY Requirements-Specification to State-Evolution-Model;
5.  FORALL ( Channel_i ∈ State-Evolution-Model ) DO:
6.      FORALL ( Signal_j ∈ Channel_i Signal List ) DO:
7.          TAG Signal_j = External;
8.      ENDFOR;
9.      FORALL ( CEFSM_j ∈ State-Evolution-Model ) DO:
10.         TAG CEFSM_j = Internal;
11.         IF ( Channel_i connected-to CEFSM_j ) OR
12.            ( Channel_i connected-from CEFSM_j ) THEN:
13.             TAG CEFSM_j = External;
14.             BREAK;
15.         ENDIF;
16.     ENDFOR;
17. ENDFOR;
18. FORALL ( CEFSM_i ∈ State-Evolution-Model ) DO:
19.     IF ( READTAG ( CEFSM_i ) == Internal ) THEN:
20.         PRUNE ( CEFSM_i );
21.     ENDIF;
22. ENDFOR;
```

```
23. FORALL ( Timer_i ∈ State-Evolution-Model ) DO:
24.     FORALL ( Signal_j ∈ Timer_i Signal List ) DO:
25.         TAG Signal_j = Significant;
26.     ENDFOR;
27. ENDFOR;
28. FORALL ( SRoute_i ∈ State-Evolution-Model ) DO:
29.     TAG SRoute_i = Insignificant;
30.     FORALL ( CEFSM_m ∈ State-Evolution-Model ) DO:
31.         FORALL ( CEFSM_n ∈ State-Evolution-Model ) DO:
32.             IF ( SRoute_i connected-from CEFSM_m ) AND ( SRoute_i connected-to CEFSM_n ) THEN:
33.                 TAG SRoute_i = Significant;
34.                 BREAK;
35.             ENDIF;
36.         ENDFOR;
37.     ENDFOR;
38.     FORALL ( Signal_j ∈ SRoute_i Signal List ) DO:
39.         TAG Signal_j = READTAG ( SRoute_i );
40.     ENDFOR;
41. ENDFOR;
42. FORALL ( SRoute_i ∈ State-Evolution-Model ) DO:
43.     IF ( READTAG ( SRoute_i ) == Insignificant ) THEN:
44.         PRUNE ( SRoute_i );
45.     ENDIF;
46. ENDFOR;
47. FORALL ( CEFSM_i ∈ State-Evolution-Model ) DO:
48.     FORALL ( State_j ∈ CEFSM_i ) DO: Q
49.         TAG State_j = Transient;
50.         FORALL ( Input_k ∈ State_j ) DO:
51.             IF ( READTAG ( Input_k.Signal ) == External ) OR
52.                  ( READTAG ( Input_j.Signal ) == Significant ) THEN:
53.                 TAG State_j = Quiescent;
54.                 BREAK;
55.             ENDIF;
56.         ENDFOR;
57.     ENDFOR;
58. FORALL ( Construct_j ∈ CEFSM_i ) DO:
59.         IF ((( Construct_j.Type == Input ) AND
60.              ( READTAG (Construct_j ) == Insignificant )) OR
61.              ( Construct_j.Type == Output ) OR
62.              ( Construct_j.Type == Task ) OR
63.              ( Construct_j.Type == Timer ) THEN:
64.             PRUNE ( Construct_j );
65.             JOIN-TRANSITION ( Construct_j-1, Construct_j+1 );
66.         ELSEIF ( Construct_j.Type == Save ) THEN:
67.             PRUNE (Construct_j );
68.         ELSEIF ( Construct_j.Type == Decision ) THEN:
69.             SET Construct_j.Type = State;
70.             TAG Construct_j = Transient;
71.         ENDIF;
72.     ENDFOR;
73.     FORALL ( Construct_j ∈ CEFSM_i ) DO:
74.         IF (( Construct_j.Type == State ) AND
75.              ( READTAG ( Construct_j ) == Transient ) AND
76.              ( Construct_j+1.Type == State ) AND
77.              ( READTAG ( Construct_j+1 ) == Transient )) THEN
78.             PRUNE ( Construct_j );
```

```
79.             JOIN-TRANSITION ( Construct_j-1, Construct_j+1 );
80.         ENDIF;
81.     ENDFOR;
82.     FORALL ( Construct_j ∈ CEFSM_i ) DO:
83.         IF (( Construct_j.Type == State ) AND ( READTAG ( Construct_j ) == Quiescent )) THEN
84.             FORALL ( Input_k ∈ Construct_j ) DO:
85.                 REPLACE ( Input_k, generateMonitorConstruct( Entry, j+k ));
86.             ENDFOR;
87.             IF ( Construct_j-1.Type == State AND READTAG( Construct_j-1 ) == Transient ) THEN
88.                 INSERT ( Construct_j, generateMonitorConstruct( Exit, j ) );
89.             ELSEIF:
90.                 Newconstruct.Type := State;
91.                 TAG Newconstruct = Transient;
92.                 INSERT ( Construct_j, Newconstruct );
93.                 INSERT ( Construct_j, generateMonitorConstruct( Exit, j ) );
94.             ENDIF;
95.         ENDIF;
96.     ENDFOR;
97. ENDFOR;
98. RETURN ( State-Evolution-Model );
```

In lines 1-4, the input and output to the algorithm is the target's CEFSM-based software requirements specification and the state evolution model, respectively. The initial *state evolution model* is set to a copy of the software requirements specification. Lines 5-17 deal with *external signals* and CEFSMs. In lines 5-8, every signal belonging to a channel's signal list is tagged as an External signal. In line 9-16, every CEFSM connected to a channel is tagged as External CEFSM in the model. In lines 18-22, every CEFSM tagged as Internal is pruned from the model. Lines 23-27 deal with timer signals. Every signal belonging to a timer's signal list is tagged as a Significant internal signal in the model. Lines 28-41 deal with internal signals. In lines 28-37, a signalroute connected between a pair of external CEFSMs is tagged as a Significant signalroute in the model; otherwise the signalroute is tagged as an Insignificant signalroute. In lines 38-41, a signal is tagged as a Significant signal if the signal belongs to a Significant signalroute's signal list in the model; otherwise, the signal is tagged as an Insignificant signalroute. In lines 42-46, every signalroute tagged as Insignificant is pruned from the model. Lines 47-57 deal with identifying Transient state or a Quiescent state. If a state is followed by a signal tagged as either External or Significant, then the state is tagged as a Quiescent state in the model; otherwise, the state is tagged as a Transient state. Lines 58-72 deal pruning from, and replacing insignificant constructs in the state evolution model. In lines 59-65, if the current construct under consideration is either an input with a signal identifier that was tagged as Insignificant, an output construct, a task construct or a timer construct, then the construct is pruned from the model and the transition between the preceding and succeeding constructs is joined. In lines 66-67, if the current construct is a save construct, it is pruned from the

model. In lines 68-71, if the current construct is a decision construct, then the decision is replaced with a state construct that is tagged as a Transient state in the model. Lines 73-81 deal with collecting succeeding transient states into a single transient state. In line 78, each predecessor transient state in a chain of transient states is pruned until only one transient state remains from the original chain. Lines 82-96 deal with embedding EoB Entry and Exit monitoring constructs into the structurally reduced, state evolution model. In lines 83-86, every input construct that follows each quiescent state is replaced with a unique EoB Entry monitoring construct. In lines 87-89, a unique EoB Exit monitoring construct is inserted immediately preceding the quiescent state and after an existing transient state. In lines 90-96, no transient state precedes the quiescent state and therefore a unique EoB Exit monitoring construct and transient state combination is inserted immediately preceding the quiescent state. In line 98, the resultant structurally reduced state evolution model is returned.

#### 7.4.2.2 EoB Entry Point Monitoring Scenario

Consider the EoB entry point monitoring scenario presented in Figure 7-9. In the requirements excerpt, as shown in Figure 7-9(i), the transition from state ***S1*** is initiated after input signal ***A*** is consumed by CEFSM-instance ***21***. The state evolution model derivation replaces signal ***A*** by an entry monitoring construct ***N01***, as shown in Figure 7-9(ii). The transition from state ***S1*** is then fired if and only if monitoring command ***N01*** is consumed. The interpreter loads and begins interpreting the EoB model associated with ***N01***. The CEFSM-instance's logical clock advances to ***10***.

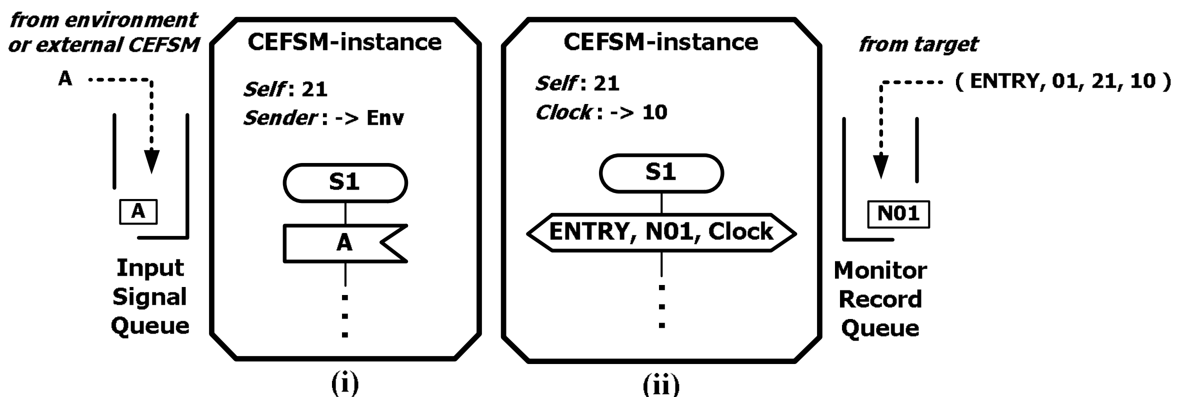



**Figure 7-9: EoB entry point monitoring scenario.**

### 7.4.2.3 EoB Exit Point Monitoring Scenario

Consider the EoB exit point monitoring scenario presented in Figure 7-10. In the software requirements excerpt, as shown in Figure 7-10(i), the transition from state ***S3*** is initiated after input signal ***C*** is consumed by CEFSM-instance ***34***. The flag in the decision constructs determines whether the behavioral alternative ending in state ***S4*** or state ***S5*** is chosen. The state evolution model derivation process replaces the decision construct with transient state ***NS1*** as shown in Figure 7-10(ii), followed by two exit monitoring constructs, ***X01*** and ***X02***, inserted into the transitions leading from transient state ***NS1*** to quiescent states ***S4*** and ***S5***, respectively. The transition from state ***S3*** is fired when ***N03*** is consumed. The interpreter loads and interprets the EoB model for ***N03***. When interpretation of the current EoB model completes, the interpreter waits in ***NS1***. Whether the operational target produces exit monitoring command ***X01*** or ***X02*** determines whether the interpreter enters the quiescent state ***S4*** or S5, respectively. In this example, the CEFSM-instance consumes ***X01*** and enters ***S4***, causing the logical clock to advance from ***100*** to ***125***.

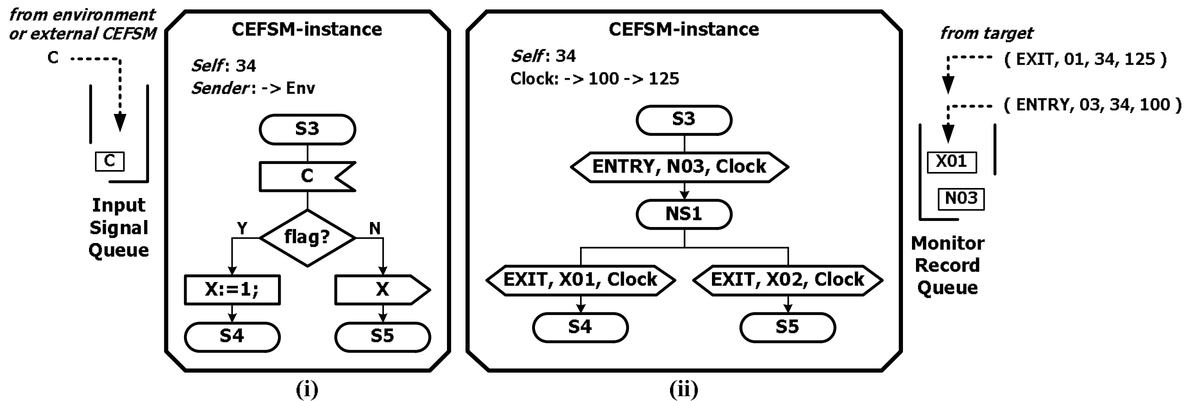



**Figure 7-10: EoB exit point monitoring scenario.**

## 7.4.3 Structural Transaction Monitoring

There are five context-sensitive monitoring command types, as shown in Figure 7-11. The **ACQUIRE**, **RELEASE**, **REGISTER**, and **UNREGISTER** monitoring commands are for reporting concern-specific structural transactions. The **INSTANCE** monitoring command is used for initialization purposes. The four structural transaction monitoring commands are each associated with monitoring constructs that were into various EoB models. Structural transaction monitoring construct

allow the greybox interpreter to handle and process the completion of certain successful structural transactions, as reported by the instrumented operational target, the selected structural concern. The **INSTANCE** monitoring command is a special and has no corresponding monitoring construct. As will be described later in this chapter, the **INSTANCE** monitoring command is handled and processed directly by the interpreter's system meta-process.

**( ACQUIRE, LocId, ConId, SupId, ResId, Timestamp )**
**( RELEASE, LocId, ConId, SupId, ResId, Timestamp )**

**( REGISTER, LocationId, ConId, SupId, $ResId_1$, ..., $ResId_n$, Timestamp )**
**( UNREGISTER, LocationId, ConId, SupId, $ResId_1$, ..., $ResId_n$, Timestamp )**

**( INSTANCE, ObjectType, ObjectId, Timestamp )**

**Figure 7-11: Structural transaction monitoring commands.**

### 7.4.3.1 Instance Monitoring Command Types

The special initialization monitoring command has no matching monitoring construct. Instead, the **INSTANCE** monitoring command directs the greybox interpreter to create the appropriate tuple, in accordance with Table 7-1, and insert or remove the resultant tuple into or from the *dynamic knowledge base*, respectively. The greybox interpreter uses an **INSTANCE** monitoring command's **ObjectType** to determine what object tuple type. The interpreter uses a static ***Object-Type-to-ResOwn-Role*** lookup table. The lookup table is derived using the following procedure:

- An application-specific *specialized ResOwn instance* is extended (i.e., specialized) by creating an implementation specific ontological class for each object class in the target software system.
- All created *application-specific* classes are then classified under ResOwn using the RacerPro reasoner on the *specialized ResOwn instance*.
- For each object tuple representing a Resource role, the greybox interpreter automatically creates and stores an associated tuple representing the named **Titledeed** instance that is logically bound to the **Resource** instance, as shown in Table 7-1.

These object and association tuples are important for processing structural transaction monitoring commands as the set of object tuples stored in the *dynamic knowledge base* is used to translate the physical object Ids from a structural monitoring command to their corresponding logical class Id. If

an object plays a **Resource** role, then the resulting logical **Resource** class *Id* is also used to locate the specific **Titledeed** class *Id* bound to that **Resource** class.

| ResOwn Role | Object Tuple Format | Defined Class | Titledeed Specialization | Association Tuple Format (tuples are timestamped) |
|---|---|---|---|---|
| Resource | RES, Id, Oid | Transferable Resource<br>Nontransferable Resource<br>Embedded Resource | TransferableTitledeed<br>NontransferableTitledeed<br>EmbeddedTitledeed | TTD, Id, Rid, Hid, Iid, Eid<br>NTD, Id, Rid, Hid, Iid, Eid<br>ETD, Id, Rid, Hid, Iid, Eid |
| Supplier | SUP, Id, Oid | PooledResourceSupplier<br>ManagedResourceSupplier<br>SurrogateResourceSupplie<br>r<br>CachedResourceSupplier<br>CompoundResource | n/a | |
| Consumer | CON, Id, Oid | Dedicated Consumer<br>Dispatchable Consumer | n/a | |

**Table 7-1: INSTANCE monitoring command.**

#### 7.4.3.1.1 Object and Association Tuples

An object tuple represents a corresponding physical object in the operational target that is specified by the **INSTANCE** monitoring command's **ObjectType** and **ObjectId**. The object tuple parameters, as shown in Table 7-1, are: *ResOwn Role Type* (**RES**, **SUP**, **CON**), *ResOwn Object Id* (*Id*), *Object Id* (*Oid*), where **RES** is a **Resource**, **SUP** is a **Supplier** and **CON** is a **Consumer**. An association tuple represents a corresponding logical proof of ownership Instrument instance maintain by the greybox interpreter. For an **INSTANCE** command, all association tuples pertain to **Titledeed** instances. The association tuple parameters, as shown in Table 7-1, are: Instrument Type (**TTD**, **NTD**, **ETD**), *InstrumentId*, *ResourceId* (***Rid***), *HolderId* (***Hid***), *IssuerId* (***Iid***), *ExtentId* (***Eid***), where **TTD** is a **Transferable Titledeed**, **NTD** is a **Nontransferable Titledeed**, and **ETD** is an **Embedded Titledeed**.

### 7.4.3.2 Structural Transaction Monitoring Command Types

Structural transaction monitoring commands have a payload. The four structural transaction monitoring commands have the following fields:

- The **ACQUIRE**, **RELEASE**, **REGISTER** or **UNREGISTER** identifies the command as a typed current structural transaction monitoring command for the current structural transaction.
- The **LocId** corresponds to a specific monitoring construct for the current structural transaction

that will consume the monitoring command.

- The **ConId** is the **ObjectId** of the object in the operational target that is playing the role of Consumer for the current structural transaction.
- The **SupID** is the **ObjectId** of the object in the operational target that is playing the role of Supplier for the current structural transaction.
- The set of one or more **ResId** are the set of **ObjectIds** of the objects in the operational target playing the role of **Resource** for the current structural transaction
- The timestamp commands the logical sequence number for the monitoring command.

#### 7.4.3.2.1 ACQUIRE Monitoring Command and Construct Type

An **ACQUIRE** command is produced by the instrumented target when the embedded software sensor corresponding to a specific ACQUIRE monitoring construct in an EoB model is encountered. The **ACQUIRE** monitoring command informs the interpreter of the successful completion of an associated structural resource ownership transaction in which a **Consumer** instance in the operational target has obtained *beneficiary ownership* of a **Resource** instance from a **Supplier** instance, as shown in Figure 7-12(i) and Figure 7-12(ii). For an **ACQUIRE** command, an association tuple is inserted into or updated in the *dynamic knowledge base* with parameters, as shown in Table 7-2: *InstrumentType* (**TTD**, **LIC**, **PRX**), *InstrumentId* (***Id***), *ResourceId* (***Rid***), *TitledeedId* (***Tid***) *HolderId* (***Hid***), *IssuerId* (***Iid***), *ExtentId* (***Eid***), where **LIC** is a **License** and **PRX** is a **Proxy**.

#### 7.4.3.2.2 RELEASE Monitoring command and Construct Type

A **RELEASE** command is produced by the instrumented target when the embedded software sensor corresponding to a specific **RELEASE** monitoring construct in an EoB model is encountered. The **RELEASE** monitoring command informs the interpreter of the successful completion of a corresponding resource ownership in which an object playing a Consumer role in the operational target has relinquished beneficiary ownership of an object playing a **Resource** role to an object playing a **Supplier** role, as shown in Figure 7-12(i) and Figure 7-12(ii). For a **RELEASE** command, an association tuple is removed from or updated in the dynamic knowledge base with parameters, as shown in Table 7-2.

| Consumer Class | Defined Supplier Class | Defined Resource Class | Instrument Class | Association Tuple (tuples are timestamped) |
|---|---|---|---|---|
| Consumer | PooledResourceSupplier | TransferableResource | TTD | TTD, Id, Rid, Hid, Iid, Eid |
| Consumer | ManagedResourceSupplier | NontransferableResource | NTD - LIC | LIC, Id, Tid, Hid, Iid, Eid |
| Consumer | CompoundResource | EmbeddedResource | ETD - PRX | PRX, Id, Tid, Hid, Iid, Eid |

**Table 7-2: ACQUIRE / RELEASE commands.**

### 7.4.3.2.3 REGISTER Monitoring Command and Construct Type

A **REGISTER** command is produced by the instrumented target when the embedded software sensor corresponding to a specific **REGISTER** monitoring construct in an EoB model is encountered. The **REGISTER** monitoring command informs the interpreter of the successful completion of an associated structural resource ownership transaction in which a Consumer instance in the operational target assigned *nonbeneficiary surrogate ownership* of a **Resource** instance to a **Supplier** instance, as shown in Figure 7-12(iii). For a **REGISTER** command, an association tuple is inserted into or updated in the *dynamic knowledge base* with parameters, as shown in Table 7-3: *InstrumentType* (**POA**, **PTH**), *InstrumentId* (***Id***), and *TitledeedId* (***Tid***). *LicenceId* (***Lid***). *ProxyId* (***Pid***), *HolderId* (***Hid***), *IssuerId* (***Iid***), where **POA** is a **Power Of Attorney** and **PTH** is a **Permit To Hold**.

| Defined Supplier Class | Consumer Class | Defined Resource Class | Instrument Class | Association Tuple (tuples are timestamped) |
|---|---|---|---|---|
| Surrogate | Consumer | TransferableResource<br>NontransferableResource<br>EmbeddedResource | TTD<br>LIC<br>PRX | POA, Id, Tid, Hid, Iid<br>POA, Id, Lid, Hid, Iid<br>POA, Id, Pid, Hid, Iid |
| Cached | Consumer | TransferableResource<br>NontransferableResource<br>EmbeddedResource | TTD<br>LIC<br>PRX | PTH, Id, Tid, Hid, Iid<br>PTH, Id, Lid, Hid, Iid<br>PTH, Id, Pid, Hid, Iid |

**Table 7-3: REGISTER / UNREGISTER command types.**

### 7.4.3.2.4 UNREGISTER Monitoring Command and Construct Type

An **UNREGISTER** command is produced by the instrumented target when the embedded software sensor corresponding to a specific **UNREGISTER** monitoring construct in an EoB model is encountered. The **UNREGISTER** monitoring command informs the interpreter of the successful completion of an associated structural resource ownership transaction in which a **Consumer** instance in the operational target has relinquished *nonbeneficiary surrogate ownership* **Resource** instance

from a **Supplier** instance, as shown in Figure 7-12(iii). For a **UNREGISTER** command, an association tuple is inserted into or updated in the *dynamic knowledge base* with parameters, as shown in Table 7-3.

#### 7.4.3.3 Structural Transaction Signaling

Consider the message sequence charts presented in Figure 7-12. For the selected structural software concern, there are several main types of *structural transaction signaling* considered. As shown, each structural transaction signal of interest travels between a **Consumer** instance and **Supplier** instance. Normally, it is the type of **Supplier** instance that ultimately determines which type of monitoring command should be produced by the corresponding code in the instrumented target as well as which type of monitoring construct should be embedded in the corresponding EoB model. The one exception is the scenario presented in Figure 7-13 where the presence of a **Compound Resource** instance predicates the need to also consider the **Embedded Resource** instance(s) contained there within. The ResOwn signal set to identify successful structural resource ownership transaction from includes:

- For a **Consumer**, a **Pooled Resource Supplier** and a **Transferable Resource**, as shown in Figure 7-12(i), the *Grant* signal and the *Return* signal. The corresponding monitoring command / construct types are **ACQUIRE** and **RELEASE**, respectively.
- For a **Consumer, Managed Resource Supplier** and a **Nontransferable Resource**, as shown in Figure 7-12(ii), the *Activate* signal and the *Deactivate* signal. The corresponding monitoring command / construct types are **ACQUIRE** and **RELEASE**, respectively.
- For a **Consumer**, **Surrogate Resource Supplier** and a **Resource**, as shown in Figure 7-12(iii): the *Start* signal and the *Stop* signal. The corresponding monitoring command / construct types are **REGISTER** and **UNREGISTER**, respectively.

Consider the scenario, as given in Figure 7-13, for a **Consumer**, a **Pooled Resource Supplier**, a **Surrogate Resource Supplier** and a **Transferable Compound Resource** that contains an **Embedded Resource**. In this scenario, the *grant* from the **Pooled Resource Supplier** to the **Consumer** causes two **ACQUIRE** monitoring commands to be produced: (1) one for the **Transferable Compound Resource**, and (2) one for the **Embedded Resource** that is contained inside the **Compound Resource**. As shown, the **Embedded Resource** can be registered with a **Surrogate Resource Supplier**, as indicated by the **REGISTER** monitoring command, independently

of the **Transferable Compound Resource**. When the **Transferable Compound Resource** is finally returned to its original **Pooled Resource Supplier**, two **RELEASE** monitoring commands are produced, one each for the **Transferable Compound Resource** and the **Embedded Resource**.

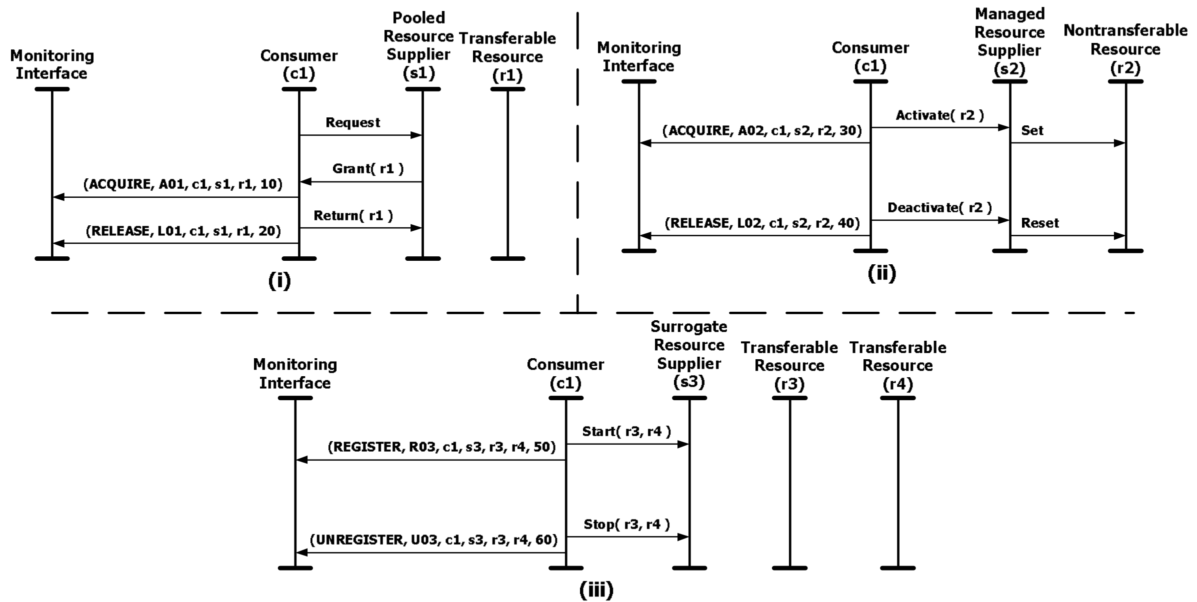



**Figure 7-12: MSC for Consumer and Supplier.**

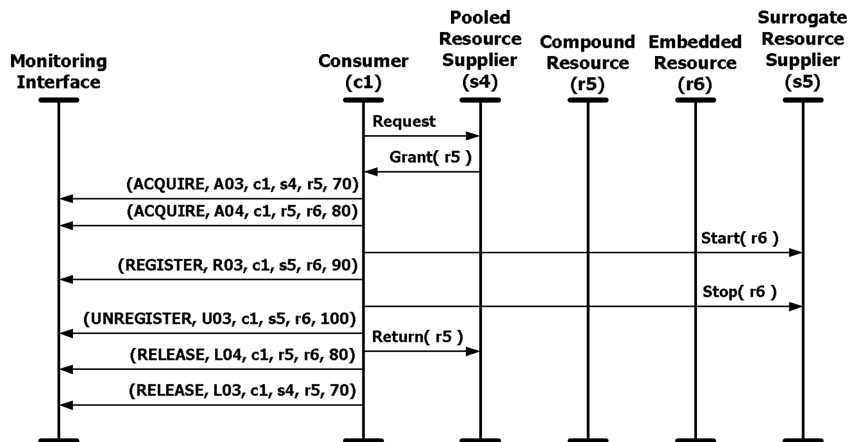



**Figure 7-13: MSC Consumer and Compound Resource.**

### 7.4.3.4 EoB Model Derivation Algorithm

This section describes EoB model derivation. Unlike the general derivation approach for the state transition model, each EoB model requires application-specific signal and CEFSM role-based knowledge to determine which aspects of the software design specification should go into an EoB model. The EoB model derivation algorithm may be automated. However, at present, a systematic manual procedure is required to identify the required structural transaction signal constructs for the selected structural concern from the software design specification. Each signal indicates when a specific structural transaction has successfully completed. Structural transaction signals are stored in a ResOwn signal set.

In addition, EoB entry and exit information identified in the software requirements is required to demarcate each EoB specification in the software design specification since, unlike the state evolution model derivation, the EoB model algorithm is applied to software design specification slices. Each specification slice contains a set of transitions, demarcated by a single EoB entry point, and a set of one or more EoB exit points, that will constitute the resultant EoB model. The algorithm itself systematically traverses an EoB model's transitions-set using a depth-first approach. During a transition traversal, structural transactions input or output signals are identified and replaced with an appropriate structural transaction monitoring construct. Further, during traversal, any encountered insignificant constructs are pruned from the resulting EoB model. The resulting, self-contained EoB model is a roadmap containing all the required structural transaction monitoring constructs for the selected structural concern being monitored. A line-by-line description follows the algorithm.

```
99. ALGORITHM: Derive-EoB-Specification
100.    INPUT: Design-Specification, CEFSM, State, Input, Exit-State-Set, ResOwn-Signal-Set
101.    OUTPUT: EoB-Specification
102.    FORALL ( State_i ∈ CEFSM ) DO:
103.        IF ( State_i == Entry-State) THEN:
104.            FORALL ( Input_j ∈ State_i ) DO:
105.                IF ( Input_j == Entry-Input ) THEN:
106.                    EoB-Entry == Input_j;
107.                    Break;
108.                ENDIF;
109.            ENDFOR;
110.        ENDIF;
111.    ENDFOR;
112.    COPY CEFSM to EoB-Specification and CREATE EoB Entry point;
113.    Transition := EoB-Entry;
114.    Transition-Stack := EMPTY;
115.    EoB-Complete := FALSE;
116.    WHILE ( EoB-Complete == FALSE ) DO:
```

```
117.        FORALL ( Construct_j ∈ Transition ) DO:
118.            IF ( Construct_j ∈ Exit-State-Set ) THEN:
119.                CREATE EoB exit point in EoB-Specification
120.                IF ( TOP (Transition-Stack) != EMPTY ) THEN:
121.                    Transition := POP (Transition-Stack);
122.                ELSEIF:
123.                    EoB-Complete:= TRUE;
124.                ENDIF;
125.            ELSEIF:
126.                FORALL ( Input_k ∈ Current-State ) DO: /* transient state */
127.                    If ( Input_k != Construct_j+1 ) THEN: /* don't push current transition */
128.                        PUSH ( Input_k, Transition-Stack );
129.                    ENDIF;
130.                ENDFOR;
131.            ENDIF;
132.            ELSEIF ( Construct_j == Task ) THEN:
133.                PRUNE ( Construct_j );
134.                JOIN-TRANSITION (Construct_j-1, Construct_j+1);
135.            ELSEIF ( Construct_j == Timer ) THEN:
136.                PRUNE ( Construct_j );
137.                JOIN-TRANSITION (Construct_j-1, Construct_j+1);
138.            ELSEIF ( Construct_j == Save ) THEN:
139.                PRUNE ( Construct_j );
140.            ELSEIF ( Construct_j == Decision ) THEN:
141.                FORALL ( NextTransition_n ∈ Decision ) DO: /* convert to transient state */
142.                    If ( NextTransition_n != Construct_j+1 ) THEN: /* don't push current */
143.                        PUSH ( NextTransition_n, Transition-Stack );
144.                    ENDIF;
145.                ENDFOR;
146.                REPLACE ( Construct_j, generateTransientState () );
147.            ELSEIF ( Construct_j == Input ) THEN:
148.                FORALL ( Signal_k ∈ ResOwn-Signal-Set ) DO:
149.                    IF ( Signal_k == GRANT-SIGNAL ) THEN:
150.                        REPLACE (Construct_j, generateMonitorConstruct( ACQUIRE, j+k );
151.                    ELSE
152.                        PRUNE ( Construct_j );
153.                    ENDIF;
154.                ENDFOR;
155.            ELSEIF ( Construct_j == Output ) THEN:
156.                FORALL ( Signal_k ∈ ResOwn-Signal-Set ) DO:
157.                    IF (( Signal_k == REQUEST-SIGNAL ) OR
158.                         ( Signal_k == ACTIVATE-SIGNAL ) THEN
159.                        INSERT ( Construct_j, NewState );
160.                        REPLACE (Construct_j, generateMonitorConstruct( ACQUIRE, j+k );
161.                    ELSEIF (( Signal_k == RETURN-SIGNAL ) OR
162.                             ( Signal_k == DEACTIVATE-SIGNAL ) THEN
163.                        INSERT ( Construct_j, NewState );
164.                        REPLACE ( Construct_j, generateMonitorConstruct( RELEASE, j+k );
165.                    ELSEIF (( Signal_k == START-SIGNAL ) THEN
166.                        INSERT ( Construct_j, NewState );
167.                        REPLACE ( Construct_j, generateMonitorConstruct( REGISTER, j+k );
168.                    ELSEIF (( Signal_k == STOP-SIGNAL ) THEN
169.                        INSERT ( Construct_j, NewState );
170.                        REPLACE ( Construct_j, generateMonitorConstruct( UNREGISTER, j+k );
171.                    ELSEIF
172.                        PRUNE ( Construct_j );
```

```
173.                ENDIF;
174.              ENDFOR;
175.          ENDIF;
176.      ENDFOR;
177.  ENDWHILE;
178.  FORALL ( Constructi ∈ EoB-Specification ) DO:
179.      IF (( Constructj.Type == State ) AND ( Constructj+1.Type == State ) THEN
180.          PRUNE ( Constructj );
181.          JOIN-TRANSITION ( Constructj-1, Constructj+1 );
182.      ENDIF;
183.  ENDFOR;
184.  RETURN ( EoB-Specification );
```

In lines 99-101, the inputs to the algorithm are the target's CEFSM-based software design specification, the CEFSM-state-input construct combination for the current EoB entry point, the set of quiescent end states for the current set of EoB Exit points and the set of ResOwn structural transaction signals that have been manually identified for the selected structural resource ownership concern from the software design specification. Lines 102-111 deal with locating the EoB entry point in the corresponding CEFSM of the software design specification. Line 113 initializes the current transition to the EoB entry point. Line 114 initializes the transition stack to empty. Line 115 initializes the EoB completion flag to false. Lines 116 to 132 deal with the handling of transient and quiescent states in the current transition. Line 118 loops until every transition from the EoB entry point to each of the EoB exit points has been traversed and processed. Line 117 creates a loop for all constructs in the current transition. In lines 118 to 124, if a quiescent end state is encountered, an EoB exit point is created at the end of the current transition in the EoB-Specification and the next transition to be traversed is popped from the Transition-Stack. If the stack is empty, the flag is set to indicate that the current EoB-specification is complete. In lines 125 to 131, if the current state is a transient state (i.e., not an EoB exit point), the first transition following the state construct is selected for traversal, and the remaining transitions emanating from the transient state are pushed onto the Transition-Stack for future exploration. Lines 132 to 139 deal with the pruning of insignificant constructs. Lines 132 to 134 prune Task constructs, lines 135 to 137 prune *Timer* constructs and lines 129 to 139 prune Save constructs. Lines 140 to 146 deal with the *Decision* constructs. The first transition following a decision is selected for traversal, and the remaining transition(s) emanating from the decision are pushed onto the Transition-Stack for future exploration. In line 146, the decision construct is replaced with a newly generated transient state. Lines 147 to 154 deal with the Input constructs. In lines 147 to 150, if the input signal has been identified as a *GRANT* from the set of manually identified ResOwn signals, then the Input construct is replaced by a corresponding

**ACQUIRE** monitoring construct. In lines 151 to 154, the insignificant Input is pruned. Lines 155 to 177 deal with the Output constructs. In lines 155 to 160, if the output signal has been identified as a *REQUEST* or **ACTIVATE** from the set of manually identified ResOwn signals, then a transient state is inserted into the transition and the Output construct is replaced by a corresponding **ACQUIRE** monitoring construct. In lines 161 to 164, if the output signal has been identified as a *RETURN* or *DEACTIVE*, then a transient state is inserted and the construct is replaced by a **RELEASE** monitoring construct. In lines 165 to 167, if the output signal has been identified as a *START*, then a transient state is inserted and the construct is replaced by a **REGISTER** monitoring construct. In lines 168 to 170, if the output signal has been identified as a *STOP*, then the construct is replaced by a **UNREGISTER** monitoring construct. In line 171 to 175, an insignificant Output is pruned. Lines 178 to 183 deal with collecting superfluous transient states from transitions in the EoB model. In line 184, the resultant EoB model is returned.

# Chapter 8

# Evaluation

*"There is a theory which states that if ever anybody discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable. There is another theory which states that this has already happened."*

- D. Adams (1952-2001)

## 8.1 Introduction

This chapter presents an analytical evaluation of the research presented in this thesis. The first section will discuss the ResOwn ontology and the second section will discuss the greybox approach to concern-specific dynamic software structure monitoring.

## 8.2 ResOwn

Software systems change over time in response to reliability and security needs, advances in technology and changing end user requirements. According to [Kir04], a resource allocation and management scheme should be formally specified, monitorable, traceable, extensible and upgradeable. The ResOwn ontology presented in this thesis is all that and more.

### 8.2.1 Advantages

- ResOwn integrates resource and ownership concepts into a single, unified model. The author is not aware of other ongoing research or published literature that uses an ontology to model application resource ownership structure. This makes ResOwn a novel contribution.

- ResOwn provides software monitoring and engineering with a new opportunity or way of thinking about resource allocation and management. ResOwn is specified using a well-defined vocabulary consisting of both resource and ownership concepts and properties that have been borrowed from other, existing domains (Section 4.3.2). There is a major advantage of ResOwn importing it's existing terminology and concepts from both the legal and real property domains. These long established domains have an extremely rich and well understood set of ownership concepts that because of ResOwn, can now be used to support resource allocation and management in the software domain. For example:
  - ResOwn formally distinguishes between managing (or servicing) a resource as a *nonbeneficiary owner* versus actually benefiting from the use of a resource as a *beneficiary owner* (Section 4.2.2 and Section 4.2.3). This distinction is important from the design and implementation viewpoints when modeling both static and evolving software structure. ResOwn now provides the necessary vocabulary and concepts.
  - ResOwn incorporates the notion of single user capacity and multi user resource capacity (Section 4.6.1.1) as both a structural and legal, role-based, ownership concept. Only objects playing a beneficiary owner role are included when determining whether a resource's structural ownership capacity has been exceeded. This approach provides a formally defined way to deal with evolving cardinality (i.e., multiplicity) restrictions that can be modeled and monitored at runtime. For example, consider a *touch tone receiver (ttrx) card* from the PBX (Section 2.10) with a specified *user capacity* of one (1) *beneficiary owner*:
    - If the *ttrx card* is owned by a *phone handler*, and also owned by a *ttrx scanner*, then the *card* has one *beneficiary owner* and one *nonbeneficiary owner*; the *capacity* is not violated, and
    - If the *ttrx card* is owned by two *phone handlers* at the same time, then the *card* has two *beneficiary owners*; the capacity is violated.
- The use of ResOwn terminology, concepts and object properties has been applied successfully to at least one software system from the interactive session-oriented service domain - the control program of the PBX application (see Section 2.10 for PBX description).
- A methodology (Section 5.2) has been devised and tested for creating a *specialized ResOwn instance* (Section 4.3.3) for a software product line and/or application instance from the *baseline*

*ResOwn ontology* (Section 4.3.3). This methodology was successfully applied to the PBX.

- The author was able to define a ResOwn ontological class for every object class in the PBX using the current set of concepts and object properties provided by ResOwn (Section 5.2).
- The author was then able to successfully create a specialized ResOwn instance (Section 5.2) for the PBX by inserting each of the new ontological classes, defined for the PBX, into the current baseline ResOwn class hierarchy. The resulting ResOwn instance was automatically shown to be consistent using a reasoner.
- The author was able to automatically classify the specialized ResOwn instance (Section 5.3) using a reasoner and create an *inferred* ResOwn instance *class hierarchy* for the PBX.

- The *baseline* ResOwn ontology, and any resulting *specialized* ResOwn instances created from the *baseline*, are formally defined using OWL-DL which provides a major advantage over visual UML-based models which lack formal semantics.
- The *modularity* of ResOwn promotes reuse of its concern-specific, *core subontology* (Section 4.3.1) with different, application-general *support subontologies* (Section 4.3.1). Since ResOwn models *support* and *value partitions* classes only as properties (Section 4.4), it is relatively easy and straightforward to identify where specific changes to the ResOwn core class definition are required, if a new support class or value partition is added or, if an entirely new application domain is to replace an existing one.
- One advantage of ResOwn is that it deals with the implementation bias in ontology construction in a systematic and structured manner. Before creating ResOwn, the author reviewed literature on existing ontologies (general and OWL-based) [Cha99, Dar06, Hor04, Noy97, Rec04, Sta06]. A common theme was that, traditionally, ontologies are to be created with careful consideration so as not to introduce implementation bias into the taxonomy, classes or properties [Noy01].
  - ResOwn's modular construction logically compartmentalizes and separates concern-specific knowledge from application-general knowledge through implementation of the *core* and *support* subontologies (Section 4.1). Doing this helped to control the scope and the amount of implementation bias that inevitably needed to be introduced into the ontology to customize ResOwn for the interactive session-oriented service domain.
  - ResOwn's creation and construction approach followed an iterative development procedure.

A concern-specific scope (i.e., structural resource ownership) was selected and any terminology listed. Over time, the list evolved. Terms not directly related to the scope were removed or revised. Sets of related concepts were collected into separate concept and property groups. Terms were added or revised based on new knowledge obtained from sources such as existing resource, scheduling, legal, real property and ownership ontologies, patterns for resource management and terminology and natural language definitions from legal and financial dictionaries. Any application-specific terms were separated out and placed into a separate list to limit the influence of implementation bias on core concepts. The next phase involved constructing the ontology itself in an iterative fashion. Separate top-level core class hierarchies (Section 4.2.2) were constructed and subclasses defined. At each iteration, the reasoner was used to test and debug the ontology by ensuring the class definitions were concerned and inferred class hierarchy correctly. In addition, the author constructed UML-based models of the ontology to visually check the author's interpretation and understanding of the ontological concepts and properties and ensure that, in fact, the ontology could be used in the software domain. Once the core ontology was reasonably mature, the author created the support subontology to add the required application-general concepts and properties to ResOwn. At each iteration, a conscious effort was made to ensure that implementation did not creep into the core subontology.

- The *extensibility* of ResOwn allows resource ownership knowledge to be created, encoded, categorized and classified at multiple levels of abstraction (i.e., levels of specialization) within a single, unified model. Using specialization to create new, application-specific instances of ResOwn also promotes reuse across many different applications within a single domain.
- For software product lines, it is possible to create a hierarchy of specialized ResOwn instances emanating from a single, baseline ResOwn ontology at the root. Beneath the root, the first tier in the hierarchy would contain a number of specialized ResOwn instances. Each first tier instance would be specialized with application-specific classes for a particular software product line from a family of products that belong to the selected interactive session-oriented service domain. On the second tier, each first tier instance would subsume one specialized ResOwn instance for every application instance in the product line. This approach has the advantage that only those ontological classes pertaining to a specific application need to be created to determine whether a new application adheres to the existing application structural resource ownership scheme.

### 8.2.2 Known Limitations

- A limitation of ResOwn is that the ontology has not been exposed to the public domain so that it can be further evaluated by experts from both the knowledge representation domain as an ontology, and by experts from the software engineering domain as an ownership-based resource allocation and management schema.
- ResOwn is currently limited to a single structural software concern: the resource ownership. There is a legitimate question as to whether the ResOwn approach to ontology modeling is adaptable to other software concerns such as data security or system adaptation.
- ResOwn has only been tested and evaluated for the interactive session-oriented service domain using the PBX. ResOwn need to be tested on other applications. This imposes a number of limitations on the current version of ResOwn:
    - ResOwn needs to be tested and evaluated using other applications from the interactive session-oriented service domain to see if its applicability is general enough for applications across the interactive session-oriented service domain beyond telephony.
    - ResOwn needs to be tested and evaluated with other application domains to see if ResOwn's structural approach to resource ownership is applicable to application domains other than interactive session-oriented services.
- To be truly an ontology, ResOwn should be capable of supporting specializations of real application classes; that is, the existing ResOwn classes and properties must be capable of encoding all relevant knowledge necessary to represent the application-level knowledge of real-world software systems for the selected application-domain. While the PBX provided a rich set of application-specific **Resource**, **Consumer** and **Supplier** subclasses, test cases are required to evaluate, identify and address the limitations of the current ResOwn implementation across a broader selection of applications.
- The terminology and concepts from the legal and real property domains in ResOwn are mostly new to the software engineering domain, placing a limitation on ResOwn, at this time, as a resource ownership modeling and management standard for software. The question remains as to whether ResOwn will be understood and readily adopted for practical use in the software domain. To this end, more examples are needed to clearly demonstrate the meaning and applicability of

ResOwn's legal- and real property-based concepts.

- One of the limitations of the ResOwn construction process was the lack of empirical or formal methods or techniques to determine how much, if any, implementation bias had either crept into the core subontology, or was need in the support subontology for core class definitions. Determining how much, or how little, implementation bias is good for a concern-specific ontology such as ResOwn remains an open question.
- A major limitation for adopting ResOwn in the software engineering domain is that most traditional software engineers are not familiar with the philosophy of ontology construction, OWL-DL, description logic, or tools such as Protégé-OWL and RacerPro.
- Another related limitation, is that tools like Protégé-OWL lack the same rich, visual modeling experience provided by the current generation of UML-based modeling tools such as IBM's Rational. However, it appears as though help is on the way. For example, UML profiles for OWL ontologies have emerged and may allow UML modeling tools to also facilitate ontological modeling. One possible solution is new tools that use a UML front-end for visual modeling, but keep the formal OWL-DL back-end for consistency checking and automatic classification purposes.

# 8.3 Dynamic Software Structure Monitoring

ResOwn implements application resource ownership as an individual structural software concern that can be monitored orthogonally from the operational software system's functional behavior. The greybox approach to concern-specific dynamic software structure monitoring (Chapter 7) has advantages and known limitations.

## 8.3.1 Advantages

- Being concern-specific is a major advantage for the proposed monitoring approach. Today's interactive session-oriented services are delivered by large and complex software systems. It is not possible to monitor everything. A greybox concern-specific approach offers a reasonable comprise.
- Concern-specific monitoring provides an opportunity to create models of the evolving, concern-

specific structure of operational software systems by fusing structural and behavioral knowledge into a concern-specific monitoring model.

- In general, the concern-specific monitoring approach integrates nicely with autonomic systems that support concepts such as self-healing, self-adaptation and self-optimization. For example, a set of individual concern-specific monitors, each focused on a particular autonomic concept could be turned on or off at runtime, according to the dynamic monitoring requirements of the operational autonomic system.
- The greybox aspect of the monitoring approach has several advantages and disadvantages over other blackbox and whitebox monitoring approaches as shown in Table 8-1.

| Blackbox Approaches | Whitebox Approaches | Greybox Approach |
|---|---|---|
| Runtime knowledge about evolving concern-specific structure enhances blackbox monitoring | Runtime knowledge about evolving concern-specific structure including in whitebox monitoring | Runtime knowledge about evolving structure focused on one, concern-specific aspect of the target software system |
| Non-intrusive | Highly intrusive | Some intrusion required |
| Monitoring model derived using external behavioral knowledge | Monitoring model derived from detailed, internal behavioral, structural and data flow knowledge | Monitoring models derived using only that internal behavioral knowledge relevant to the concern-specific structure to be monitored |
| Tracks global specification state | Shadows runtime implementation state | Tracks state-dependent, evolving software structure for a selected software concern on a session-by-session basis |
| Detects behavioral failure and must infer internal runtime knowledge | Detects behavioral and data errors and possesses detailed internal runtime knowledge | Detects only context-sensitive, structural integrity errors. Internal runtime knowledge limited to select structural context |
| Specification nondeterminism may lead to large space and time complexity | Volume of monitored data to be transported and processed may lead to large error detection latencies | Limited structural context reduces volume of monitored data to be transported and processed leading to reduced detection latency but limited monitoring focus may reduce error detection probability |
| Limited error diagnosis and fault localization capability | Enhanced error diagnosis and fault localization capability | Not intended for detecting data and behavioral failures or errors |

**Table 8-1: Comparison of monitoring approaches.**

- The greybox approach has an advantage over blackbox approaches because the greybox approach can detect structural errors and eliminates the specification nondeterminism encountered in blackbox models by embedding software sensors in the operational target implementation that maps monitoring constructs into the concern-specific model.
- The greybox approach has an advantage over whitebox approaches because the greybox approach abstracts away unnecessary implementation details and focuses on an individual software concern

within the evolving structure of the operational target. This approach reduces or limits both the amount of required intrusion into the target and the volume of monitored data that must be collected, transported and processed by the monitor's interpreter.

- The monitoring application research ownership has benefits for detecting operational performance bottlenecks as well as implications for detecting performance degradation due to runtime phenomena such as software aging.
- The monitoring approach is capable of detecting (and possibly localizing) resource ownership errors that a purely blackbox approach, which is dependent on externally observable knowledge, could not. For example, the orphaned power of attorney scenario (Section 5.4.5) is only detectable using the proposed approach for monitoring the evolving resource ownership structure of the PBX. Other examples of structural resource ownership errors are detailed in the other resource acquisition and ownership scenarios described in Section 5.4 of Chapter 5.

### 8.3.2 Known Limitations

- The current approach is limited to only manual placement of embedded software sensors in the instrumented target software system (Section 7.3.3). To be widely practical and on the same level of independence as autonomic software systems, an automated methodology or technique is necessity that would be capable of using the formal software specifications and the source code to automatically locate and then weave sensors into the appropriate location in the software source code. This is a nontrivial problem and would require the ability to locate features in source code.
- A major limitation of the monitoring approach is that it does not have formal algorithms for resynchronizing the interpreter once an application resource ownership error is detected or an inconsistency in the dynamic knowledge base is found. These algorithms are nontrivial and would need to include how to deal with discrepancies such as missing, lost or extra monitoring commands. Presently, the greybox interpreter only reports warnings (Section 7.2.4).
- The approach prunes away most of the SDL constructs when deriving the concern-specific model (Section 7.4.2 and Section 7.4.3.4). It is unclear whether the derivation approach used for application resource ownership monitoring would work with other software concerns such as data security or system adaptation.

# Chapter 9

# Conclusions

*"The sooner you get behind in your work, the more time you have to catch up."*

- Anonymous

## 9.1 Introduction

This chapter contains conclusions drawn from the research work described in this thesis, followed by a summary of the novel research contribution of this thesis. The last section suggests some areas for possible future work.

## 9.2 Conclusions

This thesis introduced and presented ResOwn, a novel ontology for application resource ownership. ResOwn provides a vocabulary along with a set of concepts and properties for modeling the application resource ownership structure of operational software systems. ResOwn is OWL-DL-based, role-based, modular, extensible and automatically classifiable by a reasoner. ResOwn models an application resource ownership structure scheme that is not hard-coded into the resource and owner model concepts, but instead built upon a dedicated concept of proof of ownership instruments. These instruments support a rich notion of resource ownership that allows different owners to play different ownership roles, each with different ownership rights, even with the same resource. This thesis also proposes a greybox approach for monitoring the state-dependent, evolving resource ownership structure in interactive session-oriented services. The monitor executes as a separate unit. A monitoring interface comprised of embedded software sensors is associated with the monitor, but woven into the target software system's implementation according to a sensor plan. The monitor

interprets a concern-specific model that has been derived (i.e., abstracted) from the target's formal specifications. During the derivation process, the model is extended with special monitoring constructs. Each monitoring construct in the model is associated with a specific monitoring command produced by a corresponding software sensor in the instrumented target. The top-level architecture of the monitor is comprised of a greybox interpreter and a tuple-based dynamic knowledge base. The greybox interpreter uses the dynamic knowledge base to maintain a representation of the operational target software system's evolving software structure. The greybox interpreter receives monitoring commands while interpreting the concern-specific model and updates the contents of the dependent dynamic knowledge base to match the monitored portion of the target's actual software structure.

# 9.3 Research Contributions

These are the major novel contributions that were presented in this thesis:

- A reusable and extensible, concern-specific ontology called ***ResOwn*** provides enriched concepts of application resource ownership borrowed from real-world legal and ownership ontologies. ResOwn is defined in the *Web Ontology Language Description Logic (OWL-DL)*, verified with a reasoner and tested using the PBX example.
- A methodology to create an application-specific ResOwn instance that specializes the concern-specific portion of the ResOwn ontology with application-level knowledge for a particular software system.
- A dual-view, *Session-Oriented Model of Computation (SOMOC)* for interactive session-oriented services that relates observable, external service behavior to internal, evolving software structure.
- A greybox, concern-specific dynamic software structure monitoring *approach* and *architecture* devised for tracking the *evolving software structure* of an operational software system.
- A pair of algorithms for deriving the *concern-specific monitoring model*:
    - An *algorithm* for deriving a *state evolution model* from the target's *software requirements* specification. The state evolution model allows the greybox interpreter to track the *specification state* (i.e., *macro-steps* in the evolving structure) of the operational target.
    - An *algorithm* for deriving a set of *epoch of behavior (EoB) models* from certain *slices* of the

target's *software design* specification. Each EoB model contains the monitoring constructs that allow the greybox interpreter to track certain concern-specific *structural transactions* (i.e., *micro-steps* in the evolving structure) as they are reported by the instrumented target.

# 9.4 Future Work

Some key areas of future work for the ResOwn ontology can be investigated along two related dimensions. In the first dimension, ResOwn needs to be evaluated with other interactive session-oriented service applications. In the second dimension, the modular approach of separating ResOwn into a core and a support subontology needs to be evaluated with other software concerns.

Another key area of future work pertains to the formalization of algorithms for both structural error detection and monitor resynchronization in the presence of detected errors. Once this important future work is completed, a implementation of a prototype monitor could be constructed, and a practical evaluation of the monitoring approach conducted. A prototype monitor could also facilitate the collection and study of empirical data on type and frequency of software structure errors.

SDL is a very rich specification language, and presently, the approach presented only dealt with SDL-based requirements and design specifications. Although SDL is expressive enough to allow for the specification of quite complex systems, another area of future work might pertain to further extending the approach to other CEFSM-based specification languages, including UML Statecharts.

The monitoring approach has been intentionally constrained to consider only application resource ownership in interactive session-oriented services whose real-time software systems have been specified in SDL. An opportunity exists to extend the research of this thesis by widening the scope to other individual software concerns such as data security, intrusion detection, and adaptive systems.

Another potential area of future work is to investigate the formal specification of a standard greybox monitoring interface for software components. This idea is similar to the notion of a built-in component testing interface already proposed in other related work that is designed into the component during development and can be used both during testing and software maintenance.

Another important area of future work involves extending behavioral monitoring in a modular fashion to incorporate dynamic software structure monitoring. In this hybrid approach, the software structure monitor would maintain a dynamic knowledge base of the evolving concern-specific

structure of the operational target. The behavioral monitor queries the software structure monitor's knowledge base whenever the behavior monitor needs to verify some concern-specific knowledge. For example, in the case of resource ownership, when the behavioral monitor detects that a particular event handler has received data from a particular resource, the behavioral monitor would query the dynamic resource ownership knowledge base and check if the event handler has the correct and necessary ownership rights to receive the resource's benefit.

One last area of related research that was not considered in this thesis is the effects of dynamic load on the integrity of the concern-specific evolving structure in general, and on the evolving resource ownership structure of the operational software system in particular. Investigation of the impacts of high offered loads on resource ownership structure could be valuable in a number of interesting ways. For example, there is the notion of concern-specific failure creep, in which the level of resource ownership structure errors continues to build at higher loads until a failure or service outage occurs. Failure creep, as it pertains to the corruption of the application resource ownership structure, may result from the long-term exposure of a soft, real-time software system to heavy loads. It would be useful to have a method to document and categorize different types of resource ownership errors and their likelihood of causing a failure. Another aspect to investigate is the rate of decline in service quality as a function of the evolving resource ownership structure consistency at high loads.

# Appendix A

# Protégé-Owl and RacerPro Screenshots

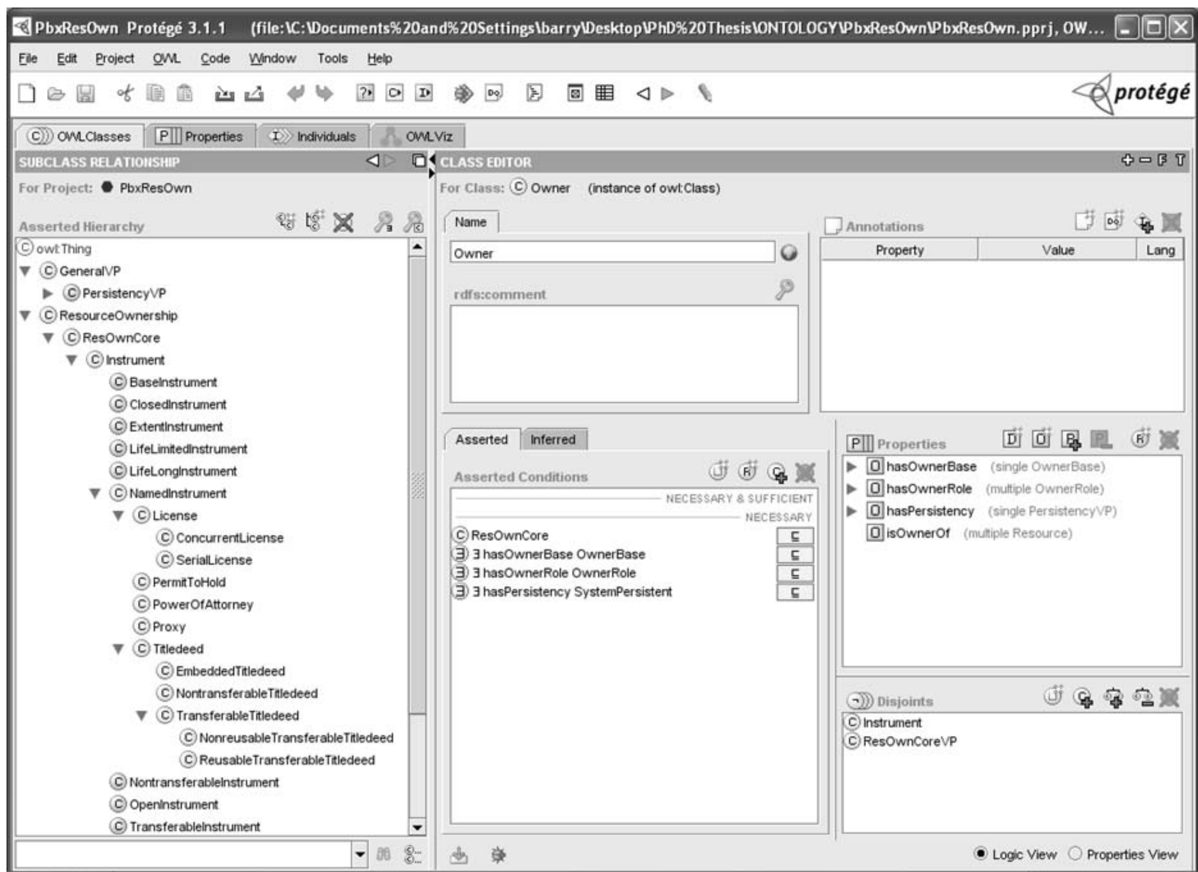



**Figure A-1: Protégé-Owl classes view screenshot.**

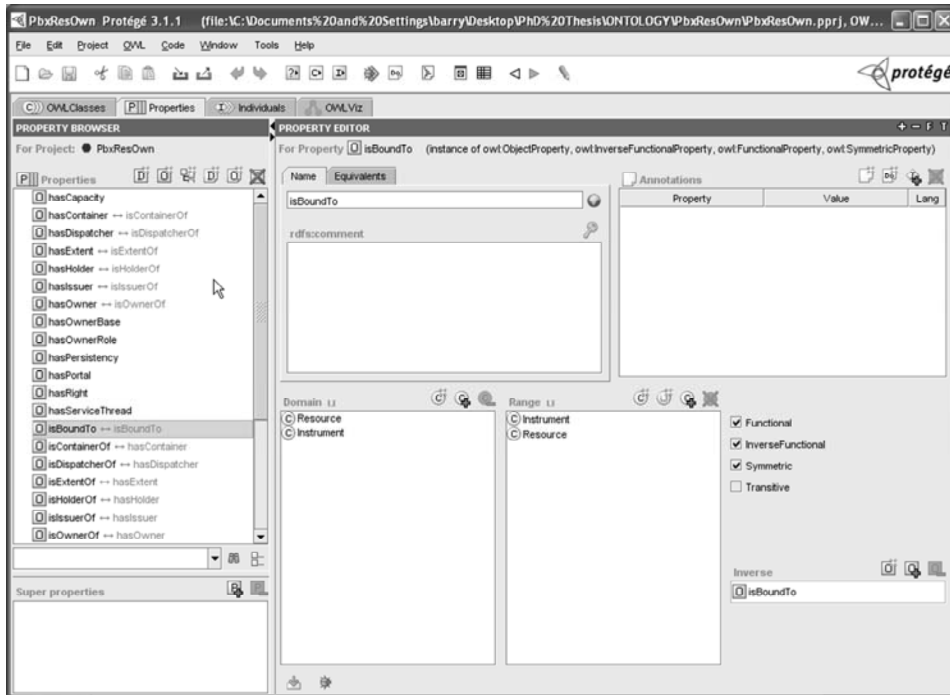



**Figure A-2: Protégé-Owl properties view screenshot.**

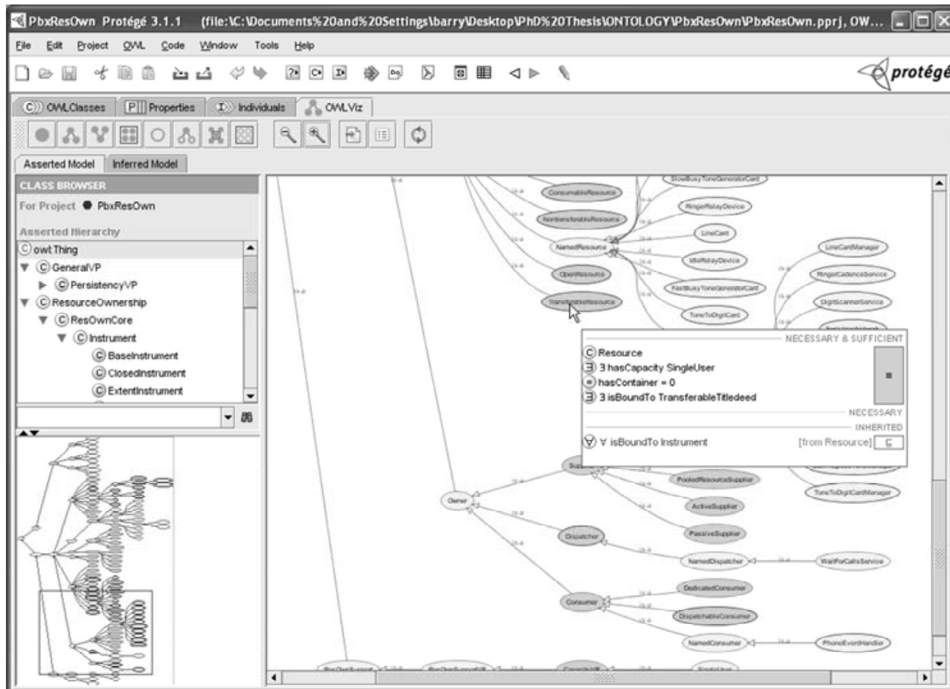



**Figure A-3: Protégé-Owl OWL Viz view screenshot.**

```
RacerPro
;;; Welcome to RacerPro Version 1.9.0 2005-12-05!

;;; Racer: Renamed Abox and Concept Expression Reasoner
;;; Supported description logic: ALCQHIr+(D)-
;;; Supported ontology web language: subset of OWL DL (no so-called nominals)

;;; Copyright (C) 2004, 2005 by Racer Systems GmbH & Co. KG
;;; All rights reserved. See license terms for permitted usage.
;;; Racer and RacerPro are trademarks of Racer Systems GmbH & Co. KG
;;; For more information see: http://www.racer-systems.com
;;; RacerPro comes with ABSOLUTELY NO WARRANTY; use at your own risk.

;;; RacerPro is based on:
;;; International Allegro CL Enterprise Edition 7.0 (Oct 19, 2004 13:28)
;;; Copyright (C) 1985-2004, Franz Inc., Oakland, CA, USA.  All Rights Reserved.
;;; The XML/RDF/RDFS/OWL parser is implemented with Wilbur developed
;;; by Ora Lassila. For more information on Wilbur see
;;; http://wilbur-rdf.sourceforge.net/.

;;; ==========================================================================
;;; Found license file
;;;  C:\Program Files\RacerPro-1-9-0\license.racerlicense
;;; This copy of RacerPro is licensed to:
;;;
;;; Barry R Pekilis
;;; University Of Waterloo
;;; Generative Programming Lab
;;; 200 University Avenue West
;;; N2L3G1, Waterloo, Ontario
;;; CA
;;;
;;; Initial license generated on 04-20-2006, 13:34 for 1.9.0.
;;; Desktop, Educational, on X86 Win32.
;;; This license is valid up to version 1.9.999.
;;; This license is valid until 10-31-2006, 17:59.
;;;
;;; This timelimited demo license expires in 90 days, 21 hours and 32 minutes.
;;;
;;; ==========================================================================

HTTP service enabled for: http://localhost:8080/
TCP service enabled for: http//localhost:8088/
```

**Figure A-4: RacerPro Reasoner console screenshot.**

```
Connected to Racer 1.9.0
Finished: Classification complete

Reasoner log
  Check concept consistency
    Time to build query = 0.011 seconds
    Time to send and receive from reasoner = 1.592 seconds
    Time to update Protege-OWL = 0.02 seconds
  Compute inferred hierarchy
    Time to build query = less that 0.001 seconds
    Time to query reasoner = 3.465 seconds
    Time to update Protege-OWL = 0.18 seconds
  Compute equivalent classes
    Time to build query = less that 0.001 seconds
    Time to query reasoner = 0.19 seconds
    Time to update Protege-OWL = 0.03 seconds
  Total time: 5.629 seconds

Cancel    OK
```

**Figure A-5: RacerPro reasoner log screenshot.**

# Appendix B

# Instrumentation Examples

```
String state = IDLE_STATE;
. . .
while (true)
{
    Object msg = msgBuffer.get ();
    switch (state)
    {
    case IDLE_STATE:
        switch (msg.type)
        {
        case OFF_HOOK_MSG:
            state = handleOffHook (msg);
            break;
        case CALL_REQ_MSG:
            state = handleCallReq (msg);
            break;
        }
    . . .
    }
}
```

(i)

```
String state = IDLE_STATE;
 . . .
while (true)
{
    Object msg = msgBuffer.get ();
    switch (state)
    {
    case IDLE_STATE:
        switch (msg.type)
        {
        case OFF_HOOK_MSG:
            monEventDirective.send (E1, msg.from, msg.to, msg.param, clk.ts ());
            state = handleOffHook (msg);
            break;
        case CALL_REQ_MSG:
            monEventDirective.send (E2, msg.from, msg.to, msg.param, clk.ts ());
            state = handleCallReq (msg);
            break;
        }
    . . .
    }
}
```

(ii)

**Figure B-1: State-oriented structure: (i) original code; (ii) instrumented code.**

```
String state = IDLE_STATE;
. . .
while (true)
{
    Object msg = msgBuffer.get ();
    switch (msg.type)
    {
    case OFF_HOOK_MSG:
        switch (state)
        {
        case IDLE_STATE:
            state = callOrigination (msg);
            break;
        case RINGER_STATE:
            state = callTermination (msg);
            break;
        }
    . . .
    }
}
```

(i)

```
String state = IDLE_STATE;
 . . .
while (true)
{
    Object msg = msgBuffer.get ();
    switch (msg.type)
    {
    case OFF_HOOK_MSG:
        switch (state)
        {
        case IDLE_STATE:
            monEventDirective.send (E1, msg.from, msg.to, msg.param, clk.ts ());
            state = callOrigination (msg);
            break;
        case RINGER_STATE:
            monEventDirective.send (E2, msg.from, msg.to, msg.param, clk.ts ());
            state = callTermination (msg);
            break;
        }
    . . .
    }
}
```

(ii)

**Figure B-2: Input-oriented structure: (i) original code; (ii) instrumented code.**

# Appendix C
# ResOwn Class Hierarchy

The full *asserted* and *inferred* class hierarchies for the ResOwn ontology are shown in **Error! Reference source not found.** and **Error! Reference source not found.**, respectively. The purpose of the diagrams is to give the reader a feel for the size, complexity, and scope of the *asserted* and *inferred* class hierarchies. Class shown in *Yellow* are named or *primitive* OWL classes. Classes shown in *Orange* are *defined* OWL classes. Classes shown with a *Blue* outline are specified using *multiple inheritance*; that is, those classes from the *asserted* class hierarchy that have been automatically classified and are subsumed by more than one *Superclass* in the *inferred* class hierarchy. Unfortunately, due to the size of the two class hierarchy diagrams, and the margin limits imposed on the page, it is recognized that many of the labels in the two class hierarchies are difficult to read. That is why these diagrams are in the appendix and smaller, more readable portions or entire sections of the two class hierarchies are presented, where relevant, in the main body of this thesis.

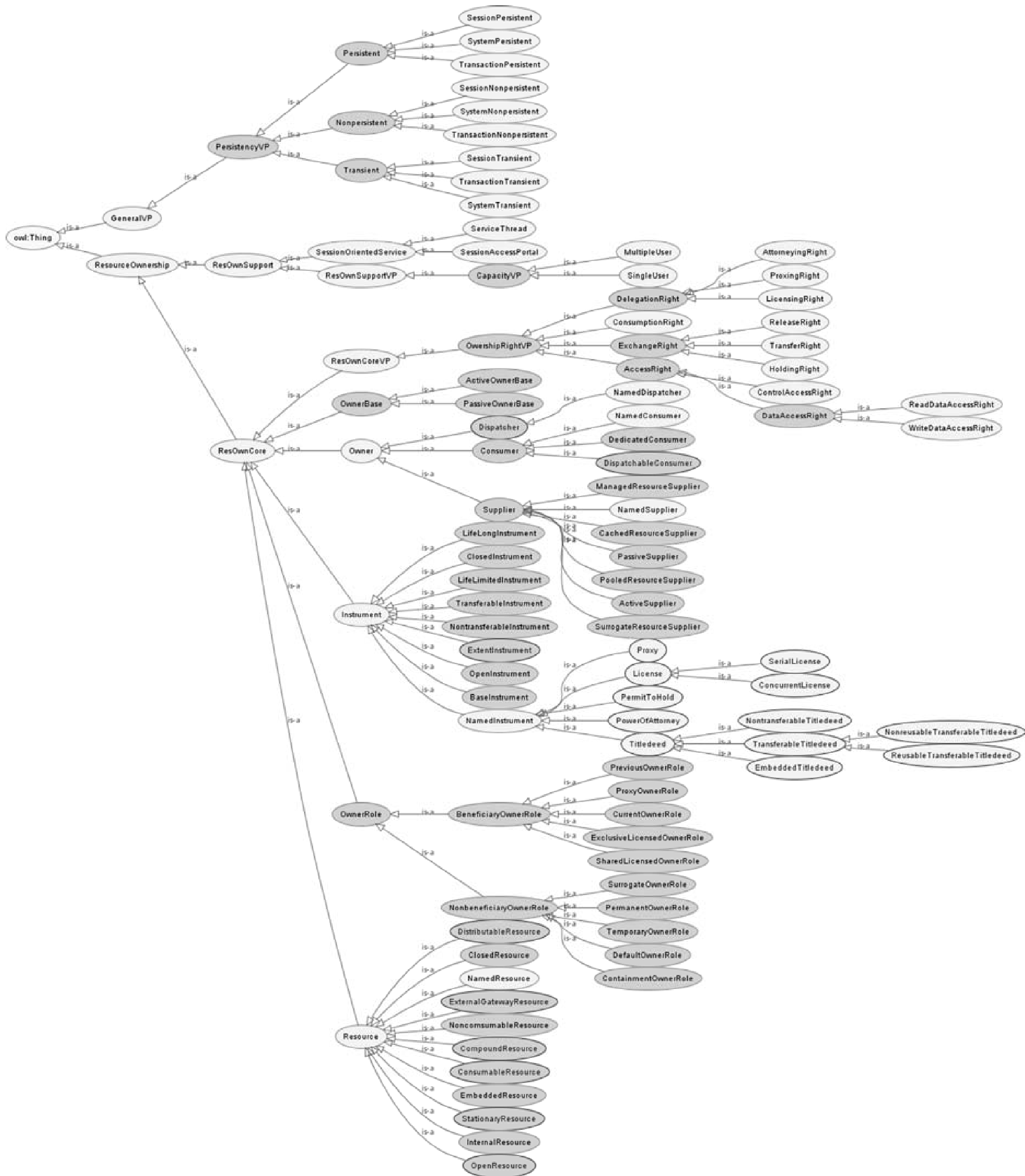



**Figure C-1: The ResOwn ontology's asserted class hierarchy.**

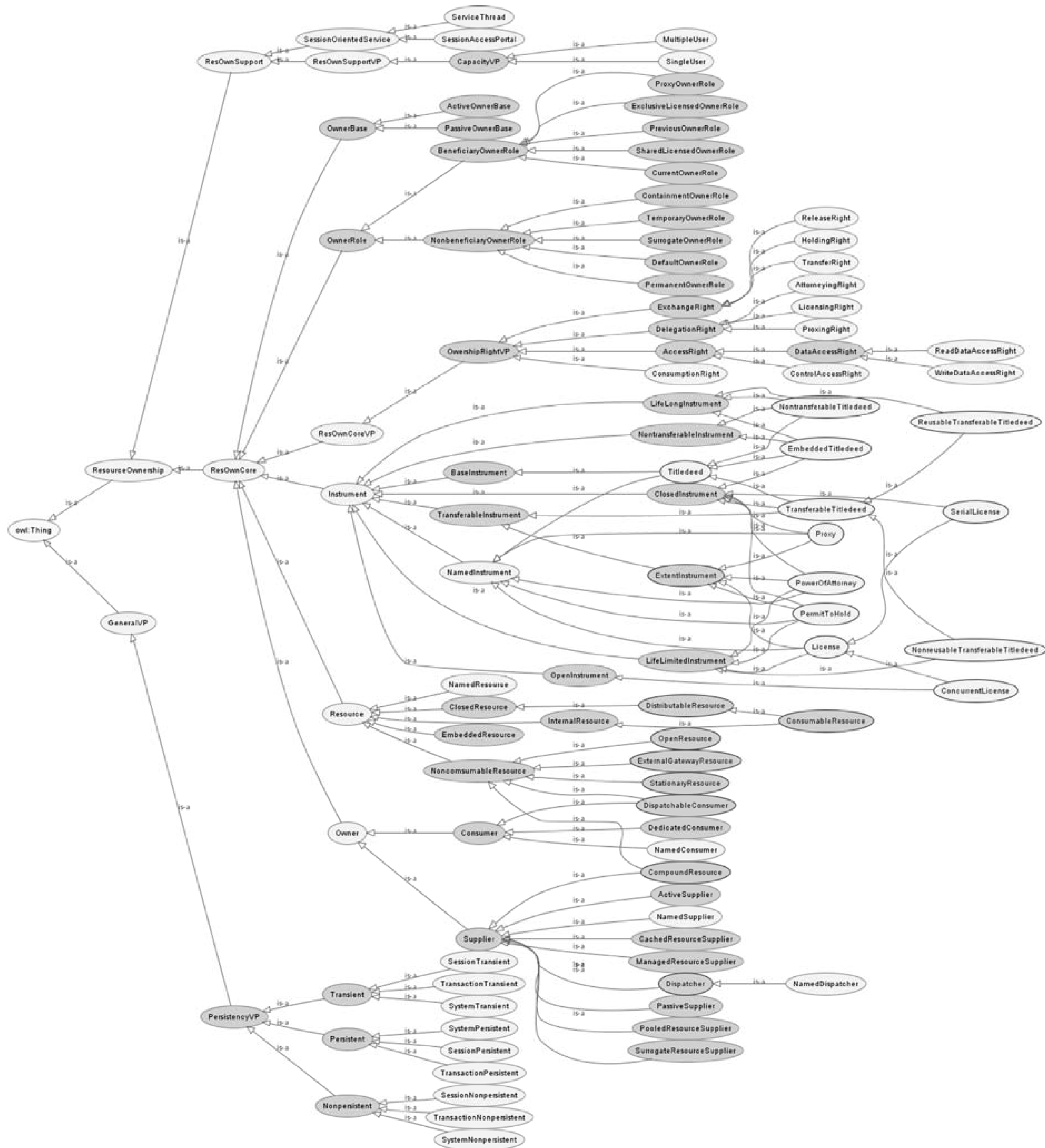



**Figure C-2: The ResOwn ontology's inferred class hierarchy.**

# Appendix D
# Phone Handler Example

The first part of this appendix shows an original SDL requirements specification excerpt for the Phone Handler from the example PBX described in Section 2.10, followed by the transformed Phone Handler excerpt shown as a sample excerpt of the entire state evolution model.

The second part of this appendix shows an original SDL design specification excerpt for the Phone Handler from the example PBX described in Section 2.10, followed a number of corresponding EoB models derived from the excerpt representing just a sample of the entire EoB Library.

The derivation algorithms are described in detail in Chapter 7.

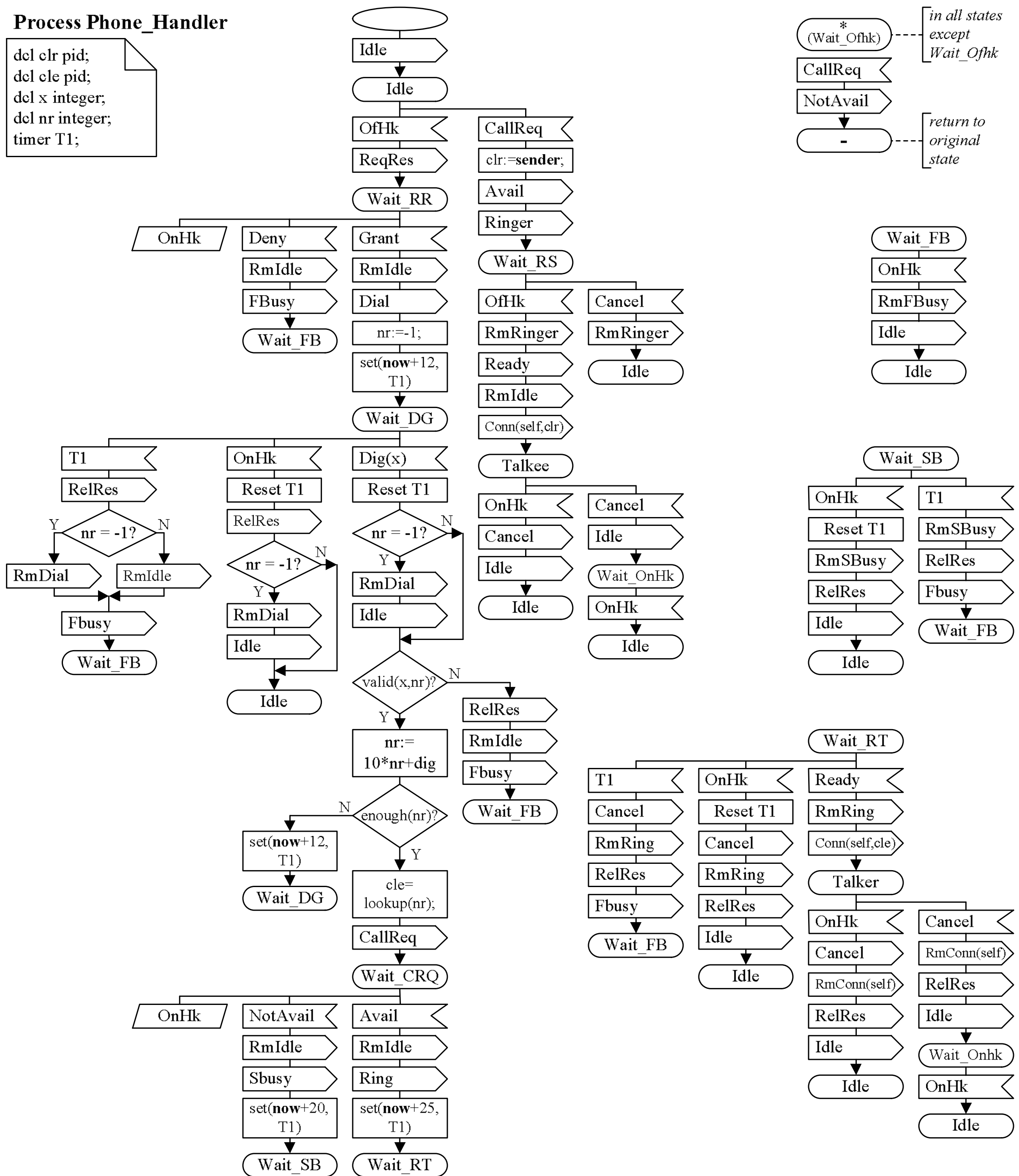



**Figure D-1: SDL requirements excerpt of Phone Handler.**

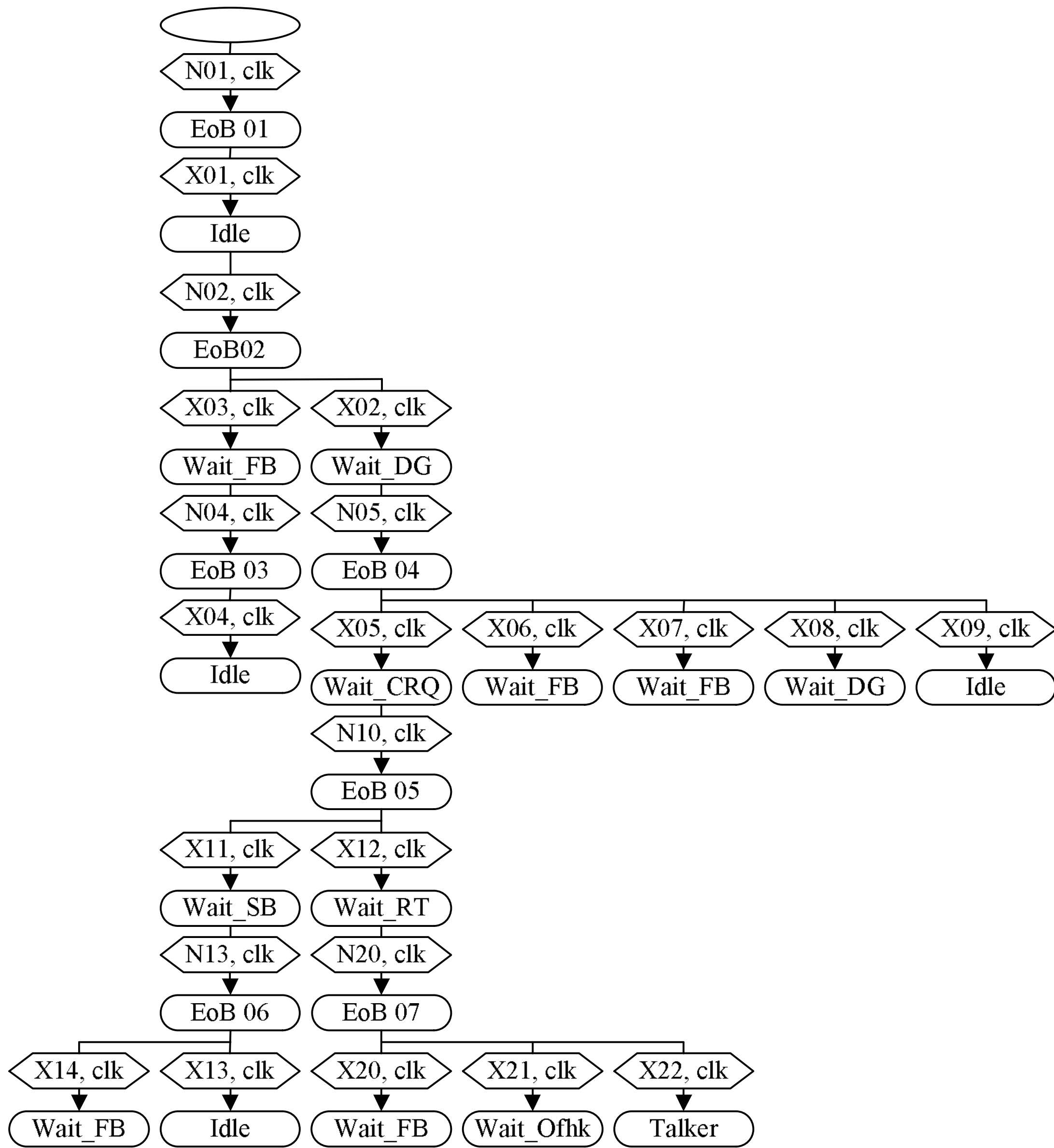



**Figure D-2: Sample excerpt from state evolution model for phone handler.**

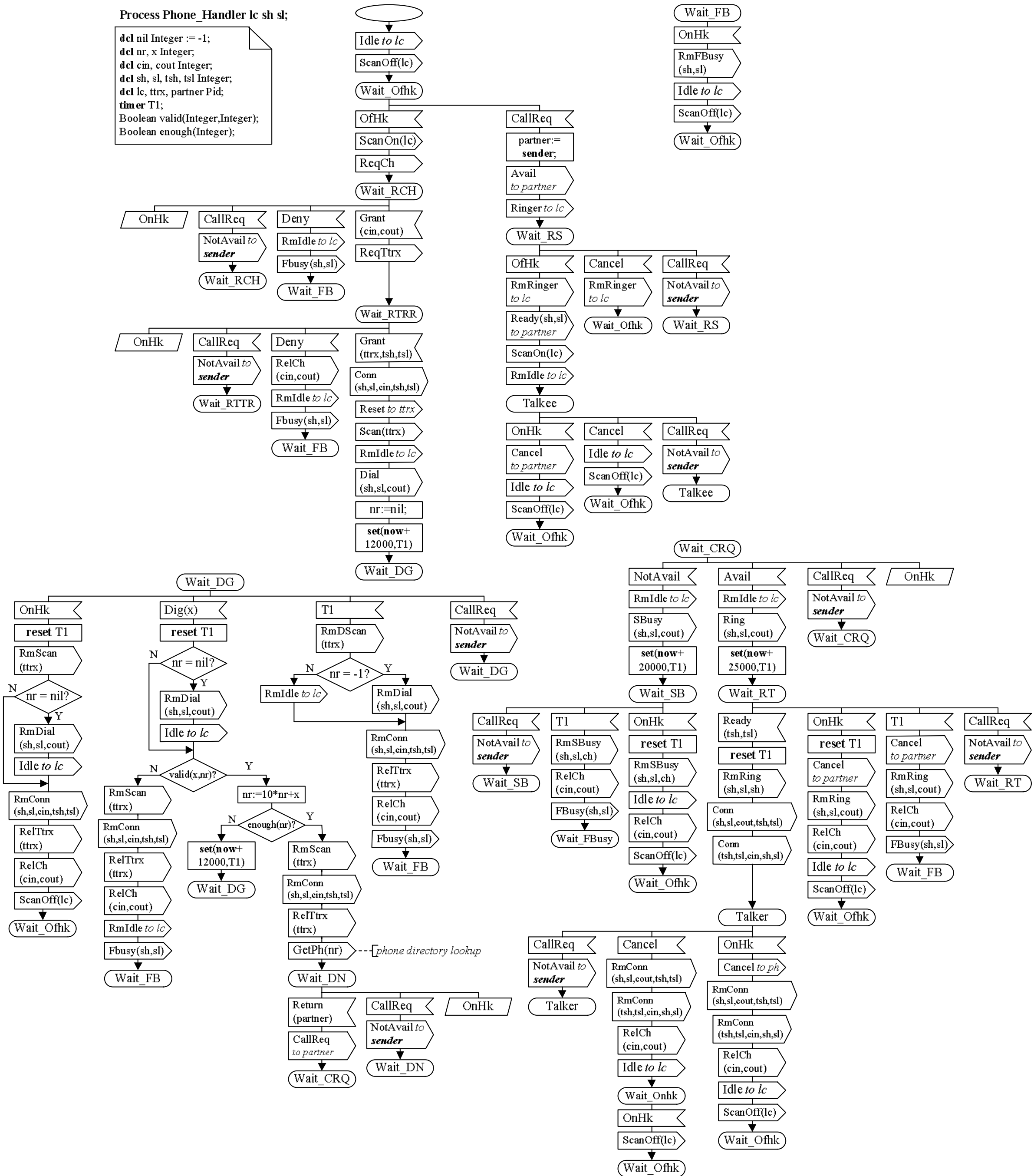


**Figure D-3: SDL design excerpt for phone handler.**

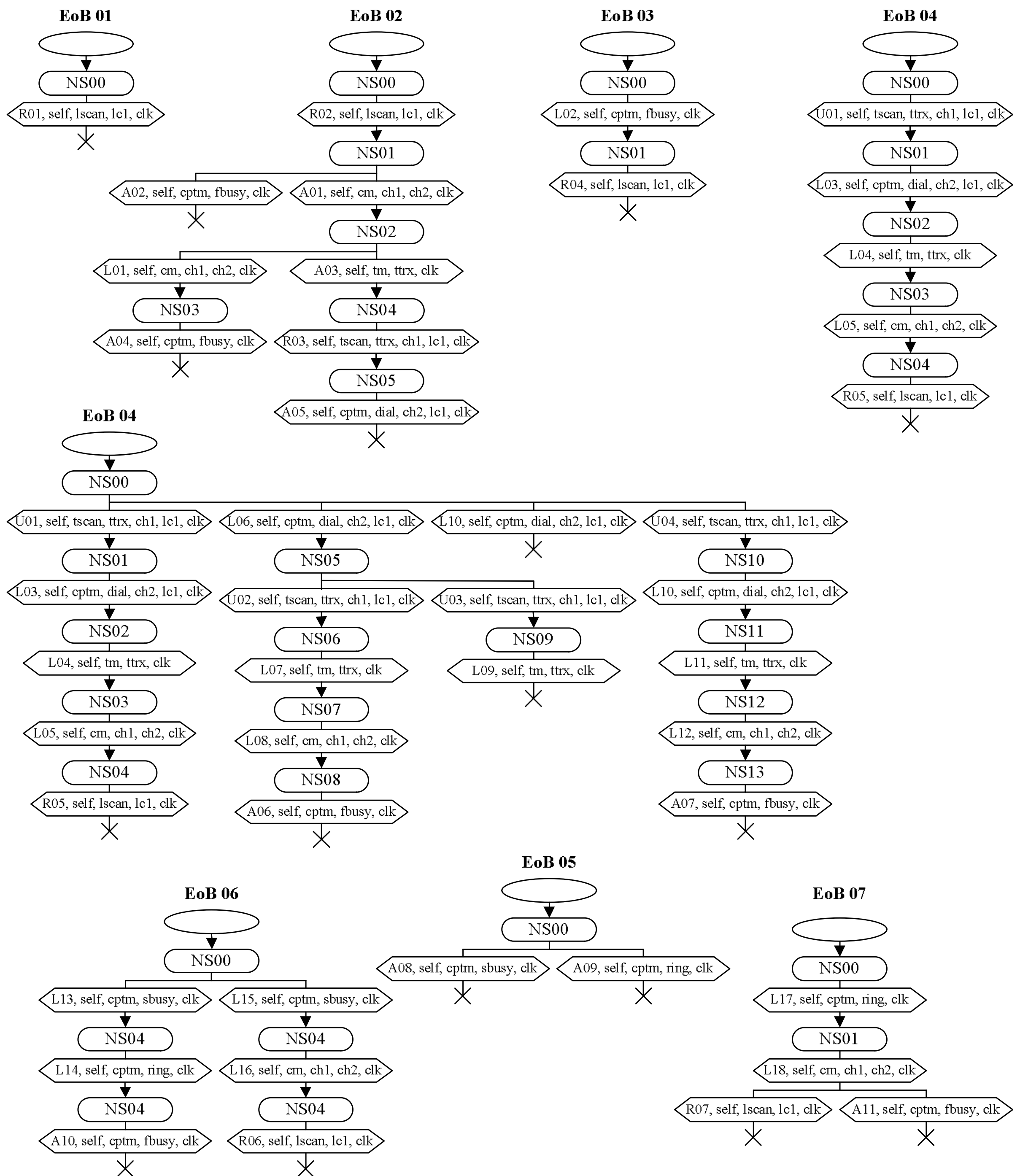



**Figure D-4: Sample excepts from EoB library for phone handler.**

# References

[Avr97] A. Avritzer, E.J. Kolettis, “Monitoring smoothly degrading systems for increased dependability,” *In: Empirical Software Engineering*, Vol. 2, No. 1, Kluwer, 1997, pp. 59-77.

[Aba91] M. Abadi and L. Lamport, “The existence of refinement mappings,” *In: Theoretical Computer Science*, Vol. 82, No. 2, Elsevier, 1991, pp. 253-284.

[Amb88] V. Ambriola, D. Notkin, “Reasoning about interactive systems,” *In: IEEE Transactions on Software Engineering*, Vol. 14, No. 2, IEEE, 1988, pp. 272-276.

[Atk03] C. Atkinson, T. Kuhne, “Model-driven development: a metamodeling foundation,” *In: IEEE Software*, Vol. 20, No. 5, IEEE, 2003, pp. 36-41.

[Baa03] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. Patel-Schneider (Eds), *The description logic handbook: theory, implementation, and applications*, Cambridge University Press, 2003.

[Bac99] R. Back, A. Mikhajlova, J. von Wright, “Reasoning about interactive systems,” *In: Proceedings of the World Congress on Formal Methods in the Development of Computing Systems, (FM Vol. 2),* Lecture Notes in Computer Science, Vol. 1709, Springer, 1999, pp. 1460-1576.

[Bel91] F. Belina, D. Hogrefe, A. Sarma, *SDL with applications from protocol specification*, Prentice-Hall, 1991.

[Ben03] A. Benveniste, S. Haar, E. Fabre, C. Jard, *Distributed monitoring of concurrent and asynchronous systems*, Report No. 4842, The French National Institute for Research in Computer Science and Control (INRIA), 2003.

[Bid66] B.J. Biddle, E.J. Thomas, *Role theory: concepts and research*, Wiley, 1966.

[Bid79] B.J. Biddle, *Role theory: expectations, identities, and behaviors*, Academic Press, 1979.

[Bræ93] R. Bræk, “SDL Basics,” *In: Computer Networks and ISDN Systems*, Vol. 28, No. 12, Elsevier, 1996, pp. 1585-1602.

[Bræ93] R. Bræk, Ø. Haugen, *Engineering real time systems: an object-oriented methodology using SDL*, Prentice Hall, 1993.

[Bri06] Britannica, *Online Encyclopedia Britannica*, http://www.britannica.com/.

[Bro01] M. Broy, K. Stølen, “Specification and development of interactive systems: focus on streams, interfaces, and refinement,” *In: Monographs in Computer Science*, Springer, 2001.

[Cab03] G. Cabri, L. Leonardi, F. Zambonelli, “Implementing role-based interactions for Internet agents,” *In: Proceedings of the 2003 Symposium on Applications and the Internet (SAINT)*, IEEE, 2003, pp. 380-389.

[Cas01] V. Castelli, R.E. Harper, P. Heidelberger, S.W. Hunter, K.S. Trivedi, K. Vaidyanathan, W.P. Zeggert, "Proactive management of software aging," *In: IBM Journal*, Vol. 45, No. 2, IBM, 2001, pp. 311-332.

[Cha99] B. Chandrasekaran, J.R. Josephson, V.R. Benjamins, "What are ontologies, and why do we need them?" *In: Intelligent Systems*, Vol. 14, No. 1, IEEE, 1999, pp. 20-26.

[Cha00] S. Chandra, P.M. Chen, "Whither generic recovery from application faults? a fault study using open-source software," *In: Proceedings of the International Conference Dependable Systems and Networks (DSN)*, IEEE, 2000, pp. 97-106.

[Chr01] V. Christophides, R. Hull, G. Karvounarakis, A. Kumar, G. Tong, M. Xiong, "Beyond discrete e-services: composing session-oriented services in telecommunications," *In: Proceedings of the Second Annual Very Large Data Bases International Workshop on Technologies for e-Services (VLDB-TES)*, Lecture Notes in Computer Science, 2193, Springer, 2001, pp 58-73.

[Con72] J.R. Connet, E.J. Pasternak, B.D. Wagner, "Software defenses in real-time control systems," *In: Proceedings of the Second Symposium on Fault-Tolerant Computing*, IEEE, 1972, pp. 94-99.

[Cou01] G. Coulouris, J. Dollimore, T. Kindberg, *Distributed systems: concepts and design*, Addison-Wesley, 2001.

[Cra03] J. Crampton, G. Loizou, "Administrative scope: a foundation for role-based administrative models," *In: ACM Transactions on Information and System Security*, Vol. 6 , No. 2, ACM, 2003, pp. 201-231.

[Dam00a] DAML, *Capacity Ontology*, http://www.kestrel.edu/DAML/2000/12/CAPACITY.daml, DARPA, 2000.

[Dam00b] DAML, *Resource Ontology*, http://www.kestrel.edu/DAML/2000/12/RESOURCE.daml, DARPA, 2000.

[Dar06] Defense Advanced Research Projects Agency (DARPA), *The DARPA Agent Markup Language (DAML) Program*, http://www.daml.org/, DARPA, 2006.

[Det01] M. Deters, R.K. Cytron, "Introduction of program instrumentation using aspects." *In: Proceedings of the OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, ACM, 2001.

[Dev99] V. Devedzic, "Ontologies: borrowing from software patterns," *In: Intelligence*, Vol. 10, No. 3, ACM, 1999, pp. 14-24.

[Dju05] D. Djurić, D. Gašević, V. Devedžić, "Ontology modeling and MDA," *In: Journal of Object Technology*, Vol. 4, No. 1, Chair of Software Engineering at ETH Zurich, Switzerland, 2005, pp. 109-128.

[Dod92] P.S. Dodd, C.V. Ravishankar, "Monitoring and debugging distributed real-time programs," *In: Software Practice and Experience*, Vol. 22, No. 6, Wiley, 1992, pp. 863-877.

[Dos05] B. Doshi, L. Benmohamed, A. DeSimone, "A hybrid end-to-end QoS architecture for heterogeneous networks (like the global information grid)," *In: Proceedings of the Military Communications Conference (MILCOM)*, IEEE, 2005, pp. 1-9.

[Dvo91] D. Dvorak, B. Kuipers, "Process monitoring and diagnosis: a model-based approach," *In: IEEE Expert*, Vol. 6, No. 3, IEEE, 1991, pp. 67-74.

[Ece00] Department of Electrical and Computer Engineering, *ECE 355 software engineering course project, PBX hardware description - Version 1.2*, University of Waterloo, Ontario, Canada, 2000.

[Eng95] D.R. Engler, M.F. Kaashoek, J. O'Toole Jr., "Exokernel: an operating system architecture for application-level resource management," *In: Proceedings of the Fifteenth Symposium on Operating Systems Principles (SIGOPS)*, ACM, 1995, pp. 251-266.

[Fac97] M. Faci, L. Logrippo, B. Stepien, "Structural models for specifying telephone systems," *In: Computer Networks and ISDN Systems*, Vol. 29, No. 4, Elsevier, 1997, pp. 501-528.

[Far06a] Farlex, *The Free Dictionary*, http://thefreedictionary.com/, 2006.

[Far06b] Farlex, *The Free Dictionary (Legal)*, http://legal-dictionary.thefreedictionary.com/, 2006.

[Far06c] Farlex, *The Free Dictionary (Financial)*, http://financial-dictionary.thefreedictionary.com/, 2006.

[Floc03] J. Floch, R. Bræk, "Using projections for the detection of anomalous behaviors," *In: Lecture Notes on Computer Science*, Vol. 2708, Springer, 2003, pp. 251-268.

[Fow03] M. Fowler, *Patterns of enterprise application architecture*, Addison-Wesley, 2003.

[Fow04] M. Fowler, *UML distilled: a brief guide to the standard object modeling language*, Addison-Wesley, 2004.

[Gao03] J.Z. Gao, H.-S.J. Tsao, Y. Wu, *Testing and quality assurance for component-based software*, Artech House, 2003.

[Gar00] J. García, J. Entialgo, F.J. Suárez, D.F. García, "Model-driven monitoring for the multi-view performance analysis of parallel embedded applications," *In: Performance Evaluation*, Vol. 39, No. 1-4, Elsevier, 2000, pp. 81-98.

[Gar79] M.R. Garey, D.S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*, W.H. Freeman and Company, 1979.

[Gar98] S. Garg, A. Puliafito, M. Telek, K. Trivedi, "Analysis of preventive maintenance in transaction-based software systems," *In: IEEE Transactions on Computers*, Vol. 47, No. 1, IEEE, 1998, pp. 96-107.

[Gat04] A.Q. Gates, S. Roach, N. Delgado, "A taxonomy and catalog of runtime software-fault monitoring tools, *In: IEEE Transactions on Software Engineering*, Vol. 30, No. 12, IEEE, 2004, pp. 859-872.

[Gha02] N. Ghaffari, A. Lau, B. Pekilis, J. Thai, R. Seviora, "Horizontal and vertical (H&V) consistency checking for software health monitoring," *In Proceedings of the First AOSD Workshop on Aspects, Components and Patterns for Infrastructure Software*, ACM, 2002, pp. 30-35.

[Gha03] N.H. Ghaffari, *State inconsistency detection in semi-stationary distributed systems*, Masters Thesis, Bell Canada Software Reliability Laboratory, University of Waterloo, Waterloo, Ontario, Canada, 2003.

[Gel85] D. Gelernter, N. Carriero, S. Chandran, S. Chang, “Parallel programming in Linda,” *In: Proceedings of the Fourteenth International Conference on Parallel Processing (ICPP)*, IEEE, 1985, pp 255-263.

[Gez03] C. Gezzi, M. Jazayeri, D. Mandrioli, *Fundamentals of Software Engineerin*g, Prentice Hall, 2003.

[Gru93] T.R. Gruber, “A translation approach to portable ontology specification”, *In: Knowledge Acquisition*, Vol. 5, No. 2, Elsevier, 1993, pp. 199-220.

[Gua98] N. Guarino, “Formal ontology and information systems,” *In: Proceedings of the First International Conference on Formal Ontology in Information Systems*, IOS Press, 1998, pp. 3-15.

[Hau01] R. Hauck, I. Radisic, “Service-oriented application management – do current techniques meet requirements?,” *In: Proceedings of the IFIP TC6 / WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, Klumer, 2001, pp. 295-303.

[Hay91] D. Hay, R. Seviora, “A real-time validator,” *In: Proceedings of the Third International Conference on Software Engineering for Real-Time Systems*, Vol. 344, IEE, 1991.

[Haz01] P. Hazy, *Event trace based software supervision*, Masters Thesis, Bell Canada Software Reliability Laboratory, University of Waterloo, Waterloo, Ontario, Canada, 2001.

[Hen01] J. Hendler, “Agents and the semantic web,” *In: Intelligent Systems*, Vol. 16, No. 2, IEEE, 2001, pp. 30-37.

[Hie01] R.M. Hierons, “Checking states and transitions of a set of communicating finite state machines,” *In: Microprocessors and Microsystems*, Vol. 24, No. 9, Elsevier, 2001, pp. 443-452.

[Hla95] M. Hlady, R. Kovacevic, J.J. Li, B.R. Pekilis, D. Prairie, T. Savor, and R.E. Seviora, "An Approach to Automatic Detection of Software Failures," *In: Proceedings of the Fifth International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 1995, pp. 314-323.

[Hon95] Honderich, T. (Ed.), *The oxford companion to philosophy*, Oxford University Press, 1995.

[Hor04] M. Horridge, H. Knublauch, A. Rector, R. Stevens, C. Wroe, *Building OWL Ontologies Using The Protégé-OWL Plugin and CO-ODE Tools (Edition 1.0)*, University Of Manchester, http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial.pdf , 2004.

[Hua99] L. Huang, *Evaluation of Resynchronization Methods for Real-Time Software Supervision*, Masters Thesis, Bell Canada Software Reliability Laboratory, University of Waterloo, Waterloo, Ontario, Canada, 1999.

[Iee90] *IEEE 610.12-1990: standard glossary of software engineering terminology*, IEEE, 1990.

[Ior94] R. Iorgulescu, R.E. Seviora, “A resynchronization method for real-time supervision,” *In: Proceedings of the Sixth Euromicro Workshop on Real-Time Systems*, IEEE, 1994, pp. 66-71.

[Itu91] International Telecommunications Union / Telecommunications Standards Sector (ITU/TSS), “Functional Specification and Description Language, In*: ITU-T Blue Books: Recommendation Z.100-104*, , 1991.

[Jia05] S. Jiang, C. Carrez, F.A. Aaagesen, "Automatic translation of service specification to a behavioral type of dynamic service verification," *In: Proceedings of the First International Workshop on Rapid Integration of Software Engineering Techniques (RISE)*, Lecture Notes in Computer Science (LNCS), Vol. 3475, Springer, 2005, pp. 34-44.

[Ken99] E.A. Kendall, "Role models – patterns of agent system analysis and design," *In: British Telecom (BT) Technical Journal*, Vol. 17, No. 4, Springer, 1999, pp. 46-56.

[Kep03] J.O. Kephart, D.M. Chess, "The vision of autonomic computing," *In: IEEE Computer*, Vol. 36, No. 1, IEEE, 2003, pp. 41-50.

[Kir04] M. Kircher, P. Jain, Pattern-oriented software architecture: patterns for resource management (Volume 3), Wiley, 2004.

[Kol98] M. Kolberg, R. Sinnott, E.H. Magill, "Engineering of internetworking TINA-based telecommunication services," *In: Proceedings of the Fourth International Conference on Telecommunications Information Networking Architecture (TINA)*, IEEE, 1998, pp. 205-213.

[Kri95] B.B. Kristensen, "Object-oriented modeling with roles," *In: Proceeding of the International Conference on Object-Oriented Information Systems (OOIS)*, Springer, 1995, pp. 57-71.

[Knu04a] H. Knublauch, R.W. Fergerson, N.F. Noy, M.A. Musen, "The Protégé-OWL plugin: an open development environment for semantic web applications," *In: Proceedings of the Third International Semantic Web Conference (ISWC)*, Lecture Notes In Computer Science, Vol. 3298, Springer, 2004, pp. 229-243.

[Knu04b] H. Knublauch, M.A. Musen, A.L. Rector, "Editing description logic ontologies with the Protégé-OWL plugin," *In: Proceedings of the International Workshop on Description Logics (DL)*, http://CEUR-WS.org/Vol-104/, CEUR-WS.org, 2004, #8.

[Kuh87] D. Kuhn, "Sources of failure in the public switched telephone network," *In: IEEE Computer*, Vol. 30, No. 4, IEEE, 1997, pp.31-36.

[Kur01] J. Kurose, K. Ross, *Computer Networking: A Top-Down Approach Featuring The Internet*, Addison-Wesley, 2001.

[Lam83] L. Lamport, "Specifying concurrent program modules," *In ACM Transactions on Programming Languages and Systems*, Vol. 5. No. 2, ACM, 1983, pp. 190-222.

[Lam84] S.S. Lam, A.U. Shankar, "Protocol verification via projections," *In: IEEE Transactions on Software Engineering*, Vol. 10, No. 4., IEEE, 1984, pp. 325-342.

[Lar98] C. Larman, *Applying UML and patterns: an introduction to object-oriented analysis and design*, Prentice-Hall, 1998.

[Lau05] A. Lau, R.E. Seviora, "Design patterns for software health monitoring," *In: Proceedings of the Tenth International Conference Engineering of Complex Computer Systems (ICECCS)*, IEEE, 2005, pp. 467-476.

[Law90] K.H. Law, T. Barsalou, G. Wiederhold, "Management of complex structural engineering objects in a relational framework," *In: Engineering with computers*, Vol. 6, No. 2, Springer, 1990, pp.81-92.

[Lee90] P.A. Lee, T. Anderson, “Fault-tolerance: principles and practice,” *In: Dependable Computing and Fault-Tolerant Systems Series*, Vol. 3, Springer, 1990.

[Lew98] S.M. Lewandowski, “Frameworks for component-based client/server computing,” *In: Computing Surveys*, Vol. 30, No. 1, ACM, 1998, pp. 3-27.

[Liu02] S. Liu, “Capturing complete and accurate requirements by refinement,” In*: Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems (ICECCS)*, IEEE, 2002, pp. 57-67.

[Lyu95] M.R. Lyu (Ed.), *Software Fault Tolerance*, Wiley, 1995.

[Lyu96] M.R. Lyu (Ed.), *Handbook of software reliability engineering*, McGraw-Hill, 1996.

[Mcc02] L.T. McCarty, “Ownership: a case study in the representation of legal concepts,” *In: Artificial Intelligence and the Law*, Vol. 10., No. 1-3, Springer, 2002, 135-161.

[Mem06] A.M. Memon, *Testing event-driven software applications: issues, challenges, and solutions*, Invited Presentation, IEEE Computer Society Chapter, University of Waterloo, http://www.cs.umd.edu/~atif/presentations/Waterloo031606.pdf, 2006.

[Mil00] K.L. Mills, H. Gomaa, “A knowledge-based method for inferring semantic concepts from visual models of system behavior,” *In: ACM Transactions on Software Engineering and Methodology*, Vol. 9, No. 3, ACM, 2000, pp. 306-337.

[Mit99] A. Mitschele, B. Müller-Clostermann, “Performance engineering of SDL/MSC systems,” *In: Computer Networks*, Vol. 31, No. 7, Elsevier, 1999, pp. 1801-1815.

[Mol93] K-H. Moller, D.J. Paulish, "An empirical investigation of software fault distribution," *In: Proceedings of the First International Software Metrics Symposium (METRICS)*, IEEE, 1993, pp. 82-90.

[Mur98] K. Murakami, R.W. Buskens, R. Ramjee, Y-J. Lin, T.F. LaPorta, “Design, implementation, and evaluation of highly available distributed call processing systems,” *In: Digest of Papers of the Twenty-Eighth International Symposium of Fault-Tolerant Computing (FTCS-98)*, IEEE, 1998, pp. 118-127.

[Noy97] N.F. Noy, C.D. Hafner, “The state of the art in ontological design: a comparative review.” *In: AI Magazine*, Vol. 18, No. 3, AAAI, 1997, 53-74.

[Noy01] N.F. Noy, D.L. McGuinness, *Ontology development 101: a guide to creating your first ontology*, Knowledge Systems Laboratory, Technical Report, KSL-01-05, Stanford, 2001.

[Omg05a] Object Management Group, *Unified Modeling Language (UML) Version 2.0 – infrastructure specification*, OMG, 2005.

[Omg05b] Object Management Group, *Unified Modeling Language (UML) Version 2.0 - superstructure specification*, OMG, 2005.

[Ost02] T. J. Ostrand, E.J. Weyuker, "The distribution of faults in a large industrial software system," *In: Proceedings of the SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, ACM. 2002, pp. 55-64.

[Par72] D.L. Parnas, “On the criteria to be used in decomposing systems into modules,” *In: Communications of the ACM*, Vol. 15, No. 12, ACM, 1972, pp. 1053-1058.

[Par00] Y. Park, S.S. Rappaport, "Performance analysis of session oriented data communications for mobile computing in cellular system," *In: Wireless Networks*, Vol. 6, No. 6, Kluwer, 2000, pp. 441-456.

[Pav01] D. Pavlovic, D.R. Smith, "Composition and refinement of behavioral specifications," *In: Proceedings of the Twenty-first International Conference on Automated Software Engineering (ASE)*, IEEE, 2001, pp. 157-166.

[Peh83] B. Pehrson, "Abstraction by structural reduction," *In: Proceedings of the IFIP WG 6.1 Third International Symposium on Protocol Specification, Testing, and Verification (PSTV)*, North-Holland, 1983, pp. 87-94.

[Pek97] B. Pekilis, R. Seviora, "Detection of Response Time Failures of Real-Time Software," *In: Proceedings of the Seventh International Symposium On Software Reliability Engineering (ISSRE)*, IEEE, 1997, pp. 38-47.

[Pek03] B.R. Pekilis, R.E. Seviora, "Automatic response performance monitoring for real-time software with nondeterministic behaviors," *In: Performance Evaluation*, Vol. 53, No. 1, Elsevier, 2003, pp. 1-22.

[Pen80] B. Penney, "The DMS-100: a switch that takes care of itself," *In: Telesis*, Vol. 4, Bell Canada, 1980, pp. 41-43.

[Pfl06] S. Lawrence-Pfleeger, J.M. Atlee, *Software engineering: theory and practice*, Prentice-Hall, 2006.

[Pla84] B. Plattner, "Real-time execution monitoring," *In: IEEE Transactions on Software Engineering*, Vol. SE-10, No. 6, IEEE, 1984, pp. 756-764.

[Rec04] A.L. Rector, N. Drummond, M. Horridge, J. Rogers, H. Knublauch, R. Stevens, H. Wang, C. Wroe, "OWL pizzas: practical experience of teaching OWL-DL: common errors and common patterns," *In: Proceedings of the Fourteenth International Conference on Knowledge Engineering and Knowledge Management (EKAW)*, Lecture Notes in Computer Science, Vol. 3257, Springer, 2004, pp. 63-81.

[Rod97] K.J. Rodham, D.R. Olsen Jr., "Nanites: an approach to structure-based monitoring," *In: ACM Transactions on Computer-Human Interactions*, Vol. 4, No. 2, ACM, 1997, pp. 103-136.

[Ros98] J. Rosenberg, H. Schulzrinne, "Internet Telephony Gateway Location," *Seventeenth Annual Conference on Computer Communications (INFOCOM)*, IEEE, 1998, pp. 488-496.

[Ros04] J-M. Rosengard, M.F. Ursu, "Ontological representations of software patterns," *In: Proceedings of the Eighth International Conference on Knowledge-Based Intelligent Information and Engineering Systems (KES)*, Lecture Notes in Computer Science (LNCS), Vol. 3215, Springer, 2004, pp. 31-37.

[Sal84] J.H. Saltzer, D.P. Reed, D.D. Clark, "End-to-end arguments in system design," *In: ACM Transactions on Computer Systems*, Vol. 2, No. 4, ACM, 1984, pp. 277-288.

[San93] S. Sankar, M. Mandal, "Concurrent run-time monitoring of formally specified programs, *In: IEEE Computer*, Vol. 26, No. 3, IEEE, 1993, pp. 32-41.

[Sav97] T. Savor, R.E. Seviora, “Hierarchical supervisors for automatic detection of software failures,” *In: Proceedings of the Eighth International Symposium On Software Reliability Engineering (ISSRE)*, IEEE, 1997, pp. 48 -59.

[Sch95] B. Schroeder, “On-line monitoring: a tutorial,” *In: IEEE Computer*, Vol. 28, No. 6, IEEE, 1995, pp. 72-78.

[Sei03] E. Seidewitz, “What models mean,” *In: IEEE Software*, Vol. 5, No. 2, IEEE, 2003, pp. 26-32.

[Sel03] B. Selic, “The pragmatics of model-driven development,” *In: IEEE Software*, Vol. 5, No. 2, IEEE, 2003, pp. 19.25.

[Sha96] M. Shaw, D. Garlan, *Software architecture: perspectives on a emerging discipline*, Prentice-Hall, 1996.

[Sha05] J. Shaheed, A. Yip, J. Jim Cunningham, “A top-level language-biased legal ontology,” *In: Workshop Proceedings, Legal Ontologies and Artificial Intelligence Techniques*, International Association for Artificial Intelligence and Law, Workshop Series No 4, Wolf Legal Publishers, 2005, pp. 13-24.

[Son02] H. Song, H.H. Chu, N. Islam, S. Kurkake, M. Katagiri, “Browser state repository service,” *In: Lecture Notes in Computer Science (LNCS)*, Vol. 2414, Springer, 2002, pp. 253-266.

[Sta06] Stanford University, *OWL Ontology Library*, http://protege.stanford.edu/plugins/owl/owl-library/, 2006.

[Sto05] O. Storz, A. Friday, N. Davies, “Supporting ordering and consistency in a distributed event heap for ubiquitous computing,” *In: Personal and Ubiquitous Computing*, Vol. 10, No. 1, ACM, 2005, pp. 45-49.

[Sul91] M. Sullivan, R. Chillarge, “Software defects and their impact on system availability: a study of field failures in operating systems,” *In: Digest of Papers of the Twenty-First International Symposium on Fault-Tolerant Computing (FTCS-21)*, IEEE, 1991, pp. 2-9.

[Swa99] W. Swartout, A. Tate, “Ontologies: Guest Editors' Introduction,” *In: Intelligent Systems: Special Issue on Ontologies*, Vol. 14, No. 1. IEEE, 1999, pp. 18-19.

[Szp02] C. Szyperski, D. Gruntz, S. Murer, *Component software: beyond object-oriented programming*, Addison-Wesley, 2002.

[Tae03] G. Taentzer, “AGG: a graph transformation environment for modeling and validating software,” *In: Proceedings of the Second International Workshop on Applications of Graph Transformations with Industrial Relevance (ACTIVE)*, Lecture Notes in Computer Science, Vol. 3062, Springer, 2004, pp. 446-453.

[Tar99] P. Tarr, H. Ossher, W. Harrison, S.M. Sutton Jr., "N degrees of separation: multi-dimensional separation of concerns, " *In: Proceedings of the Twenty-First International Conference on Software Engineering (ICSE)*, ACM, 1999, pp. 107-

[Tas02] G. Tassey, *The economic impacts of inadequate infrastructure for software testing*, National Institute of Standards and Technology (NIST), Final report, 2002.

[Tex03] G. Texier, N. Plouzeau, “Automatic management of sessions in shared spaces,” *In: Journal of Supercomputing*, Vol. 24, No. 2, Kluwer, 2003, 173-181.

[Tha01a] J. Thai, B. Pekilis, A. Lau, R. Seviora, "Detection of errors using aspect-oriented state consistency checks," *In: Supplemental Proceedings of the Twelfth International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2001, pp. 29-30.

[Tha01b] J. Thai, B. Pekilis, A. Lau, R. Seviora, "Aspect-oriented implementation of software health indicators," *In: Proceedings of the Eighth Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, 2001, 96-104.

[Tur93] K. Turner (ed), *Using Formal Description Techniques: An Introduction to Estelle, Lotos, and SDL*, Wiley, 1993.

[Usc06] M. Uschold, *Introduction to ontologies and the semantic web*, Presentation, Boeing Technology, Phantom Works, http://courses.washington.edu/imt530/schedule/8b.pdf, 2006.

[Vai01] K. Vaidyanathan, K. Trivedi, "Extended classification of software faults based on aging," *In: Proceedings of the Twelfth International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2001, pp. 27-28.

[War98] J.B. Warmer, A.G. Keppe, *The object constraint language: precise modeling with UML*, Addison-Wesley, 1998.

[Wik06] Wikipedia, *The Free Encyclopedia*, http://en.wikipedia.org/wiki/, 2006.

[Woo00] M. Wooldridge, N.R. Jennings, D. Kinny, The Gaia Methodology for agent-oriented analysis and design, *In: Autonomous Agents and Multi-Agent Systems*, Vol.3, No. 3, Kluwer, 2000, 285-312.

[W3c04a] World Wide Web Consortium (W3C), *OWL Web Ontology Language Overview*, Recommendation 10, http://www.w3.org/, February 2004.

[W3c04b] World Wide Web Consortium (W3C), *OWL Web Ontology Language Guide*, Recommendation 10, http://www.w3.org/, February 2004.

[W3c05] World Wide Web Consortium (W3C), *Representing Specified Values in OWL: value partitions and value sets*, Working Group Note 17, http://www.w3.org/, May 2005.

[Yip02] A. Yip, J. Cunningham, "Some issues in agent ownership," *In: Proceedings of the Workshop on the Law and Electronic Agents (LEA)* , CIRSFID, Universita di Bologna 2002, pp. 13-22.

[Yip03] A. Yip, J. Cunningham, "Ontological issues in agent ownership," *In: Proceedings of the Workshop on the Law and Electronic Agents (LEA)*, Norwegian Research Center for Computers and Law, Oslo, 2003, pp. 113-126.

[Yip04] A. Yip, J. Cunningham, "Legal event reasoning for software agents," *In: Proceedings of the Workshop on the Law and Electronic Agents (LEA)*, CIRSFID, Universita di Bologna, 2004, pp. 35-53.

[Yu05] Y. Yu, H. Jin, "An ontology-based host resources monitoring approach in grid environments." *In: Proceedings of the Sixth International Conference on Web-Age Information Management (WAIM)*, Lecture Notes in Computer Science, Vol. 3739, Springer, 2005, pp. 834-839.

[Zsc04] S. Zschaler, “Toward a semantic framework for non-functional specification of component-based systems,” *In: Proceedings of the Thirtieth EUROMICRO Conference*, IEEE, 2004, pp. 92-99.

[Zul04] M. Zulkernine and R.E. Seviora, "Towards automatic monitoring of component-based software systems," *In: Journal of Systems and Software,* Special Issue on Automated Component-Based Software Engineering, Vol. 74, No. 1, Elsevier, 2004, pp. 15-24.