DYNAMIC SYSTEM RECONFIGURATION

By

D'Arcy Walsh

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

School of Computer Science

CARLETON UNIVERSITY

Ottawa, Ontario

April, 2007

© Copyright by D'Arcy Walsh, 2007

Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 978-0-494-27115-5

Our file Notre référence

ISBN: 978-0-494-27115-5

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

To Victoria

Abstract

This thesis defines what dynamic reconfiguration means in the context of software components that interact to implement services by formulating models that explicitly specify the relationships that exist among changing global or local system properties, changing behavioral or structural system signatures, and intra- or inter- component change. Using a running example of the dynamic inter-operation of two financial analysis systems, it identifies and categorizes the various types of change that are required, the relationships among those types, and the system integrity characteristics that need to be considered when such changes take place.

A use case specifies the common and variable causal flow of dynamic reconfiguration. It describes the different levels of dynamism as alternatives based upon change groups. The following change groups are identified: *Pure Architectural Change*, *Pure Behavioral Change*, *Behavior-driven Structural Change*, *Comprehensive Behavior-driven Change*, *Pure Structural Change*, *Structure-driven Behavioral Change*, *Comprehensive Structure-driven Change*, *Pure Behavioral and Structural Change* and *Comprehensive Change*.

Each change group is defined to be composed of a set of related change sequences that describe how a particular change group unfolds. A change sequence combines a subset of the change types that apply within the sequence; the change types of a change sequence determine whether the particular change sequence implies architectural, behavioral, and/or structural change.

The following types of change are identified: *Architectural Change*, *Topology Change*, *Substitution*, *Protocol Change*, *Interface Change*, and *Internal Change*.

Architectural Change affects the model elements *Global Property* and *Local Property*.

There is an indirect affect upon a system's structural or behavioral signature that in turn affects the configuration of other domain model elements. *Topology Change* affects the model elements *Component* and *Communication Path*. *Substitution* affects the model elements *Component* and *Connection*. *Protocol Change* affects the model element *Service Protocol*. *Interface Change* affects the model elements *Required Service* or *Provided Service*. *Internal Change* affects the model elements *Operation*, *State Element*, and *Service Invocation*. To ensure overall system consistency in the face of change, change types are related to seven system integrity management areas through the domain model elements that underpin the interdependency.

Acknowledgements

This thesis happened because of the support of many people.

First, let me acknowledge the Communications Security Establishment for partial funding, leave time during critical time periods, and ongoing support over many years as it matured towards its current state.

I would like to thank Prof. Robert France for participating during an early briefing of formative material.

Let me acknowledge the numerous conversations I had with many different experts in the field. I want to especially thank those particular reviewers and participants of the Models conference (Models 2005), Engineering Computer-based Systems conference (ECBS 2006), Hawaiian International Conference on System Sciences (HICSS 2007), and the Software and System Modeling Journal (Springer Verlag) who I had direct contact with or who reviewed manuscripts or papers.

I would like to thank Prof. Hasan Ural, Prof. Robert France, Prof. Dorina Petriu, Prof. Tony White, and Prof. Timothy Lethbridge for participating on my Ph.D. comprehensives, proposal defense and/or examination committees.

I must acknowledge the many conversations I had with my friend Mike Schulte and his ongoing support of this endeavor. He did not live to see its completion.

I am deeply grateful to Prof. Bran Selic. I want to thank Bran for agreeing to be on my Ph.D. comprehensives, proposal defense, and examination committees and for becoming a second supervisor. This outcome could not have been achieved without his breadth and depth of understanding of computation and many years of experience. His advice,

comments, observations, and help with practical matters were crucial with respect to approach, content, form, and style. His judgment and timing was impeccable.

I cannot begin to express my gratitude to Prof. Francis Bordeleau. Francis was there from the very beginning of the process. I certainly want to thank him for being my supervisor and for being on my Ph.D. comprehensives, proposal defense, and examination committees. More importantly I want to thank Francis for being a mentor as a first class academic and by demonstrating how cutting edge techniques can be realized within an industrial context. This had an untold affect on how this thesis came to be. Over an extended period of time, he helped me navigate from the start to the finish during a journey that was sometimes arduous. Through his deep knowledge of the domain, insight, instincts, and enthusiasm for the subject matter, this thesis was achievable.

I must thank my parents, Barbara and John, who support me in everything I do. Among many other things, they have instilled in me the desire to discover, learn, and create.

Most importantly, I must thank my wife Victoria who has been with me in every conceivable way. Through the whole process, she has talked me through issues, challenges, and intellectual roadblocks. More than anyone else, this thesis happened because of her passion for life, clear thinking and perspective, strength of character, and unwavering support in every way through thick and thin.

Table of Contents

ABSTRACT	III
ACKNOWLEDGEMENTS	V
TABLE OF CONTENTS.....	VII
LIST OF TABLES	X
LIST OF FIGURES.....	X
LIST OF APPENDICES.....	XII
1. INTRODUCTION.....	1
1.1. Motivation.....	2
1.2. Context	3
1.3. Problem Statement.....	8
1.4. Approach to the Problem.....	8
1.5. Objectives.....	11
1.6. Contributions.....	12
1.7. Outline.....	13
2. RELATED RESEARCH REVIEW	14
2.1. Introduction	14
2.2. Related Research	18
2.3. Discussion.....	27
3. DOMAIN ANALYSIS.....	30
3.1. Introduction	30
3.2. Dynamic System Reconfiguration Case Study.....	32
3.2.1. Description of Original System A	36
3.2.2. Description of Original System B	38
3.2.3. Description of Systems Evolution	39
3.2.4. Description of Evolved Systems.....	40

3.2.5. Examples Global and Local System Properties.....	42
3.3. Integrity Management Areas.....	43
3.3.1. Global Consistency Management.....	44
3.3.2. Local Consistency Management.....	46
3.3.3. Active Reference Management.....	48
3.3.4. Dependent Operation Management.....	50
3.3.5. Composite Component Management.....	52
3.3.6. Constrained Operation Management.....	54
3.3.7. State Management.....	56
3.4. Change Types.....	57
3.4.1. Architectural Change.....	61
3.4.2. Topology Change.....	62
3.4.3. Substitution.....	63
3.4.4. Protocol Change.....	65
3.4.5. Interface Change.....	66
3.4.6. Internal Change.....	67
3.5. Common and Variable Causal Flow of Dynamic Reconfiguration.....	68
3.5.1. General Context of Dynamic Reconfiguration.....	69
3.5.2. Pure Architectural Change.....	72
3.5.3. Pure Behavioral Change.....	73
3.5.4. Behavior-Driven Structural Change.....	73
3.5.5. Comprehensive Behavior-Driven Change.....	74
3.5.6. Pure Structural Change.....	76
3.5.7. Structure-Driven Behavioral Change.....	76
3.5.8. Comprehensive Structure-Driven Change.....	77
3.5.9. Pure Behavioral and Structural Change.....	79
3.5.10. Comprehensive Change.....	80
3.6. Feature Model.....	81
3.6.1. Primary Concepts of Interacting Components.....	85
3.6.2. Primary Concepts of Context of Change.....	97
3.6.3. Primary Concepts of Dynamic Reconfiguration.....	108
3.7. Alloy Model.....	112
3.7.1. Alloy Model of Interacting Components.....	113
3.7.2. Alloy Model of Context of Change.....	119
3.7.3. Alloy Model of Dynamic Reconfiguration.....	123
3.7.4. Discussion.....	123
3.8. Summary and Conclusion.....	125
3.8.1. Summary.....	125
3.8.2. Conclusion.....	127
4. DOMAIN DESIGN.....	129
4.1. Introduction.....	129
4.2. Architectural Change Model.....	133
4.2.1. Domain Level.....	133
4.2.2. UML Modeling Level.....	135
4.3. Topology Change Model.....	142

4.3.1. Domain Level	143
4.3.2. UML Modeling Level	145
4.4. Substitution Model	152
4.4.1 Domain Level	152
4.4.2. UML Modeling Level	154
4.5. Protocol Change Model.....	161
4.5.1. Domain Level	161
4.5.2. UML Modeling Level	163
4.6. Interface Change Model	169
4.6.1. Domain Level	169
4.6.2. UML Modeling Level	171
4.7. Internal Change Model.....	178
4.7.1. Domain Level	178
4.7.2. UML Modeling Level	181
4.8. Composed Models	189
4.8.1. Composed Models of Pure Architectural Change	189
4.8.2. Composed Models of Pure Behavioral Change	193
4.8.3. Composed Models of Behavior-driven Structural Change	199
4.8.4. Composed Models of Comprehensive Behavior-driven Change	205
4.8.5. Compose Models of Pure Structural Change	209
4.8.6. Composed Models of Structure-driven Behavioral Change	213
4.8.7. Composed Models of Comprehensive Structure-driven Change	219
4.8.8. Composed Models of Pure Behavioral and Structural Change	224
4.8.9. Composed Models of Comprehensive Change	228
4.9. Executable Model	232
4.9.1. Using Explicit Metaclasses as an Executable Representation	233
4.9.2. Executing an Application-Specific Example	239
4.9.3. Discussion	248
4.10. Summary and Conclusion.....	250
4.10.1. Summary	251
4.10.2. Conclusion	252
5. SUMMARY, CONCLUSION, AND FUTURE WORK	254
5.1. Summary	254
5.2. Conclusion	255
5.3. Future Work	260
APPENDIX A – FINANCIAL ANALYSIS SYSTEM CASE STUDY.....	263
A.1. Introduction	263
A.2. Description of Original Systems	265
A.2.1 FAS-A Description	267
A.2.2 FAS-B Description	281
A.3. Planned Evolution of FAS-A and FAS-B.....	295

A.4. Dynamic Reconfiguration of FAS-A and FAS-B	297
A.4.1 New Representations of FAS-A and FAS-B	297
A.4.2 How FAS-A and FAS-B Evolve	303
A.5. Summary and Conclusion.....	311
 APPENDIX B – ARCHITECTURAL PATTERNS FOR DYNAMIC RECONFIGURATION	 313
B.1 Catalog of Patterns for Dynamic Reconfiguration.....	314
B.2. How these Patterns Interact with One Another.....	328
 REFERENCES	 330

List of Tables

Table 1. Common and Variable Causal Flow of Dynamic Reconfiguration	71
Table 2. Examples of Configuration Information of Interacting Components	87
Table 3. Examples of Configuration Information of External Interactions	90
Table 4. Examples of Configuration Information of Internal Behavior	93
Table 5. Examples of Configuration Information of Dependencies	96
Table 6. Examples of Configuration Information of Context of Change	99
Table 7. Examples of Configuration Information of Condition Change Criteria	103
Table 8. Examples of Configuration Information of Types of Change	105
Table 9. Examples of Configuration Information of System Integrity Characteristics	107
Table 10. Examples of Configuration Information of Dynamic Reconfiguration	110
Table 11. Pattern Catalog	314
Table 12. Change Groups and Change Sequence	329

List of Figures

Figure 1. Context of Dynamic System Reconfiguration.....	4
Figure 2. Behavioral and Structural Signatures of a Component	6
Figure 3. Broad Context of Financial Analysis	33
Figure 4. Original Control Style of System A	36
Figure 5. External Interactions of Components of System A	37
Figure 6. External Interactions and Internal Behavior of Components of System A	37
Figure 7. Original Control Style of System B	38
Figure 8. External Interactions of Components of System B	39
Figure 9. External Interactions and Internal Behavior of Components of System B.....	39
Figure 10. New Control Style of System A	40
Figure 11. New Control Style of System B	41
Figure 12. New External Interactions.....	41
Figure 13. New Internal Behavior.....	42
Figure 14. Domain of Global Consistency Management	45
Figure 15. Domain of Local Consistency Management	47
Figure 16. Domain of Active Reference Management.....	49
Figure 17. Domain of Dependent Operation Management.....	51
Figure 18. Domain of Composite Component Management.....	52
Figure 19. Domain of Constrained Operation Management.....	54
Figure 20. Domain of State Management.....	56
Figure 21. Change Types and Dependencies.....	60
Figure 22. Activity Model of General Context of Dynamic Reconfiguration	70
Figure 23. Pure Architectural Change	72
Figure 24. Pure Behavioral Change.....	73

Figure 25. Behavior-Driven Structural Change.....	74
Figure 26. Comprehensive Behavior-Driven Change	75
Figure 27. Pure Structural Change	76
Figure 28. Structure-Driven Behavioral Change.....	77
Figure 29. Comprehensive Structure-Driven Change	78
Figure 30. Pure Behavioral and Structural Change	79
Figure 31. Comprehensive Change	80
Figure 32. High Level Feature Model of Software Evolution.....	82
Figure 33. Feature Models Composed as Dynamic Reconfiguration	85
Figure 34. Feature Model of Interacting Components.....	86
Figure 35. Feature Model of External Interactions.....	88
Figure 36. Feature Model of Internal Behavior.....	91
Figure 37. Feature Model of Dependencies.....	94
Figure 38. Complete Feature Model of Interacting Components	97
Figure 39. Feature Model of Context of Change.....	98
Figure 40. Feature Model of Condition Change Criteria.....	100
Figure 41. Feature Model of Types of Change.....	104
Figure 42. Feature Model of System Integrity Characteristics.....	106
Figure 43. Complete Feature Model of Context of Change	108
Figure 44. Feature Model of Dynamic Reconfiguration	109
Figure 45. Complete Feature Model of Dynamic Reconfiguration	111
Figure 46. Example Visual Representation Generated by the Alloy Analyzer	125
Figure 47. Domain Concepts as UML Classes.....	130
Figure 48. Primary Feature Interactions of Architectural Change.....	134
Figure 49. Activity Model of Architectural Change.....	136
Figure 50. Class Model of Architectural Change	138
Figure 51. Sequence Model of Architectural Change	139
Figure 52. State Machine of Architectural Change	142
Figure 53. Primary Feature Interactions of Topology Change	143
Figure 54. Activity Model of Topology Change	147
Figure 55. Class Model of Topology Change.....	148
Figure 56. Sequence Model of Topology Change.....	150
Figure 57. State Machine Model of Topology Change	151
Figure 58. Primary Feature Interactions of Substitution	153
Figure 59. Activity Model of Substitution	155
Figure 60. Class Model of Substitution	157
Figure 61. Sequence Model of Substitution	159
Figure 62. State Machine Model of Substitution.....	160
Figure 63. Primary Feature Interactions of Protocol Change	162
Figure 64. Activity Model of Protocol Change	164
Figure 65. Class Model of Protocol Change.....	165
Figure 66. Sequence Model of Protocol Change.....	166
Figure 67. State Machine Model of Protocol Change	168
Figure 68. Primary Feature Interactions of Interface Change	170
Figure 69. Activity Model of Interface Change	172
Figure 70. Class Model of Interface Change.....	174
Figure 71. Sequence Model of Interface Change	176
Figure 72. State Machine model of Interface Change	177
Figure 73. Primary Feature Interactions of Internal Change	179
Figure 74. Activity Model of Internal Change	181
Figure 75. Class Model of Internal Change.....	183
Figure 76. Sequence Model of Internal Change	186
Figure 78. Activity Model of Pure Architectural Change	189
Figure 79. Class Model of Pure Architectural Change.....	190
Figure 80. Sequence Model of Pure Architectural Change	191
Figure 81. State Machine Model of Pure Architectural Change.....	192

Figure 82. Activity Model of Pure Behavioral Change	193
Figure 83. Class Model of Pure Behavioral Change	195
Figure 84. Sequence Model of Pure Behavioral Change	197
Figure 85. State Machine Model of Pure Behavioral Change	198
Figure 86. Activity Model of Behavioral-Driven Structural Change	200
Figure 87. Class Model of Behavior-driven Structural Change	201
Figure 88. Sequence Model of Behavior-Driven Structural Change	203
Figure 89. State Machine Model of Behavior-Driven Structural Change	204
Figure 90. Activity Model of Comprehensive Behavior-Driven Change	206
Figure 91. Class Model of Comprehensive Behavior-driven Structural Change	207
Figure 92. Sequence Model of Comprehensive Behavior-Driven Change	208
Figure 93. State Machine Model of Comprehensive Behavior-Driven Change	208
Figure 94. Activity Model of Pure Structural Change	209
Figure 95. Class Model of Pure Structural Change	210
Figure 96. Sequence Model of Pure Structural Change	212
Figure 97. State Machine Model of Pure Structural Change	213
Figure 98. Activity Model of Structure-Driven Behavioral Change	214
Figure 99. Class Model of Structure-driven Behavioral Change	216
Figure 100. Sequence Model of Structure-Driven Behavioral Change	218
Figure 101. State Machine Model of Structure-Driven Behavioral Change	219
Figure 102. Activity Model of Comprehensive Structure-Driven Change	220
Figure 103. Class Model of Comprehensive Structure-driven Change	222
Figure 104. Sequence Model of Comprehensive Structure-Driven Change	223
Figure 105. State Machine Model of Comprehensive Structure-Driven Change	223
Figure 106. Activity Model of Pure Behavioral and Structural Change	225
Figure 107. Class Model of Pure Behavioral and Structural Change	226
Figure 108. Sequence Model of Pure Behavioral and Structural Change	227
Figure 109. State Machine Model of Pure Behavioral and Structural Change	227
Figure 110. Activity Model of Comprehensive Change	229
Figure 111. Class Model of Comprehensive Change	230
Figure 112. Sequence Model of Comprehensive Change	231
Figure 113. State Machine Model of Comprehensive Change	231
Figure 114. Base versus meta-objects	235
Figure 115. Using the reflective substrate	236
Figure 116. Encoding global and local constraints	237
Figure 118. Broad Context of Financial Analysis	264
Figure 119. Off-Line Financial Analysis Architectural Style	268
Figure 120. FAS-A Use Case Maps	269
Figure 121. FAS-A Provided and Required Services	270
Figure 122. FAS-B On-Line Financial Analysis Architectural Style	282
Figure 123. FAS-B Use Case Maps	283
Figure 124. FAS-B Provided and Required Services	284
Figure 125. FAS-A Evolved Use Case Maps	298
Figure 126. FAS-B Evolved Use Case Maps	299
Figure 127. Evolved Required and Provided Services	299
Figure 128. Pattern Interaction Diagram	328

List of Appendices

Appendix A – Financial Analysis System Case Study	263
Appendix B – Architectural Patterns for Dynamic Reconfiguration	313

1. Introduction

Deployed software systems can be characterized as being ever more complex, ever more prevalent, and ever more critical in many ways to society as a whole. Examples include the trend toward mobile and distributed systems, embedded systems and embedded controlling devices, personal digital devices, and others. Since these systems serve increasingly useful roles in many application domains, it is important they are engineered for future change as the demands placed upon them vary over time.

However, predicting future user requirements or anticipating changing computing environment imposed requirements is extremely difficult and error prone, often resulting in over- or under-engineered solutions. The very nature of these systems (and by inference the requirements placed upon them) seems to continually grow in levels of complexity, interdependence, dynamism, and in other dimensions, and to continually outpace the state-of-the-art of software engineering capabilities required to respond to these driving concerns.

Adaptive computing through dynamic system reconfiguration techniques can enable the systematic evolution of software systems due to unanticipated changes in specification or requirements. The kinds of change are dynamic in the sense they have not necessarily been pre-programmed as part of the current capability of a deployed system but instead represent a realignment of the implementation of a system. Software engineering techniques that enable dynamic system reconfiguration are viewed as an important basis for building software systems that can adjust their structural and behavioral make-up in phase with their run-time contexts.

This thesis presents a well-formed domain model that is useful at build time, and at run time, to enable the process of systematically changing software due to changes in software specification or software requirements. The domain model is useful in the roles of a software architecture, metamodel, and reference model of dynamic system reconfiguration. When utilizing the domain model, the greater intent is the provision of a set of guiding principles, and an associated suite of techniques, which together help to ensure that a system, or a family of systems, can better adjust in a systematic way to dynamically changing run-time environments.

The component-based and service-oriented systems paradigm is adopted as a realization platform because complex systems can be decomposed as software components that implement services, that are units of deployment, that can in principle be dynamically reconfigured as distinct entities (through intra-component dynamic reconfiguration) and/or as cooperative entities (through inter-component dynamic reconfiguration). A component can be realized in many ways including through metaobjects that define its runtime properties or as a batch-compiled entity of an embedded system.

The rest of this chapter presents the motivation, context, problem statement, approach to the problem, objectives, contributions, and outline of the thesis.

1.1. Motivation

This thesis addresses the need for the dynamic change of a software-based Information Technology (IT) system. A primary motivation for dynamically reconfiguring these systems is the fact that existing software-intensive deployments need to evolve multiple times during their lifetime. It is assumed that a business requirement or business concern

is ultimately driving the need for change and can be expressed as a technical system specification that can be acted upon. The kind of dynamic change that is investigated is limited to automated system interactions (for example, responding to changing user interaction patterns is not in scope).

As one example, financial systems must be continually changed in order to satisfy new and emerging system requirements for the provision of software services for financial risk measurement and analysis. In this case, continual system evolution is necessary to effectively check deal pricing, generate valuation and risk numbers, stress test investment positions, perform scenario analysis, generate cash flow projections, and so on, because of the dynamically changing financial conditions which drive these kinds of systems Walsh et al [1].

Due to the increasing dependence on software intensive systems, Oreizy et al [2] identifies the need for a system to dynamically evolve in a consistent fashion to be continuously available. The source of requirements for change varies depending upon the context. Oreizy et al [3] describe an unmanned air vehicle scenario and an airborne sensor platform designed for land-use monitoring dynamically reconfigured to provide damage surveys following an earthquake.

1.2. Context

The dynamic reconfiguration of software can be viewed as an aspect of software evolution, which is the process of changing software due to changes in software specification or software requirements. Importantly, since this is dynamic change with regard to an existing running system, the existing system specification may impose hard constraints that limit the degree to which the need for change can be addressed.

A fundamental assumption is that a technical system specification may be expressed as a set of constraints. A global property is a functional or non-functional constraint that requires global knowledge for conformance. A local property is a functional or non-functional constraint that requires local (i.e. component-level) knowledge for conformance. Examples of global properties include system level performance, availability, synchronization, distribution, security, or control style constraints. Examples of local properties include limits on the number of service invocations or service bindings, state element immutability, local performance, or local availability constraints.

Dynamic system reconfiguration is addressed within the context shown in Figure 1. What follows is a description of each of the main elements that makeup this context in order to more clearly define what is in scope and what is out of scope for this investigation.

Figure 1. Context of Dynamic System Reconfiguration

A component-based software system comprises *Interacting Components* and *Component Framework*. *Computing Environment* is the run-time context of the system. A component-based software system is deployed and executes within this environment.

Component Framework is the context for the instantiation of components and for the provision of services for coordinating those components that have been realized within the framework. The framework constrains how a component interacts with other

components for the component to be an independent yet cooperative member of an overall system of components. More specifically, *Component Framework* provides capabilities for their installation and initial configuration, loading and instantiation, configuration and connection to any required system artifacts, and for governing dynamic system reconfiguration.

Interacting Components form a system that implements required functional and non-functional properties through provided and required services (and associated service protocols). As a member of a greater system, a component is a unit of deployment whose encapsulated internal behavior satisfies its external interaction obligations. Its internal behavior may be implemented through the recursive composition of other components.

A component has a behavioral signature and a structural signature (See Figure 2). A component's behavioral signature is specific kinds of dependencies that determine its internal behavior within a component-based system. The behavioral signature is composed of (i) *service to service protocol*, (ii) *operation to required service*, (iii) *operation to provided service*, (iv) *operation to operation*, (v) *operation to state element*, and (vi) *operation to composite component service* dependencies. A component's structural signature is specific kinds of dependencies that determine its external interactions within a component-based system. The structural signature is composed of (a) *component to component* and (b) *required service to provided service* dependencies.

Figure 2. Behavioral and Structural Signatures of a Component

The following are examples of each kind of dependency in the context of the airborne land use monitoring system:

- An example of a *service to service protocol* dependency is a provided or required service of a new component that implements damage survey capabilities of hurricanes that conforms to a land-use monitoring service protocol of an airborne land use monitoring system.
- An example of an *operation to required service* dependency is a damage assessment operation of the damage survey component that requires a provided service for land use information of another component of the airborne land use monitoring system.

- An example of an *operation to provided service* dependency is a damage assessment operation of the damage survey component that implements hurricane damage assessments as a provided service for other components of the airborne land use monitoring system.
- An example of an *operation to operation* dependency is a damage assessment operation of the damage survey component whose damage assessments are computed based on future damage projections implemented by another local operation of the component.
- An example of an *operation to state element* dependency is a damage assessment operation of the damage survey component that accesses a state element of the component for previous damage assessments as part of its implementation.
- An example of an *operation to composite component service* dependency is a damage assessment operation of the damage survey component whose damage assessments are computed based on a service provided by a composite component that specializes in hurricane predictions and damage assessments.
- An example of a *component to component* dependency is a communication path that has been established between a new damage survey component and an existing land-use monitoring component of a component-based airborne land-use monitoring system.
- An example of a *required service to provided service* dependency is a connection that has been established between a required service of a damage survey component and a provided service of a land use component of the airborne land-use monitoring system.

A component framework may change just as well as the configuration of components that it instantiated. This thesis focuses on component-level dynamic reconfiguration. However, the model execution environment section provides an example, through simulation, of a component framework that does change to support the dynamic reconfiguration models that are presented.

Dynamic change may well lead to the need for further dynamic change (a system may not reach an acceptable equilibrium). Managing this kind of ‘cycle of change’ is considered out of the scope of this thesis and to be future work. This investigation focuses on ‘discrete change events’ and the formulation of a model-based specification of such events.

Finally, in this context, a context-aware component-based software system is viewed to be a system that can continually dynamically reconfigure itself to stay in phase with its (changing) computing environment. This investigation describes an approach that enables context-aware computing. However, the latter is considered future work.

1.3. Problem Statement

The problem addressed is the formulation of a domain model of the dynamic reconfiguration of interacting software components due to user-driven or computing environment-imposed discrete change events.

1.4. Approach to the Problem

In response to the problem, and as a way to manage the inherent complexity of representing dynamic change, models are used to abstract away details of the underlying implementation technologies in order to specify separate dynamic reconfiguration

concerns that may then be composed to represent different levels of dynamism. A model-based approach is used to identify, abstractly, a set of change types and system integrity characteristics that together can affect the comprehensive change of a component-based software system in a consistent manner. The evolution of a particular system conforms to one or more of these general types but within the context of the particular conditions that motivated the change and the concrete representations of that system.

First, a domain analysis is presented that defines the core concepts that make up a reconfigurable system. It is inherently component based since it represents a software system as a configuration of autonomous cooperating components. Dynamic reconfiguration is therefore viewed as a run-time realignment of that configuration or the individual elements in that configuration. The essentials of the proposed domain model are first expressed in the form of a conceptually simple feature model. This in turn is used as a basis for deriving a more refined formal model expressed using the Alloy language. This allows the proposed domain model to be formally validated for consistency.

Second, a domain design of a software architecture of dynamic reconfiguration of the domain analysis is presented. The domain design is intended to apply for a range of applications; as a meta-model it describes the common aspects of dynamic reconfiguration of specific instance systems. As a demonstration, the domain design is implemented using a computing environment that supports explicit metaclass programming to illustrate the simulation of an application-specific example of dynamic change.

To summarize, when analyzing the domain of dynamic reconfiguration, the following assumptions are made:

- The kind of dynamic change that is investigated is limited to automated system interactions (for example, responding to changing user interaction patterns is not in scope);
- A technical system specification may be expressed as a set of constraints;
- A component is assumed to be a unit of deployment whose encapsulated internal behavior satisfies its external interaction obligations;
- A component has a behavioral signature and a structural signature;
- Dynamic change may well lead to the need for further dynamic change (a system may not reach an acceptable equilibrium). Managing this kind of ‘cycle of change’ is considered out of the scope of this thesis and to be future work;
- A component framework may change just as well as the configuration of components that it instantiated. This thesis focuses on component-level dynamic reconfiguration; and
- A context-aware component-based software system is viewed to be a system that can continually dynamically reconfigure itself to stay in phase with its (changing) computing environment. Context-aware computing is considered to be out of the scope of this thesis and to be future work.

To fully realize the proposed model-based specification, certain facilities or infrastructure are assumed to be in place (usually through an appropriate component framework). This includes:

- A facility that supports the instantiation and deployment of software as autonomous components;
- A communication infrastructure (which could support atomic broadcasting);

- A facility that supports introspection and intercession (which could include a meta-circular interpreter);
- A facility that supports constraint-based computing (including a constraint solver);
- A dynamic change mechanism (including a connection/reconnection capability); and
- A facility that supports different kinds of control (for example, centralized/distributed/hierarchical, open/closed loop, and feedback/feedforward control).

1.5. Objectives

The overall objective of this thesis is to provide a domain model to enable deployed software systems to evolve during their lifetime in response to discrete change events.

The particular objectives are:

- (O-1) Analyze existing software architecture, infrastructure and patterns to understand the different dimensions of dynamic reconfiguration;
- (O-2) Specify a set of model-based constructs that enable the representation of distinct dimensions of dynamic reconfiguration;
- (O-3) Specify a model-based framework that enables modeling combined dimensions of dynamic reconfiguration;
- (O-4) Realize executable representations of different dimensions of dynamic reconfiguration;
- (O-5) Provide a model-based foundation for future work which extends these dynamic reconfiguration modeling constructs to enable managing the lifecycle of a system, advanced fault-tolerance techniques, more formal methods, and representations of context-aware systems; and

- (O-6) Leverage, and be compatible with, existing modeling standards and approaches that are applicable (in particular Model-Driven Engineering (MDE) and Service-Oriented Architecture (SOA)-based technology).

1.6. Contributions

To meet these objectives, this thesis defines what dynamic reconfiguration means in the context of components that interact to implement services by formulating models that explicitly specify the relationships that exist among changing global or local system properties, changing behavioral or structural system signatures, and intra- or inter-component change.

The specific contributions of this thesis that relate to these objectives are:

- (C-1) Definition of a domain model of dynamic reconfiguration through the identification of relevant terms and concepts that define interacting components, context of change, and dynamic reconfiguration;
- (C-2) Validation of the domain analysis;
- (C-3) Definition of a Unified Modeling Language (UML) 2 representation of the distinct dimensions of dynamic reconfiguration;
- (C-4) Definition of a model-based dynamic reconfiguration composition framework that integrates the distinct modeling dimensions of the UML representation that were formulated as part of C-3; and
- (C-5) Demonstration of the applicability of the composition framework through a representative case study.

1.7. Outline

Chapter 2 reviews significant relevant research with a view to understanding and characterizing the different aspects of the dynamic reconfiguration of component-based software systems that implement services. Four broad areas of concern are identified which sets the context when formulating the domain model.

Chapter 3 presents a domain analysis of the primary concepts of interacting components, context of change and dynamic reconfiguration. A running example is used to illustrate a set of system integrity characteristics and change types and to define a use case of the common and variable causal flow of dynamic change. Using a satisfiability solver, the domain analysis is validated as an executable specification.

Chapter 4 presents a domain design of dynamic system reconfiguration. First, UML based representations of the different types of system evolution are defined and then composed models that represent the common and variable casual flow are described. A model execution environment is then presented that demonstrates the execution of the composed models within an application-specific context.

Chapter 5 concludes by summarizing the model, reviewing the contributions that have been made, discussing how the use case of dynamic change is satisfied, reviewing future work to be undertaken, and making final statements.

2. Related Research Review

2.1. Introduction

The problem of software evolution is long standing and solutions have been attempted for a least thirty-five years Selic [4]. Over the past twenty years, a number of research activities have been conducted to investigate dynamic reconfiguration.

Jarke [5] presents a meta model of requirements engineering that distinguishes between user-defined and computing environment imposed requirements which must be reconciled in order to satisfy user needs within an application-specific context. Importantly, both sources of requirements may change over time and their reconciliation is a continual process. As stated, the difficulty of predicting future user requirements or anticipating changing computing environment imposed requirements, and the subsequent problems of over- or under- engineering software solutions because of this, lead to the notion of deploying software systems that can evolve at run-time in a systematic fashion.

In this regard, Oreizy et al [3] observe that the greater objective of software engineering is to aim for the systematic design and deployment of systems that retain full plasticity throughout their lifecycle. To this end, they motivate the need for software that can systematically self-adapt and they identify a range of open issues that must be addressed.

Open issues identified in Oreizy [2] include the following. How are runtime changes to the architectural model governed? Are the model and implementation consistent with one another? When can runtime changes be applied to the system? How are changes to the architectural model mapped to corresponding changes in the implementation? What is the impact of architectural styles on analyzing and implementing runtime changes? What is the impact of architectural connectors on

analyzing and implementing runtime changes? How should non-instantaneous changes be addressed? What is the impact of supporting decentralized software evolution?

Open issues identified in Oreizy et al [3] include the following. Under what conditions does the system undergo adaptation? Should the system be open-adaptive (allowing new application behaviors) or closed-adaptive (allowing only environmental changes)? What type of autonomy must be supported? Under what circumstances is adaptation cost-effective? How often is adaptation considered? What kind of information must be collected to make adaptation decisions? How accurate and current must the information be?

They present a description of the high-level processes of a general-purpose approach to self-adaptive software. The spectrum of self-adaptation is described ranging from selecting from predetermined alternatives that support localized change to evolutionary techniques that utilize knowledge of the environment and knowledge gained from previous evolutionary activities. This approach distinguishes between adaptation management and evolution management of a system. The sub-processes of adaptation management include planning changes, deploying change descriptions, enacting changes and collecting observations, evaluating and monitoring observations, planning changes, and so on. The sub-processes of evolution management encompass an architectural model, an implementation of the model, maintaining consistency and system integrity, and, overlapping with adaptation management, enacting changes and collecting observations.

The following are important recent research initiatives that illustrate different approaches for deploying software that can systematically self-adapt. These initiatives

have indirectly informed the work presented in this thesis; the next section of this chapter reviews specific investigations that have had a more direct impact.

Karsai and Szitipánovits [6] present a comprehensive approach to self-adaptive software based on model integrated computing techniques. Models are used to represent a computer application, its runtime environment, and the relationship between them. They describe a multi-graph architecture of a self-adaptive system and model interpretation within this architecture. The architecture is composed of an application-level, model integrated programs synthesis level, and metalevel. The key issues identified are (i) how self-adaptive systems are represented with respect to separating time-variant and time invariant elements and formalizing the time variant aspects and (ii) the reconfiguration mechanism that maps these representations to executable systems.

Colman and Han [7] identify levels of autonomy of software entities and explores the relationships among these levels with respect to different modes of software organization. An encapsulated object exhibits process autonomy. An intelligent agent/object exhibits process and goal state autonomy. A proactive cooperative or competitive agent exhibits process, goal state, and intentional autonomy. An anti-social or malicious agent exhibits process, goal state, intentional, and constraint autonomy. Colman and Han [8, 9] reviews Role-Oriented Adaptive Design (ROAD) as a software architectural approach that incorporates entities with differentiated autonomy using association aspects to implement organizational contracts. Colman et al [10] presents adaptive application-specific middleware services (based on a service composite role structure) that can be recursively composed and distributed using the ROAD framework.

Minsky et al [11] and Wermelinger et al [12-14] investigate more formal techniques for describing architectures that support dynamic re-configuration. For systems that operate under Law Governed Interaction (LGI), Minsky et al [11] describe a reconfiguration suite as a set of primitive operations. The reconfiguration suite is used to carry out any reconfigurations of a system based on system interactions that are governed by an explicit and strictly enforced set of rules.

Wermelinger et al [12-14] present a graph-based formal language for specifying changes over the configuration of a system based on labeled graph rewriting. The architecture of a system is represented as a configuration built from (transient) connectors and individual programs which can be transformed into a single, semantically equivalent, program that represents the whole system (this transformation operation is given by a universal categorical construction called a colimit). Dynamic system reconfiguration is represented as the interleaving of graph rewriting rules performed on the architecture and computations performed on the colimit.

Wermelinger et al [14] investigates the semantics of reconfiguration by identifying two kinds of reconfiguration commands: basic commands that perform the actual reconfiguration and composite commands and scripts, that control the flow of execution. Examples of basic commands include component creation, component refinement, connector creation, removal, query, and assignment. They also describe composite commands and scripts for enacting control flow.

Based on Kramer et al [15], Wermelinger et al [13, 16] presents a dynamic reconfiguration model that can describe the mapping and validation of reconfigurations when enacting a configuration change from one configuration specification to another

configuration specification. A configuration manager is responsible for issuing appropriate commands to ensure valid changes are realized at the system level.

Wermelinger et al [13, 17] present a chemical model as a formalism for representing dynamic software architecture. The Chemical Abstract Machine (CHAM) model of computation is adopted to represent system reconfiguration as a sequence of reactions between data elements (i.e. component, connection, data and command) that are represented as molecules within the model. With this approach there is no explicit control mechanism. The current state of the system is represented as a multi-set of data element molecules (the solution). This set changes when rules are applied (which are chosen non-deterministically and possibly applied in parallel) that govern subsequent system reactions until no further reaction rule is applicable.

The next section reviews significant relevant research with a view to establishing a direct understanding of the different aspects of dynamic system reconfiguration as a unified domain model that applies within the context of the component-based systems paradigm (where components implement services). It is intended that future research activities extend the domain model when investigating how it can support higher-order change, different levels of autonomy, or more formal techniques.

2.2. Related Research

Medvidovic et al [18] provides an extensive summary and comparison of software Architecture Description Languages (ADLs). In the context of dynamic systems, Magee et al [15, 19-22] makes fundamental contributions. They present the basic context and many of the concepts that subsequent work builds upon. This includes a description of the basic properties required of a system to support dynamic reconfiguration and dynamic

structure, the nature of dynamic change with respect to the architecture of a system, and more recently, the feasibility of systems whose components can correctly and automatically configure their interaction.

Georgiadis et al [21] investigates minimizing the explicit management of the evolution of systems, and therefore maximizing their self-evolution, by preserving architectural properties while still allowing an individual component's state to influence reconfiguration. Architectural properties are preserved through compliance with architectural constraints. A component may influence re-configuration through its interface attributes. The goal is to allow components to automatically interact in a way that is compliant with an overall architectural specification.

As an example, a component may automatically emerge as a result of user action (a user requirement for particular functionality) or as a result of computing environment-imposed requirements (such as a failure). The idea is to ensure a system remains well formed even with the dynamic addition or removal of components. The architecture specification in this context is not an explicit description of component instances and their interconnection but instead a set of constraints, similar to an architectural style, which are used to check a particular system reconfiguration for conformance.

Based on the Darwin Architecture Description Language (ADL) Magee et al [22], Alloy is adopted as a language to express structural constraints and the Alloy Analyzer is used to analyze or verify these specifications Jackson [23] and Jackson et al [24]. It is noted that a subset of constraints require global knowledge of the system (i.e. the constraint that the pipeline forms a single chain) while the rest may be enforced by a

component with no knowledge of the rest of the system (i.e. the cardinality constraint that a provided service has at most one required service bound to it and vice-versa).

The runtime architecture presented relies on the use of an atomic broadcast mechanism to maintain a consistent view of the current configuration of the architecture of a system Birman et al [25]. Georgiadis et al [21] note maintaining a consistent configuration view is a crucial property for a correctly self-organizing system. With this approach, a runtime component's implementation is packaged with a component manager and a configuration view. In addition to providing an interface between component implementations, port objects provide an interface for interaction between the component implementation and its component manager. The component implementation interacts with the component manager to indicate a significant change of its internal state. The component manager interacts with the component implementation when a required port becomes bound or unbound from a remote interface.

The configuration view is updated in response to local or remote binding or unbinding actions (including the emergence of new components). The component manager is responsible for evaluating the set of architectural constraints of the system with respect to the current configuration view when determining the binding or unbinding actions that are needed to conform to these constraints. Self-organization by constraint satisfaction is achieved by evaluating a selector function for each required port to find a provided port given the current configuration and associated binding or unbinding actions.

The following key concepts are identified which impact the make-up of the domain model: conditions for change, architectural properties as constraints, architectural change, topology change, state capture and restoration, managing active references (in this case to

establish quiescence), elements of the system model, and global versus local consistency in the context of the component-based systems paradigm.

Bailek [26] presents a three-layer model architecture of a dynamically updateable, component-based system. The model consists of an architecture layer (with an architecture manager), a configuration layer (with a component manager), and an application layer (with component interface and implementation specifications). He identifies three levels of dynamism in Architecture Description Languages (ADLs) including interactive dynamism (which allows changing data only in fixed connections between fixed components); structural dynamism (which allows adding/removing components and connections to a system); and architectural dynamism (which allows changing whole configurations of components).

He also identifies three kinds of dynamic software updates which include dynamic linking (where new objects are deployed by changing all object references from old to new object versions); static state transfer (in which state is saved and the system stopped while the new version replaces the old version with the state of the old version rolled into the new version); and dynamic state transfer (in which a state transfer function transfers state within an executing system to its new version). Finally, he reviews update requests, performing updates, and maintaining application consistency through examples of local update and global update operation sequences.

The following key concepts are identified which impact the make-up of the domain model: higher-order architectural change, structural versus protocol change, state capture and restoration, managing dependencies during internal change (in this case when managing different object versions), managing active references (in this case to establish

quiescence), and global versus local consistency in the context of the component-based systems paradigm.

Warren and Sommerville [27] suggest an important aspect of dynamic configuration is the ability to realize continually running and flexible systems. They note that a system may be dynamically (re)configured to effect reactive change or evolutionary change. Reactive change is viewed as a system response to an (unpredictable) dynamic environment. Evolutionary change is viewed as a more predictable response to a change in system requirements (including software updates or more static environment changes).

They investigate maintaining system consistency in the presence of dynamic configuration. Their approach is to manage dynamic configuration through a system model that captures each component's structural and behavioral information through system constraint specifications. In this context, applying a version descriptor triggers a transition from one system configuration state to another system configuration state. Important aspects of dynamic configuration include the synchronization of dynamic configuration information with respect to global consistency, local consistency, dependent operations, constrained operations, and composite components. In addition, persistent state management is viewed as important to component state capture and restoration.

The following key concepts are identified which impact the make-up of the domain model: predictable versus unpredictable stimulus for change, structural versus behavioral aspects of the system model, and global, local, dependent operation, constrained operation, composite component, and component state consistency management.

Guo [28] suggests a component is comprised of four types of elements which include component interaction elements, data elements, processing (or behavioral elements) and state elements. Given these elements, he identifies affected-by, affects, and related relationships. Based on these three types of relationships, he then distinguishes between type adaptation (for substitution), interface adaptation (to alter the interaction of component elements), and behavioral adaptation (that changes the function of a component). He also identifies different kinds of component-based specifications when matching these specifications with formal interface specifications to enable system reconfiguration.

The following key concepts are identified which impact the make-up of the domain model: elements of the system model and substitution, interface change, and internal change.

Appavoo et al [29] investigate hot swapping as a technique towards autonomic computing systems through code inter-positioning or code replacement. Interposition allows for the insertion of a component between existing components (as an example, a progressive monitoring capability is discussed). Replacement allows an active component to be replaced (as an example, upgraded components better suited to a new environment is discussed).

The three key issues identified are (a) establishing a quiescent state so that it is safe to transfer state and swap references, (b) determining what state needs to be transferred safely and efficiently, and (c) the need to swap all references of clients to the new component. For (a), locking strategies are discussed. For (b), it is noted transferring state

to a canonical form could lose context and be expensive. For (c), front-ending a swappable component with a special component to forward references is discussed.

The following key concepts are identified which impact the make-up of the domain model: conditions for change, topology change and substitution, and active reference and component state consistency management.

Yellin [30] characterize the adaptive component problem as the challenge of dynamically switching between multiple implementations of a component when each implementation is optimized for a particular request workload. The goal of this investigation is an algorithm for determining the runtime cost of switching component implementations. To this end an algorithm named Delta is demonstrated for the distributed publisher/subscribe problem and the data structure selection problem, which optimizes at the level of an individual component. He suggests these kinds of algorithms will be important within the context of dynamic architecture description languages noting that domain knowledge should be utilized to obtain better results.

The following key concepts are identified which impact the make-up of the domain model: conditions for change and its impact, internal change, and local consistency management.

Whisnant et al [31] describe an approach for capturing the structure, runtime behavior and dependencies of a distributed system that represents dynamic reconfiguration as changing code blocks. In this model, operations invoke code blocks to bring about state changes. The variables accessed during the execution of an operation constitute the signature of the operation. The set of operation signatures of the system is used to determine whether any data flow dependencies are affected by a dynamic change.

They note their approach, in contrast to ADL-based models, is implementation based and bottom-up without the need for a larger architecture description model. It is further observed that approaches such as Kramer et al [19] require the system to be quiescent before reconfiguration in order to preserve consistency. An alternative approach of incorporating architectural styles (expressed as pre-determined constraints) and models as first class entities within an adaptation framework is also recognized.

The following key concepts are identified which impact the make-up of the domain model: constraint-based architectural properties, conditions for change and its impact, internal change, structural versus behavioral dependencies, and elements of the system model.

Batista and Rodriguez [32] defines dynamic reconfiguration as being classified according to the moment of its definition as either a programmed reconfiguration or an ad-hoc reconfiguration. In the case of programmed reconfiguration (e.g. automatic reconfiguration), changes are defined at system design time and are enacted when known conditions become true. In the case of ad-hoc reconfiguration (e.g. software maintenance), changes occur during system execution at an unpredictable time. They suggest that the operations that are typically provided by systems for reconfiguration are the addition and removal of components and links (connectors).

They describe a configuration language and an interconnection model that is employed to allow the specification of a system and the definition of component interactions in order to plug components together at the configuration level. The configuration language makes use of component interface information that describes required and provided communication objects. The configuration program then determines which components

will be used in an application and links output to input channels to define the flow of information. The importance of facilities to describe the global organization of a system is emphasized.

The following key concepts are identified which impact the make-up of the domain model: conditions for change and its impact, topology change, global consistency, and elements of the system model.

de Paula et al [33] investigate the use of Architecture Description Languages (ADLs) for the specification and verification of re-configurable architectures that allow the evolution of a system during its execution. They observe that the most common dynamic configuration operations are the addition of new components, the upgrading of existing components, the removal of unnecessary components, the reconfiguration of application architecture, and the reconfiguration of system architecture.

The following key concepts are identified which impact the make-up of the domain model: architectural change, topology change, substitution, global consistency, and elements of the system model.

Truyen et al [34] pursue the dynamic customization of component-based systems through a set of potential extensions that can be selectively integrated within the core functionality of a system. In particular, they investigate run-time customization, modular customization, system-wide and consistent integration of extensions, client-specific integration of extensions, context-specific and dynamic combination of extensions, scalability of the customization process, and ease of use.

The following key concepts are identified which impact the make-up of the domain model: architectural change, constraint-based architectural properties, global consistency, and elements of the system model.

2.3. Discussion

This chapter reviewed related research work that investigates techniques or mechanisms to enact and manage dynamic change. From this review, the following areas of concern are identified in order to better frame the general context when defining a domain model of dynamic system reconfiguration:

- ◆ Conditions for evolution establish under what circumstances a system adapts;
- ◆ Types of change determine how a system evolves given that the conditions for change had been met;
- ◆ Circumstances when evolution is cost effective determines whether the change anticipated is worthwhile enacting; and
- ◆ Management of system evolution ensures consistency constraints are not violated.

The conditions for evolution occur because change was user-driven or computing environment-imposed. In either case, there are new or augmented functional or non-functional requirements of the system currently deployed. The conditions for functional change may or may not invalidate the original non-functional conditions and vice-versa. Also, the conditions for change may be expected, and therefore planned for within the design of the system, or unexpected, and therefore involve more fundamental, and possibly more pervasive, behavioral or structural system evolution. Finally, the conditions for evolution may imply that the time required to enact the required change was relatively bounded or unbounded compared to other conditions for evolution.

The types of change indicate what common behavioral and/or structural patterns or configurations of change were needed of a system under evolution. These are (metalevel) specifications that are realized by any evolutionary system; these specifications define abstractly how an evolutionary system could evolve. The process of a particular system's evolution depends on the conditions for change, which determines what type of change was required, and the particular concrete representations that composed the system under evolution.

System evolution proceeds depending on the cost of affecting change. This depends on the nature of the conditions for change and the impact of enacting the type of change required with respect to the current system. At one end of the spectrum, the conditions for change may be driven by the emergence of a new mandatory requirement that must be met no matter what the cost of enacting this change. At the other end of the spectrum, the type of change may be restricted when enacted to ensure existing functional and/or non-functional properties of the system are preserved.

The management of system evolution is composed of system level characteristics that explicitly manage dynamic reconfiguration to help ensure overall system integrity. Which integrity characteristics apply depend upon the conditions of change, the type of change required, and the concrete representations that composed the currently deployed system under evolution.

No one approach reviewed defines a comprehensive conceptual framework that addresses the four broader concerns just described. For this reason, attention is focused on this body of research work, together, contributing source material towards the definition of this domain model. This source material is then augmented when

formulating an integrated and comprehensive domain model. To this end, a domain model of dynamic system reconfiguration is presented in Walsh et al [35] and is more fully reported on in Walsh et al [36]. A case study of an application-specific example of changing global and local properties, leading to comprehensive change, is presented in Walsh et al [1] and is more fully reported on in Walsh et al [36]. A constrained executable model of dynamic system reconfiguration is reported on in Walsh et al [37].

Finally, based on the related research, it becomes clear that the ability to allow for unanticipated extension and composition has direct applicability for addressing each of the broader concerns. This thesis utilizes model-based techniques to achieve separation of concerns when modeling different aspects of dynamic change. These multiple-dimensional models are then composed to define different levels of dynamism.

3. Domain Analysis

3.1. Introduction

As was motivated in Chapter 1, software systems that are dynamically reconfigurable are more responsive to changing functional and non-functional requirements, including changing operational settings that may change their preferred architecture. This chapter uses domain analysis techniques to define what dynamic reconfiguration means in the context of interacting components [38-41].

Domain analysis is "the process of identifying, collecting, organizing, and representing the relevant information in a domain, based upon the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within a domain" Software Engineering Institute [42]. The analysis should "carefully bound the domain being considered, consider commonalties and differences of the systems in the domain, organize an understanding of the relationships between the various elements in the domain, and represent this understanding in a useful way" Software Engineering Institute [42].

The particular approach adopted is Feature-Oriented Domain Analysis (FODA), developed at the Software Engineering Institute (SEI). FODA defines a process for domain analysis that is characterized by three basic phases:

- Context Analysis: defining the extent (or bounds) of a domain for analysis
- Domain Modeling: providing a description of the problem space in the domain that is addressed by software
- Domain Design: creating the software architecture(s) that implement solutions to the problems in the domain

Chapter 1 presented the context of the dynamic reconfiguration problem domain investigated in this thesis. Within this context, a change to a system may be either *user driven* or *computing environment imposed*. In either case, there will be new or augmented functional and non-functional requirements on the system. Importantly, it may be required that the functional change must not violate the original non-functional conditions (e.g., response time, availability) or vice-versa. A change may either be *expected*, and therefore planned for within the design of the system, or *unexpected*, and therefore involving more fundamental and possibly more pervasive behavioral and structural changes. Finally, the time to enact the required change may be *bounded* or *unbounded* compared to other conditions for evolution Oriey et al [2, 43].

This chapter defines a domain model that identifies a set of common features that comprise a dynamically reconfigurable component-based software system that implements services. It is component based since it represents a software system as a configuration of autonomous cooperating components [44, 45]. Dynamic reconfiguration is therefore viewed as a run-time realignment of that configuration [3, 21, 29, 31, 46]. The particular kinds of realignment are represented as types of change. The overall intent is, within a well-defined scope, the accurate and complete description of the concepts that compose this particular problem domain in a manner that leads to the formulation of a domain design, which is presented in Chapter 4.

Using a running example, the chapter identifies and categorizes the various types of change that may be required, the relationships among those types, and the system integrity characteristics that need to be considered when such changes take place. A

system under evolution implements one or more of these general types within the context of the conditions that motivated the change and the concrete representations of that system. As a set, the types of change represent a comprehensive capability of how a component-based software system may evolve. They are governed by the system integrity characteristics that help ensure overall system consistency in the face of change.

The rest of this chapter first presents a case study of examples of global and local properties (which is based on the complete description given in Appendix A). It then identifies integrity management areas and types of change that occur in a dynamically evolving system. The common and variable causal flow of dynamic reconfiguration is then described. The essentials of the proposed domain model are then expressed in the form of a conceptually simple feature model [38, 39]. This is used as a basis for deriving a more refined formal model expressed using the Alloy language [23, 24, 47-49]; this allows the proposed domain model to be formally validated for consistency. In Chapter 4, the feature and Alloy models are used to produce a UML model [39, 40, 50, 51] that combines the various aspects into a software architecture of dynamic reconfiguration [52-59].

3.2. Dynamic System Reconfiguration Case Study

An application-specific case study is used to motivate the utility of dynamic system reconfiguration and provide a sufficiently complex context for modeling different kinds of change and related system integrity characteristics, and for formalizing a complete set of change sequences. The case study is formulated based on well-known and accepted concepts originating from the financial analysis and component-based systems domains. The complete case study description is given in Appendix A.

A Financial Analysis System (FAS) is a system of financial relationships and cash flows. This is considered a complex modeling domain in which adaptability is required because of the volatile nature of the world of finance. As shown in Figure 3, affective investment, operating, and financing business decisions are ultimately expressed as a linkage between cash flows and the creation of economic value Lee and Smith [60] and Helfert [61].

The Broad Context of Financial/Economic Analysis

Figure 3. Broad Context of Financial Analysis

This problem domain encompasses the challenge of the continual reconciliation of user-defined needs and computing environment-imposed requirements. A financial analysis system must provide affective modeling techniques that facilitate the integration of prior knowledge and adaptation to changing legal, market, or regulatory conditions. In this context, data-driven change, that minimizes human intervention and bias, must be

systematically integrated with human-driven information that is the result of naturally adaptive and perceptive processes. The challenge may be summarized as the need for techniques for incorporating, enhancing or augmenting prior knowledge through systematic evolution.

Valuation is concerned with whether shareholder value is being created and is primarily based on the present value analysis of future cash flows. It is in essence a future-oriented process that encompasses projecting current economic performance over a time span applicable to the nature of a given business activity and its market segment. To do this, there is often a distinction made between short-term and long-term prediction and any analytic outputs may be indicator and/or magnitude based information. Definitions of different value types include economic, market, book, liquidation, break-up, reproduction, collateral, assessed, appraised, going concern, and shareholder value.

Using time-adjusted measures, net investment, operating cash inflow, economic life and terminal value typically represent cash flow. Methods of cash flow analysis include relatively simple payback or Return on Investment (ROI) calculations or more complex economic investment measures such as net present value, present value payback, profitability index, internal rate of return, or annualized net present value.

The following is an example of changing global and local properties leading to comprehensive change. The example describes the components and the dynamic interoperation of two initially decoupled financial systems that specialize in maintaining knowledge and providing predictions about a particular sector of the economy. System A's clients are concerned with shorter-term predictions. System B's clients are concerned with longer-term predictions.

The following is a high-level description of the components that compose these systems:

Scenario Analysis: This component is responsible for generating shorter-term or longer-term indicator and magnitude-based common or preferred stock or bond values predictions. These predictions are based on current market conditions available through the energy sector knowledge base component. The shorter-term or longer-term predicted values are then made available to clients. The energy sector knowledge representations are updated accordingly.

Energy Sector Knowledge: This component is responsible for managing raw data about the energy sector that is retained over time as immutable information. In addition, it is responsible for retaining energy sector knowledge representations inferred from this raw information. The knowledge representations may change but their composition must ultimately be governed by human-driven information.

Energy Sector Conditions: This component is responsible for accessing on-line raw information about the energy sector market conditions. This data is streamed to follow-on components as the basis for subsequent financial analysis methods.

Cash Flow Projections: This component is responsible for providing future cash flows financial scenarios based on appropriate methods of cash flow analysis and time adjusted measures using the techniques previously described.

Valuation Assessment: This component is responsible for projecting current economic performance over a time-span applicable to the nature of the given business within the energy market sector. This projection is formulated in conjunction with available cash

flow information in order to derive ranges of valuation based on the present value analysis of future cash flows.

3.2.1. Description of Original System A

Figure 4 shows the original control style of System A. Use Case Maps (UCMs) Buhr and Casselman [62] are used to illustrate the causal flow that is required of System A's components to provide shorter-term predictions. For example, for timeliness reasons, cash flow projections and valuation assessment are done on-line.

Figure 4. Original Control Style of System A

The dot represents the starting point of a scenario, scenario execution follows the sequence of actions expressed on the thread, and the scenario terminates at the 'T' bar. For example, the starting point of one scenario is shown associated with the Energy Sector Conditions component. It terminates at the Energy Sector Knowledge component indicating the following sequence of actions are executed: (a1) generate on-line financial conditions, (a2) provide cash flow projections, (a3) provide valuation assessment and (a4) update on-line financial conditions and knowledge information about market sector.

Each run-time component of System A implements required and/or provided services (and the service protocol of these services) as shown in Figure 5.

Figure 5. External Interactions of Components of System A

For the scenario described above, the following service protocols represent the necessary external interactions to execute the scenario: “Cash Flow Projections of Energy Conditions”, “Valuation Assessment of Cash Flow Projections”, and “Knowledge Representations of Valuation Assessment”.

Figure 6 shows the internal operations and state elements of each component of System A.

Figure 6. External Interactions and Internal Behavior of Components of System A

For the external interactions described above, the following operations and state elements implement the necessary internal behavior to support the interactions: *currentConditions()* and *financialDataInputs* of Energy Sector Conditions, *shortTermCashFlowProjections()* and *CashFlowProjectionsReferenceData* of Cash Flow Projections, *shortTermValuation()* and *PresentValueAnalysisReferenceData* of Valuation Assessment, and *updateShortTermValuations()*, *updatePriorKnowledge()*, *RawFinancialData*, and *KnowledgeInformation* of Energy Sector Knowledge.

3.2.2. Description of Original System B

Figure 7 shows the original control style of System B. UCMs are used to illustrate the causal flow that is required of System B's components to provide longer-term predictions. For example, for reasons of accuracy, cash flow projections and valuation assessment is done off-line on demand.

Figure 7. Original Control Style of System B

Each run-time component of System B implements required and/or provided services (and the service protocol of these services) as shown in Figure 8.

Figure 8. External Interactions of Components of System B

Figure 9 shows the internal operations and state elements of each component of System B.

Figure 9. External Interactions and Internal Behavior of Components of System B

3.2.3. Description of Systems Evolution

The systems need to be dynamically reconfigured so that System A and System B inter-operate to leverage each other's preferred stock predictions. To do this, each system's architectural constraints are reconciled and changes are constrained to be backward compatible. Each system is then able to provide improved analytic results for their respective clients based upon the new information that is available.

System evolution happens as follows:

- Evolution is localized to the scenario analysis components of each system
- Scenario analysis component of System A evolves in place; for System B, a new scenario analysis component is substituted
- A communication path is established between the two systems
- External interactions evolve to support new required and provided services
- Internal behavior of System A's scenario analysis component evolves in place so that it can process the new information that is exchanged

3.2.4. Description of Evolved Systems

Figure 10 shows the new control style of System A. A new UCM represents the causal flow linking the Scenario Analysis component of System A to the Scenario Analysis component of System B. This enables System A to use System B's longer-term predictions to validate its shorter-term predictions.

Figure 10. New Control Style of System A

Figure 11 shows the new control style of System B. A new UCM represents the causal flow linking the Scenario Analysis component of System B to the Scenario Analysis

component of System A. This enables System B to use System A's shorter-term predictions to validate its longer-term predictions.

Figure 11. New Control Style of System B

Figure 12 shows the new required and provided services linking Systems A and B.

Figure 12. New External Interactions

The new service protocol “Short Term Predictions Based on Long Term Predictions” enables System A to refine its short term predictions based upon long term predictions provided by System B. The new service protocol “Long Term Predictions Based on Short Term Predictions” enables System B to refine its long term predictions based upon short term predictions provided by System A.

Figure 13 shows the new internal behavior of Systems A and B.

Figure 13. New Internal Behavior

The internal behavior of System A's Scenario Analysis component dynamically evolved to support the new service protocols. The implementation of *generateShortTermPredictions()* must change to be informed by long term predictions provided by System B. The component also has a new operation, *shortTermPredictions()*, to provide System B with short term values when it computes long term values. System B's Scenario Analysis component was implemented off-line to support the new service protocols.

3.2.5. Examples Global and Local System Properties

The following are examples of global system properties:

- (GP1) The control style of each system (as depicted by use case maps);
- (GP2) The operations of different components provide needed behavior within limited time constraints (scenario analysis must not be invalidated by current financial conditions); and
- (GP3) The state elements of different components are updated in a synchronized fashion (present value analysis, cash flow projection, and scenario analysis reference data is synchronized).

The following are examples of local system properties:

- (LP1) A component has an upper bound on the number of service invocations that may be spawned in response to remote service requests;

- (LP2) A provided service has at most one required service bound to it and vice versa; and
- (LP3) The values of certain state elements may not change (Scenario Analysis reference data, once synchronized, remains immutable).

3.3. Integrity Management Areas

Integrity management governs change to ensure system consistency constraints are not violated. Within the context of dynamic reconfiguration described in Chapter One, and informed by the related research review given in Chapter Two, a set of system level characteristics are identified that, together, explicitly manage the dynamic reconfiguration of a component-based software system to help ensure overall system integrity.

The system integrity characteristics are:

- *Global Consistency Management* ensures the preservation of global architectural properties of a system, as implied by its specification;
- *Local Consistency Management* ensures the preservation of local architectural properties of a system, as implied by its specification;
- *Active Reference Management* ensures that any client that is actively bound to a changing component has a valid binding after the change is completed (the binding may have to be dynamically updated);
- *Dependent Operation Management* ensures that a change to an operation is valid for client operations that have a dependency on the operation about to change;

- *Composite Component Management* ensures that a change to a composite component is valid for a composing component that has a dependency on the changing component;
- *Constrained Operation Management* ensures that an operation with a dependency on a state element is not enacted until the state element is restored as necessary after its component's change; and
- *State Management* ensures that any needed transfers of state from a changing component to its replacement component(s) occurred through appropriate mechanisms.

The rest of this section reviews each integrity management area in more detail by (i) describing its domain, (ii) indicating interdependencies with other integrity management areas, and (iii) providing further explanation using the examples of global or local properties from the FAS case study when applicable. The next section describes how these integrity management areas are related to the different kinds of dynamic change that can occur.

3.3.1. Global Consistency Management

Global Consistency management ensures the preservation of any global properties of a system, as implied by its specification. An illustrative example of global consistency is the following. If you have a savings account and a checking account and wish to transfer money from the savings account to the checking account, the fact that the balance of the savings account must be reduced by the amount that is added to the checking account represents a global consistency condition.

Given that a specification may be formulated as a set of constraints, Global Consistency management consists of conformance to the subset of those constraints that require global knowledge for conformance. This could be accomplished through an architecture evolution manager, by synchronizing and ordering distributed event information (such as through an appropriate atomic broadcast mechanism) or by some other means [2, 3, 21, 43, 49]. By definition, global properties are reconciled with local properties. An example is the required reconciliation of GP2 and LP1 from the FAS example.

Figure 14. Domain of Global Consistency Management

Figure 14 shows the domain of Global Consistency Management. By ensuring the integrity of global properties, this management area affects external interactions and internal behavior as required through changing the structural or behavioral signature of a component. Affecting a component's external interactions is interdependent with Active Reference Management because components and communication paths may have to change. Affecting a component's internal behavior is interdependent with Dependent Operation, Composite Component, Constrained Operation and State Management because the profile of operations and state elements may have to change.

The following is an example of Global Consistency management affecting the structural signature of a component. To ensure the integrity of GP1 for System A, a specific topology must be established amongst the *Energy Sector Conditions*, *Cash Flow Projections*, *Valuation Assessment*, and *Energy Sector Knowledge* components to conform to the structural signature of a pipeline. Specifically, streaming financial data

inputs are accessed by *Energy Sector Conditions* >> *CurrentConditions()* which feeds *Cash Flow Projections* >> *ShortTermCashFlowProjections()* which feeds *Valuation Assessment* >> *ShortTermValuation()* which feeds *Energy Sector knowledge* >> *UpdateShortTermValuations()* in that order. This specific ordering affects the structural signature of each of these components.

The following is an example of Global Consistency Management affecting the behavioral signature of a component. To ensure the integrity of GP2, the implementation of the operations of this pipeline of components must ensure the *Energy Sector Knowledge* component is populated in time for short-term predictions. Based upon performance analysis, certain operations may have to be optimized. For example, *Cash Flow Projections* >> *ShortTermCashFlowProjections()* may have to in-line behavior that previously had been computed by the invocation of local or composite component operations, or cache results previously recomputed every time, to meet performance requirements. These optimizations affect the behavioral signature of *Cash Flow Projections* component.

3.3.2. Local Consistency Management

Local Consistency management ensures the preservation of any local properties of a system, as implied by its specification. An illustrative example of local consistency is the following. If you are transferring money from a savings account to a checking account, the fact that the savings account balance must not be a negative amount represents a local consistency condition.

Given that a specification may be formulated as a set of constraints, Local Consistency management consists of conformance to the subset of those constraints that require local

(i.e. component-level) knowledge for conformance [21, 49]. By definition, local properties are reconciled with global properties.

Figure 15. Domain of Local Consistency Management

Figure 15 shows the domain of Local Consistency Management. By ensuring the integrity of local properties, this management area affects external interactions and internal behavior through changing the structural or behavioral signature of a component. In a similar fashion to global consistency management, affecting a component's external interactions is interdependent with Active Reference Management because components and communication paths may have to change. Again, affecting a component's internal behavior is interdependent with Dependent Operation, Composite Component, Constrained Operation, and State Management because the profile of operations and state elements may have to change.

The following is an example of Local Consistency Management affecting the structural signature of a component. To ensure the integrity of LP2, each component of the system must conform to the structural signature of one provided service having one, and only one, required service bound to it, and vice versa. For example, in the case of the *Energy Sector Conditions* component of System A, it requires a provided service provided by the *Cash Flow Projections* component (because of GP1). To conform to LP2, the required service of *Energy Sector Conditions* is only bound to the provided service of *Cash Flow Projections* to ensure the cardinality constraint of at most one required service bound to a provided service. Likewise, the provided service of *Cash Flow Projections* servicing

Energy Sector Conditions is only bound to the required service of *Energy Sector Conditions*.

The following is an example of Local Consistency Management affecting the behavioral signature of a component. To ensure the integrity of LP1, the number of service invocations spawned by the *Energy Sector Conditions* component must be limited to conform to the required upper bound. This limits the number of run time instantiations of *Energy Sector Conditions* >> *CurrentConditions()* that in turn constrains the processing capacity of *Energy Sector Conditions* that in turn constrains the throughput of the financial system processing pipeline. Constraining the processing of *Energy Sector Conditions* affects its behavioral signature.

3.3.3. Active Reference Management

Active Reference Management ensures that any client component bindings bound to a component under evolution are managed. An illustrative example of active reference management is the following. If you are at a banking ATM in the middle of a transaction and the bank machine is being serviced at the same time, the fact that you are notified of this and told to try again later or that your transaction continues to be processed while the machine is being serviced is an example of active reference management.

When a component evolves, a communication path may have to be (re)established with a new provided service. At one extreme it may be necessary for any active references to become non-active (to establish quiescence) before component evolution (and subsequent rebinding) may take place - in the case of continuous references service is interrupted. At another extreme, active bindings and references may not be impacted because a special front-end component, meta-object protocol, or some other mechanism enabled

transparent component evolution and therefore protected them [29, 63]. Figure 16 shows the domain of this integrity management area.

Figure 16. Domain of Active Reference Management

Constrained by global and local properties, Active Reference Management affects external interactions by ensuring the integrity of communication paths and connections, and indirectly any service invocations executing provided services that implement the service protocol of these connections. Ensuring the integrity of service invocations is interdependent with Dependent Operation, Composite Component, and Constrained Operation Management.

The following is an example of a global property constraining Active Reference Management. To ensure the integrity of GP1 for System A, only a pipeline configuration of connections may connect *Energy Sector Conditions*, *Cash Flow Projections*, *Valuation Assessment* and *Energy Sector Knowledge*. The following active component references must exist to satisfy GP1. A required service of *Energy Sector Conditions* must connect to a provided service of *Cash Flow Projections*. A required service of *Cash Flow Projections* must connect to a provided service of *Valuation Assessment*. Finally, a required service of *Valuation Assessment* must connect a provided service of *Energy Sector Knowledge*. When constrained by GP1, Active Reference Management must ensure these particular connections exist.

The following is an example of a local property constraining Active Reference Management. To ensure the integrity of LP2, at most one communication path can connect a required service of any component to a provided service of any other component of the system, and vice versa. For example for System A, the required service of *Energy Sector Conditions* is bound to one communication path that links it to a provided service of *Cash Flow Projections*. Likewise, the provided service of *Cash Flow Projections* is bound to one communication path that links it to a required service of *Energy Sector Conditions*. When constrained by LP2, Active Reference management must ensure provided and required services of all components are only bound to one communication path.

3.3.4. Dependent Operation Management

Dependent Operation Management ensures that any change to an operation was valid with respect to any client operations that had a dependency on the operation about to change (for example through data flow dependency analysis techniques) Whisnant et al [31]. An illustrative example of dependent operation management is the following. If you are an employee at a company who deals directly with the human resources department, the fact that the company changes its processes from an manual paper-based system to an automated employee self-service system is an example of dependent operation management when clients are informed of the process change. Figure 17 shows the domain of this integrity management area.

Figure 17. Domain of Dependent Operation Management

Constrained by global and local properties, whether an operation invokes other operations determines how that operation is constrained for the purpose of Dependent Operation Management. This management area affects internal behavior by ensuring the integrity of operations and indirectly service invocations since the interoperation invocation patterns of a service invocation's operations may change. Ensuring the integrity of operations is interdependent with Constrained Operation and Composite Component Management. Ensuring the integrity of service invocations is interdependent with Active Reference Management.

The following is an example of a global property constraining Dependent Operation Management. To ensure the integrity of GP2 for System A, behavior provided by the local operations of *Energy Sector Conditions*, *Cash Flow Projections*, *Valuation Assessment*, and *Energy Sector Knowledge* must populate the raw financial conditions maintained by *Energy Sector Knowledge* in time for short-term predictions. For example, based on performance analysis, *Cash Flow Projections* >> *ShortTermCashFlowProjections()* may need to be optimized by in-lining behavior previously computed by the invocation of another local operation. To conform to GP2, Dependent Operation management must ensure the integrity of *Cash Flow Projections* when its operation dependencies change to satisfy GP2.

The following is an example of a local property constraining Dependent Operation Management. To ensure the integrity of LP1, if an optimized local operation of *Energy Sector Knowledge* is deployed due to a limit on the number of service invocations that may be spawned, other local operations may also become dependent upon this new operation. To conform to LP1, dependent operation management must ensure the integrity of *Energy Sector Knowledge* when its operation dependencies change to satisfy LP1.

3.3.5. Composite Component Management

Similar to Dependent Operations Management, Composite Component Management ensures that any change to a component was valid with respect to any composing components that had a direct dependency on composite component operations (again for example through data flow dependency analysis techniques) Whisnant et al [31]. An illustrative example of composite component management is the following. If you are an employee at a company who interacts with the human resources department, the fact that the company has outsourced human resources processes is an example of composite component management when clients are informed of the process change. Figure 18 shows the domain of this integrity management area.

Figure 18. Domain of Composite Component Management

Constrained by global and local properties, whether an operation invokes a composite component operation determines how that operation was constrained for the purpose of Composite Component Management. This management area affects internal behavior by ensuring the integrity of operations and indirectly service invocations since the interoperation invocation patterns of a service invocation's operations may change. Ensuring the integrity of operations is interdependent with Constrained Operation and Dependent Operation Management. Ensuring the integrity of service invocations is interdependent with Active Reference Management.

The following is an example of a global property constraining Composite Component Management. To ensure the integrity of GP2 for System A, behavior provided by the local operations of *Energy Sector Conditions*, *Cash Flow Projections*, *Valuation Assessment*, and *Energy Sector Knowledge* must populate the raw financial conditions maintained by *Energy Sector Knowledge* in time for short-term predictions. Based on performance analysis, a local operation may need to be optimized by in-lining behavior provided by a composite component operation. For example, *Cash Flow Projections* >> *ShortTermCashFlowProjections()* may have to in-line behavior provided by a composite component that computed net present value. To conform to GP2, Composite Component management must ensure the integrity of *Cash Flow Projections* when its composite component dependencies change to satisfy GP2.

The following is an example of a local property constraining Composite Component Management. To ensure the integrity of LP1, if an optimized composite component operation of *Energy Sector Knowledge* is deployed due to a limit on the number of service invocations that may be spawned, other local operations may also become

dependent upon this new composite component operation. To conform to LP1, Composite Component Management must ensure the integrity of *Energy Sector Knowledge* when its composite component dependencies change to satisfy LP1.

3.3.6. Constrained Operation Management

This management area ensures that any operation with a dependency on a component's state element (i.e. a persistent element in a particular state in its lifecycle) is not invoked until the state element's lifecycle is restored as necessary after its component's evolution Appavoo et al [29]. An illustrative example of constrained operation management is the following. A banker that relies on daily updates to mortgage rate conversion tables in order to calculate fixed or variable rate mortgages over different time periods is an example of constrained operation management. Figure 19 shows the domain of this integrity management area.

Figure 19. Domain of Constrained Operation Management

How an operation accesses state elements determines how that operation is constrained for the purpose of Constrained Operation Management. Constrained Operation Management ensures that operations access state elements in conformance to global and local properties. This management area affects internal behavior by ensuring the integrity of operations and state elements, and indirectly service invocations since state element access patterns by a service invocation's operations may change. Ensuring the integrity of

operations is interdependent with Composite Component and Dependent Operation Management. Ensuring the integrity of service invocations is interdependent with Active Reference Management. Ensuring the integrity of state elements is interdependent with State Management.

The following is an example of a global property constraining Constrained Operation Management. To ensure the integrity of GP3, Constrained Operation Management must ensure *Valuation Assessment* is notified whenever an operation updates the reference data of *Cash Flow Projections* or that *Cash Flow Projections* is notified whenever an operation updates the reference data of *Valuation Assessment*. For example, based on changing financial conditions, *Cash Flow Projections* may change the financial model it uses to compute present value payback. When *Cash Flow Projections* changes its internal reference data, the inputs provided to *Valuation Assessment* are affected, which means *Valuation Assessment* must change its financial model for computing the present value of future cash flows.

The following is an example of a local property constraining Constrained Operation Management. To ensure the integrity of LP3, Constrained Operation Management must ensure, once raw financial data about the energy sector is received by *Energy Sector Knowledge*, any local operation that accesses this information accesses it in a read only manner. For example, *Energy Sector Knowledge>>UpdatePriorKnowledge()* may only access *Raw Financial Data* in a read only manner when updating *Knowledge Information*.

3.3.7. State Management

This management area ensures that any needed synchronization of state of the component under evolution with replacement component(s) occurred through appropriate mechanisms if required Appavoo et al [29]. An illustrative example of state management is the following. If you leave one organization and join another organization, which recognizes the pension plan of the organization you left, and your pension is transferred from the organization you left to the organization you joined is an example of state management.

At one extreme, state synchronization policy may indicate that no state transfer was required for synchronization (in this case any required results are recomputed by the new component). At another extreme, the policy may mean that all state elements must be transferred from one component to the other using an appropriate mechanism (such as via a canonical format, a special purpose technique for efficiency reasons, or some other mechanism). Figure 20 shows the domain of this integrity management area.

Figure 20. Domain of State Management

Whether state capture or restoration is required determines state synchronization policy. State management ensures that state element synchronization policy conforms to global and local properties. This management area directly affects internal behavior by ensuring the integrity of state elements. Ensuring the integrity of state elements is interdependent with Constrained Operation Management because an operation's state element access patterns may change.

The following is an example of a global property constraining State Management. To ensure the integrity of GP3, State management must ensure *Present Valuation Analysis Data* is synchronized with *Cash Flow Projections Data*, and vice versa. If *Cash Flow Projections Data* is updated by *Cash Flow Projections >> ShortTermCashFlowProjections()*, *Present Valuation Analysis Data* must be updated before *Valuation Assessment >> ShortTermValuation()* accesses this information. Likewise, *Cash Flow Projections >> ShortTermCashFlowProjections()* must only access *Cash Flow Projections Data* that is synchronized with *Present Valuation Analysis Data*.

The following is an example of a local property constraining State Management. To ensure the integrity of LP3, State management must ensure *Raw Financial Data* of *Energy Sector Knowledge* is retained over time as immutable information. This constrains the state synchronization policy that can be applied to this state element. For example, *Raw Financial Data* could not be two-way synchronized with another state element in the manner *Cash Flow Projections Data* and *Present Valuation Analysis Data* are.

3.4. Change Types

As stated in Chapter One, the component-based systems paradigm is adopted as a realization platform because complex systems can be decomposed as software components, that are units of deployment, that can in principle be dynamically reconfigured as distinct entities (through intra-component dynamic reconfiguration) and/or as cooperative entities (through inter-component dynamic reconfiguration). Because dynamic reconfiguration is viewed to be an open problem, a “reduced instruction set” of dynamic change is identified with respect to a particular system model

of a component-based software system. The system model, described in Section 3.6, is specified in a manner that minimizes complexity but is aligned with recent work and standards that address component-based software systems and service-oriented architecture. The system model is formulated in a way to explicitly address component dependencies, external interactions, and internal behavior.

In this context, change types are shown effecting inter-component or intra-component change. In addition, they are shown effecting structural change when the configuration of components is changed or behavioral change when the functionality of existing components is changed. The particular change types and their relationships are explicitly identified to minimize the number of types of change and their interdependencies while ensuring critical aspects of component dependencies, external interactions, and internal behavior can be dynamically reconfigured.

In the context of the system model, the set of change types, together, represent a comprehensive capability for dynamically adjusting a system because each change type applies to unique system model elements. When change types are combined they have the capacity to affect all critical aspects of the system model. The change types are related to each other in a manner that limits the number of their combinations and enables a systematic treatment of the problem without a loss of generality.

The change types themselves are not direct representations of different levels of autonomy or adaptive system or learning techniques such as evolutionary algorithms, neural networks, fuzzy-logic, machine learning or other techniques. With this approach, the view is held that they are building blocks that support higher-order dynamic change. For example, in the context of cognitive radios, the affect of the cognition cycle is

realized by using these change types to realign the current configuration of components that compose the system.

The type of a change defines, at an abstract level, how a system evolves to adapt to new conditions. The evolution of a particular system conforms to one or more of these general types but within the context of the particular conditions that motivated the change and the concrete representations of that system.

Figure 21 shows the six types of change that could be applied to a running system and their mutual relationships. Together they provide a comprehensive dynamic reconfiguration capability. Each type is described below in terms of whether structural or behavioral change is affected, whether inter-component or intra-component change is required, and its dependencies on other change types. The types of change determine whether the behavioral or structural signature of a system must change, which in turn determines whether a system's internal behavior or external interactions need to be realigned.

The diagram also identifies the system integrity characteristics that govern each change type and which ensure global and local consistency across the system in the face of change. Depending upon the combination of change types needed, the same integrity characteristics may apply for different types of change (such as global and local consistency constraints).

Figure 21. Change Types and Dependencies

The types of system change are the following:

- *Architectural Change* changes global and/or local system properties. It is an inter-component change that may be structural, behavioral, or both. Ultimately, it may require topology and/or protocol changes;
- *Topology Change* changes the topology of a system through component addition and/or removal. This is a structural inter-component change. Ultimately, it may drive substitution or protocol changes;
- *Substitution* replaces a particular component by another within a given system topology. This is a structural intra-component change. It does not require any other types of change;

- *Protocol Change* changes the control flow and/or data flow of components. This is a behavioral inter-component change. Ultimately, it may require a topology change or an interface change;
- *Interface Change* changes the externally observable behavior of a component. It is a behavioral intra-component change. Ultimately, it may require internal change of one or more components to realize the new behavior; and
- *Internal Change* changes the internal implementation of a component. This is an intra-component type of change that can be either behavioral, structural, or both. It does not require any other types of change.

This section describes each change type in more detail by (i) providing a definition, (ii) indicating interdependencies with other change types and system integrity management areas, and (iii) providing further explanation using the examples of global or local properties from the FAS case study when applicable.

3.4.1. Architectural Change

Architectural Change means changing global or local system properties. An illustrative example of Architectural Change is the following. If a football quarterback calls an audible to change a running play to a throwing play, the fact the football team is instructed to change how it lines up for the play is an example of Architectural Change.

A change to global or local properties can imply pervasive behavioral and/or structural system evolution to conform to these changed properties. This may mean changing whole configurations of components of a greater system. Therefore, Topology Change or Protocol Change may also be required. Global and Local Consistency management

ensures the integrity of global and local properties by changing the behavioral and/or structural signature of a system as needed.

In the FAS example, an Architectural Change means the Scenario Analysis components of each system dynamically evolve in order to inter-operate. This is represented as a change to GP1 as shown by Figure 4 and Figure 10 for System A and by Figure 7 and Figure 11 for System B. To achieve this inter-operation, there is a cascading requirement for Topology and Protocol Change as described below.

Global and Local Consistency management must ensure the integrity of GP1 to GP3 and LP1 to LP3, respectively, to maintain the consistency of each system during Architectural Change.

3.4.2. Topology Change

Topology Change involves the addition and/or removal of components within a component-based system and therefore means a change to components and communication paths. An illustrative example of Topology Change is the following. The fact that a football team on defense changes how the team lines up on the field in order to blitz a quarterback is an example of Topology Change.

Topology Change is a structural kind of change because it affects the topology of a system. It is inter-component evolution because it requires other components to change as well. By changing the structural signature of a system, Global and Local Consistency management ensures system consistency during Topology Change. Active Reference management ensures the integrity of existing components and communication paths.

Protocol Change or Substitution may be required because a change of system topology may require new or updated service protocols, or, in the special case when a component must be replaced by another component, component substitution.

In the FAS example, Architectural Change (see Figures 10 and 11) drives the topology of each system to change because a new communication path must be established between the Scenario Analysis components of each system as shown in Figure 12. Topology Change establishes the new communication path. In the case of System A, this depends upon a Protocol Change of an existing component. In the case of System B, this depends upon the substitution of a new component that provides capability that satisfies the new external interactions required.

Global Consistency ensures GP1 and Local Consistency ensures LP2 for both systems because the new communication path is constrained by these properties during Topology Change. Active Reference management ensures the integrity of any active references when the new communication path is established.

3.4.3. Substitution

Substitution involves substituting one component for another within a component-based system and therefore also means a change to components and communication paths. An illustrative example of Substitution is the following. If a football quarterback calls a special play that involves a player currently off the field, the fact that the player off the field replaces a player on the field is an example of Substitution.

Substitution is a structural kind of change because it is a restricted form of component removal and then addition. It is intra-component evolution because it does not require other components having to change as well. By changing the structural signature of a

system, Global and Local Consistency management ensures system consistency during Substitution. Active Reference management ensures the integrity of the interaction of any clients of the original component. State management ensures state capture and restoration as required.

Substitution implies that the properties of the new component are a superset of the characteristics of the original component. An input provided to the original component must be substitutable for an input expected by the new component (principle of contra-variance). Any results returned by the new component must be substitutable for any results returned by the original component (principle of co-variance).

The ability to substitute one component for another is not dependent upon other types of structural or behavioral change. This assumes the internal implementation of the substitute component occurred off-line.

In the FAS example, a new Scenario Analysis component is substituted for System B's original Scenario Analysis component. The new component is implemented off-line to satisfy the external interactions of the service protocol of the new communication path and existing communication paths. The new component is shown in Figures 11, 12, and 13 supporting the new control style and new required and provided services of System B. State Management ensures the capture and restoration of *Scenario Analysis Reference Data*, *Long Term Preferred Stock Values*, and *Long Term Bond Values* state elements.

Global Consistency ensures GP2 and GP3 and Local Consistency ensures LP1, LP2 and LP3 for the Scenario Analysis component of System B because the new component is constrained by these properties during Substitution. Active Reference management

ensures the integrity of any active references when the new component is substituted for the old component.

3.4.4. Protocol Change

Protocol Change involves changing the control and/or data flow of a component's interaction with other components and therefore means a change to service protocols. An illustrative example of Protocol Change is the following. The fact that a quarterback must change how he communicates with his team because of crowd noise from using audibles on the line of scrimmage to calling plays in the huddle is an example of Protocol Change.

Protocol Change is a behavioral kind of change because it changes the behavioral interaction of components. It is inter-component evolution because it requires other components to change as well. By changing the behavioral signature of a system, Global and Local Consistency management ensures system consistency during Protocol Change. Protocol Change may require Interface Change or Topology Change because a change of control and/or data flow may require new or updated required or provided services, or new components that implement these services.

In the FAS example, new service protocols are established between the Scenario Analysis components of the two systems. These protocols are "Short Term Predictions Based On Long Term Predictions" and "Long Term Predictions Based On Short Term Predictions" as shown in Figure 12. Protocol Change establishes the new service protocol. The need for the new protocol is implied by the need for a new communication path linking the Scenario Analysis components of each system. Hence, Protocol Change is driven by Topology Change.

Global Consistency ensures GP1 for the service protocol of the new communication path because the service protocol is constrained by this property during Protocol Change.

3.4.5. Interface Change

Interface Change involves changing the interface properties of a component and therefore means changing provided and/or required services. An illustrative example of Interface Change is the following. The fact that a pitcher in baseball has learned to throw a new pitch (such as a fork ball), which the catcher can now call, is an example of Interface Change.

Interface Change is a behavioral kind of change because it changes the exposed behavior of a component. It is intra-component evolution because it does not require other components having to change as well. By changing the behavioral signature of a system, Global and Local Consistency management ensures system consistency during Interface Change. Internal Change may be required because a change to provided and/or required services may require changing the internal implementation of a component.

In the FAS example, new service protocols lead to the Interface Change of System A's Scenario Analysis component in order to support the protocols. This is represented by the new required service (the 'Socket') and the new provided service (the 'Ball') shown in Figure 12 of System A's Scenario Analysis component. The required service implements the "Short Term Predictions Based On Long Term Predictions" service protocol. The provided service implements the "Long Term Predictions Based On Short Term Predictions" service protocol. It is assumed System B's Scenario Analysis component was implemented off-line to support these protocols.

Global Consistency ensures GP1 and Local Consistency ensures LP2 for the new provided and required services of System A's evolved Scenario Analysis component because the component is constrained by these properties during Interface Change.

3.4.6. Internal Change

Internal Change involves changing the internal implementation of a component and therefore means changing operations, state elements, and service invocations. An illustrative example of Internal Change is the following. The fact that a pitcher in baseball changes how he throws when he throws a new pitch (such as a fork ball) is an example of Internal Change.

Internal Change may involve structural and/or behavioral change internally. It is intra-component evolution because it may be accomplished without other components having to change as well. Internal Change does not depend upon other types of structural or behavioral change.

Constrained Operation management ensures the integrity of operations with dependencies upon other operations. Composite Component management ensures the integrity of operations with dependencies upon composite component services. Constrained Operation management ensures the integrity of operations with dependencies upon state elements. State management ensures the integrity of state elements.

In the FAS example, the encapsulated implementation of capability provided by System A's Scenario Analysis component (see Figures 6 and 13) must change in order to support the external interaction obligations of its new provided and required services as shown in Figure 12. In this example, the operation *generateShortTermPredictions()* must change when computing short term preferred stock values in order to process information

provided by System B's Scenario Analysis component (via the invocation of *longTermPredictions()*). In addition, System A's Scenario Analysis component must dynamically evolve to provide short term predictions through System B's invocation of the new operation *shortTermPredictions()*.

Global Consistency ensures GP2 and GP3, and Local Consistency ensures LP1 and LP3 during Internal Change, of System A's Scenario Analysis component because the component's internal implementation is constrained by these properties. Dependent Operation, Composite Component, Constrained Operation, and State management ensure the consistency of System A's Scenario Analysis component when operations, state elements, or service invocations change to ensure global and local consistency constraints.

3.5. Common and Variable Causal Flow of Dynamic Reconfiguration

This section describes the common and variable causal flow of dynamic reconfiguration. The intent is to provide a description of the full spectrum of architectural, behavioral, and structural changes that could be exhibited by a dynamically reconfigurable component-based software system. The following nine *change groups*, that specify the different levels of dynamism that could be exhibited by such a system, are identified: *Pure architectural change*; *pure behavioral change*; *behavior-driven structural change*; *comprehensive behavior-driven change*; *pure structural change*; *structure-driven behavioral change*; *comprehensive structure-driven change*; *pure behavioral and structural change*; and *comprehensive change*.

Each change group is composed of a set of related *change sequences* that describe how a particular change group unfolds. A change sequence combines up to six different

change types (Figure 21 shows the possible change types and their mutual relationships).

The change types of a change sequence determine whether the particular change sequence implies architectural, behavioral, and/or structural change.

A use-case description is used to specify both the common and variable causal flows of each grouping Wirfs-Brock and Schwartz [64] and Cockburn [65]. These descriptions will be expanded further into a series of behavioral and structural UML models that represent the different levels of dynamism in Chapter 4. The rest of this section presents the common and variable causal flow of dynamic reconfiguration by describing the general context and the nine categories of change.

3.5.1. General Context of Dynamic Reconfiguration

Dynamic reconfiguration progresses according to the activity model shown in Figure 22. The decision points immediately after ‘*Reconcile Change Properties with Existing System Properties*’ indicates this reconciliation may lead to change rejection. The decision point immediately after ‘*Estimate Impact of Architectural Change upon System Integrity*’ indicates this impact analysis may lead to change rejection. The parallel fork indicates ‘*Enact Change by Realigning System Constructs*’ and ‘*Ensure System Integrity Characteristics*’ happens concurrently. The parallel join indicates a system has enacted dynamic change while attempting to ensure overall system consistency.

Figure 22. Activity Model of General Context of Dynamic Reconfiguration

Given the interactions implied by Figure 22, Table 1 is a use case of the general context of dynamic reconfiguration. This use case describes the different levels of dynamism as alternatives based upon the change groups.

Use Case Description: General Context of Dynamic Reconfiguration	
Description:	This use case describes the overall casual flow of dynamically reconfiguring a component-based system.
Actors:	System User and/or Processing Environment
Pre-Condition:	Component-based system provides adequate capability to respond to run time change stimuli, characterize the nature of change required as global or local properties (including their reconciliation), enact generic types of change that may be required, and govern change in a manner to help ensure overall system consistency.
Triggering Events:	<i>User-Driven Change</i> and/or <i>Computing Environment Imposed Change</i>
Sequence:	<p>Step 1. Sense <i>User-Driven</i> and/or <i>Computing Environment Imposed Change</i>.</p> <p>Step 2. Interpret and then represent the particular <i>Origin of Change</i> as <i>Global Properties</i> or <i>Local Properties</i> with an associated <i>Reconciliation Policy</i> (also reconcile with existing <i>System Properties</i>).</p> <p>Step 3. Determine what feasible change group (if any) satisfy the <i>Condition Change Criteria</i> and enact change (See Alternatives).</p> <p>Step 4. Realign <i>Dependencies</i> (if necessary) and therefore possibly <i>External Interactions</i> and <i>Internal Behavior</i>.</p> <p>Step 5. Validate preservation of applicable <i>System Integrity Characteristics</i> during change sequence.</p>
Post-condition:	Component-based system is dynamically reconfigured based upon a particular change groups' change sequence in a manner that helps to ensure overall system integrity.
Resulting Events:	Further <i>User-Driven Change</i> and/or <i>Computing Environment Imposed Change</i>
Alternatives for Step 3:	<p>Alternative 1 – <i>Pure architectural change</i>;</p> <p>Alternative 2 – <i>Pure behavioral change</i>;</p> <p>Alternative 3 – <i>Behavior-driven structural change</i>;</p> <p>Alternative 4 – <i>Comprehensive behavior-driven change</i>;</p> <p>Alternative 5 – <i>Pure structural change</i>;</p> <p>Alternative 6 – <i>Structure-driven behavioral change</i>;</p> <p>Alternative 7 – <i>Comprehensive structure-driven change</i>;</p> <p>Alternative 8 – <i>Pure behavioral and structural change</i>; or</p> <p>Alternative 9 – <i>Comprehensive change</i>.</p>

Table 1. Common and Variable Causal Flow of Dynamic Reconfiguration

In what follows, the term “signature” is used to denote the observable behavior and structure of a system, as discussed in Chapter One. Different types of change are categorized based on the effect that they have on a system’s signatures and the relationships between them.

The following reviews each alternative by providing a description of each change group. Within each description, a determination is made about (a) whether the behavioral and/or the structural signature of a system must change and (b) how these system signatures are related.

3.5.2. Pure Architectural Change

Pure architectural change is a change group that consists of a single change sequence involving a single change type: *architectural change*. In its purest form, this is a change to only global and/or local system *properties*. By implication, the existing behavioral and structural signatures of the system can accommodate the change and are unaffected.

Figure 23. Pure Architectural Change

In the FAS case study, the following dynamic reconfiguration sequence is an example of pure architectural change. If LP1 changes for System B to a lower threshold as a result of a change to GP1 (which dynamically links System A and System B), the change to LP1 could be accommodated without changing System B's behavioral or structural signatures (which did change as a result of a change to GP1).

3.5.3. Pure Behavioral Change

Pure behavioral change is a change group that encompasses three change sequences, each driven by *architectural change*: (i) *protocol change* only, (ii) *protocol change* leading to *interface change*, or (iii) *protocol change* leading to *interface change* leading to *internal change*. By implication, *pure behavioral change* means that only the behavioral signature of a system is realigned.

Figure 24. Pure Behavioral Change

In the FAS case study, the following dynamic reconfiguration sequence of System A is an example of pure behavioral change. A change to *GPI* (architectural change) leading to the deployment of the new service protocol “*Short Term Predictions Based on Long Term Predictions*” shown in Figure 12 (protocol change) leading to the new required service that implements the new service protocol (interface change) leading to the operation *generateShortTermPredictions()* changing when computing short term values in order to process information provided by System B (internal change).

3.5.4. Behavior-Driven Structural Change

Behavior-driven structural change is a change group that comprises two change sequences, each driven by *architectural change*: (i) *protocol change* leading to *topology*

change or (ii) *protocol change* leading to *topology change* leading to *substitution*. By implication, *behavior-driven structural change* means that, when the behavioral signature of a system is realigned, it causes the structural signature of the system to be realigned.

Figure 25. Behavior-Driven Structural Change

In the FAS case study, the following dynamic reconfiguration sequence is an example of behavior-driven change. A change to *GPI* (architectural change) leading to the deployment of the new service protocol “*Short Term Predictions Based on Long Term Predictions*” shown in Figure 12 (protocol change) leading to a new communication path linking System A and System B (topology change) leading to a new Scenario Analysis component being substituted for System B’s original Scenario Analysis component (substitution).

3.5.5. Comprehensive Behavior-Driven Change

Comprehensive behavior-driven change is a change group that is a combination of *pure behavioral change* and *behavior-driven structural change*. By implication, *comprehensive behavior-driven change* means that, when the behavioral signature of a

system is realigned, it causes the structural signature of the system to be realigned. All combinations are possible as long as a behavioral change leads to a structural change.

Figure 26. Comprehensive Behavior-Driven Change

In the FAS case study, the following dynamic reconfiguration sequence is an example of comprehensive behavior-driven change. A change to *GPI* (architectural change) leading to:

- the deployment of the new service protocol “*Short Term Predictions Based on Long Term Predictions*” shown in Figure 12 (protocol change) leading to:
 - a new communication path linking System A and System B (topology change) leading to a new Scenario Analysis component being substituted for System B’s original Scenario Analysis component (substitution); and
 - the new required service that implements the new service protocol (interface change) leading to the operation *generateShortTermPredictions()* changing when computing short term values in order to process information provided by System B (internal change).

3.5.6. Pure Structural Change

Pure structural change is a change group that encompasses two change sequences, that are each driven by *architectural change*: (i) *topology change* only or (ii) *topology change* leading to *substitution*. By implication, *pure structural change* means the structural signature, and only the structural signature, of a system is realigned.

Figure 27. Pure Structural Change

In the FAS case study, the following dynamic reconfiguration sequence is an example of pure structural change. A change to *GPI* (architectural change) leading to a new communication path linking System A and System B (topology change) leading to a new Scenario Analysis component being substituted for System B's original Scenario Analysis component (substitution).

3.5.7. Structure-Driven Behavioral Change

Structure-driven behavioral change is a change group that comprises three different change sequences, each driven by *architectural change*: (i) *topology change* leading to *protocol change*, (ii) *topology change* leading to *protocol change* leading to *interface change*, or (iii) *topology change* leading to *protocol change* leading to *interface change*

leading to *internal change*. By implication, *structure-driven behavioral change* means that, when the structural signature of a system is realigned, it causes the behavioral signature of the system to be realigned.

Figure 28. Structure-Driven Behavioral Change

In the FAS case study, the following dynamic reconfiguration sequence is an example of structure-driven behavioral change. A change to *GPI* (architectural change) leading to a new communication path linking System A and System B (topology change) leading to the deployment of the new service protocol “*Short Term Predictions Based on Long Term Predictions*” shown in Figure 12 (protocol change) leading to the new required service that implements the new service protocol (interface change) leading to the operation *generateShortTermPredictions()* changing when computing short term values in order to process information provided by System B (internal change).

3.5.8. Comprehensive Structure-Driven Change

Comprehensive structure-driven change is a change group that is a combination of *pure structural change* and *structure-driven behavioral change*. By implication, *comprehensive structure-driven change* means that, when the structural signature of a

system is realigned, it causes the behavioral signature of the system to be realigned. All combinations are possible as long as a structural change leads to a behavioral change.

Figure 29. Comprehensive Structure-Driven Change

In the FAS case study, the following dynamic reconfiguration sequence is an example of comprehensive structure-driven change. A change to *GPI* (architectural change) leading to:

- a new communication path linking System A and System B (topology change) leading to:
 - a new Scenario Analysis component being substituted for System B's original Scenario Analysis component (substitution); and
 - the deployment of the new service protocol "*Short Term Predictions Based on Long Term Predictions*" shown in Figure 12 (protocol change) leading to the new required service that implements the new service protocol (interface change) leading to the operation *generateShortTermPredictions()* changing when computing short term values in order to process information provided by System B (internal change).

3.5.9. Pure Behavioral and Structural Change

Pure behavioral and structural change is a change group that is a combination of *pure behavioral change* and *pure structural change*. By implication, *pure behavioral and structural change* means that the behavioral signature and the structural signature of a system are realigned but one does not drive the other to change. All combinations are possible as long as behavioral change does not lead to structural change nor does structural change lead to behavioral change.

Figure 30. Pure Behavioral and Structural Change

In the FAS case study, the following dynamic reconfiguration sequence is an example of pure behavioral and structural change. A change to *GPI* (architectural change) leading to:

- a new communication path linking System A and System B (topology change) leading to a new Scenario Analysis component being substituted for System B's original Scenario Analysis component (substitution); and
- the deployment of the new service protocol "*Short Term Predictions Based on Long Term Predictions*" shown in Figure 12 (protocol change) leading to the new required service that implements the new service protocol (interface change) leading to the

operation *generateShortTermPredictions()* changing when computing short term values in order to process information provided by System B (internal change).

3.5.10. Comprehensive Change

Comprehensive change is a change group that is a combination of *comprehensive behavior-driven change* and *comprehensive structure-driven change*. By implication, *comprehensive change* means that the behavioral signature and the structural signature of a system are realigned as one drives the other to change. All combinations are possible as long as behavioral change leads to structural change and structural change leads to behavioral change.

Figure 31. Comprehensive Change

In the FAS case study, the following dynamic reconfiguration sequence is an example of comprehensive change. A change to *GPI* (architectural change) leading to:

- a new communication path linking System A and System B (topology change) leading to:
- a new Scenario Analysis component being substituted for System B's original Scenario Analysis component (substitution); and

- the deployment of the new service protocol “*Short Term Predictions Based on Long Term Predictions*” shown in Figure 12 (protocol change) leading to the new required service that implements the new service protocol (interface change) leading to the operation *generateShortTermPredictions()* changing when computing short term values in order to process information provided by System B (internal change); and
- the deployment of the new service protocol “*Short Term Predictions Based on Long Term Predictions*” shown in Figure 12 (protocol change) leading to:
 - the new required service that implements the new service protocol (interface change) leading to the operation *generateShortTermPredictions()* changing when computing short term values in order to process information provided by System B (internal change); and
 - a new communication path linking System A and System B (topology change) leading to a new Scenario Analysis component being substituted for System B’s original Scenario Analysis component (substitution).

3.6. Feature Model

This section uses feature-modeling techniques to define the concepts that underpin *Dynamic Reconfiguration*. Feature modeling is adopted because it is an appropriate notation for concept modeling at the domain analysis level [38]. The complete model can be expressed in a visual and concise way. The notation is elegant and because of this is ‘readable’ when specifying relatively complex semantics (i.e. mandatory-and features, and so on). Features are used instead of UML classes because features are ‘higher’ order concepts that provide an easily understood context for subsequent Alloy and UML

models. UML classes are not strictly semantically equivalent even when augmented with OCL (for example, a feature does not necessarily have identity).

As stated in Chapter One, the dynamic reconfiguration of software systems can be viewed as an aspect of software evolution where software evolution is the process of changing software due to changes in software specification or software requirements. Figure 32 shows the concept *Dynamic Reconfiguration*, the focus area of this thesis, within the greater context of a high-level feature model of *Software Evolution*. Depending on the details, *Software Evolution* may be viewed as more static or more dynamic in nature. In any of these cases (those more static or those more dynamic), the stimulus for change may be manual or may be automatic.

Figure 32. High Level Feature Model of Software Evolution

The two main classes of *Static Software Evolution* as shown in Figure 32 include software (system) configuration and software (system) deployment. The OMG Deployment and Configuration of Component-based Distributed Applications RFP addresses installation configuration when deploying component-based systems in order to establish the initial configuration of components after their creation. From OMG [66],

...the terms deployment and configuration address all activities to be performed in order to deploy the software components of an application as the result of the construction phase onto entities of the target infrastructure and incarnate them.

Incarnation includes the installation and configuration of the components, their loading and finally the instantiation, configuration and connection of the artifacts...

Two of the three main classes of *Dynamic Software Evolution*, as shown in Figure 32, include the upgrade of continually running (24X7) systems and software systems that are dynamically self-reconfigured to reflect changes within their context of execution.

The OMG Online Upgrades RFP addresses the need to upgrade or evolve individual components or objects while a system continues to operate; this is meant to address the issue of the upgrade of continually running systems. From OMG [67],

...To make use of the rapidly changing technology and to maintain technical freshness, it must be possible to upgrade and evolve (these) systems by replacing individual software and hardware components. Many systems cannot be taken out of service to perform an upgrade... Upgrading the interfaces of an object is a more difficult problem, because it involves other objects that invoke the methods of the object that is being upgraded. This RFP does not solicit proposals for upgrading the interfaces of an object...

In OMG [68], an on-line upgrade is achieved via an UpgradeManager who primarily manages the transfer of state from an instance of the old implementation to an instance of the new implementation. Supported on-line upgrade scenarios include pushed upgrade, pulled upgrade, and upgrades in the presence of smart-clients and system management. In terms of dynamic software evolution, the main objective of OMG [68] is to transparently provide the ability to:

...transfer state from an instance of the old implementation of the object to an instance of the new implementation of the object, with the provision for such state transfers where the representation of the old state and the new state are different...

With respect to context-aware change mechanisms, on-going research work on reflective systems, ubiquitous computing, (self) adaptive systems, and context modeling combine at some level to address dynamic software evolution based on context aware condition change criteria. At this level, models of computation are employed to represent the nature of the processing environments of running systems as well as the intrinsic nature of systems running within these environments.

In this thesis, dynamic reconfiguration is viewed to lie in between on-line upgrades and facilities for context-aware changes on the spectrum of dynamic software evolution - it is within this area that the dynamic reconfiguration of interacting components is addressed. The position is adopted that a dynamically re-configurable component-based software system must provide adequate capability (the level of which will depend on the particular environment in which the system is running) to:

- Respond to run time change stimuli;
- Characterize the nature of change required as global or local properties (including their reconciliation);
- Enact the generic types of change that are required; and
- Govern change in a manner to ensure overall system consistency.

To specify this as a feature model, the primary concepts of *Interacting Components* are first defined. Then, the primary concepts of *Context of Change* are defined. Finally, the primary concepts of *Dynamic Reconfiguration* are defined by composing the primary concepts of *Context of Change* with the feature model of *Interacting Components*, and by explaining how these specifications interact as a combined model.

This is shown diagrammatically in Figure 33 as *Context of Change* being composed with *Interacting Components* to specify *Dynamic Reconfiguration*:

Figure 33. Feature Models Composed as Dynamic Reconfiguration

As noted in Chapter One, a component framework exists as a mechanism for the instantiation of components which knows about its components but components may also have explicit dependencies on the framework. For example, a component may rely upon a Service Access Point (SAP) provided by its framework as part of its implementation. This dependency is represented as a global or local property of the component.

The rest of this section first defines the primary concepts of *Interacting Components*. Then it defines the primary concepts of *Context of Change*. Finally, these feature models are composed to define the primary concepts of *Dynamic Reconfiguration*.

3.6.1. Primary Concepts of Interacting Components

Figure 34 shows the decomposition of the concept *Interacting Components*. It is decomposed into:

- *External Interactions* that represent the cooperating components of a system;
- *Internal Behavior* that represents the encapsulated implementation of capabilities required to satisfy a component's external interaction obligations; and
- *Dependencies* that represent the configuration of a component with respect to other components.

This separation of features is intended to ease the understanding of a complex system from the viewpoint of dynamic reconfiguration. It allows, at the modeling level, the internal implementation of components to be viewed independently of component interactions, and it enables dependencies that drive internal implementations to be considered separately from those that drive external interactions. Each of these concepts is described in more detail below.

Figure 34. Feature Model of Interacting Components

There are, of course, dependencies between the various sub-features. This decomposition enables better focus on the areas most directly related to dynamic reconfiguration. As one example of a sub-feature dependency not shown, a component may be recursively composed of other components; dependencies between the various sub-features are represented in the Alloy specification given in the next section. Using the conceptual framework identified in Czarnecki and Eisennecker [38], Table 2 provides further comments on configuration or composition information about these features.

Configuration Information	Comment
<i>External Interactions</i> , <i>Internal Behavior</i> , and <i>Dependencies</i> are mandatory sub-features of <i>Interacting Components</i> .	Each of these features must be present. They are the basic constructs that allow <i>Interacting Components</i> to be modeled in a manageable fashion.
Default settings specifications are basic configurations of <i>External Interactions</i> , <i>Internal Behavior</i> , or <i>Dependencies</i> . They can be overridden during the lifetime of a system.	For example, default settings may indicate that a component is a service provider only.
Default dependency specifications are basic interdependencies among <i>External Interactions</i> , <i>Internal Behavior</i> , or <i>Dependencies</i> .	For example, dependencies drive the configuration of external interactions and internal behavior. Internal behavior satisfies system external interaction obligations.
Illegal feature combinations are aspects of <i>External Interactions</i> , <i>Internal Behavior</i> , or <i>Dependencies</i> that cannot co-exist or may only co-exist under special conditions that mitigated the incompatibility.	For example, if system dependency information indicated that a component was strictly a service provider then required services could not exist.
Construction rules determine how <i>External Interactions</i> , <i>Internal Behavior</i> , or <i>Dependencies</i> are instantiated and interrelated.	For example, if components are members of a pipeline of components, system construction rules ensure that the member components only had one provided and one required service.
Optimization strategies embody space and time efficiency techniques applicable for <i>External Interactions</i> , <i>Internal Behavior</i> , or <i>Dependencies</i> during system realization.	For example, internal behavior may be realized using caching techniques to avoid needlessly re-computing results.

Table 2. Examples of Configuration Information of Interacting Components

In the FAS case study, Figure 5 shows an example of external interactions of interacting components. Figure 6 shows an example of the internal behavior of interacting components. These examples also indicate dependencies of interacting components. More detailed examples of each of these sub-features is given below.

The core concepts that compose *External Interactions*, *Internal Behavior*, and *Dependencies* are now further described.

3.6.1.1. Primary Concepts of External Interactions

Figure 35 is a feature model diagram of the concept *External Interactions*. *External Interactions* represent the nature of cooperation that exists among the independent

software entities that compose a system. The concept is further decomposed into components and communication paths that connect required services to provided services and which conform to service interaction protocols. This decomposition is consistent with the concepts found in architecture description languages (ADLs) Medvidovic and Taylor [18]. Table 3 provides further comments on configuration or composition information about these features.

Figure 35. Feature Model of External Interactions

Component represents a software entity as a unit of system composition that has, when possible, contractually specified interfaces and internal context dependencies only, which can be deployed independently within a component framework. Structurally, a component is de-coupled and fully encapsulated from other system entities based on its provided and/or required services. As a unit of deployment, it may be recursively composed of other composite components.

Required Service implements a service protocol, is a connection point from which a component binds to a provided service of another component that provides the kind of service required, and isolates the internal implementation of its component.

Provided Service implements a service protocol, is a connection point for other components to bind to this component that may require this kind of service, and isolates the internal implementation of its component.

Communication Path represents a communication link that is established between a required service and a provided service of one component with another component. A *Communication Path* specifies the protocol of component interactions as a service protocol.

Service Protocol represents the behavior that is expected of two interacting compatible components. It encompasses a set of roles, that describe the behavior that is expected of the interacting components that are connected, and a specification of how the activities of the role instances are coordinated.

Connection implements a communication path definition. It represents an active interaction whose semantics are governed by the underlying communication path binding that links the two components that have been connected. There may be many connections based on a communication path specification.

In the FAS case study, Figure 5 shows an example of these sub-features. Energy Sector Conditions is an example of a component that implements a Required Service. Cash Flow Projections is an example of a component that implements a Provided Service. “Cash Flow Projections of Energy Conditions” is an example of a Service Protocol. The fact that these two components are interacting implies there is a Communication Path linking them and that a Connection exists that binds the Required Service to the Provided Service.

Configuration Information	Comment
<p><i>Component</i> and <i>Communication Path</i> are mandatory sub-features of <i>External Interactions</i>.</p> <p><i>Required Service</i> and <i>Provided Service</i> are mandatory or-features of <i>Component</i>.</p> <p><i>Service Protocol</i> is a mandatory sub-feature of <i>Communication Path</i>. <i>Connection</i> is an optional feature of <i>Communication Path</i>.</p>	<p>Each of these features must be present. Components interconnected through communication paths are the basic constructs that enable complex external interactions to be modeled in a manageable fashion.</p> <p>One or both of these features must be present. To be a useful member within a component-based system, a component must provide a needed service to other components and/or require service of other components.</p> <p><i>Service Protocol</i> must be present. <i>Connection</i> is shown as an optional feature because it is possible for components to be interconnected yet to have no active connections. To be useful, a communication path must define a protocol for external interactions and have run-time instantiations that make use of this protocol definition.</p>
Default settings specifications are basic configurations of <i>Component</i> or <i>Communication Path</i> . They can be overridden during the lifetime of a system.	For example, default settings may indicate that provided or required services have a limit on the number of bound connections or that a communication path may have a limit on the number of supported connections.
Default dependency specifications are basic interdependencies among <i>Component</i> and <i>Communication Path</i> .	For example, required and provided services satisfy service protocol definitions to enable required system external interaction patterns, or the number of connections are constrained by the number of bindings allowed for a given required or provided service.
Illegal feature combinations are aspects of <i>Component</i> and <i>Communication Path</i> that cannot co-exist or may only co-exist under special conditions that mitigated the incompatibility.	For example, if required or provided service configuration information restricted the cardinality of bound connections then it is illegal for any connections to exist that exceeded these limits.
Construction rules determine how <i>Component</i> and <i>Communication Path</i> are instantiated and interrelated.	For example, each required service of a component must be interconnected to other components for the component to be well formed at run-time.
Optimization strategies embody space and time efficiency techniques applicable for <i>Component</i> and <i>Communication Path</i> during system realization.	For example, the system topology may be configured in a manner to restrict the number of external interactions, or in the case of a communication path, the number of round trips for efficiency reasons.

Table 3. Examples of Configuration Information of External Interactions

3.6.1.2. Primary Concepts of Internal Behavior

Figure 36 is a feature model diagram of the concept *Internal Behavior*. *Internal Behavior* represents the encapsulated implementation of required capability to satisfy an entity's external interaction obligations. The concept is further decomposed to enable

modeling of internal computation (as operations, state elements, and service invocations), systematic management of system integrity (by managing dependencies between operations, state elements, and service invocations) and mapping of internal behavior to external interactions (by linking certain operations to required or provided services). Table 4 provides further comments on configuration or composition information about these features.

Figure 36. Feature Model of Internal Behavior

Operation represents the implementation of a method of a component. A component has a set of operations that implement the internal behavior of the component. A subset of these operations represent public methods that implement provided service protocol behavior specifications.

An operation is defined based on a set of input parameters, actions, local (temporary) and global (persistent) state elements and results. An input parameter is an encapsulated entity (an object) which has been passed to the associated operation; the behavior of an input parameter is governed by the set of types implemented by the parameter. A result is an encapsulated entity (an object) which is a possible outcome of invoking an associated operation; the behavior of a result is governed by the set of types implemented by the result.

The body of an operation is composed of a sequence of actions that transform input parameters as results. During this transformation, permanent or temporary state elements are read from, or written to, depending on the control flow that defines the exact sequence of actions based on the run-time values of input parameters and state elements. In addition, an action may invoke another operation depending on the computation context. The control flow of the operation determines what kind of result is generated.

State Element represents local (temporary) or permanent (persisted) computation results generated and maintained by a component or temporarily by a service invocation. Whether permanent or temporary, state elements are read from, or written to, depending on the action sequences of the public or private operations of a component.

Service Invocation represents the run-time behavior of a component (for example, by representing the run time sequence of its operations as a computation stack). A service invocation is created to service a new external interaction and may spawn new service invocations depending on the context. Any concurrent service invocations of a component share the component's permanent state elements; they do not share temporary state elements as defined by an operation.

In the FAS case study, Figure 6 is an example of these sub-features. An example of a State Element is `financialDataInputs` of Energy Sector Conditions. An example of an Operation is `currentConditions()` of Energy Sector Conditions. Given there is an active Connection linking Energy Sector Conditions and Cash Flow Projections, Service Invocation(s) represent the run-time behavior of each Component.

Configuration Information	Comment
<i>Operation</i> and <i>Service Invocation</i> are mandatory sub-features of <i>Internal Behavior</i> . <i>State Element</i> is an optional feature.	Operation and Service Invocation must be present. Operation definitions, and their run time instantiation as service invocations, are the basic constructs that enable the internal behavior of a component to meet its external interaction obligations within a component-based system. A state element is required when operational results must be persisted temporarily or permanently.
Default settings specifications are basic configurations of <i>Operation</i> , <i>State Element</i> , or <i>Service Invocation</i> . They can be overridden during the lifetime of a system.	For example, default settings may indicate that certain operations remain private, that certain state elements be immutable or that there is a limit on the number of secondary service invocations that may be spawned by a primary service invocation.
Default dependency specifications are basic interdependencies among <i>Operation</i> , <i>State Element</i> and <i>Service Invocation</i> .	For example, if a state element is immutable then no operation can update it or if an operation cannot be invoked concurrently (because it represented a critical region for example) then the operation only appears on one computation stack at a time.
Illegal feature combinations are aspects of <i>Operation</i> , <i>State Element</i> and <i>Service Invocation</i> that cannot co-exist or may only co-exist under special conditions that mitigated the incompatibility.	For example, an operation that updated state information could not be combined with a state element that was immutable or an operation that required sub-threading at run-time could not be combined with a service invocation's computation stack that precluded sub-threading.
Construction rules determine how <i>Operation</i> , <i>State Element</i> and <i>Service Invocation</i> are instantiated and interrelated.	For example, the construction of these features happen in a manner that satisfies the concurrency model of the component (i.e. with respect to concurrent updates and the number of service invocations required).
Optimization strategies embody space and time efficiency techniques applicable for <i>Operation</i> , <i>State Element</i> and <i>Service Invocation</i> during system realization.	For example, operations may cache results to avoid re-computing them or multi-threading models may be adopted to increase throughput through maximizing concurrent operations.

Table 4. Examples of Configuration Information of Internal Behavior

3.6.1.3. Primary Concepts of Dependencies

Figure 37 is a feature model diagram of the concept *Dependencies*. *Dependencies* represent the configuration of a component with respect to other components as global and local constraints and behavioral and structural signatures. The concept is therefore further decomposed to allow the specification of global or local system properties as constraints Marriott and Stuckey [49], the reconciliation of these constraints, and the

specification of system profiles that constrain external interactions and internal behavior. Table 5 provides further comments on configuration or composition information about these features.

Figure 37. Feature Model of Dependencies

As stated in Chapter One, a fundamental assumption is that a technical system specification may be expressed as a set of constraints. Given this assumption, *System Properties* are a reconciled set of constraints. The sub-features of *System Properties* are *Global Properties*, *Local Properties*, and *Reconciliation Policy* that represent global and local properties and their reconciliation respectively.

Global Properties exist as implied by the specification of an overall system formulated as a set of architectural constraints. By definition, global system properties require global (system-level) knowledge for constraints conformance.

Local Properties exist as implied by the specification of a component formulated as a set of architectural constraints. By definition, local system properties only require local (component-level) knowledge for constraints conformance.

Reconciliation Policy exists to enable local system properties to be reconciled with respect to global system properties as necessary. Due to reconciliation, not all system

properties may be satisfied concurrently. In addition, the system signature, and therefore the configuration of external interactions and internal behavior constructs, may be effected.

System Signature represents the observable behavior and structure of a system, as discussed in the introduction. It is composed of *Structural Signature* and *Behavioral Signature*. The signature of a system is ultimately driven by its system properties.

Structural Signature represents specific kinds of dependencies that determine a component's external interactions within a component-based system. See Figure 2.

Behavioral Signature represents specific kinds of dependencies that determine a component's internal behavior within a component-based system. See Figure 2.

In the FAS case study, GP1-GP3 and LP1-LP3 are examples of Global Properties and Local Properties respectively. Examples of System Properties that require a Reconciliation Policy are GP2 and LP1. Driven by GP1, an example of a Structural Signature is the pipeline configuration of Energy Sector Conditions, Cash Flow Projections, Valuation Assessment, and Energy Sector Knowledge components of System A. Since Valuation Assessment is a member of this pipeline, an example of a Behavioral Signature is the implied invocation of `shortTermValuation()` and its access of `PresentValueAnalysisReferenceData`.

Configuration Information	Comment
<p><i>System Signature</i> and <i>System Properties</i> are mandatory sub-features of <i>Dependencies</i>.</p> <p><i>Structural Signature</i> and <i>Behavioral Signature</i> are mandatory sub-features of <i>System Signature</i>.</p> <p><i>Global Properties</i> and <i>Local Properties</i> are mandatory or-features of <i>System Properties</i>.</p>	<p>Each of these features must be present. The system properties that must be conformed to, and the behavioral and structural profile of a component that reflects this conformance, are the basic constructs that indicate how a component is interdependent with other components within a component-based system.</p> <p>Each of these features must be present. A component has both a behavioral and a structural profile when conforming to its system properties. One or both of these features must be present. In addition, <i>Reconciliation Policy</i> is shown as a mandatory feature. This feature must be present. A component has global and/or local properties it must conform to; these properties are reconciled when they are in conflict.</p>
Default settings specifications are basic configurations of <i>System Signature</i> or <i>System Properties</i> . They can be overridden during the lifetime of a system.	For example, default settings of system signature and system properties can reflect a layered system architecture, or some other architectural style.
Default dependency specifications are basic interdependencies among <i>System Signature</i> and <i>System Properties</i> .	For example, reconciled global and local properties of system properties drive the behavioral and structural profile of system signature as represented by structural signature and behavioral signature.
Illegal feature combinations are aspects of <i>System Signature</i> and <i>System Properties</i> that cannot co-exist or may only co-exist under special conditions that mitigated the incompatibility.	For example, a global property that demanded minimum system response times might not be compatible with a local property that demanded common behavior to be re-factored as distinct operations.
Construction rules determine how <i>System Signature</i> and <i>System Properties</i> are instantiated and interrelated.	For example, a system signature is constructed that reflected a component being a member of a pipeline of components if system properties indicated this was required.
Optimization strategies embody space and time efficiency techniques applicable for <i>System Signature</i> and <i>System Properties</i> during system realization.	For example, as an optimization, system properties may drive in-lining any behavior that normally is provided by composite component operations. This also drives the system signature of a component.

Table 5. Examples of Configuration Information of Dependencies

Given the descriptions of the primary concepts just given, the following sequence determines the constraints hierarchy of *Interacting Components*:

- System specification constrains *System Properties*;

- *System Properties* constrain the *System Signature*;
- *Structural Signature* constrains *External Interactions*; and
- *Behavioral Signature* constrains *Internal Behavior*.

Figure 38 is a complete feature model diagram of the concept *Interacting Components*:

Figure 38. Complete Feature Model of Interacting Components

3.6.2. Primary Concepts of Context of Change

Figure 39 is a feature model diagram of the concept *Context of Change*. *Context of Change* represents software evolution as a feature that will enable the system model to represent dynamic reconfiguration when this feature becomes one of the system model's mandatory sub-features. The concept is further decomposed of Condition Change Criteria, Type of Change, and System Integrity Characteristics sub-features to separate concerns to better manage complexity when specifying the context of system change. This separation of features allows different aspects of the context of change to vary

independently and causal flow to be modeled in a fashion more easily understood. Table 6 provides further comments on configuration or composition information about these features.

Figure 39. Feature Model of Context of Change

Condition Change Criteria represents under what conditions a system adapts or evolves. The conditions for adaptation or evolution occur because change was user-driven or computing environment-imposed. In either case, there are new or augmented functional or non-functional requirements of the system currently deployed. Importantly, the conditions for functional change may or may not invalidate the original non-functional conditions and vice-versa. Also, the conditions for change may be expected, and therefore planned for within the design of the system, or unexpected, and therefore involve more fundamental, and possibly more pervasive, behavioral or structural system change. Finally, the conditions for evolution may imply that the time required to enact the required change was relatively bounded or unbounded compared to other conditions for evolution.

Type of Change represents the different kinds of software reconfiguration as described in Section 3.4. Software realignment activities are performed in a way that maintains the global consistency of the greater system.

System Integrity Characteristics represents the set of system characteristics that require explicit management to effectively and efficiently enact system evolution while maintaining overall system integrity as described in Section 3.3.

Configuration Information	Comment
<i>Condition Change Criteria, Type of Change, and System Integrity Characteristics</i> are mandatory sub-features of <i>Context of Change</i> .	Each of these features must be present. These sub-features are the basic constructs that enable complex system change to be modeled in a manageable fashion.
Default settings specifications are basic configurations of <i>Condition Change Criteria, Type of Change, or System Integrity Characteristics</i> . They can be overridden during the lifetime of a system.	For example, default settings may time box how long it may take to enact change to make the problem tractable, limit the number of types of change that may happen concurrently, or give precedence to one aspect of system integrity characteristics.
Default dependency specifications are basic interdependencies among <i>Condition Change Criteria, Type of Change and System Integrity Characteristics</i> .	For example, condition change criteria determine what type of change was required which in turn determine what aspects of system integrity characteristics (if any) are required to help ensure overall system consistency during system evolution.
Illegal feature combinations are aspects of <i>Condition Change Criteria, Type of Change and System Integrity Characteristics</i> that cannot co-exist or may only co-exist under special conditions that mitigated the incompatibility.	For example, if condition change criteria indicated that only protocol change is required then it is illegal for other types of change to exist.
Construction rules determine how <i>Condition Change Criteria, Type of Change and System Integrity Characteristics</i> are instantiated and interrelated.	For example, depending on condition change criteria only certain types of change are constructed which in turn determines what aspects of system integrity characteristics are constructed.
Optimization strategies embody space and time efficiency techniques applicable for <i>Condition Change Criteria, Type of Change and System Integrity Characteristics</i> during system realization.	For example, if different combinations of types of change were possible to satisfy the condition change criteria, certain combinations may be more optimal to enact in terms of space or time requirements in order to effect the required change.

Table 6. Examples of Configuration Information of Context of Change

In the FAS case study, an example of *Condition Change Criteria* is the need for System A and System B to be dynamically reconfigured so that they inter-operate to leverage each other's short or long term predictions. An example of *Type of Change* is the new

Component that is substituted by System B. An example of *System Integrity Characteristics* is the management of active references to the existing *Scenario Analysis Component* of System B that the new *Component* is replacing. More detailed examples of each of these sub-features are given below.

The core concepts that compose *Condition Change Criteria*, *Type of Change*, and *System Integrity Characteristics* are now further described.

3.6.2.1. Primary Concepts of Condition Change Criteria

Figure 40 is a feature model diagram of the concept Condition Change Criteria. The concept is decomposed as Origin of Change, User-Driven Change, Computing Environment Imposed Change, Change Properties, Global Properties, Local Properties, and Reconciliation Policy sub-features to allow a direct mapping of change characteristics to equivalent system model constructs when effecting dynamic change to a running system. Table 7 provides further comments on important configuration or composition information about these features.

Figure 40. Feature Model of Condition Change Criteria

Dynamic reconfiguration may be needed based on anticipated or unanticipated properties that become apparent as the runtime environment of a system changes over time. *Origin of Change* represents the different kinds of stimulus that drive the need for software to evolve. A distinction is made between user-defined and computing

environment-imposed requirements that must be addressed in order to satisfy user needs within the context of a given application domain. Importantly, both sources of requirements may change over time and these requirements are continually reconciled.

In addition, the origin of change may represent reactive/programmed stimulus for change that is directly accommodated as anticipated changes defined and planned for at system design time. Alternatively, the origin of change may represent evolutionary/ad-hoc change which is initiated due to changes within a dynamic runtime environment and which is unplanned and could, if not properly managed, lead to system inconsistency.

User-Driven Change represents user-defined requirements that have been imposed upon a deployed system in order to satisfy changing user needs within the context of an application domain. User-driven change may be reactive/programmed or evolutionary/ad-hoc depending on how a system is engineered.

Computing Environment Change represents change stimulus whose origin is the runtime context or environment of a deployed system. Computing environment-imposed change may be reactive/programmed or evolutionary/ad-hoc depending on how a system is engineered.

Change Properties represent the global or local characteristics that distinguish the nature of user-driven or computing environment-imposed change. *Change Properties* are interpreted, in context, to determine the cost effectiveness of proceeding to enact change. The sub-features *Global Properties* and *Local Properties* represent global and local change properties, respectively.

Global Properties exist as implied by the specification of an overall system change formulated as a set of architectural constraints. By definition, global system change properties require global (system-level) knowledge for constraints conformance.

Local Properties exist as implied by the specification of a change formulated as a set of architectural constraints. By definition, local system change properties only require local (component-level) knowledge for constraints conformance.

Reconciliation Policy exists to enable local change properties to be reconciled with respect to global change properties as necessary. Due to reconciliation, not all change properties may be satisfied concurrently. In addition, the configuration of external interactions and internal behavior constructs may be impacted when these change properties are enacted to affect system evolution.

In the FAS case study, an example of User-Driven Change is the need for System A and System B to inter-operate. An example of Computing Environment Change is a fluctuation of financial data inputs processed by the Energy Sector Conditions components of either system. An example of Change Properties is the new control styles required for System A and System B to inter-operate (i.e. a change to GPI for each system).

Configuration Information	Comment
<p><i>Origin of Change</i> and <i>Change Properties</i> are mandatory sub-features of Condition Change Criteria.</p> <p><i>User-Driven Change</i> and <i>Computing Environment Change</i> are mandatory or-features of <i>Origin of Change</i>.</p> <p><i>Global Properties</i> and <i>Local Properties</i> are mandatory or-features of <i>Change Properties</i>. <i>Reconciliation Policy</i> is as a mandatory feature.</p>	<p>Each of these features must be present. Change properties characterize the adaptation that is conformed to by a system undergoing evolution. The origin of change reflects any stimuli that drive the existence of these properties. These features are the basic constructs that indicate what kind of change is represented within a particular evolution context.</p> <p>One or both of these features must be present. The origin of a component's context of evolution is due to one or one both of these features; the <i>Change Properties</i> feature represents that nature of change required.</p> <p>One or both of <i>Global Properties</i> and <i>Local Properties</i> must be present. <i>Reconciliation Policy</i> must be present. A component's context of evolution has global and/or local properties it must conform to; these properties are reconciled when they are in conflict.</p>
Default settings specifications are basic configurations of <i>Origin of Change</i> or <i>Change Properties</i> . They can be overridden during the lifetime of a system.	For example, default settings may give precedence to user-driven change over computing environment-imposed change when they occur concurrently.
Default dependency specifications are basic interdependencies among <i>Origin of Change</i> and <i>Change Properties</i> .	For example, given user-driven change is given precedence over computing environment-imposed changed, this may impact the policy for how global and local properties are reconciled.
Illegal feature combinations are aspects of <i>Origin of Change</i> and <i>Change Properties</i> that cannot co-exist or may only co-exist under special conditions that mitigated the incompatibility.	For example, the makeup of global and local properties and their reconciliation satisfy the origin of change. It is illegal for change properties to exist that do not reflect the nature of change that was desired or required.
Construction rules determine how <i>Origin of Change</i> and <i>Change Properties</i> are instantiated and interrelated.	For example, change properties are reconciled and constructed to best reflect the nature of change required.
Optimization strategies embody space and time efficiency techniques applicable for <i>Origin of Change</i> and <i>Change Properties</i> during system realization.	For example, change properties are reconciled and constructed in a manner to optimize the effort (in terms of space and/or time) to enact change within the context of a particular system environment.

Table 7. Examples of Configuration Information of Condition Change Criteria

3.6.2.2. Primary Concepts of Types of Change

Figure 41 is a feature model diagram of the concept *Types of Change*. They are described in Section 3.4. Table 8 provides further comments on configuration or composition information about these features.

Figure 41. Feature Model of Types of Change

The sub-features of *Types of Change* are *Architectural Change*, *Topology Change*, *Substitution*, *Protocol Change*, *Interface Change*, and *Internal Change*. *Architectural Change* is described in Section 3.4.1. *Topology Change* is described in Section 3.4.2. *Substitution* is described in Section 3.4.3. *Protocol Change* is described in Section 3.4.4. *Interface Change* is described in Section 3.4.5.

In the FAS case study, an example of *Architectural Change* is a change to GP1 for either system. An example of *Topology Change* is the new *Communication Path* that is established between them. An example of *Substitution* is the new *Scenario Analysis Component* that is substituted for the existing *Scenario Analysis Component* of System B. An example of *Protocol Change* are the new service protocols shown in Figure 12. Examples of *Interface Change* are the new required and provided services that support the new service protocols shown in Figure 12. An example of *Internal Change* is the new internal behavior of System A's *Scenario Analysis component* shown in Figure 13.

Configuration Information	Comment
<i>Internal Change, Substitution, Interface Change, Protocol Change, Topology Change</i> and <i>Architectural Change</i> are mandatory or-features of <i>Types of Change</i> .	One or more of these features must be present. Depending upon a component's condition change criteria, one or more types of dynamic reconfiguration may be required.
Default settings specifications are basic configurations associated with <i>Internal Change, Substitution, Interface Change, Protocol Change, Topology Change</i> or <i>Architectural Change</i> .	For example, default settings may time box change to help ensure system evolution in a timely and consistent fashion. Basic configurations may then be overridden during the lifetime of a system.
Default dependency specifications are basic interdependencies among <i>Internal Change, Substitution, Interface Change, Protocol Change, Topology Change</i> and <i>Architectural Change</i> .	For example, architectural change might make use or topology change and protocol change; topology change might make use of substitution, and so on.
Illegal feature combinations are aspects of <i>Internal Change, Substitution, Interface Change, Protocol Change, Topology Change</i> and <i>Architectural Change</i> that cannot co-exist or may only co-exist under special conditions that mitigated the incompatibility.	For example, it is legal to only compose change types in a certain order (i.e. architecture evolution may then require topology change but not the other way around).
Construction rules determine how <i>Internal Change, Substitution, Interface Change, Protocol Change, Topology Change</i> and <i>Architectural Change</i> are instantiated and interrelated.	For example, only compatible subsets of change types are constructed depending on the nature of change required.
Optimization strategies embody space and time efficiency techniques applicable for <i>Internal Change, Substitution, Interface Change, Protocol Change, Topology Change</i> and <i>Architectural Change</i> during system realization.	For example, depending on the nature of change required multiple distinct subsets of change types with varying space and/or time requirements may be possible. An optimization strategy selects the subset that minimizes these requirements.

Table 8. Examples of Configuration Information of Types of Change

3.6.2.3. Primary Concepts of System Integrity Characteristics

Figure 42 is a feature model diagram of the concept *System Integrity Characteristics*. They are described in Section 3.3. Table 9 provides further comments on configuration or composition information about these features.

Figure 42. Feature Model of System Integrity Characteristics

The sub-features of System Integrity Characteristics are Global Consistency Management, Local Consistency Management, Active Reference Management, Dependent Operation Management, Composite Component Management, Constrained Operation Management, and State Management. Global Consistency Management is described in Section 3.3.1. Local Consistency Management is described in Section 3.3.2. Active Reference Management is described in Section 3.3.3. Dependent Operation Management is described in Section 3.3.4. Composite Component Management is described in Section 3.3.5. Constrained Operation Management is described in Section 3.3.6. State Management is described in Section 3.3.7.

In the FAS case study, Global Consistency Management is ensuring GP1-GP3. An example of Local Consistency Management is ensuring LP1-LP3. An example of Active Reference Management is ensuring the integrity of active references to the existing Scenario Analysis Component of System B. When the internal behavior of System A's Scenario Analysis Component changed, the remaining integrity management areas ensure the integrity of any dependent operations, composite components, constrained operations, and state elements (these details were not shown in the example).

Configuration Information	Comment
<i>Active Component References, Component State, Constrained Operations, Dependent Operations, Composite Components, Local Consistency and Global Consistency</i> are optional or-features of <i>System Integrity Characteristics</i> .	Depending upon a component's condition change criteria and types of change required, one or more of these features may be required to ensure overall system integrity.
Default settings specifications are basic configurations of <i>Active Component References, Component State, Constrained Operations, Dependent Operations, Composite Components, Local Consistency</i> or <i>Global Consistency</i> . They can be overridden during the lifetime of a system.	For example, global consistency may be given precedence over local consistency.
Default dependency specifications are basic interdependencies among <i>Active Component References, Component State, Constrained Operations, Dependent Operations, Composite Components, Local Consistency</i> or <i>Global Consistency</i> .	For example, depending on the type of change, different subsets of system integrity characteristics may be required to help ensure system consistency during system evolution.
Illegal feature combinations are aspects of <i>Active Component References, Component State, Constrained Operations, Dependent Operations, Composite Components, Local Consistency</i> or <i>Global Consistency</i> that cannot co-exist or may only co-exist under special conditions that mitigated the incompatibility.	For example, component state is not applied if the nature of change demanded only realigning active component references.
Construction rules determine how <i>Active Component References, Component State, Constrained Operations, Dependent Operations, Composite Components, Local Consistency</i> or <i>Global Consistency</i> are instantiated and interrelated.	For example, only certain aspects of system integrity characteristics are constructed depending on the type of change required (i.e. if system integrity is required at all, global and local consistency are always constructed but other management areas may not be).
Optimization strategies embody space and time efficiency techniques applicable for <i>Active Component References, Component State, Constrained Operations, Dependent Operations, Composite Components, Local Consistency</i> or <i>Global Consistency</i> during system realization.	For example, optimization strategies guarantee system constructs exist to ensure relevant system integrity characteristics are applied efficiently (i.e. such as a special front-end component that automatically tracks active component references).

Table 9. Examples of Configuration Information of System Integrity Characteristics

Given the descriptions of the primary concepts just given, the following sequence drives how Context of Change is configured:

- Changing system specification drives *Origin of Change*;

- *User-Driven and/or Computing Environment Change drive Change Properties;*
- *Change Properties drives Type of Change;* and
- *Type of Change drives System Integrity Characteristics.*

Figure 43 is a complete feature model diagram of the concept *Context of Change*:

Figure 43. Complete Feature Model of Context of Change

3.6.3. Primary Concepts of Dynamic Reconfiguration

Figure 44 is a feature model diagram of the concept *Dynamic Reconfiguration*. The dynamic reconfiguration of a component-based software system is represented as an interaction among sub-features of *Context of Change* and *Interacting Components*. The concept is therefore shown as composed of *External Interactions*, *Internal Behavior*, *Dependencies*, and *Context of Change* sub-features.

Figure 44. Feature Model of Dynamic Reconfiguration

Table 10 provides further comments on configuration information about these features.

The FAS case study as a whole provides an application-specific example of comprehensive change.

Configuration Information	Comment
<i>External Interactions, Internal Behavior, Dependencies, and Context of Change</i> are mandatory sub-features of <i>Dynamic Reconfiguration</i> .	Each of these features must be present. They are the basic constructs that enable dynamic reconfiguration to be modeled in a manageable fashion, including context of evolution when functional or non-functional requirements change over time.
Default settings specifications are basic configurations of <i>External Interactions, Internal Behavior, Dependencies, or Context of Change</i> . They can be overridden during the lifetime of a system.	For example, limitations may be placed on what aspects of the system may be dynamically reconfigured such as only internal behavior.
Default dependency specifications are basic interdependencies among <i>External Interactions, Internal Behavior, Dependencies, and Context of Change</i> .	For example, condition change criteria change properties of the context of change are reconciled with system dependency properties during component evolution.
Illegal feature combinations are aspects of <i>External Interactions, Internal Behavior, Dependencies, and Context of Change</i> that cannot co-exist or may only co-exist under special conditions that mitigated the incompatibility.	For example, any existing system properties that are not compatible with dynamic reconfiguration change properties could not exist without a reconciliation strategy that reconciled this incompatibility.
Construction rules determine how <i>External Interactions, Internal Behavior, Dependencies, and Context of Change</i> are instantiated and interrelated.	For example, system external interaction and internal behavior constructs are re-constructed, if necessary, to align with reconciled change properties and existing system properties.
Optimization strategies embody space and time efficiency techniques applicable for <i>External Interactions, Internal Behavior, Dependencies, and Context of Change</i> during system realization.	For example, existing system properties and context of change properties may be reconciled in different ways with different tradeoffs in terms of space and time requirements. An optimization strategy attempts to choose a reconciliation strategy that minimized the space and/or the time required to effect system change.

Table 10. Examples of Configuration Information of Dynamic Reconfiguration

Figure 45 is a complete feature model diagram of the concept *Dynamic Reconfiguration*.

Figure 45. Complete Feature Model of Dynamic Reconfiguration

3.7. Alloy Model

The intent of formulating a more formal model is to refine the feature model in a precise way and to use this formal model as a bridge to a UML representation of dynamic reconfiguration. A rigorous methodology is employed to specify the primary feature interactions among *Context of Change* and *Interacting Components*. Alloy, a purely relation-oriented language, is used to describe the feature model as a formal specification. The Alloy Analyzer is then used to automatically validate its consistency, generate snapshots, execute operations, and check various properties of the model Jackson [69].

Alloy was chosen because it is a relatively simple yet powerful language for expressing executable specifications. Its built-in analysis engine can explore the possible executions of a system specification looking for conditions that are unexpected or which were not planned to happen. At the time of writing, Alloy version 2.1 was used [23, 24, 47, 48]. Fragments of the Alloy Model are used at the domain analysis level when specifying UML models in Chapter 4.

The following outlines the general approach taken:

- Each feature defined in the feature model is specified as a signature in Alloy;
- Based upon these signatures, containment relations are then specified to formally relate these features to each other;
- Based upon these containment relations, Alloy facts and assertions are specified as further constraints; and
- Based upon these facts and assertions, Alloy checks and functions are specified to generate snapshots, execute operations, or check properties.

With this approach, the feature model is transformed to a semantically richer Alloy model that specifies many constraints. The complete model is partitioned into separate Alloy modules based upon dependencies (for example, the *Condition Change Criteria* module is dependent upon the *System Properties* module, and so on). These dependencies build until the highest-level Alloy module specifies a dynamically reconfigurable system as fully constrained by the signatures, facts, and assertions defined in constituent modules. With this approach, each module may be independently checked for consistency representing well-formed building blocks of the complete specification.

3.7.1. Alloy Model of Interacting Components

This section presents an Alloy model that specifies, first at a high-level, the concept *Interacting Components*, and then in more detail, the concepts *External Interactions*, *Internal Behavior*, and *Dependencies*.

Figure 34 defines the domain model elements that underpin the following Alloy signature and containment relations that specify *Interacting Components*:

```
sig InteractingComponents {externalInteractions: ExternalInteractions,
internalBehavior: InternalBehavior, dependencies: Dependencies}
```

InteractingComponents is specified as a relation that contains *externalInteractions*, *internalBehavior*, and *dependencies* that are each constrained based on the Alloy models that are summarized below.

Figure 35 defines the domain model elements that underpin the following Alloy signatures and containment relations that specify *External Interactions*:

```
sig ServiceProtocol {}

sig Service {serviceProtocol: ServiceProtocol, implementation: set
PublicOperation}
```

```

disj sig RequiredService, ProvidedService extends Service {}

sig CommunicationPath {serviceProtocol: ServiceProtocol, connections:
set Connection}

sig Connection {requiredService: RequiredService, providedService:
ProvidedService}

sig Component {dependencies: Dependencies, requiredServices: set
RequiredService, providedServices: set ProvidedService,
communicationPaths: set CommunicationPath, compositeComponents: set
Component, internalBehavior: InternalBehavior}

sig ExternalInteractions {components: set Component, communicationPaths:
set CommunicationPath}

```

Driven by the domain analysis, *ExternalInteractions* is specified as a relation that contains a set *components* and a set *communicationPaths*. *Component* is specified as a relation that contains *dependencies*, a set *requiredServices*, a set *providedServices*, a set *communicationPaths*, a set *compositeComponents*, and *internalBehavior*.

CommunicationPath is specified as a relation that contains *serviceProtocol* and a set *connections*. A *Connection* is specified as a relation that contains *requiredService* and *providedService*. *RequiredService* and *ProvidedService* extend *Service* in a disjoint fashion. *Service* itself is specified as a relation that contains *serviceProtocol* and *implementation* (which is a public operation).

The following are representative Alloy facts, assertions, checks, and functions from the complete model related to *ExternalInteractions*:

```

fact {all c: Connection | some c1, c2: Component | c.requiredService in
c1.requiredServices && c.providedService in c2.providedServices}

fact {all c: Connection | some c1, c2: Component | c.providedService in
c1.providedServices && c.requiredService in c2.requiredServices} //same
effect

fact {all c: Connection | some comp: Component | c.requiredService in
comp.requiredServices && c.providedService !in comp.providedServices}

fact {all c: Connection | some comp: Component | c.providedService in
comp.providedServices && c.requiredService !in comp.requiredServices}

fact {all comp: Component | comp !in comp.^compositeComponents}

```

```
fact {all comp: Component, comm: CommunicationPath | all s: Service |
CommunicationPathConnected(comp, comm) && ((s in comp.providedServices
&& comm.serviceProtocol = s.serviceProtocol) || (s in
comp.requiredServices && comm.serviceProtocol = s.serviceProtocol))}
```

These facts establish that (i) a connection connects a provided service of one component to a required service of another component, (ii) a connection does not connect a provided service to a required service, or a required service to a provided service, of the same component, (iii) a component is not a member of its own composite components, and (iv) if a component and a communication path are associated, the service protocol of the communication path is the same service protocol as that of a provided or required service of the component.

The last fact is formed using a helper function `CommunicationPathConnected()`, which is shown below.

The assertions can be made, and checked, that some components can be connected and that some components can be actively connected.

```
assert SomeComponentsConnected {some c1, c2: Component | some c:
CommunicationPath | ComponentsConnected(c1, c2, c)}

check SomeComponentsConnected for 2

assert SomeComponentsActivelyConnected {some c1, c2: Component | some c:
CommunicationPath | ComponentsActivelyConnected(c1, c2, c)}

check SomeComponentsActivelyConnected for 2
```

Both of these examples of assertions are formed using helper functions `ComponentsConnected()` and `ComponentsActivelyConnected()`, which are shown below.

```
fun CommunicationPathConnected (comp: Component, c: CommunicationPath)
{c in comp.communicationPaths}

run CommunicationPathConnected for 3

fun ComponentsConnected (c1, c2: Component, c: CommunicationPath) {c in
c1.communicationPaths && c in c2.communicationPaths}

run ComponentsConnected for 3
```



```

fun ComponentsActivelyConnected (c1, c2: Component, c:
CommunicationPath) {ComponentsConnected(c1, c2, c) && some
c.connections}

run ComponentsActivelyConnected for 3

```

The first function ensures that the communication path is in a component's set of communication paths. The second function ensures that a communication path is in the set of communication paths of two different components, *c1* and *c2*. The third function ensures that a communication path connects two components and that the communication path has active connections. As shown, the three functions can be run directly.

Figure 36 defines the domain model elements that underpin the following Alloy signatures and containment relations that specify *Internal Behavior*:

```

sig StateElement {}

disj sig PermanentStateElement extends StateElement {operations: set
Operation}

disj sig TemporaryStateElement extends StateElement {localOperation:
Operation}

sig Operation {operations: set Operation, compositeComponentServices:
set Service, temporaryStateElements: set TemporaryStateElement,
permanentStateElements: set PermanentStateElement}

disj sig PublicOperation extends Operation {ExternalService: Service}

disj sig PrivateOperation extends Operation {}

sig ServiceInvocation {operations: set Operation}

sig InternalBehavior {operations: set Operation, stateElements: set
PermanentStateElement, serviceInvocations: set ServiceInvocation}

```

Driven by the domain analysis, *Internal Behavior* is specified as a relation that contains a set *operations*, a set *stateElements* (which are permanent state elements), and a set *service invocations*. The model formally specifies that a particular state element is either permanent or it is temporary. A permanent state element is specified as a relation that

contains the operations that access this state element. A temporary state element is specified as a relation that contains the local operation that has defined it.

Operations are specified to be public or private. Public operations are specified as a relation that contains the external service that the operation implements. By extension, all operations are specified as a relation that contains a set *operations* (that are invoked by this operation), a set *composite Component Services* (that are invoked by this operation), a set *temporary State Elements* (that are defined and accessed by this operation), and a set *permanent State Elements* (that are accessed by this operation).

Finally, *Service Invocation* is specified as a relation that contains a set *operations* (that are executed by the service invocation).

Figure 37 defines the domain model elements that underpin the following Alloy signatures and containment relations that specify *Dependencies*:

```

sig SpecificationAsConstraint {}

sig Property {constraint: SpecificationAsConstraint}

disj sig GlobalProperty, LocalProperty extends Property {}

sig ReconciliationPolicy {disj compatibleProperties,
incompatibleProperties: Property -> Property}

sig SystemProperties {globalProperties: set GlobalProperty,
localProperties: set LocalProperty, reconciliationPolicy:
ReconciliationPolicy}

sig BehavioralSignature {ProtocolDependencies: Service ->
ServiceProtocol, RequiredServiceDependencies: Operation ->
RequiredService, ProvidedServiceDependencies: Operation ->
ProvidedService, OperationDependencies: Operation -> Operation,
StateElementDependencies: Operation -> StateElement,
CompositeComponentDependencies: Operation -> CompositeComponentService}

sig StructuralSignature {ConnectedComponentDependencies: Component ->
Component, ConnectedServiceDependencies: RequireService ->
ProvidedService}

sig SystemSignature {structuralSignature: StructuralSignature,
behavioralSignature: BehavioralSignature}

sig Dependencies {systemProperties: SystemProperties, systemSignature:
SystemSignature}

```

Driven by the domain analysis, *Dependencies* is formally specified as a relation that contains *systemProperties* and *systemSignature*. *SystemProperties* is specified as a relation that contains a set *globalProperties*, a set *LocalProperties*, and an associated *reconciliationPolicy*. *SystemSignature* is specified as a relation that contains *structuralSignature* and *behavioralSignature*.

At the root of the Alloy model is the assumption that a system specification may be represented by a set of constraints Marriott and Stuckey [49]. Given this, global or local system or change properties can be specified as constraints.

Property is specified as a relation that contains *SpecificationAsConstraint*. *Global Property* and *Local Property* are specified as extensions to *Property* as disjoint signatures. *Reconciliation Policy* is specified as containing the disjoint relations *compatibleProperties* and *incompatibleProperties*. These relations specify compatible or incompatible properties as *Property* to *Property* relations.

The following are representative Alloy facts from the complete model related to properties:

```
fact{no p: Property | some rp: ReconciliationPolicy | p -> p in
rp.compatibleProperties || p -> p in rp.incompatibleProperties}

fact{no p1, p2: Property | some rp: ReconciliationPolicy | p1 -> p2
in rp.compatibleProperties && p1 -> p2 in rp.incompatibleProperties}

fact{no p1, p2: Property | some rp: ReconciliationPolicy | p1 -> p2 in
rp.compatibleProperties && p2 -> p1 in rp.incompatibleProperties}
```

These particular facts indicate: (i) a given property may not be related to itself as compatible or incompatible properties, (ii) if *p1* is related to *p2* as a compatible property, *p1* may not be related to *p2* as an incompatible property, and (iii) if *p1* is related to *p2* as a compatible property, *p2* may not be related to *p1* as an incompatible property.

The Alloy specification formally constrains the behavioral and structural profile of a system. *BehavioralSignature* is specified as relations between a service and a service protocol, an operation and a required service, an operation and a provided service, an operation and another operation, an operation and a state element, or an operation and a composite component service. *StructuralSignature* is specified as relations between a component and another component or a provided service and required service.

The following is a fact from the complete model related to dependencies:

```
fact{some op1, op2: Operation | some b: BehavioralSignature | op1 in
op2.operations || op2 in op1.operations => op1 -> op2 in
b.operationDependencies}
```

This particular fact indicates: if op1 relies on op2 or op2 relies op1, this implies that op1 is related to op2 as an operation dependency.

3.7.2. Alloy Model of Context of Change

This section presents an Alloy model that specifies, first at a high-level, the concept *Context of Change*, and then in more detail, the concepts *Condition Change Criteria*, *Type of Change*, and *System Integrity Characteristics*.

Figure 39 defines the domain model elements that underpin the following Alloy signature and containment relations that specify *Context of Change*:

```
sig ContextOfChange {context: ConditionChangeCriteria -> ChangeType ->
SystemIntegrityCharacteristics}
```

The Alloy model fragment above specifies *Context of Change* as containing a *ConditionChangeCriteria* to *ChangeType* to *SystemIntegrityCharacteristics* relation. When analyzed, it resolves to a relation that associates particular instances of *OriginOfChange* (which itself contains particular instances of user-driven and/or computing environment-imposed change related to an instance of *ChangeProperties*) with

a particular change type with a set of system integrity characteristics (which may be empty since its sub-features are optional-or).

Figure 40 defines the domain model elements that underpin the following Alloy signatures and containment relations that specify *Condition Change Criteria*:

```
sig OriginOfChange {}

disj sig UserDrivenChange, DomainImposedChange extends OriginOfChange{}

sig ChangeProperties {globalProperties: set GlobalProperty,
localProperties: set LocalProperty, reconciliationPolicy:
ReconciliationPolicy}

sig ConditionChangeCriteria {originOfChange: set OriginOfChange,
changeProperties: ChangeProperties}
```

Driven by the domain analysis, *Condition Change Criteria* is specified as a relation that contains the set *originOfChange* and *changeProperties*. The members of *originOfChange* may be *UserDrivenChange* and/or *DomainImposedChange*. Similar to *SystemProperties*, *ChangeProperties* is specified as a relation containing a set of *globalProperties*, a set of *localProperties* and an associated *reconciliationPolicy*. This reflects the domain analysis that user and/or computing environment-imposed stimulus for change can be represented as reconciled global and/or local properties.

If, for example, *ReconciliationPolicy* was an optional sub-feature (which it is not) it would be specified as optional in the Alloy model:

```
sig ChangeProperties {globalProperties: set GlobalProperty,
localProperties: set LocalProperty, optional reconciliationPolicy:
ReconciliationPolicy}
```

The following is a fact from the complete model related to change properties:

```
fact{all p: Property | some cp: ChangeProperties | p in
cp.globalProperties || p in cp.localProperties}
```

This particular fact indicates that all properties exist as global or local properties of some change properties.

Figure 41 defines the domain model elements that underpin the following Alloy signatures and containment relations that specify *Type of Change*:

```
sig ChangeType {systemIntegrityCharacteristics: set
SystemIntegrityCharacteristic}

disj sig ArchitecturalChange extends ChangeType {changeProperties:
ChangeProperties, systemProperties: SystemProperties}

disj sig TopologyChange extends ChangeType {components: set Component,
communicationPaths: set CommunicationPath}

disj sig Substitution extends ChangeType {components: set Component,
communicationPaths: set CommunicationPath }

disj sig ProtocolChange extends ChangeType {serviceProtocols: set
ServiceProtocol}

disj sig InterfaceChange extends ChangeType {services: set Service}

disj sig InternalChange extends ChangeType {operations: set Operation,
stateElements: set StateElement, serviceInvocations: set
ServiceInvocation}
```

Driven by the domain analysis, *Types of Change* is specified as *ChangeType* that is extended by each type of change in a disjoint fashion. Each extension is a relation that contains the domain model elements that are affected by the particular *ChangeType* that is specified. *ChangeType* itself is specified as a relation that contains *changeProperties* and *systemIntegrityCharacteristics* since change properties lead to particular kinds of change which are governed by particular system integrity characteristics.

Figure 42 defines the domain model elements that underpin the following Alloy signatures and containment relations that specify *System Integrity Characteristics*:

```
sig SystemIntegrityCharacteristics {integrityCharacteristics: set
SystemIntegrityCharacteristic}

sig SystemIntegrityCharacteristic {}

disj sig GlobalConsistency extends SystemIntegrityCharacteristic
{globalProperties: set GlobalProperty}
```

```

disj sig LocalConsistency extends SystemIntegrityCharacteristic
{localProperties: set LocalProperty}

disj sig ActiveReference extends SystemIntegrityCharacteristic
{communicationPaths: set CommunicationPath, components: set Component}

disj sig DependentOperation extends SystemIntegrityCharacteristic
{dependentOperations: set Operation}

disj sig CompositeComponent extends SystemIntegrityCharacteristic
{compositeOperations: set Operation}

disj sig ConstrainedOperation extends SystemIntegrityCharacteristic
{constrainedOperations: set Operation, constrainedStateElements: set
StateElement}

disj sig ComponentState extends SystemIntegrityCharacteristic
{stateElements: set StateElement}

```

Driven by the domain analysis, *System Integrity Characteristics* is specified a relation that contains a set *integrityCharacteristics*. Specific system integrity characteristics are each specified as disjoint signatures that extend *SystemIntegrityCharacteristic* one for each of the seven kinds of system integrity previously described. Each extension is a relation that contains the domain model elements that are governed by the particular *System Integrity Characteristic* that is specified.

The following is a representative fact of a list of facts of the complete model that constrains certain integrity characteristics from being related to certain change types:

```

fact{all c: ContextOfChange | some ccc: ConditionChangeCriteria | some
ac: ArchitecturalChange | some si: SystemIntegrityCharacteristics | no
ar: ActiveReference | c.context = ccc -> ac -> si && ar in
si.integrityCharacteristics}

```

This particular fact indicates that active references are excluded from being an integrity characteristic of architectural change.

An assertion can be made, and checked, that when there are active references there is not architectural change.

```

assert ActiveReferencesButNotArchitecturalChange {all c: ContextOfChange
| ActiveReferencesExist(c) && NotArchitecturalChange(c)}

```

check `ActiveReferencesButNotArchitecturalChange` for 1

This example of an assertion is formed using helper functions

`NotArchitecturalChange()` and `ActiveReferencesExist()`, which are shown below.

```
fun NotArchitecturalChange(c: ContextOfChange) {some ccc:
ConditionChangeCriteria | some ct: ChangeType | some si:
SystemIntegrityCharacteristics | all ac: ArchitecturalChange | c.context
= ccc -> ct -> si && ct != ac}
```

```
fun ActiveReferencesExist(c: ContextOfChange) {some ccc:
ConditionChangeCriteria | some ct: ChangeType | some si:
SystemIntegrityCharacteristics | some ar: ActiveReference | c.context =
ccc -> ct -> si && ar in si.integrityCharacteristics}
```

The first function ensures that the context of change does not include architectural change. The second function ensures that the context of change does include active references within its system integrity characteristics.

3.7.3. Alloy Model of Dynamic Reconfiguration

This section presents an Alloy model that specifies, at a high-level, the domain concept *Dynamic Reconfiguration*.

Figure 44 defines the domain model elements that underpin the following Alloy signature and containment relations that specify *Dynamic Reconfiguration*:

```
sig DynamicallyReconfigurableSystem {externalInteractions:
ExternalInteractions, internalBehavior: InternalBehavior, dependencies:
Dependencies, contextOfChange: ContextOfChange}
```

DynamicallyReconfigurableSystem is specified as a relation that contains *externalInteractions*, *internalBehavior*, *dependencies*, and *contextofChange* that are each constrained based on the Alloy models that were summarized previously in this section.

3.7.4. Discussion

The Alloy modeling language is expressive enough to enable the domain analysis to be specified in a manner that can be validated for consistency within the Alloy run-time

environment. The run-time environment is the analyzer component of Alloy, which implements a satisfiability (SAT) solver. The Analyzer can search within a user defined search space for possible counter examples of a design specification.

Using the interactive features of the Analyzer, the constraint space of each Alloy module can be automatically explored when searching for constraint-based solutions (snapshots) that represent conditions that are not intended. This capability is powerful when validating semantically richer models that specify many constraints because the constraint space of any model of sufficient complexity quickly explodes to a size beyond the realm of manual inspection attempting to validate the model's consistency.

As an example, Figure 46 shows a visual representation generated by the Analyzer. It represents one valid solution that was generated testing the *System Properties* module when run for three properties.

```
sig PState {db: List [ReconciliationPolicy]}
```

```
fun registerPState (pre, post: PState, cps: GlobalProperty ->
LocalProperty, ips: GlobalProperty -> LocalProperty) {some rp:
ReconciliationPolicy {rp.compatibleProperties = cps
rp.incompatibleProperties = ips post.db =
pre.db.(ListOps [ReconciliationPolicy].append) [rp]}}
```

```
run registerPState for 3
```

The counter example reveals a discrepancy in the intended design because existing constraints can be resolved as an unacceptable outcome. In this example, the Alloy Analyzer output is useful because it indicates that the module is under specified. It shows visually that *Property_2* and *Property_3* are not related when they should be related (as compatible properties or as incompatible properties).

Figure 46. Example Visual Representation Generated by the Alloy Analyzer

3.8. Summary and Conclusion

This chapter described a domain analysis that identifies a set of common features that define a dynamically reconfigurable component-based software system that implements services. Using a running example of the dynamic inter-operation of two financial analysis systems, the chapter identified and categorized the various types of change that may be required, the relationships among those types, and the system integrity characteristics that need to be considered when such changes take place.

3.8.1. Summary

The chapter first presented a case study of examples of global and local properties. It then identified integrity management areas and types of change that occur in a dynamically evolving system. The common and variable causal flow of dynamic reconfiguration was then described. The essentials of the proposed domain model were then expressed in the form of a feature model. This was used as a basis for deriving a more refined formal model expressed using the Alloy language that allowed the domain model to be formally validated for consistency.

A use case was used to specify the common and variable causal flow of dynamic reconfiguration; this use case described the different levels of dynamism as alternatives based upon change groups. The following change groups were identified: *Pure Architectural Change*, *Pure Behavioral Change*, *Behavior-driven Structural Change*, *Comprehensive Behavior-driven Change*, *Pure Structural Change*, *Structure-driven Behavioral Change*, *Comprehensive Structure-driven Change*, *Pure Behavioral and Structural Change* and *Comprehensive Change*.

Each change group was defined to be composed of a set of related change sequences that describe how a particular change group unfolds. A change sequence combines a subset of the change types that apply within the sequence; the change types of a change sequence determine whether the particular change sequence implies architectural, behavioral, and/or structural change.

The following types of change were identified: *Architectural Change*, *Topology Change*, *Substitution*, *Protocol Change*, *Interface Change*, and *Internal Change*.

Architectural Change affects global and local properties. There is an indirect affect upon a system's structural or behavioral signature that in turn affects the configuration of other domain model elements. Topology Change affects component and communication path. Substitution affects component and connection. Protocol Change affects service protocol. Interface Change affects required service or provided service. Internal Change affects operation, state element, and service invocation.

To help ensure overall system consistency in the face of change, the following system integrity characteristics were identified: *Global Consistency Management*, *Local Consistency Management*, *Active Reference Management*, *Dependent Operation*

Management, Composite Component Management, Constrained Operation Management and State Management.

Change types are related to system integrity management areas through the domain model elements that underpin the interdependency. Global Consistency management is interdependent with Architectural Change through ensuring the consistency of global properties. Local Consistency management is interdependent with Architectural Change through ensuring the consistency of local properties. Both consistency management areas are indirectly interdependent with Topology Change and Substitution through structural signature. They are indirectly interdependent with Protocol, Interface, and Internal Change through behavioral signature. Active Reference management is interdependent with Topology Change and Substitution through ensuring the consistency of component and communication path. Dependent Operation and Composite Component management are interdependent with Internal Change through ensuring the consistency of operation. Constrained Operation management is interdependent with Internal Change through ensuring the consistency of operation and state element. State management is interdependent with Substitution and Internal Change through ensuring the consistency of state element.

3.8.2. Conclusion

By explicitly abstracting dependencies, external interactions, and internal behavior, the system model is a simple but powerful model of computation that is based on widely known and accepted concepts used for reasoning about complex software systems. In addition, this particular decomposition enables the explicit identification of key

relationships that integrate context of change within the model as a whole and in doing so facilitates a systematic treatment of the problem of dynamic system reconfiguration.

Because the system model is aligned or easily mappable to ADL-based techniques, component-based systems, and service-oriented architectures it is broadly applicable and is a useful context for assessing the comprehensiveness of the change types and related system integrity characteristics that are defined in conjunction with it. Also, when the model is translated to Alloy, the richer semantics of the domain analysis are validated as an executable specification of the domain design that is well formed.

Finally, the system model provides a narrow enough scope to allow a 'reduced instruction set of change' to be identified and the common and variable causal flow of dynamic reconfiguration to be described that provides the building blocks and a universal composition framework for the domain design. In Chapter 4, the feature and Alloy models are used to produce a UML model that combines the various aspects as a software architecture of dynamic reconfiguration.

4. Domain Design

4.1. Introduction

Chapter 3 presented the primary feature interactions and the complete feature model (See Figure 45) of dynamic reconfiguration. Based upon the domain analysis, this chapter presents a domain design that is expressed using UML [39, 40, 50, 51]. “A domain architecture is a reference model for a range of applications that share similar structure, functionality, and behavior ... it is a meta-model that describes how specific instance systems are created” Duffy [40].

In conjunction with the domain analysis, the domain design is useful as a software architecture that specifies distinct dimensions of change that can be realized individually, or in their entirety, when forward-engineering an application-specific system with dynamic reconfiguration capability. It is useful as a metamodel when engineering a component framework that manages a dynamically reconfigurable component-based system. It is useful as a reference model when assessing existing or to be built systems for their level of dynamism.

UML was chosen to express the domain design because it is inherently object-oriented and it is the industry standard notation for representing the software architecture of a system or an application. It is a family of graphical notations that are useful for describing the structural and behavioral (and other) aspects of a software system. Throughout this section four different UML notations are used. Activity models are used to show the primary feature interactions of each change type or change group. Based on the primary feature interaction models, Class models show the structural entities of the architecture. Based on the class models, Sequence models show how the structural

entities inter-operate to realize the primary feature interactions. Finally, State Machine models show the life-cycle of each change type or change group when these interactions occur.

Figure 47. Domain Concepts as UML Classes

First, Figure 47 shows a class model representation of the domain concepts as classes. The intent is to present a UML context for the domain design highlighting the importance of change properties reconciliation, change enactment, and further dynamic reconfiguration concerns.

With respect to change properties reconciliation, the classes *System Properties* and *Change Properties* represent the current system properties and (new) change properties, respectively, as global and local constraints. Each grouping of properties is internally reconciled according to reconciliation policies. In addition, these groupings are reconciled with respect to each other. The overall reconciliation is modeled as a binary association linking *System Properties* and *Change Properties*. The reconciled change properties in turn determine which subsets of change types are required to perform the necessary system reconfiguration.

With respect to change enactment, the associations among *ChangeType*, *SystemIntegrityCharacteristics*, and *SystemModel* specify how the required change types, as constrained by system integrity characteristics, realign the system model. Depending upon the types of change required, the structural and/or behavioral signature of the system may have to be re-aligned, which, in turn, may impact its internal behavior or external interactions.

With respect to further dynamic reconfiguration, given that the internal behavior or the external interactions of a system have changed, the system may reach a failure state or require further change based upon emergent properties. This is represented by the group of classes *External Interactions*, *Internal Behavior*, *Run Time Environment*, and *Origin Of Change*. *External Interactions* and *Internal Behavior* are related to *Run Time*

Environment upon failure (or emergent properties). *Run Time Environment* then drives *Origin Of Change* (which is user driven and/or computing environment imposed change) which in turn indicates needed change properties. This completes the cycle of system dynamic reconfiguration described by this domain model.

As stated, UML activity, class, sequence, and state machine models representations of the domain analysis are formulated. Each of the six change types, shown as *ChangeType* in Figure 47, is modeled separately, first at the domain analysis level and then at the UML level [41, 70-73]. Relevant fragments of the domain analysis are used to populate the domain level; this includes domain level concepts as features and as Alloy relations. The UML constructs are then formulated based on the domain analysis level models. The change type UML models are then composed [52, 74-85] based upon the change sequences described in Chapter 3 to represent each of the nine change groups.

In addition, an executable representation of the domain design is presented. “By specializing the architecture to the domain it is possible to increase the descriptive power of structures. Indeed, in many cases the architecture is sufficiently constrained that an executable system can be generated automatically or semi-automatically from the architectural description itself ...” Shaw and Garlan [53]. This chapter illustrates the use of a programming environment that supports explicit metaclass programming techniques to demonstrate an application-specific example of dynamic reconfiguration. The purpose of constructing an executable model is: (i) to gain a more direct understanding of the conceptual framework and (ii) to validate, through empirical proof and simulation, that the constrained model is applicable within application-specific contexts.

The rest of this Chapter first presents a domain design of each change type. The change type design models are then composed as a domain design for each of the change groups. Finally, based on the domain design, an executable realization of the case study is described.

4.2. Architectural Change Model

Architectural Change affects global and local properties. There is an indirect affect upon a system's structural or behavioral signature that in turn can effect the configuration of other domain model elements. *Global Consistency Management* is interdependent with *Architectural Change* through ensuring the consistency of global properties. *Local Consistency Management* is interdependent with *Architectural Change* through ensuring the consistency of local properties. First, the domain level concepts that define *Architecture Change* are presented and then a UML representation of the domain level specification is formulated.

4.2.1. Domain Level

This sub-section presents the specific domain-level concepts that define *Architectural Change* first as features and then as Alloy relations.

4.2.1.1. Domain Concepts as Features

Figure 48 shows the primary feature interactions of *Architectural Change*. *Change Properties* indicate that *Architectural Change* is required to satisfy condition change criteria. *Architectural Change* in turn indicates the need for *Global Consistency* and *Local Consistency* system integrity characteristics. *Architectural Change* first involves the realignment of *Global Properties* and *Local Properties*. The realignment of *System*

Signature (if necessary) occurs after the reconciliation of the new *Change Properties* with current *System Properties*.

Figure 48. Primary Feature Interactions of Architectural Change

4.2.1.2. Domain Concepts as Relations

Based on the domain analysis given in Chapter 3 and the primary feature interactions indicated in Figure 48, the following Alloy signatures and containment relations specify

Architectural Change:

```

sig ChangeProperties {globalProperties: set GlobalProperty,
localProperties: set LocalProperty, reconciliationPolicy:
ReconciliationPolicy}

sig ChangeType {systemIntegrityCharacteristics: set
SystemIntegrityCharacteristic}

disj sig ArchitecturalChange extends ChangeType {changeProperties:
ChangeProperties, systemProperties: SystemProperties}

sig SystemIntegrityCharacteristic {}

disj sig GlobalConsistency extends SystemIntegrityCharacteristic
{globalProperties: set GlobalProperty}

disj sig LocalConsistency extends SystemIntegrityCharacteristic
{localProperties: set LocalProperty}

sig Dependencies {systemProperties: SystemProperties, systemSignature:
SystemSignature}

```

```
sig SystemProperties {globalProperties: set GlobalProperty,
localProperties: set LocalProperty, reconciliationPolicy:
ReconciliationPolicy}
```

```
sig SystemSignature {structuralSignature: StructuralSignature,
behavioralSignature: BehavioralSignature}
```

The following description characterizes how *Architectural Change* is formally specified:

- Through *Architectural Change*, *Change Properties* are formally related to *System Properties*
- *Change* and *System Properties* are each formally specified as groupings of reconciled *Global* and *Local Properties*
- A particular change type, through *ChangeType*, is formally related to a set of system integrity characteristics. In this case, the set (only) includes *Global Consistency* and *Local Consistency*
- Each system integrity characteristic is formally related to the domain model element that it manages
- *Dependencies* formally relates *System Properties* to *System Signature*
- *System Signature* formally contains *Structural Signature* and *Behavioral Signature*

4.2.2. UML Modeling Level

This sub-section represents the domain-level concepts of *Architectural Change* as a series of behavioral and structural UML models.

4.2.2.1. Activity Model

Based on Figure 22, Figure 49 is an Activity Model of the primary feature interactions of *Architectural Change* shown in Figure 48.

Figure 49. Activity Model of Architectural Change

The following sequence of actions are represented:

- During the action '*Detect Origin of Change*' a system responds to user-driven or computing environment imposed change stimuli driving the need for further dynamic reconfiguration.
- During the action '*Represent Architectural Change through Global/Local Properties*' a system characterizes the nature of change as *Change Properties*.

- During the action '*Reconcile Change Properties with Existing System Properties*' a system reconciles *Change Properties* with *System Properties*. For example, through static analysis a determination is made whether mandatory global or local system properties are compromised or not.
- The decision point immediately after '*Reconcile Change Properties with Existing System Properties*' indicates this reconciliation may lead to change rejection.
- Given *Change Properties* have been reconciled with *System Properties*, during the action '*Estimate Impact of Architectural Change upon System Integrity*' a system evaluates the impact of proceeding with the change. For example, through a dynamic analysis a determination is made whether mandatory global or local system properties are compromised or not.
- The decision point immediately after '*Estimate Impact of Architectural Change upon System Integrity*' indicates this impact analysis may lead to change rejection.
- The parallel fork indicates '*Enact Architectural Change by Realigning System Constructs*' and '*Ensure System Integrity Characteristics*' happens concurrently.
- During the action '*Enact Architectural Change by Realigning System Constructs*' a system commits to a change to *System Properties* based on *Change Properties*. A change to *System Properties* may lead *System Signature* to change.
- During the action '*Ensure System Integrity Characteristics*' a system ensures the integrity of changing *Global* and *Local Properties* through *Global* and *Local Consistency Management* respectively.
- The parallel join indicates a system has enacted *Architectural Change* while attempting to ensure overall system consistency.

- During the action '*Provide Indication of Change*' a system indicates how *Change Properties* affected *System Properties*. If the *System Signature* changed this will lead to other kinds of behavioral or structural change.

4.2.2.2. Class Model

Based on the domain-level concepts, Figure 50 shows a Class Model of *Architectural Change*.

Figure 50. Class Model of Architectural Change

The model associates *Architectural Change* with *Change Properties* and *System Properties* to indicate that *Architectural Change* is an update to *System Properties* according to the nature of *Change Properties* and their overall reconciliation. *Global Consistency* and *Local Consistency* management is shown associated with *Architectural Change* and *System Properties* to indicate, during the reconciliation process, *Global Consistency Management* ensures the consistency of *Global Properties* and *Local Consistency Management* ensures the consistency of *Local Properties*. *System Properties* is shown associated with *System Signature* to indicate that an update to *System Properties*

may change the *Behavioral* or *Structural Signature* of a system. Hence, *Architectural Change* and *Global* and *Local Consistency Management* subsequently affects other change types.

4.2.2.3. Sequence Model

Given the Activity Model, shown in Figure 49, and the Class Model shown in Figure 50, Figure 51 shows a Sequence Model of *Architectural Change*.

Figure 51. Sequence Model of Architectural Change

The following interactions are represented:

- The action '*Detect Origin of Change*' is represented by the message sequence *ChangeProperties::emergentProperties()*.
- The action '*Represent Architectural Change through Global/Local Properties*' is represented by the message sequence *ChangeProperties::reconcileProperties()*.
- The action '*Reconcile Change Properties with Existing System Properties*' is represented by the message sequence *ArchitecturalChange::changeArchitecture()*, *SystemProperties::reconcileProperties()* and *ArchitecturalChange::indicationOfReconcililation()*.
- The first decision point is represented by the outer Alternative Interaction Frame.
- The parallel fork is represented by the Parallel Interaction Frame.
- The action '*Estimate Impact of Architectural Change on System Integrity*' is represented by the message sequence *ArchitecturalChange::estimateImpact()*.
- The second decision point is represented by the inner Alternative Interaction Frame.
- The action '*Enact Architectural Change by Realigning System Constructs*' is represented by the message sequence *SystemProperties::changeProperties()* and *SystemSignature::changeSignature()* of the Parallel Interaction Fragment.
- The action '*Ensure System Integrity Characteristics*' is represented by the message sequence *GlobalConsistencyManagement::ensureGlobalProperties()* and *LocalConsistencyManagement::ensurelocalProperties()* of the Parallel Interaction Fragment.
- The parallel join is represented by the completion of the Parallel Interaction Fragment.

- The action ‘*Provide Indication of Change*’ is represented by the message sequence *OriginOfChange::indicationOfChange()*.
- The action ‘*Reject Change*’ is represented by the message sequence *OriginOfChange::rejectChange()*.

4.2.2.4. State Machine Model

Given the Activity Model, shown in Figure 49, the Class Model, shown in Figure 50, and the Sequence Model, shown in Figure 51, Figure 52 shows a State Machine Model of *Architectural Change*.

The following states of *Architectural Change* are represented:

- ‘*Initial*’ is its state at the time of creation and initialization.
 - ‘*Change Properties Reconciled*’ is its state after *ArchitecturalChange::changeArchitecture()*.
 - ‘*System Properties Reconciled*’ is its state after *ArchitecturalChange::indicationOfReconciliation()*.
 - ‘*Impact Assessed*’ is its state after *ArchitecturalChange::estimateImpact()*.
 - ‘*Change Rejected*’ is its state before *OriginOfChange::rejectChange()*.
 - The seven states leading to the state ‘*Change Indicated*’ represents the combination of states possible based on the Parallel Interaction Frame shown in Figure 51.
- ‘*Change Indicated*’ is its state before *OriginOfChange::indicationOfChange()*.

Figure 52. State Machine of Architectural Change

4.3. Topology Change Model

Based on an indirect affect upon a system's *Structural Signature* due to *Architectural Change*, *Topology Change* affects *Component* and *Communication Path*. By affecting the *Structural Signature* of a system, *Global* and *Local Consistency Management* are interdependent with *Topology Change*. *Active Reference Management* is interdependent with *Topology Change* when ensuring the integrity of existing components and

communication paths. First, the domain level concepts that define *Topology Change* are presented and then a UML representation of the domain level specification is formulated.

4.3.1. Domain Level

This sub-section presents the specific domain-level concepts that define *Topology Change* first as features and then as Alloy relations.

4.3.1.1. Domain Concepts as Features

Figure 53 shows the primary feature interactions of *Topology Change*. Reconciled *Change Properties* and *System Properties* through *Architectural Change* indicate that *Topology Change* is required to satisfy condition change criteria. *Topology Change* in turn indicates the need for *Global Consistency*, *Local Consistency* and *Active Reference Management* system integrity characteristics. Due to a realignment of *Structural Signature*, *Topology Change* involves the realignment of *Component* and *Communication Path*.

Figure 53. Primary Feature Interactions of Topology Change

4.3.1.2. Domain Concepts as Relations

Based on the domain analysis given in Chapter 3 and the primary feature interactions indicated in Figure 53, the following Alloy signatures and containment relations specify

Topology Change:

```
sig StructuralSignature {ConnectedComponentDependencies: Component ->
Component, ConnectedServiceDependencies: RequireService ->
ProvidedService}

sig ChangeType {systemIntegrityCharacteristics: set
SystemIntegrityCharacteristic}

disj sig TopologyChange extends ChangeType {components: set Component,
communicationPaths: set CommunicationPath}

sig SystemIntegrityCharacteristic {}

disj sig GlobalConsistency extends SystemIntegrityCharacteristic
{globalProperties: set GlobalProperty}

disj sig LocalConsistency extends SystemIntegrityCharacteristic
{localProperties: set LocalProperty}

disj sig ActiveReference extends SystemIntegrityCharacteristic
{communicationPaths: set CommunicationPath, components: set Component}

sig Component {dependencies: Dependencies, requiredServices: set
RequiredService, providedServices: set ProvidedService,
communicationPaths: set CommunicationPath, compositeComponents: set
Component, internalBehavior: InternalBehavior}

sig CommunicationPath {serviceProtocol: ServiceProtocol, connections:
set Connection}
```

The following description characterizes how *Topology Change* is formally specified:

- Through *Structural Signature*, *Topology Change* formally specifies a set of components and communication paths that must change
- A particular change type, through *Change Type*, is formally related to a set of system integrity characteristics. In this case, the set (only) includes *Global Consistency*, *Local Consistency* and *Active Reference Management*
- Each system integrity characteristic is formally related to the domain model element that it manages.

- The domain model elements that are realigned are formally specified as *Component* and *Communication Path*.

4.3.2. UML Modeling Level

This sub-section represents the domain-level concepts of *Topology Change* as a series of behavioral and structural UML models.

4.3.2.1. Activity Model

Based on Figure 22, Figure 54 is an Activity Model of the primary feature interactions of *Topology Change* shown in Figure 53.

The following sequence of actions are represented:

- Driven by *Architectural Change*, during the action '*Change Structural Signature*' a system responds by realigning *Component to Component* and *Required Service to Provided Service* dependencies.
- The decision point immediately after '*Change Structural Signature*' indicates this realignment may be rejected and lead to change rejection.
- Given a new *Structural Signature*, during the action '*Estimate Impact of Topology Change upon System Integrity*' a system evaluates the impact of proceeding with the change. For example, through a dynamic analysis a determination is made whether mandatory global or local system properties are compromised or not.
- The decision point immediately after '*Estimate Impact of Topology Change upon System Integrity*' indicates this impact analysis may lead to change rejection.

- If change is accepted, the parallel fork indicates '*Enact Topology Change by Realigning System Constructs*' and '*Ensure System Integrity Characteristics*' happens concurrently.
- During the action '*Enact Topology Change by Realigning System Constructs*' a system commits to a change to *Component and Communication Path* based on *Structural Signature*.
- During the action '*Ensure System Integrity Characteristics*' a system ensures the integrity of changing *Component* and *Communication Path* through *Active Reference Management*.
- The parallel join indicates a system has enacted *Topology Change* while attempting to ensure overall system consistency.
- During the action '*Provide Indication of Change*' a system indicates how *Components* and *Communication Paths* changed. *Topology Change* may lead to *Substitution* or *Protocol Change*.

Figure 54. Activity Model of Topology Change

4.3.2.2. Class Model

Based on the domain-level concepts, Figure 55 shows a Class Model of *Topology Change*. The model associates *Topology Change* with *System Signature* and with *Component* and *Communication Path* to indicate that, driven by a change to a system's *Structural Signature*, the affect of *Topology Change* is the additional and/or removal of *Components* and *Communication Paths*. *Active Reference Management* is shown associated with *Topology Change* and *Component* and *Communication Path* to indicate that it ensures the consistency of these model elements in the face of this kind of dynamic evolution.

Figure 55. Class Model of Topology Change

4.3.2.3. Sequence Model

Given the Activity Model, shown in Figure 54, and the Class Model shown in Figure 55, Figure 56 shows a Sequence Model of *Topology Change*.

The following interactions are represented:

- The action '*Change Structural Signature*' is represented by the message sequence *StructuralSignature::changeSignature()*.
- The first decision point is represented by the outer Alternative Interaction Frame.
- The action '*Estimate Impact of Topology Change on System Integrity*' is represented by the message sequence *TopologyChange::changeTopology()* and *TopologyChange::estimateImpact()*.
- The second decision point is represented by the inner Alternative Interaction Frame.
- The parallel fork is represented by the Parallel Interaction Frame.
- The action '*Enact Topology Change by Realigning System Constructs*' is represented by the message sequence *Component::addOrRemoveComponent()* and *CommunicationPath::changeCommunicationPath()* of the Parallel Interaction Frame.

- The action '*Ensure System Integrity Characteristics*' is represented by the message sequence *ActiveReferenceMangement::ensureActiveReferences()*,
Component::ensureComponent(),
CommunicationPath::ensureCommunicationPaths(),
StructuralSignature::ensureGlobalProperties(), and
StructuralSignature::ensureLocalProperties() of the Parallel Interaction Frame.
- The parallel join is represented by the completion of the Parallel Interaction Fragment.
- The action '*Provide Indication of Change*' is represented by the message sequence *ArchitecturalChange::indicationOfChange()*.
- The action '*Reject Topology Change*' is represented by the message sequence *ArchitecturalChange::rejectChange()*.

Figure 56. Sequence Model of Topology Change

4.3.2.4. State Machine Model

Given the Activity Model, shown in Figure 54, the Class Model, shown in Figure 55, and the Sequence Model, shown in Figure 56, Figure 57 shows a State Machine Model of *Topology Change*.

Figure 57. State Machine Model of Topology Change

The following states of *Topology Change* are represented:

- '*Initial*' is its state at the time of creation and initialization.
- '*Structural Signature Changed*' is its state after *TopologyChange::changeTopology()*.
- '*Impact Assessed*' is its state after *TopologyChange::estimateImpact()*.
- '*Change Rejected*' is its state before *ArchitecturalChange::rejectChange()*.
- The seven states leading to the state '*Change Indicated*' represents the combination of states possible based on the Parallel Interaction Frame shown in Figure 56. *Note:* For

readability reasons the states ‘*Global Consistency Assured*’ and ‘*Local Consistency Assured*’ are not shown.

- ‘Change Indicated’ is its state before `ArchitecturalChange::indicationOfChange()`.

4.4. Substitution Model

Based on an indirect affect upon a system’s *Structural Signature* due to *Architectural Change*, and driven-by *Topology Change*, *Substitution* affects *Component* and *Connection*. By affecting the *Structural Signature* of a system, *Global* and *Local Consistency Management* are interdependent with *Substitution*. *Active Reference Management* is interdependent with *Substitution* when ensuring the integrity of existing components and connections. *State Management* is interdependent with *Substitution* when ensuring the integrity of state elements. First, the domain level concepts that define *Substitution* are presented and then a UML representation of the domain level specification is formulated.

4.4.1 Domain Level

This sub-section presents the specific domain-level concepts that define *Substitution* first as features and then as Alloy relations.

4.4.1.1. Domain Concepts as Features

Figure 58 shows the primary feature interactions of *Substitution*. Reconciled *Change Properties* and *System Properties* through *Architectural Change* and then *Topology Change* indicate that *Substitution* is required to satisfy condition change criteria. *Substitution* in turn indicates the need for *Global Consistency Management*, *Local Consistency Management*, *Active Reference Management* and *State Management* system

integrity characteristics. Due to a realignment of *Structural Signature* and *Topology Change*, *Substitution* involves the realignment of *Component* and *Connection*.

Figure 58. Primary Feature Interactions of Substitution

4.4.1.2. Domain Concepts as Relations

Based on the domain analysis given in Chapter 3 and the primary feature interactions indicated in Figure 58, the following Alloy signatures and containment relations specify

Substitution:

```
sig StructuralSignature {ConnectedComponentDependencies: Component ->
Component, ConnectedServiceDependencies: RequireService ->
ProvidedService}
```

```
sig ChangeType {systemIntegrityCharacteristics: set
SystemIntegrityCharacteristic}
```

```
disj sig Substitution extends ChangeType {components: set Component,
communicationPaths: set CommunicationPath }
```

```
sig SystemIntegrityCharacteristic {}
```

```
disj sig GlobalConsistency extends SystemIntegrityCharacteristic
{globalProperties: set GlobalProperty}
```

```
disj sig LocalConsistency extends SystemIntegrityCharacteristic
{localProperties: set LocalProperty}
```

```
disj sig ActiveReference extends SystemIntegrityCharacteristic
{communicationPaths: set CommunicationPath, components: set Component}
```

```

disj sig ComponentState extends SystemIntegrityCharacteristic
{stateElements: set StateElement}

sig Component {dependencies: Dependencies, requiredServices: set
RequiredService, providedServices: set ProvidedService,
communicationPaths: set CommunicationPath, compositeComponents: set
Component, internalBehavior: InternalBehavior}

sig CommunicationPath {serviceProtocol: ServiceProtocol, connections:
set Connection}

```

The following description characterizes how *Substitution* is formally specified:

- Through *Structural Signature*, *Substitution* formally specifies a set of components and communication paths that must change
- A particular change type, through *Change Type*, is formally related to a set of system integrity characteristics. In this case, the set (only) includes *Global Consistency*, *Local Consistency*, *Active Reference* and *State Management*
- Each system integrity characteristic is formally related to the domain model element that it manages.
- The domain model elements that are realigned are formally specified as *Component* and *Communication Path*.

4.4.2. UML Modeling Level

This sub-section represents the domain-level concepts of *Substitution* as a series of behavioral and structural UML models.

4.4.2.1. Activity Model

Based on Figure 22, Figure 59 is an Activity Model of the primary feature interactions of *Substitution* shown in Figure 58.

Figure 59. Activity Model of Substitution

The following sequence of actions are represented:

- Driven by *Architectural Change* and *Topology Change*, during the action ‘*Change Structural Signature*’ a system responds by realigning *Component to Component* and *Required Service to Provided Service* dependencies.
- The decision point immediately after ‘*Change Structural Signature*’ indicates this realignment may be rejected and lead to change rejection.
- Given a new *Structural Signature*, during the action ‘*Estimate Impact of Substitution upon System Integrity*’ a system evaluates the impact of proceeding with the change. For example, through a dynamic analysis a determination is made whether mandatory global or local system properties are compromised or not.

- The decision point immediately after '*Estimate Impact of Substitution upon System Integrity*' indicates this impact analysis may lead to change rejection.
- If change is accepted, the parallel fork indicates '*Enact Substitution by Realigning System Constructs*' and '*Ensure System Integrity Characteristics*' happens concurrently.
- During the action '*Enact Substitution by Realigning System Constructs*' a system commits to a change to *Component and Connection* based on *Structural Signature*.
- During the action '*Ensure System Integrity Characteristics*' a system ensures the integrity of changing *Component* and *Connection* through *Active Reference Management and State Management*.
- The parallel join indicates a system has enacted *Substitution* while attempting to ensure overall system consistency.
- During the action '*Provide Indication of Change*' a system indicates how *Components* and *Connections* changed.

4.4.2.2. Class Model

Based on the domain-level concepts, Figure 60 shows a Class Model of *Substitution*. Similar to *Topology Change*, the model associates *Substitution* with *System Signature* and with *Component* and *Communication Path* to indicate that, driven by a change to a system's *Structural Signature*, the affect of *Substitution* is the replacement of one component for another while maintaining existing connections. *Active Reference Management* is shown associated with *Substitution* and *Component* and *Communication Path* to indicate that it ensures the consistency of these model elements in the face of this kind of dynamic evolution. *State Management* is shown associated with *Substitution* and

Component to indicate that it ensures the consistency of *State Elements* during *Substitution*.

Figure 60. Class Model of Substitution

4.4.2.3. Sequence Model

Given the Activity Model, shown in Figure 59, and the Class Model shown in Figure 60, Figure 61 shows a Sequence Model of *Substitution*.

The following interactions are represented:

- The action '*Change Structural Signature*' is represented by the message sequence *StructuralSignature::changeSignature()*.
- The first decision point is represented by the outer Alternative Interaction Frame.
- The action '*Estimate Impact of Topology Change on System Integrity*' is represented by the message sequence *Substitution::substituteComponent()* and *Substitution::estimateImpact()*.
- The second decision point is represented by the inner Alternative Interaction Frame.

- The parallel fork is represented by the Parallel Interaction Frame.
- The action ‘*Enact Substitution by Realigning System Constructs*’ is represented by the message sequence *Component::substituteComponent()* and *Connection::changeConnection()* of the Parallel Interaction Frame.
- The action ‘*Ensure System Integrity Characteristics*’ is represented by the message sequence *ActiveReferenceMangement::ensureActiveReferences()*, *Component::ensureComponent()*, and *CommunicationPath::ensureCommunicationPaths()* and *StateManagement::ensureComponentState()* and *Component::ensureState()* and *StructuralSignature::ensureGlobalProperties()* and *StructuralSignature::ensureLocalProperties()* of the Parallel Interaction Frame.
- The parallel join is represented by the completion of the Parallel Interaction Fragment.
- The action ‘*Provide Indication of Change*’ is represented by the message sequence *ArchitecturalChange::indicationOfChange()*.
- The action ‘*Reject Substitution*’ is represented by the message sequence *ArchitecturalChange::rejectChange()*.

Figure 61. Sequence Model of Substitution

4.4.2.4. State Machine Model

Given the Activity Model, shown in Figure 59, the Class Model, shown in Figure 60, and the Sequence Model, shown in Figure 61, Figure 62 shows a State Machine Model of *Substitution*.

The following states of *Substitution* are represented:

- '*Initial*' is its state at the time of creation and initialization.
- '*Structural Signature Changed*' is its state after *Substitution::substituteComponent()*.

- ‘*Impact Assessed*’ is its state after *Substitution::estimateImpact()*.
- ‘*Change Rejected*’ is its state before *ArchitecturalChange::rejectChange()*.
- The fifteen states leading to the state ‘*Change Indicated*’ represents the combination of states possible based on the Parallel Interaction Frame shown in Figure 62. *Note:* For readability reasons the states ‘*Global Consistency Assured*’ and ‘*Local Consistency Assured*’ are not shown.
- ‘*Change Indicated*’ is its state before *ArchitecturalChange::indicationOfChange()*.

Figure 62. State Machine Model of Substitution

4.5. Protocol Change Model

Based on an indirect affect upon a system's *Behavioral Signature* due to *Architectural Change*, *Protocol Change* affects *Service Protocol*. By affecting the *Behavioral Signature* of a system, *Global* and *Local Consistency Management* are interdependent with *Protocol Change*. First, the domain level concepts that define *Protocol Change* are presented and then a UML representation of the domain level specification is formulated.

4.5.1. Domain Level

This sub-section presents the specific domain-level concepts that define *Protocol Change* first as features and then as Alloy relations.

4.5.1.1. Domain Concepts as Features

Figure 63 shows the primary feature interactions of *Protocol Change*. Reconciled *Change Properties* and *System Properties* through *Architectural Change* indicate that *Protocol Change* is required to satisfy condition change criteria. *Protocol Change* in turn indicates the need for *Global Consistency* and *Local Consistency* system integrity characteristics. Due to a realignment of *Behavioral Signature*, *Protocol Change* involves the realignment of *Service Protocol*.

Figure 63. Primary Feature Interactions of Protocol Change

4.5.1.2. Domain Concepts as Relations

Based on the domain analysis given in Chapter 3 and the primary feature interactions indicated in Figure 63, the following Alloy signatures and containment relations specify

Protocol Change:

```

sig BehavioralSignature {ProtocolDependencies: Service ->
ServiceProtocol, RequiredServiceDependencies: Operation ->
RequiredService, ProvidedServiceDependencies: Operation ->
ProvidedService, OperationDependencies: Operation -> Operation,
StateElementDependencies: Operation -> StateElement,
CompositeComponentDependencies: Operation -> CompositeComponentService}

sig ChangeType {systemIntegrityCharacteristics: set
SystemIntegrityCharacteristic}

disj sig ProtocolChange extends ChangeType {serviceProtocols: set
ServiceProtocol}

sig SystemIntegrityCharacteristic {}

disj sig GlobalConsistency extends SystemIntegrityCharacteristic
{globalProperties: set GlobalProperty}

disj sig LocalConsistency extends SystemIntegrityCharacteristic
{localProperties: set LocalProperty}

sig ServiceProtocol {}

```

The following description characterizes how *Protocol Change* is formally specified:

- Through *Behavioral Signature*, *Protocol Change* formally specifies a set of service protocols that must change
- A particular change type, through *Change Type*, is formally related to a set of system integrity characteristics. In this case, the set (only) includes *Global Consistency* and *Local Consistency Management*
- Each system integrity characteristic is formally related to the domain model element that it manages.
- The domain model elements that are realigned are formally specified as *Service Protocol*.

4.5.2. UML Modeling Level

This sub-section represents the domain-level concepts of *Protocol Change* as a series of behavioral and structural UML models.

4.5.2.1. Activity Model

Based on Figure 22, Figure 64 is an Activity Model of the primary feature interactions of *Protocol Change* shown in Figure 63.

Figure 64. Activity Model of Protocol Change

The following sequence of actions are represented:

- Driven by *Architectural Change*, during the action '*Change Behavioral Signature*' a system responds by realigning *Service to Service Protocol* dependencies.
- The decision point immediately after '*Change Behavioral Signature*' indicates this realignment may be rejected and lead to change rejection.
- Given a new *Behavioral Signature*, during the action '*Estimate Impact of Protocol Change upon System Integrity*' a system evaluates the impact of proceeding with the change. For example, through a dynamic analysis a determination is made whether mandatory global or local system properties are compromised or not.

- The decision point immediately after '*Estimate Impact of Protocol Change upon System Integrity*' indicates this impact analysis may lead to change rejection.
- If change is accepted, the parallel fork indicates '*Enact Protocol Change by Realigning System Constructs*' and '*Ensure System Integrity Characteristics*' happens concurrently.
- During the action '*Enact Protocol Change by Realigning System Constructs*' a system commits to a change to *Service Protocol* based on *Behavioral Signature*.
- During the action '*Ensure System Integrity Characteristics*' a system ensures the integrity of changing *Service Protocol* through *Global* and *Local Consistency Management*.
- The parallel join indicates a system has enacted *Protocol Change* while attempting to ensure overall system consistency.
- During the action '*Provide Indication of Change*' a system indicates how *Service Protocols* changed. *Protocol Change* may lead to *Topology Change* or *Interface Change*.

4.5.2.2. Class Model

Based on the domain-level concepts, Figure 65 shows a Class Model of *Protocol Change*.

Figure 65. Class Model of Protocol Change

The model associates *Protocol Change* with *System Signature* and with *Communication Path* to indicate that, driven by a change to a system's *Behavioral Signature*, the affect of *Protocol Change* is a change to the *Service Protocols* of the *Communication Paths* of the system. *Global* and *Local Consistency* is managed through the system's *Behavioral Signature*.

4.5.2.3. Sequence Model

Given the Activity Model, shown in Figure 64, and the Class Model shown in Figure 65, Figure 66 shows a Sequence Model of *Protocol Change*.

Figure 66. Sequence Model of Protocol Change

The following interactions are represented:

- The action ‘*Change Behavioral Signature*’ is represented by the message sequence *BehavioralSignature::changeSignature()*.
- The first decision point is represented by the outer Alternative Interaction Frame.
- The action ‘*Estimate Impact of Protocol Change on System Integrity*’ is represented by the message sequence *ProtocolChange::changeProtocol()* and *ProtocolChange::estimateImpact()*.
- The second decision point is represented by the inner Alternative Interaction Frame.
- The parallel fork is represented by the Parallel Interaction Frame.
- The action ‘*Enact Protocol Change by Realigning System Constructs*’ is represented by the message sequence *CommunicationPath::changeServiceProtocol()* of the Parallel Interaction Frame.
- The action ‘*Ensure System Integrity Characteristics*’ is represented by the message sequence *BehavioralSignature::ensureGlobalProperties()* and *BehavioralSignature::ensureLocalProperties()* of the Parallel Interaction Frame.
- The parallel join is represented by the completion of the Parallel Interaction Fragment.
- The action ‘*Provide Indication of Change*’ is represented by the message sequence *ArchitecturalChange::indicationOfChange()*.
- The action ‘*Reject Protocol Change*’ is represented by the message sequence *ArchitecturalChange::rejectChange()*.

4.5.2.4. State Machine Model

Given the Activity Model, shown in Figure 64, the Class Model, shown in Figure 65, and the Sequence Model, shown in Figure 66, Figure 67 shows a State Machine Model of *Protocol Change*.

Figure 67. State Machine Model of Protocol Change

The following states of *Protocol Change* are represented:

- '*Initial*' is its state at the time of creation and initialization.

- ‘*Behavioral Signature Changed*’ is its state after *ProtocolChange::changeProtocol()*.
- ‘*Impact Assessed*’ is its state after *ProtocolChange::estimateImpact()*.
- ‘*Change Rejected*’ is its state before *ArchitecturalChange::rejectChange()*.
- The seven states leading to the state ‘*Change Indicated*’ represents the combination of states possible based on the Parallel Interaction Frame shown in Figure 66.
- ‘*Change Indicated*’ is its state before *ArchitecturalChange::indicationOfChange()*.

4.6. Interface Change Model

Based on an indirect affect upon a system’s *Behavioral Signature* due to *Architectural Change*, and driven-by *Protocol Change*, *Interface Change* affects *Required Service* and *Provided Service*. By affecting the *Behavioral Signature* of a system, *Global* and *Local Consistency Management* are interdependent with *Interface Change*. First, the domain level concepts that define *Interface Change* are presented and then a UML representation of the domain level specification is formulated.

4.6.1. Domain Level

This sub-section presents the specific domain-level concepts that define *Interface Change* first as features and then as Alloy relations.

4.6.1.1. Domain Concepts as Features

Figure 68 shows the primary feature interactions of *Interface Change*. Reconciled *Change Properties* and *System Properties* through *Architectural Change* and then *Protocol Change* indicate that *Interface Change* is required to satisfy condition change criteria. *Interface Change* in turn indicates the need for *Global Consistency* and *Local Consistency* system integrity characteristics. Due to a realignment of *Behavioral*

Signature and Protocol Change, Interface Change involves the realignment of *Required Service* and *Provided Service*.

Figure 68. Primary Feature Interactions of Interface Change

4.6.1.2. Domain Concepts as Relations

Based on the domain analysis given in Chapter 3 and the primary feature interactions indicated in Figure 68, the following Alloy signatures and containment relations specify

Interface Change:

```

sig BehavioralSignature {ProtocolDependencies: Service ->
ServiceProtocol, RequiredServiceDependencies: Operation ->
RequiredService, ProvidedServiceDependencies: Operation ->
ProvidedService, OperationDependencies: Operation -> Operation,
StateElementDependencies: Operation -> StateElement,
CompositeComponentDependencies: Operation -> CompositeComponentService}

sig ChangeType {systemIntegrityCharacteristics: set
SystemIntegrityCharacteristic}

disj sig InterfaceChange extends ChangeType {services: set Service}

sig SystemIntegrityCharacteristic {}

disj sig GlobalConsistency extends SystemIntegrityCharacteristic
{globalProperties: set GlobalProperty}

disj sig LocalConsistency extends SystemIntegrityCharacteristic
{localProperties: set LocalProperty}

```

```
sig Service {serviceProtocol: ServiceProtocol, implementation:
PublicOperation}
```

```
disj sig RequiredService, ProvidedService extends Service {}
```

The following description characterizes how *Interface Change* is formally specified:

- Through *Behavioral Signature*, *Interface Change* formally specifies a set of services that must change
- A particular change type, through *Change Type*, is formally related to a set of system integrity characteristics. In this case, the set (only) includes *Global Consistency* and *Local Consistency Management*
- Each system integrity characteristic is formally related to the domain model element that it manages.
- The domain model elements that are realigned are formally specified as *Required Service* and *Provided Service*.

4.6.2. UML Modeling Level

This sub-section represents the domain-level concepts of *Interface Change* as a series of behavioral and structural UML models.

4.6.2.1. Activity Model

Based on Figure 22, Figure 69 is an Activity Model of the primary feature interactions of *Interface Change* shown in Figure 68.

Figure 69. Activity Model of Interface Change

The following sequence of actions are represented:

- Driven by *Architectural Change* and *Protocol Change*, during the action '*Change Behavioral Signature*' a system responds by realigning *Operation to Provided Service* and *Operation to Required Service* dependencies.
- The decision point immediately after '*Change Behavioral Signature*' indicates this realignment may be rejected and lead to change rejection.
- Given a new *Behavioral Signature*, during the action '*Estimate Impact of Interface Change upon System Integrity*' a system evaluates the impact of proceeding with the

change. For example, through a dynamic analysis a determination is made whether mandatory global or local system properties are compromised or not.

- The decision point immediately after '*Estimate Impact of Interface Change upon System Integrity*' indicates this impact analysis may lead to change rejection.
- If change is accepted, the parallel fork indicates '*Enact Interface Change by Realigning System Constructs*' and '*Ensure System Integrity Characteristics*' happens concurrently.
- During the action '*Enact Interface Change by Realigning System Constructs*' a system commits to a change to *Required Service* and *Provided Service* based on *Behavioral Signature*.
- During the action '*Ensure System Integrity Characteristics*' a system ensures the integrity of changing *Required Service* and *Provided Service* through *Global* and *Local Consistency Management*.
- The parallel join indicates a system has enacted *Interface Change* while attempting to ensure overall system consistency.
- During the action '*Provide Indication of Change*' a system indicates how *Required* or *Provided Services* changed. *Interface Change* may lead to *Internal Change*.

4.6.2.2. Class Model

Based on the domain-level concepts, Figure 70 shows a Class Model of *Interface Change*.

Figure 70. Class Model of Interface Change

The model associates *Interface Change* with *System Signature* and with *Component* to indicate that, driven by a change to a system's *Behavioral Signature*, the affect of *Interface Change* is a change to the *Provided Services* or *Required Services* of the *Components* of the system. Again, *Global* and *Local Consistency* is managed through the system's *Behavioral Signature*.

4.6.2.3. Sequence Model

Given the Activity Model, shown in Figure 69, and the Class Model shown in Figure 70, Figure 71 shows a Sequence Model of *Interface Change*.

The following interactions are represented:

- The action '*Change Behavioral Signature*' is represented by the message sequence *BehavioralSignature::changeSignature()*.
- The first decision point is represented by the outer Alternative Interaction Frame.
- The action '*Estimate Impact of Interface Change on System Integrity*' is represented by the message sequence *InterfaceChange::changeInterface()* and *InterfaceChange::estimateImpact()*.
- The second decision point is represented by the inner Alternative Interaction Frame.
- The parallel fork is represented by the Parallel Interaction Frame.

- The action '*Enact Interface Change by Realigning System Constructs*' is represented by the message sequence *Component::changeService()* of the Parallel Interaction Frame.
- The action '*Ensure System Integrity Characteristics*' is represented by the message sequence *BehavioralSignature::ensureGlobalProperties()* and *BehavioralSignature::ensureLocalProperties()* of the Parallel Interaction Frame.
- The parallel join is represented by the completion of the Parallel Interaction Fragment.
- The action '*Provide Indication of Change*' is represented by the message sequence *ArchitecturalChange::indicationOfChange()*.
- The action '*Reject Interface Change*' is represented by the message sequence *ArchitecturalChange::rejectChange()*.

Figure 71. Sequence Model of Interface Change

4.6.2.4. State Machine Model

Given the Activity Model, shown in Figure 69, the Class Model, shown in Figure 70, and the Sequence Model, shown in Figure 71, Figure 72 shows a State Machine Model of *Interface Change*.

Figure 72. State Machine model of Interface Change

The following states of *Interface Change* are represented:

- ‘*Initial*’ is its state at the time of creation and initialization.
- ‘*Behavioral Signature Changed*’ is its state after *InterfaceChange::changeInterface()*.
- ‘*Impact Assessed*’ is its state after *InterfaceChange::estimateImpact()*.
- ‘*Change Rejected*’ is its state before *ArchitecturalChange::rejectChange()*.
- The 15 states leading to the state ‘*Change Indicated*’ represents the combination of states possible based on the Parallel Interaction Frame shown in Figure 71.
- ‘*Change Indicated*’ is its state before *ArchitecturalChange::indicationOfChange()*.

4.7. Internal Change Model

Based on an indirect affect upon a system's *Behavioral Signature* due to *Architectural Change*, and driven-by *Interface Change*, *Internal Change* affects *Operation*, *State Element* and *Service Invocation*. By affecting the *Behavioral Signature* of a system, *Global* and *Local Consistency Management* are interdependent with *Internal Change*. *Dependent Operation Management* is interdependent with *Internal Change* when ensuring the integrity of operations. *Composite Component Management* is interdependent with *Internal Change* when ensuring the integrity of composite component operations. *Constrained Operation Management* is interdependent with *Internal Change* when ensuring the integrity of operations and state elements. *State Management* is interdependent with *Internal Change* when ensuring the integrity of state elements. First, the domain level concepts that define *Internal Change* are presented and then a UML representation of the domain level specification is formulated.

4.7.1. Domain Level

This sub-section presents the specific domain-level concepts that define *Internal Change* first as features and then as Alloy relations.

4.7.1.1. Domain Concepts as Features

Figure 73 shows the primary feature interactions of *Internal Change*. Reconciled *Change Properties* and *System Properties* through *Architectural Change* and then *Interface Change* indicate that *Internal Change* is required to satisfy condition change criteria. *Internal Change* in turn indicates the need for *Global Consistency*, *Local Consistency*, *Dependent Operation Management*, *Composite Component Management*, *Constrained Operation Management* and *State Management* system integrity

characteristics. Due to a realignment of *Behavioral Signature* and *Interface Change*, *Internal Change* involves the realignment of *Operation*, *State Element* and *Service Invocation*.

Figure 73. Primary Feature Interactions of Internal Change

4.7.1.2. Domain Concepts as Relations

Based on the domain analysis given in Chapter 3 and the primary feature interactions indicated in Figure 73, the following Alloy signatures and containment relations specify *Internal Change*:

```
sig BehavioralSignature {ProtocolDependencies: Service ->
ServiceProtocol, RequiredServiceDependencies: Operation ->
RequiredService, ProvidedServiceDependencies: Operation ->
ProvidedService, OperationDependencies: Operation -> Operation,
StateElementDependencies: Operation -> StateElement,
CompositeComponentDependencies: Operation -> CompositeComponentService}

sig ChangeType {systemIntegrityCharacteristics: set
SystemIntegrityCharacteristic}

disj sig InternalChange extends ChangeType {operations: set Operation,
stateElements: set StateElement, serviceInvocations: set
ServiceInvocation}

sig SystemIntegrityCharacteristic {}

disj sig GlobalConsistency extends SystemIntegrityCharacteristic
{globalProperties: set GlobalProperty}
```



```

disj sig LocalConsistency extends SystemIntegrityCharacteristic
{localProperties: set LocalProperty}

disj sig DependentOperation extends SystemIntegrityCharacteristic
{dependentOperations: set Operation}

disj sig CompositeComponent extends SystemIntegrityCharacteristic
{compositeOperations: set Operation}

disj sig ConstrainedOperation extends SystemIntegrityCharacteristic
{constrainedOperations: set Operation, constrainedStateElements: set
StateElement}

disj sig ComponentState extends SystemIntegrityCharacteristic
{stateElements: set StateElement}

sig StateElement {}

disj sig PermanentStateElement extends StateElement {operations: set
Operation}

disj sig TemporaryStateElement extends StateElement {localOperation:
Operation}

sig Operation {operations: set Operation, compositeComponentServices:
set Service, temporaryStateElements: set TemporaryStateElement,
permanentStateElements: set PermanentStateElement}

disj sig PublicOperation extends Operation {ExternalService: Service}

disj sig PrivateOperation extends Operation {}

sig ServiceInvocation {operations: set Operation}

```

The following description characterizes how *Internal Change* is formally specified:

- Through *Behavioral Signature*, *Internal Change* formally specifies a set of operations, state elements, and service invocations that must change
- A particular change type, through *Change Type*, is formally related to a set of system integrity characteristics. In this case, the set (only) includes *Global Consistency*, *Local Consistency*, *Dependent Operation*, *Composite Component*, *Constrained Operations*, and *State Management*
- Each system integrity characteristic is formally related to the domain model element that it manages.

- The domain model elements that are realigned are formally specified as *Operation*, *State Element* and *Service Invocation*.

4.7.2. UML Modeling Level

This sub-section represents the domain-level concepts of *Internal Change* as a series of behavioral and structural UML models.

4.7.2.1. Activity Model

Based on Figure 22, Figure 74 is an Activity Model of the primary feature interactions of *Internal Change* shown in Figure 73.

Figure 74. Activity Model of Internal Change

The following sequence of actions are represented:

- Driven by *Architectural Change* and *Interface Change*, during the action '*Change Behavioral Signature*' a system responds by realigning *Operation to Operation*, *Operation to Composite Component Service* and *Operation to State Element* dependencies.
- The decision point immediately after '*Change Behavioral Signature*' indicates this realignment may be rejected and lead to change rejection.
- Given a new *Behavioral Signature*, during the action '*Estimate Impact of Internal Change upon System Integrity*' a system evaluates the impact of proceeding with the change. For example, through a dynamic analysis a determination is made whether mandatory global or local system properties are compromised or not.
- The decision point immediately after '*Estimate Impact of Internal Change upon System Integrity*' indicates this impact analysis may lead to change rejection.
- If change is accepted, the parallel fork indicates '*Enact Internal Change by Realigning System Constructs*' and '*Ensure System Integrity Characteristics*' happens concurrently.
- During the action '*Enact Internal Change by Realigning System Constructs*' a system commits to a change to *Operation*, *State Element* and *Service Invocation* based on *Behavioral Signature*.
- During the action '*Ensure System Integrity Characteristics*' a system ensures the integrity of changing *Operation*, *State Element* and *Service Invocation* through *Dependent Operation*, *Composite Component*, *Constrained Operation* and *State Management*.

- The parallel join indicates a system has enacted *Internal Change* while attempting to ensure overall system consistency.
- During the action '*Provide Indication of Change*' a system indicates how *Operations*, *State Elements* or *Service Invocations* changed.

4.7.2.2. Class Model

Based on the domain-level concepts, Figure 75 shows a Class Model of *Internal Change*.

Figure 75. Class Model of Internal Change

The model associates *Internal Change* with *System Signature* and with *Internal Behavior* to indicate that, driven by a change to a system's *Behavioral Signature*, the affect of *Internal Change* is a change to *Operations*, *State Elements*, and/or *Service Invocations*. *Dependent Operation Management* is shown associated with *Internal Change* and *Internal Behavior* to indicate that it ensures the consistency of *Operations* in

the face of this kind of dynamic evolution. *Composite Component Management* is shown associated with *Internal Change* and *Internal Behavior* to indicate that it ensures the consistency of *Operations* that rely upon the services provided by composite components in the face of this kind of change. *Constrained Operation Management* is shown associated with *Internal Change* and *Internal Behavior* to indicate that it ensures the consistency of *State Element* constrained *Operations* in the face of this kind of dynamic evolution. *State Management* is shown associated with *Internal Change* and *Internal Behavior* to indicate that it ensures the consistency of *State Elements* in the face of this change.

4.7.2.3. Sequence Model

Given the Activity Model, shown in Figure 74, and the Class Model shown in Figure 75, Figure 76 shows a Sequence Model of *Internal Change*.

The following interactions are represented:

- The action '*Change Behavioral Signature*' is represented by the message sequence *BehavioralSignature::changeSignature()*.
- The first decision point is represented by the outer Alternative Interaction Frame.
- The action '*Estimate Impact of Internal Change on System Integrity*' is represented by the message sequence *InternalChange::changeImplementation()* and *InternalChange::estimateImpact()*.
- The second decision point is represented by the inner Alternative Interaction Frame.
- The parallel fork is represented by the Parallel Interaction Frame.

- The action '*Enact Internal Change by Realigning System Constructs*' is represented by the message sequence *InternalBehavior::changeImplementation()* of the Parallel Interaction Frame.
- The action '*Ensure System Integrity Characteristics*' is represented by the message sequence *DependentOperationMangement::ensureOperations()*, *CompositeComponentManagment::ensureCompositeComponentServices*, *ConstrainedOperationManagement::ensureConstrainedOperations()*, *StateManagement::ensureStateElements()*, *BehavioralSignature::ensureGlobalProperties()*, and *BehavioralSignature::ensureLocalProperties()* of the Parallel Interaction Frame.
- The parallel join is represented by the completion of the Parallel Interaction Fragment.
- The action '*Provide Indication of Change*' is represented by the message sequence *ArchitecturalChange::indicationOfChange()*.
- The action '*Reject Internal Change*' is represented by the message sequence *ArchitecturalChange::rejectChange()*.

Figure 76. Sequence Model of Internal Change

4.7.2.4. State Machine Model

Given the Activity Model, shown in Figure 74, the Class Model, shown in Figure 75, and the Sequence Model, shown in Figure 76, Figure 77 shows a State Machine Model of *Internal Change*.

The following states of *Internal Change* are represented:

- ‘*Initial*’ is its state at the time of creation and initialization.
- ‘*Behavioral Signature Changed*’ is its state after *InternalChange::changeImplementation()*.
- ‘*Impact Assessed*’ is its state after *InternalChange::estimateImpact()*.
- ‘*Change Rejected*’ is its state before *ArchitecturalChange::rejectChange()*.

- The thirty-one states leading to the state '*Change Indicated*' represents the combination of states possible based on the Parallel Interaction Frame shown in Figure 76. *Note:* For readability reasons the states '*Global Consistency Assured*' and '*Local Consistency Assured*' are not shown.
- 'Change Indicated' is its state before `ArchitecturalChange::indicationOfChange()`

Figure 77. State Machine of Internal Change

4.8. Composed Models

In this section, using the common and variable causal flow of dynamic reconfiguration described in Chapter 3, like-UML models of the six change types are composed as representations of each of the nine change groups.

4.8.1. Composed Models of Pure Architectural Change

Pure architectural change is a change group that consists of a single change sequence involving a single change type: *architectural change*. In its purest form, this is a change to only the global and/or local system *properties*. By implication, the existing behavioral and structural signatures of the system can accommodate the change and are unaffected.

This sub-section presents composed Activity, Class, Sequence, and State Machine models of *Pure Architectural Change*. Because this change group is composed of a single change type, there are no cross-coupling effects that occur because of the additive composition of other models.

4.8.1.1. Composed Activity Model of Pure Architectural Change

Figure 78 shows a composed Activity Model of *Pure Architectural Change*.

Figure 78. Activity Model of Pure Architectural Change

The following sequence of actions are represented:

- Because *Pure Architectural Change* is composed of just a single change type, *Architectural Change*, the action ‘*Architectural Change*’ is composed as a sub activity diagram based on Figure 49.

4.8.1.2. Composed Class Model of Pure Architectural Change

Figure 79 shows a composed Class Model of *Pure Architectural Change*.

Figure 79. Class Model of Pure Architectural Change

Because *Pure Architectural Change* is composed of just a single change type, *Architectural Change*, its composed Class Model, shown in Figure 79, is the same as the model shown in Figure 50.

4.8.1.3. Composed Sequence Model of Pure Architectural Change

Figure 80 shows a composed Sequence Model of *Pure Architectural Change*.

Figure 80. Sequence Model of Pure Architectural Change

The following interactions are represented:

- Because *Pure Architectural Change* is composed of just a single change type, *Architectural Change*, its composed Sequence Model, shown in Figure 80, is the same as the model shown in Figure 51.

4.8.1.4. Composed State Machine Model of Pure Architectural Change

Figure 81 shows a composed State Machine Model of *Pure Architectural Change*.

Figure 81. State Machine Model of Pure Architectural Change

The following states of *Internal Change* are represented:

- Because *Pure Architectural Change* is composed of just a single change type, *Architectural Change*, its composed State Machine Model, shown in Figure 81, is the same as the model shown in Figure 52.

4.8.2. Composed Models of Pure Behavioral Change

Pure behavioral change is a change group that encompasses three change sequences, each driven by *architectural change*: (i) *protocol change* only, (ii) *protocol change* leading to *interface change*, or (iii) *protocol change* leading to *interface change* leading to *internal change*. By implication, *pure behavioral change* means that only the behavioral signature of a system is realigned.

This sub-section presents composed Activity, Class, Sequence, and State Machine models of *Pure Behavioral Change*. Because this change group is composed of the sequential application of other change type models, there are no cross-coupling effects that occur because of the additive composition of other models.

4.8.2.1. Composed Activity Model of Pure Behavioral Change

Figure 82 shows a composed Activity Model of *Pure Behavioral Change*.

Figure 82. Activity Model of Pure Behavioral Change

The following sequence of actions are represented:

- The action '*Architectural Change*' is composed as a sub activity diagram based on Figure 49.
- The action '*Protocol Change*' is composed as a sub activity diagram based on Figure 64.
- The action '*Interface Change*' is composed as a sub activity diagram based on Figure 69.
- The action '*Internal Change*' is composed as a sub activity diagram based on Figure 74.

4.8.2.2. Composed Class Model of Pure Behavioral Change

Figure 83 shows a composed Class Model of *Pure Behavioral Change*. It is a composition of the class models of *Architectural Change* (See Figure 50), *Protocol Change* (See Figure 65), *Interface Change* (See Figure 70), and *Internal Change* (See Figure 75).

Figure 83. Class Model of Pure Behavioral Change

4.8.2.3. Composed Sequence Model of Pure Behavioral Change

Figure 84 shows a composed Sequence Model of *Pure Behavioral Change*. It is a composition of the sequence models of *Architectural Change* (See Figure 51), *Protocol Change* (See Figure 66), *Interface Change* (See Figure 71), and *Internal Change* (See Figure 76).

Figure 84. Sequence Model of Pure Behavioral Change

4.8.2.4. Composed State Machine Model of Pure Behavioral Change

Figure 85 is a composed State Machine Model of *Pure Behavioral Change*. It is a composition of the state machine models of *Architectural Change* (See Figure 52), *Protocol Change* (See Figure 67), *Interface Change* (See Figure 72), and *Internal Change* (See Figure 77).

Figure 85. State Machine Model of Pure Behavioral Change

4.8.3. Composed Models of Behavior-driven Structural Change

Behavior-driven structural change is a change group that comprises two change sequences, each driven by *architectural change*: (i) *protocol change* leading to *topology change* or (ii) *protocol change* leading to *topology change* leading to *substitution*. By implication, *behavior-driven structural change* means that, when the behavioral signature of a system is realigned, it causes the structural signature of the system to be realigned.

This sub-section presents composed Activity, Class, Sequence, and State Machine models of *Behavior-driven Structural Change*. Because this change group is composed of the sequential application of other change type models, there are no cross-coupling effects that occur because of the additive composition of other models.

4.8.3.1. Composed Activity Model of Behavior-driven Structural Change

Figure 86 shows a composed Activity Model of *Behavior-driven Structural Change*.

The following sequence of actions are represented:

- The action '*Architectural Change*' is composed as a sub activity diagram based on Figure 49.
- The action '*Protocol Change*' is composed as a sub activity diagram based on Figure 64.
- The action '*Topology Change*' is composed as a sub activity diagram based on Figure 54.
- The action '*Substitution*' is composed as a sub activity diagram based on Figure 59.

Figure 86. Activity Model of Behavioral-Driven Structural Change

4.8.3.2. Composed Class Model of Behavior-driven Structural Change

Figure 87 shows a composed Class Model of *Behavioral-driven Structural Change*. It is a composition of the class models of *Architectural Change* (See Figure 50), *Protocol Change* (See Figure 65), *Topology Change* (See Figure 55), and *Substitution* (See Figure 60).

Figure 87. Class Model of Behavior-driven Structural Change

4.8.3.3. Composed Sequence Model of Behavior-driven Structural Change

Figure 88 shows a composed Sequence Model of *Behavioral-Driven Structural Change*. It is a composition of the sequence models of *Architectural Change* (See Figure 51), *Protocol Change* (See Figure 66), *Topology Change* (See Figure 56), and *Substitution* (See Figure 61).

Figure 88. Sequence Model of Behavior-Driven Structural Change

4.8.3.4. Composed State Machine Model of Behavior-driven Structural Change

Figure 89 is a composed State Machine Model of *Behavior-Driven Structural Change*. It is a composition of the state machine models of *Architectural Change* (See Figure 52), *Protocol Change* (See Figure 67), *Topology Change* (See Figure 57), and *Substitution* (See Figure 62).

Figure 89. State Machine Model of Behavior-Driven Structural Change

4.8.4. Composed Models of Comprehensive Behavior-driven Change

Comprehensive behavior-driven change is a change group that is a combination of *pure behavioral change* and *behavior-driven structural change*. By implication, *comprehensive behavior-driven change* means that, when the behavioral signature of a system is realigned, it causes the structural signature of the system to be realigned. All combinations are possible as long as a behavioral change leads to a structural change.

This sub-section presents composed Activity, Class, Sequence, and State Machine models of *Comprehensive Behavior-driven Change*. Because this change group is composed of two other change groups which occur simultaneously, there can be cross-coupling effects because of the additive composition of the other two models. For example, *Architectural Change* and *Protocol Change* are in common at the beginning of both composed change groups and could therefore be unified in their application. Depending on the context, *Topology Change* and *Interface Change* could apply to the same component. When this occurs they could be serialized to aid the management of *Comprehensive Behavior-driven Change*.

4.8.4.1. Composed Activity Model of Comprehensive Behavior-driven Change

Figure 90 shows a composed Activity Model of *Comprehensive Behavior-driven Change*.

The following sequence of actions are represented:

- The action '*Pure Behavioral Change*' is composed as a sub activity diagram based on Figure 82.
- The action '*Behavior-Driven Structural Change*' is composed as a sub activity diagram based on Figure 86.

Figure 90. Activity Model of Comprehensive Behavior-Driven Change

4.8.4.2. Composed Class Model of Comprehensive Behavior-driven Change

Figure 91 shows a composed Class Model of Comprehensive *Behavioral-driven Change*. It is a composition of the class models of *Pure Behavioral Change* (See Figure 83) and *Behavior-Driven Structural Change* (See Figure 87).

Figure 91. Class Model of Comprehensive Behavior-driven Structural Change

4.8.4.3. Composed Sequence Model of Comprehensive Behavior-driven Change

Figure 92 shows a composed Sequence Model of *Comprehensive Behavioral-Driven Change*. It is a composition of the sequence models of *Pure Behavioral Change* (See Figure 84) and *Behavior-Driven Structural Change* (See Figure 88).

Figure 92. Sequence Model of Comprehensive Behavior-Driven Change

4.8.4.4. Composed State Machine Model of Comprehensive Behavior-driven Change

Figure 93 is a composed State Machine Model of *Comprehensive Behavior-Driven Change*. It is a composition of the state machine models of *Pure Behavioral Change* (See Figure 85) and *Behavior-Driven Structural Change* (See Figure 89).

Figure 93. State Machine Model of Comprehensive Behavior-Driven Change

4.8.5. Compose Models of Pure Structural Change

Pure structural change is a change group that encompasses two change sequences, that are each driven by *architectural change*: (i) *topology change* only or (ii) *topology change* leading to *substitution*. By implication, *pure structural change* means the structural signature, and only the structural signature, of a system is realigned.

This sub-section presents composed Activity, Class, Sequence, and State Machine models of *Pure Structural Change*. Because this change group is composed of the sequential application of other change type models, there are no cross-coupling effects that occur because of the additive composition of other models.

4.8.5.1. Composed Activity Model of Pure Structural Change

Figure 94 is a composed Activity Model of *Pure Structural Change*.

Figure 94. Activity Model of Pure Structural Change

The following sequence of actions are represented:

- The action '*Architectural Change*' is composed as a sub activity diagram based on Figure 49.

- The action '*Topology Change*' is composed as a sub activity diagram based on Figure 54.
- The action '*Substitution*' is composed as a sub activity diagram based on Figure 59.

4.8.5.2. Composed Class Model of Pure Structural Change

Figure 95 shows a composed Class Model of *Pure Structural Change*.

Figure 95. Class Model of Pure Structural Change

It is a composition of the class models of *Architectural Change* (See Figure 50), *Topology Change* (See Figure 55), and *Substitution* (See Figure 60).

4.8.5.3. Composed Sequence Model of Pure Structural Change

Figure 96 shows a composed Sequence Model of *Pure Structural Change*. It is a composition of the sequence models of *Architectural Change* (See Figure 51), *Topology Change* (See Figure 56), and *Substitution* (See Figure 61).

Figure 96. Sequence Model of Pure Structural Change

4.8.5.4. Composed State Machine Model of Pure Structural Change

Figure 97 is a composed State Machine Model of *Pure Structural Change*.

Figure 97. State Machine Model of Pure Structural Change

It is a composition of the state machine models of *Architectural Change* (See Figure 52), *Topology Change* (See Figure 57), and *Substitution* (See Figure 62).

4.8.6. Composed Models of Structure-driven Behavioral Change

Structure-driven behavioral change is a change group that comprises three different change sequences, each driven by *architectural change*: (i) *topology change* leading to *protocol change*, (ii) *topology change* leading to *protocol change* leading to *interface change*, or (iii) *topology change* leading to *protocol change* leading to *interface change* leading to *internal change*. By implication, *structure-driven behavioral change* means that, when the structural signature of a system is realigned, it causes the behavioral signature of the system to be realigned.

This sub-section presents composed Activity, Class, Sequence, and State Machine models of *Structure-driven Behavioral Change*. Because this change group is composed

of the sequential application of other change type models, there are no cross-coupling effects that occur because of the additive composition of other models.

4.8.6.1. Composed Activity Model of Structure-driven Behavioral Change

Figure 98 is a composed Activity Model of *Structure-driven Behavioral Change*.

Figure 98. Activity Model of Structure-Driven Behavioral Change

The following sequence of actions are represented:

- The action '*Architectural Change*' is composed as a sub activity diagram based on Figure 49.
- The action '*Topology Change*' is composed as a sub activity diagram based on Figure 54.
- The action '*Protocol Change*' is composed as a sub activity diagram based on Figure 64.

- The action '*Interface Change*' is composed as a sub activity diagram based on Figure 69.
- The action '*Internal Change*' is composed as a sub activity diagram based on Figure 74.

4.8.6.2. Composed Class Model of Structure-driven Behavioral Change

Figure 99 shows a composed Class Model of *Structure-driven Behavioral Change*. It is a composition of the class models of *Architectural Change* (See Figure 50), *Topology Change* (See Figure 55), *Protocol Change* (See Figure 65), *Interface Change* (See Figure 70) and *Internal Change* (See Figure 75).

Figure 99. Class Model of Structure-driven Behavioral Change

4.8.6.3. Composed Sequence Model of Structure-driven Behavioral Change

Figure 100 shows a composed Sequence Model of *Structure-Driven Behavioral Change*. It is a composition of the sequence models of *Architectural Change* (See Figure 51), *Topology Change* (See Figure 56), *Protocol Change* (See Figure 66), *Interface Change* (See Figure 71), and *Internal Change* (See Figure 76).

Figure 100. Sequence Model of Structure-Driven Behavioral Change

4.8.6.4. Composed State Machine Model of Structure-driven Behavioral Change

Figure 101 is a composed State Machine Model of *Structure-Driven Behavioral Change*.

Figure 101. State Machine Model of Structure-Driven Behavioral Change

It is a composition of the state machine models of *Architectural Change* (See Figure 52), *Topology Change* (See Figure 57), *Protocol Change* (See Figure 67), *Interface Change* (See Figure 72), and *Internal Change* (See Figure 77).

4.8.7. Composed Models of Comprehensive Structure-driven Change

Comprehensive structure-driven change is a change group that is a combination of *pure structural change* and *structure-driven behavioral change*. By implication, *comprehensive structure-driven change* means that, when the structural signature of a

system is realigned, it causes the behavioral signature of the system to be realigned. All combinations are possible as long as a structural change leads to a behavioral change.

This sub-section presents composed Activity, Class, Sequence, and State Machine models of *Comprehensive Structure-driven Change*. Because this change group is composed of two other change groups which occur simultaneously, there can be cross-coupling effects because of the additive composition of the other two models. For example, *Architectural Change* and *Topology Change* are in common at the beginning of both composed change groups and could therefore be unified in their application. Depending on the context, *Protocol Change* and *Substitution* could apply to the same component. When this occurs they could be serialized to ensure *Substitution* occurred first to aid the management of *Comprehensive Structure-driven Change*.

4.8.7.1. Composed Activity Model of Comprehensive Structure-driven Change

Figure 102 is a composed Activity Model of *Comprehensive Structure-driven Change*.

Figure 102. Activity Model of Comprehensive Structure-Driven Change

The following sequence of actions are represented:

- The action '*Pure Structural Change*' is composed as a sub activity diagram based on Figure 94.

- The action '*Structure-Driven Behavioral Change*' is composed as a sub activity diagram based on Figure 98.

4.8.7.2. Composed Class Model of Comprehensive Structure-driven Change

Figure 103 shows a composed Class Model of *Comprehensive Structure-driven Change*. It is a composition of the class models of *Pure Structural Change* (See Figure 95) and *Structure-Driven Behavioral Change* (See Figure 99).

Figure 103. Class Model of Comprehensive Structure-driven Change

4.8.7.3. Composed Sequence Model of Comprehensive Structure-driven Change

Figure 104 shows a composed Sequence Model of *Comprehensive Structure-Driven Change*.

Figure 104. Sequence Model of Comprehensive Structure-Driven Change

It is a composition of the sequence models of *Pure Structural Change* (See Figure 96) and *Structure-Driven Behavioral Change* (See Figure 100).

4.8.7.4. Composed State Machine Model of Comprehensive Structure-driven Change

Figure 105 is a composed State Machine Model of *Comprehensive Structure-Driven Change*.

Figure 105. State Machine Model of Comprehensive Structure-Driven Change

It is a composition of the state machine models of *Pure Structural Change* (See Figure 97) and *Structure-Driven Behavioral Change* (See Figure 101).

4.8.8. Composed Models of Pure Behavioral and Structural Change

Pure behavioral and structural change is a change group that is a combination of *pure behavioral change* and *pure structural change*. By implication, *pure behavioral and structural change* means that the behavioral signature and the structural signature of a system are realigned but one does not drive the other to change. All combinations are possible as long as behavioral change does not lead to structural change nor does structural change lead to behavioral change.

This sub-section presents composed Activity, Class, Sequence, and State Machine models of *Pure Behavioral and Structural Change*. Because this change group is composed of two other change groups which occur simultaneously, there can be cross-coupling effects because of the additive composition of the other two models. For example, *Architectural Change* is in common at the beginning of both composed change groups and could therefore be unified in its application. Depending on the context, *Protocol Change* and *Topology Change* could apply to the same component even though one does not drive the other to change. When this occurs they could be serialized to aid the management of *Pure Behavioral and Structural Change*.

4.8.8.1. Composed Activity Model of Pure Behavioral and Structural Change

Figure 106 is a composed Activity Model of *Pure Behavioral and Structural Change*.

Figure 106. Activity Model of Pure Behavioral and Structural Change

The following sequence of actions are represented:

- The action '*Pure Behavioral Change*' is composed as a sub activity diagram based on Figure 82.
- The action '*Pure Structural Change*' is composed as a sub activity diagram based on Figure 94.

4.8.8.2. Composed Class Model of Pure Behavioral and Structural Change

Figure 107 shows a composed Class Model of *Pure Behavioral and Structural Change*. It is a composition of the class models of *Pure Behavioral Change* (See Figure 83) and *Pure Structural Change* (See Figure 95).

4.8.8.3. Composed Sequence Model of Pure Behavioral and Structural Change

Figure 108 shows a composed Sequence Model of *Pure Behavioral and Structural Change*.

Figure 108. Sequence Model of Pure Behavioral and Structural Change

It is a composition of the sequence models of *Pure Behavioral Change* (See Figure 84) and *Pure Structural Change* (See Figure 96).

4.8.8.4. Composed State Machine Model of Pure Behavioral and Structural Change

Figure 109 is a composed State Machine Model of *Pure Behavioral and Structural Change*.

Figure 109. State Machine Model of Pure Behavioral and Structural Change

It is a composition of the state machine models of *Pure Behavioral Change* (See Figure 85) and *Pure Structural Change* (See Figure 97).

4.8.9. Composed Models of Comprehensive Change

Comprehensive change is a change group that is a combination of *comprehensive behavior-driven change* and *comprehensive structure-driven change*. By implication, *comprehensive change* means that the behavioral signature and the structural signature of a system are realigned as one drives the other to change. All combinations are possible as long as behavioral change leads to structural change and structural change leads to behavioral change.

This sub-section presents composed Activity, Class, Sequence, and State Machine models of *Comprehensive Change*. Because this change group is composed of two other change groups which occur simultaneously, there can be cross-coupling effects because of the additive composition of the other two models. This is the most complex case because each composed change group is in turn composed of two serialized change groups. Depending on the context, the challenge is to serialize the application of change types across all change groups to aid the management of *Comprehensive Change*. For example structural change could be given precedence when two change types apply to the same component.

4.8.9.1. Composed Activity Model of Comprehensive Change

Figure 110 is a composed Activity Model of *Comprehensive Change*.

Figure 110. Activity Model of Comprehensive Change

The following sequence of actions are represented:

- The action '*Comprehensive Behavior-Driven Change*' is composed as a sub activity diagram based on Figure 90.
- The action '*Comprehensive Structure-Driven Change*' is composed as a sub activity diagram based on Figure 102.

4.8.9.2. Composed Class Model of Comprehensive Change

Figure 111 shows a composed Class Model of *Comprehensive Change*. It is a composition of the class models of *Comprehensive Behavior-Driven Change* (See Figure 91) and *Comprehensive Structure-Driven Change* (See Figure 103).

Figure 111. Class Model of Comprehensive Change

4.8.9.3. Composed Sequence Model of Comprehensive Change

Figure 112 shows a composed Sequence Model of *Comprehensive Change*.

Figure 112. Sequence Model of Comprehensive Change

It is a composition of the sequence models of *Comprehensive Behavior-Driven Change* (See Figure 92) and *Comprehensive Structure-Driven Change* (See Figure 104).

4.8.9.4. Composed State Machine Model of Comprehensive Change

Figure 113 is a composed State Machine Model of *Comprehensive Change*.

Figure 113. State Machine Model of Comprehensive Change

It is a composition of the state machine models of *Comprehensive Behavior-Driven Change* (See Figure 93) and *Comprehensive Structure-Driven Change* (See Figure 105).

4.9. Executable Model

When a system's run-time properties are realized as instantiations of programmable metaclasses, these properties can be dynamically changed when their metaclasses change. This section describes the use of explicit metaclass programming techniques to encode certain aspects of the domain model [63, 86-94]. The purpose of constructing an executable model is: (i) to gain a more direct understanding of the conceptual framework and (ii) to validate, through empirical proof and simulation, that the constrained model is applicable within application-specific contexts.

With this approach, metaobjects are employed to implement the *Component Framework* shown in Figure 1. The *Interacting Components* shown in Figure 1 are implemented by base-level objects, which are instances of the metaobjects. In this environment metaobjects themselves are run-time objects. As run-time objects they can interact and change their characteristics. When they change their characteristics they can change the run-time properties of base-level objects that are their instances.

Using the case study, a description is given of representing and then dynamically changing an application-specific global property leading to other kinds of change. The general approach is the following. The metaobjects of system and change properties encapsulate system signature 'fragments' that are implied by their instances. The current system signature of a system is the composition of the system signature 'fragments' of its global and local system properties. To change the signature of a system, the system signature 'fragments' of change properties are composed with the existing system signature 'fragments' of system properties. In this implementation, the global or local property metaobjects implement a standardized protocol for setting up a system signature

‘fragment’. A particular fragment is expressed as a configuration of base-level objects that can be instantiated at the time change properties are reconciled with system properties.

4.9.1. Using Explicit Metaclasses as an Executable Representation

Abelson et al [95], Bouraqadi-Saadani et al [86], and Yokote [87] are based on a meta-circular and reflective approach of self-representation or self-description for effecting behavioral and/or structural change to a running system. Abelson et al [95] use the Scheme programming environment to present fundamental concepts with respect to data and procedural abstraction and manipulation, meta-circular interpretation, and other aspects.

Yokote [87] describes a reflective operating system that implements an execution environment that directly supports evolution and adaptation. The open-ended-ness and dependability properties of objects are individually accommodated through an object migration capability that is an inherent aspect of the system. Object migration is achieved by separating base and meta level objects within the design of the operating system with the meta level objects defining the semantics of base level objects’ behavior. The meta-level objects can be optimized for a particular base level object; in addition, object evolution and adaptation is further supported by allowing base level objects to change from one group of meta objects (reflectors) to another. As an example, it is shown that an object can acquire persistence properties by migrating to a reflector that supports persistence.

Similar to Yokote [87], Bouraqadi-Saadani et al [86] describe the MetaclassTalk computing environment that provides direct support for meta object composition

mechanisms to enable system evolution and adaptation. These composition mechanisms allow specific properties to be assigned to classes in order to allow the properties of their instances to dynamically change. They note that from an architectural viewpoint meta class composition allows a system to be organized into different levels of abstraction. An example is given showing the meta class composition “False + SoleInstance + Final” to create composed class properties that may then be instantiated as the sole false instance whose class may not be sub-classed.

In this section, the suitability of MetaclassTalk’s explicit metaclass programming facility is discussed and then a description is given of using this facility to represent a global property.

4.9.1.1. MetaclassTalk as a Reflective Substrate

Bouraqadi-Saadani et al [86, 96-102] describes a reflective system development approach that utilizes safe and explicit metaclass programming techniques and an implementation environment that enables this that is known as MetaclassTalk. This paradigm is based upon meta-links, which causally connect base-level and meta-level objects, and meta-object cooperation, which is explicitly programmed. Examples of the kinds of relationships that may be implemented between base-level objects and meta-level objects include: (i) a single meta-object shared between instances of the same class; (ii) a single specific meta-object private to each base object; and (iii) many meta-objects shared among many base objects [96]. This is shown in Figure 114 below:

Figure 114. Base versus meta-objects

Figure 115 shows how MetaclassTalk can be used as a reflective substrate of the domain model presented in Chapter 3. The *System Model* and *Context of Change* features, shown in Figure 47, are encoded as meta-objects that cooperate to enable the dynamic reconfiguration of base-level application-specific component interactions. The meta-link causally connecting the base-level to *Context of Change* represents emergent run-time characteristics that are ultimately manifested as change properties. *Context of Change* and the *System Model* cooperate at the meta-level to reconcile change properties with existing system properties and to realign the behavioral and/or structural profile of the system as required. The meta-link linking the base-level to the *System Model* represents the causal connection required to affect the realignment of the running system. This may in turn lead to new emergent run-time characteristics, leading to further dynamic reconfiguration, and so on.

Figure 115. Using the reflective substrate

4.9.1.2. Using Explicit Metaclass Programming

This subsection illustrates how the domain design of comprehensive change is represented within the reflective substrate using explicit metaclass programming techniques. What follows is an explanation of how *Global Properties* are encoded.

Bouraqadi-Saadani [102] explains the use of *compatibility* and *property* metaclasses to ensure the overall compatibility of composed metaclasses that are explicitly programmed. This technique is employed when encoding the constrained model and therefore *compatibility* and *property* metaclasses are part of the encoding. Bouraqadi-Saadani [102] also implements meta-level support for mixins as a mechanism to augment *property* metaclasses. This technique is also used when encoding the model. As an alternative mechanism to mixins, the use of “traits” is under review Scharli et al [103]. Because Bouraqadi-Saadani [102] explicitly uses the term *property*, property named elements of the domain model are referenced as *constraints* below.

Figure 116 shows how global and local constraints are encoded. As discussed previously, a global constraint is a system characteristic that requires global knowledge for conformance. A local constraint is a system characteristic that requires local (i.e. component-level) knowledge for conformance. Examples of constraint characteristics that may apply, globally or locally, include state element immutability, cardinality (i.e.

limits on the number of service invocations or service bindings, and so on), performance, synchronization, distribution, persistence, security, control style, and so on. These characteristics are represented as *Constraint Characteristics Meta Objects* in Figure 116. They are ‘mixedin’ when a global or local constraint is instantiated based upon a *mixin linearization list* that determines the order of composition.

Figure 116. Encoding global and local constraints

The following is a short description of the metaobjects shown in Figure 116 that are required to encode a global property:

- *Global Constraint* is defined as a concrete subclass of *Constraint*. It is an instance of *Global Constraint Property Meta Object*. When instantiated as a base-level object, a

global constraint represents a particular global property of a component-based software system. Examples of global constraints were given in Chapter 3.

- *Constraint* is defined as an instance of *Constraint Property Meta Object*. It is the abstract superclass of *Global Constraint* and *Local Constraint*. It implements *constraintSpec*, which represents the specification of a particular global or local constraint that is instantiated by *Global Constraint* or *Local Constraint*. It also implements the creation method, *new: with Mixin Linearization List*, of *Global Constraint* and *Local Constraint*. The mixin linearization list determines the mixin properties that are composed when a global or local constraint is created.
- *Constraint Property Meta Object* is defined as an instance of *Composite Class* (not shown). *Composite class* is part of the MetaclassTalk library. It is a class whose instances inherit from their superclass through a set of mixins. The inheritance relationship is via implicit subclass(es) of the ‘official superclass’ that are built by means of the mixin mechanism. *Constraint Property Meta Object* is defined as a subclass of *Constraint Compatibility Meta Object* with *Abstract* as a mixin. *Abstract* is part of the MetaclassTalk library of mixins. When it is added as a mixin to *Constraint Property Meta Class* mixins, *Constraint* becomes an abstract class.
- *Constraint Compatibility Meta Class* is defined as a subclass and as an instance of *Standard Class* (not shown). *Standard Class* is part of the MetaclassTalk library. This is the root class of all MetaclassTalk explicit metaclasses. When subclassed, it enables the explicit definition of new kinds of (meta) classes. *Constraint Compatibility Meta Class* enables the compatibility model described in [102] among

the inheritance hierarchies that underpin *Constraint*, *Global Constraint*, and *Local Constraint*.

- *Global Constraint Property Meta Object* is defined as an instance of *Composite Class* to enable mixin property composition (not shown). It is defined as a subclass of *Global Constraint Compatibility Meta Object* to ensure it complies with the meta class compatibility model. *Class With Instance Mutable Meta Objects* is part of the MetaclassTalk library of mixins (not shown). When added as a mixin, the instances of *Global Constraint Property Meta Object* (i.e. *Global Constraint*) have their own metaObject, which can be changed over time.
- *Global Constraint Compatibility Meta Class* is defined as an instance of *Standard Class* (not shown). As an explicit metaclass, it is defined as a subclass of *Constraint Compatibility Meta Object* to enable the compatibility model.

4.9.2. Executing an Application-Specific Example

The requirement for System A to dynamically inter-operate with System B is an example of emergent properties as shown in Figure 115. In this example, the outcome is reflected by Figure 10 with a new UCM causally linking System A to System B. This subsection explains how the reflective substrate affects base-level application-specific dynamic reconfiguration, when a global property changes. The following is described: (i) how a global property is instantiated using the encoded model, (ii) dynamically reconfiguring the global property, (iii) how Architectural Change reconfigures the signature of a system, and (iv) how a reconfigured system signature leads to behavioral and structural change.

4.9.2.1. Representing a Global Property

This subsection illustrates how the reflective substrate, is used to create instances of GP1, which determines the control style of System A. Figure 116 shows the base-level and meta-level interactions required. *System A - GP-1-Control Style Constraint* and *System A - GP-1-Change Property* are shown as instances of *Global Constraint*. When *System A - GP-1-Control Style Constraint* and *System A - GP-1-Change Property* are created, their global constraint properties are augmented with an appropriate control style provided by *Control Style Characteristics MetaObjects*. *System A - GP1 - Control Style Constraint* conforms to the control style shown in Figure 4. *System A - GP1 - Change Property* conforms to the control style shown in Figure 10.

For example, when *System A - GP-1-Control Style Constraint* is created, the mixin linearization list drives the composition of specific control style metaobjects that constrain the configuration of System A's components to ensure cash flow projections and valuation assessment are done on-line.

MetaclassTalk code fragments are shown below to illustrate how this is done. In the example, *gp1* is created to represent the global property *System A - GP-1-Control Style Constraint*. The specification of this property for System A is 'On-Line Control'. When *gp1* is created, the metalink policy *Class With Instance Mutable Meta Objects* is applied that enables it to have a unique metaclass (this was reported as "a single specific meta-object private to each base object" earlier). In the case of *gp1* its particular properties are augmented with *System A On Line Control Style MetaObject* in support of its 'On-Line Control' specification:

```
gp1 := GlobalConstraint new.  
gp1 constraintSpec: 'Control Style of System '.
```

```

metaclass := MetaObject
  subclass: #GP1AMetaClass
  instanceVariableNames: "
  classVariableNames: "
  poolDictionaries: "
  category: 'Dynamic Reconfiguration-Interacting Components-Instance Specific Property
MetaObjects'
  metaclass: CompositeClass.
metaclass addMixin: ClassWithInstanceMutableMetaObjects.
metaclass addMixin: SystemAOnLineControlStyleMetaObject.
metaclassInstance := metaclass new.
gp1 metaObject: metaclassInstance.

```

When *GP1AMetaClass* is created, it is a metaObject specific to *gp1*. Its mixins include

System A On Line Control Style Meta Object:

```

MetaObject subclass: #GP1AMetaClass
  instanceVariableNames: "
  classVariableNames: "
  poolDictionaries: "
  category: 'Dynamic Reconfiguration-Interacting Components-Instance Specific Property
MetaObjects'
  metaclass: CompositeClass.
GP1AMetaClass mixins: {ClassWithInstanceMutableMetaObjects.
SystemAOnLineControlStyleMetaObject}

```

System A On Line Control Style Meta Object is a mixin with instance variable 'system

Signature For System A On Line Control':

```

Mixin named: #SystemAOnLineControlStyleMetaObject
  instanceVariables: 'systemSignatureForSystemAOnLineControl '
  category: 'Dynamic Reconfiguration-Mixins'

```

System A On Line Control Style Meta Object implements a standardized protocol that is used to set up 'system Signature For System A On Line Control'. It is an instance of *System Signature*, which represents the behavioral and structural dependencies reflected by Figure 4. Below is an example of part of the protocol implemented by *System A On Line Control Style Meta Object*:

```

setUpConnectedComponentDependencies

systemSignatureForSystemAOnLineControl connectComponent: self
energySectorConditionsComponent toComponent: self cashFlowProjectionsComponent.

```

```
systemSignatureForSystemAOnLineControl connectComponent: self
cashFlowProjectionsComponent toComponent: self valuationAssessmentComponent.
```

```
systemSignatureForSystemAOnLineControl connectComponent: self
valuationAssessmentComponent toComponent: self energySectorKnowledgeComponent.
```

```
systemSignatureForSystemAOnLineControl connectComponent: self
energySectorKnowledgeComponent toComponent: self scenarioAnalysisComponent.
```

The instance specific metaObject of *System A - GP-1-Change Property* also implements the standardized protocol. In its case, its *System Signature* represents the behavioral and structural dependencies reflected by Figure 10.

4.9.2.2. Dynamically Reconfiguring a Global Property

This subsection illustrates change properties affecting existing system properties, which subsequently drives the system signature of System A to change.

In Figure 116 emergent system properties are manifested as *System A – GP1 – Change Property*. *System A - GP-1-Control Style Constraint* represents an existing reconciled global constraint of System A before any change. *System A – GP1 – Change Property* is a global constraint that implies a change to the control style of System A as reflected by Figure 10. Its particular properties are augmented with *System B Interoperation MetaObject* which represents its augmentation of the ‘On-Line Control’ specification:

```
newgp1a := GlobalConstraint new.
newgp1a constraintSpec: 'Control Style of System '.
metaclass := MetaObject
    subclass: #NewGP1AMetaClass
    instanceVariableNames: "
    classVariableNames: "
    poolDictionaries: "
    category: 'Dynamic Reconfiguration-Interacting Components-Instance Specific Property
MetaObjects'
    metaclass: CompositeClass.
metaclass addMixin: ClassWithInstanceMutableMetaObjects.
metaclass addMixin: SystemBInteroperationMetaObject.
metaclassInstance := metaclass new.
newgp1a metaObject: metaclassInstance.
```

When *NewGP1AMetaClass* is created, it is a metaObject specific to *newgp1*. Its mixins include *System B Interoperation Meta Object*:

```
MetaObject subclass: #NewGP1AMetaClass
    instanceVariableNames: "
    classVariableNames: "
    poolDictionaries: "
    category: 'Dynamic Reconfiguration-Interacting Components-Instance Specific Property
MetaObjects'
    metaclass: CompositeClass.
NewGP1AMetaClass mixins: {ClassWithInstanceMutableMetaObjects.
SystemBInteroperationMetaObject}
```

To drive system change, the original metaclass properties of *System A - GP-1-Control Style Constraint* are dynamically augmented to reflect the new control style of System A by adding the metaclass properties of *System A – GP1 – Change Property* as additional mixin properties of *System A - GP-1-Control Style Constraint*:

```
gp1 metaObject class addAllMixins: (newgp1 metaObject class mixins).
```

This is the result after adding the mixins of *newgp1*'s metaclass to the mixins of *gp1*'s metaclass. The mixins of *GP1AMetaClass* now includes *System B Interoperation Meta Object*:

```
MetaObject subclass: #GP1AMetaClass
    instanceVariableNames: "
    classVariableNames: "
    poolDictionaries: "
    category: 'Dynamic Reconfiguration-Interacting Components-Instance Specific Property
MetaObjects'
    metaclass: CompositeClass.
GP1AMetaClass mixins: {ClassWithInstanceMutableMetaObjects.
SystemAOnLineControlStyleMetaObject. SystemBInteroperationMetaObject}
```

System A On Line Control Style Meta Object and *System B Interoperation Meta Object* each encapsulate a *System Signature* as described previously.

4.9.2.3. Architectural Change Reconfiguring the Signature of a System

Ultimately, a change to the global and local properties of System A leads to the reconfiguration of its system signature. In this implementation, Architectural Change regenerates the overall system signature of System A by composing the system signature 'fragments' of property metaobjects. This drives the behavioral and structural realignment of the system. This is shown in the code fragment below:

```
Dependencies>>useSystemPropertiesToGenerateSystemSignature
self systemProperties systemSignatures do: [ :aSysSig | self systemSignature updateWith:
aSysSig].
```

The definition of *SystemSignature* is shown below. Aligned with the domain model, it is composed of *StructuralSignature* and *BehavioralSignature*.

```
Object subclass: #SystemSignature
  instanceVariableNames: 'structuralSignature behavioralSignature '
  classVariableNames: "
  poolDictionaries: "
  category: 'Dynamic Reconfiguration-Interacting Components'
  metaclass: SystemSignaturePropertyMetaObject
```

The code fragment below shows how the existing system signature is regenerated with a system signature 'fragment':

```
SystemSignature>>updateWith: aSystemSignature
self behavioralSignature updateWith: aSystemSignature behavioralSignature.
self structuralSignature updateWith: aSystemSignature structuralSignature.
```

The definition of *BehavioralSignature* is shown below. Aligned with the domain model (See Figure 2), it is composed of protocol, required service, provided service, operation, state element, and composite component dependencies.

```
Object subclass: #BehavioralSignature
  instanceVariableNames: 'protocolDependencies requiredServiceDependencies
providedServiceDependencies operationDependencies stateElementDependencies
compositeComponentDependencies '
  classVariableNames: "
  poolDictionaries: "
```

category: 'Dynamic Reconfiguration-Interacting Components'
 metaclass: BehavioralSignaturePropertyMetaObject

When *Architectural Change* regenerates the *Behavioral Signature* of a system it updates the existing *Behavioral Signature* with a new *Behavioral Signature* that reflects any changes in behavioral dependencies:

```
BehavioralSignature>>updateWith: aBehavioralSignature
self updateProtocolDependenciesWith: aBehavioralSignature protocolDependencies.
self updateRequiredServiceDependenciesWith: aBehavioralSignature
  requiredServiceDependencies.
self updateProvidedServiceDependenciesWith: aBehavioralSignature
  providedServiceDependencies.
self updateOperationDependenciesWith: aBehavioralSignature operationDependencies.
self updateStateElementDependenciesWith: aBehavioralSignature stateElementDependencies.
self updateCompositeComponentDependenciesWith: aBehavioralSignature
  compositeComponentDependencies.
```

The definition of *StructuralSignature* is shown below. Again, in phase with the domain model (See Figure 2), it is composed of connected component and connected service dependencies.

```
Object subclass: #StructuralSignature
  instanceVariableNames: 'connectedComponentDependencies
    connectedServiceDependencies '
  classVariableNames: "
  poolDictionaries: "
  category: 'Dynamic Reconfiguration-Interacting Components'
  metaclass: StructuralSignaturePropertyMetaObject
```

When *Architectural Change* regenerates the *Structural Signature* of a system it updates the existing *Structural Signature* with a new *Structural Signature* that reflects any changes in structural dependencies:

```
StructuralSignature>>updateWith: aStructuralSignature
self updateConnectedComponentDependenciesWith: aStructuralSignature
  connectedComponentDependencies.
self updateConnectedServiceDependenciesWith: aStructuralSignature
  connectedServiceDependencies.
```

In the example up to now, the metaobject of *System A - GP-1-Control Style Constraint* has been augmented with the mixin properties of the metaobject of *System A - GPI -*

Change Property. Each mixin of *System A - GP-1-Control Style Constraint* encapsulates a system signature ‘fragment’ that Architectural Change now uses to regenerate the overall *System Signature* of System A.

In this case, it is *SystemBInteroperationMetaObject*, which encapsulates the structural dependencies that reflect a new communication path linking System A with System B, which leads to the Topology Change of System A.

To do this, *SystemBInteroperationMetaObject* implements the following dependency:

```
SystemBInteroperationMetaObject >>setUpConnectedServiceDependencies
SystemSignatureForSystemBInteroperation connectRequiredServiceNamed: 'Short Term
Predictions Based On Long Term Predictions' toProvidedServiceNamed: 'Short Term Predictions
Based On Long Term Predictions'.
```

SystemSignatureForSystemBInteroperation represents the *System Signature* ‘fragment’ of *SystemBInteroperationMetaObject*. The outcome of this metaobject setup operation is a *System Signature* ‘fragment’ with a connected service dependency linking a required service of System A to a provided service of System B:

```
SystemSignature>>connectRequiredServiceNamed: requiredName toProvidedServiceNamed:
providedName
| rs ps |
rs := self behavioralSignature requiredServiceNamed: requiredName.
ps := self behavioralSignature providedServiceNamed: providedName.
self structuralSignature connectedServiceDependencies at: rs put: ps.
```

When *Architectural Change* regenerates the overall *System Signature* of System A, the connected service dependency of *SystemBInteroperationMetaObject* is composed with the existing connected service dependencies of System A when System A’s *Structural Signature* is updated (See *StructuralSignature>>updateWith: aStructuralSignature* shown above).

4.9.2.4. A Reconfigured System Signature Leading to Topology Change

The regeneration of *System Signature* drives the behavioral and structural realignment of a system. A change to *Behavioral Signature* of *System Signature* leads to a behavioral realignment through *Protocol*, *Interface*, and/or *Internal Change*. A change to *Structural Signature* of *System Signature* leads to a structural realignment through *Topology Change* and/or *Substitution*.

In this implementation, the kind of (new) dependency added to *System Signature* determines what type of change needs to be applied. New protocol dependencies are manifested as a change to service protocols via *Protocol Change*. New required or provided service dependencies are manifested as a change to required or provided services via *Interface Change*. New operation, state element, or composite component dependencies are manifested as a change to operations, state elements, or composite components via *Internal Change*. New connected component dependencies are manifested as changes to components and communication paths via *Topology Change* or *Substitution*. New connected service dependencies are manifested as changes to connections via *Topology Change* or *Substitution*.

In this example, the topology of System A is changed by adding a connection between the *Scenario Analysis* component of System A and the *Scenario Analysis* component of System B. The new connected service dependency added to the *Structural Signature* of System A results in this *Topology Change*, which leads to *Protocol Change*, *Interface Change*, and *Internal Change*.

The *Scenario Analysis* component of System A is an instance of *Deployed Component*:

Object subclass: #DeployedComponent

```

instanceVariableNames: 'name dependencies requiredServices providedServices
communicationPaths compositeComponents internalBehavior '
classVariableNames: "
poolDictionaries: "
category: 'Dynamic Reconfiguration-Interacting Components'
metaclass: ComponentPropertyMetaObject

```

Before any change the communication paths (and required services) of the *Scenario Analysis* component conform to the original control style of System A (See Figure 4):

```

CommunicationPaths: a Dictionary('EnergySectorKnowledge-ScenarioAnalysis'->a
CommunicationPath )

```

```

Required Services: a Dictionary('Knowledge Representations of Scenario Predictions'->a
ProvidedService 'Scenario Predictions of Knowledge Representations'->a ProvidedService )

```

Topology Change is manifested as a new communication path and connection added to the communication paths of the *Scenario Analysis* component. After *Topology Change*, the communication paths (and subsequently, after *Protocol* and *Interface Change*), the required services, conform to the new control style of System A (See Figure 10):

```

CommunicationPaths: a Dictionary('EnergySectorKnowledge-ScenarioAnalysis'->a
CommunicationPath 'ScenarioAnalysis-ScenarioAnalysis'->a CommunicationPath)

```

```

Required Services: a Dictionary('Knowledge Representations of Scenario Predictions'->a
ProvidedService 'Scenario Predictions of Knowledge Representations'->a ProvidedService 'Short
Term Predictions Based On Long Term Predictions'->a ProvidedService)

```

4.9.3. Discussion

It was learned that the explicit metaclass programming facility of MetaclassTalk can effectively be used to encode the constrained model, as a static representation, at the metalevel. Base level objects could then be instantiated, based upon these static representations, as application-specific interactions. With respect to dynamic reconfiguration, it was learned that a base-level object could be an instance of a property metaclass that is unique to that base-level object. Through MetaclassTalk's mixin mechanism, emergent run-time properties could be dynamically composed that applied

just to that object. The set of available mixins should also be adjusted dynamically. This was not addressed in this thesis but is the subject of future work.

As a simulator of dynamic system reconfiguration, the MetaclassTalk programming environment is a build-time and run-time facility that is useful for validating that the domain design can be instantiated within an application-specific context. This level of validation is complementary with what is validated by the Alloy Analyzer. The Analyzer is used to validate that the domain model itself is well-formed while the MetaclassTalk environment is used to validate that the domain model is useful for dynamically reconfiguring a particular application (like the financial analysis system).

As a framework, the simulator implements 'plugin points' for realizing the general context of dynamic reconfiguration. In doing so, it provides the macro-level ordering of operations and the contextual information required at each step for a more sophisticated implementation. Generally, there is a need for a facility that allows backing out and restoring the configuration of the system if dynamic change is rejected.

The current implementation of the reconciliation of change properties with system properties is based on name matching (to associate a change property with a system property), with the change property taking precedence. A more sophisticated implementation would provide a constraint solver-like capability to determine whether multiple constraints were compatible.

This implementation simply accepts the cost of enacting reconfiguration when estimating the impact of change. A more sophisticated implementation would assess this cost taking into account previous results (through heuristics), the current configuration

(through dataflow dependency analysis), and future results (through predictive simulation).

The current implementation of the 'don't care' ordering of the application of a change type and its integrity characteristics is arbitrary with the change made first. The implementation assumes integrity characteristics can be assured. A more sophisticated implementation would ensure system integrity including assessing what was the optimal order of application based on current conditions. For example, in the case of Substitution, if possible the change could be made during a time window when there were not any active references. As another example, in the case of *Internal Change*, when a new operation is deployed to replace an existing operation, both operations could co-exist for a limited time period to ensure continuity of any active dependencies on the old operation.

Finally, as discussed, a change to system dependencies is manifested as realigning external interactions or internal behavior through an appropriate type of change. Multiple global or local properties of one change can be represented as multiple kinds of dependency when updating the signature of a system. A more sophisticated implementation would establish an optimal change strategy when there is more than one way to dynamically reconfigure a system to achieve the same end result.

4.10. Summary and Conclusion

This chapter used UML to express a domain design (or a software architecture) of the domain analysis presented in Chapter 3.

4.10.1. Summary

In order to help manage the inherent complexity of formulating a software architecture of dynamic system reconfiguration, distinct dimensions of change were modeled separately. To do this, relevant fragments of the domain analysis were identified (as features and as Alloy relations) that specifically defined the domain context of a particular change type. UML Activity, Class, Sequence, and State Machine models representations were then presented. The Activity models define the primary feature interactions of the type of change being modeled. The class models define relevant structural entities. The Sequence models define how the structural entities interoperate. The state machine models define the life cycle of change.

Based upon the common and variable causal flow of dynamic reconfiguration described in Chapter 3, like-UML models were composed as representations of each of the nine change groups. Depending upon the change group, the UML composition is either directly a function of the change type UML models or it is a function of the UML models of other change groups. Ultimately, the change group models build upon one another and are manifested as Comprehensive Change. Performing static impact analysis, performing dynamic impact analysis, enacting change consistently, and indicating the outcome is a common pattern that is applied for every change scenario.

Finally, to validate the domain model within an application-specific context, the domain design was realized as an executable representation using a computing environment that supports explicit metaclass programming techniques.

4.10.2. Conclusion

The reconciliation of change properties with (existing) system properties is represented as a distinct action. When properties are expressed as constraints this reconciliation could be computed through a constraint solver (such as the Alloy analyzer or a different implementation of a similar capability). Importantly, depending upon the context, there may not be a solution. In this case the change may be rejected or the change constraints re-expressed in a manner that enables a resolution. This issue is not addressed in this thesis but is the subject of future work.

Given that change constraints can be reconciled with (existing) system constraints, the UML models represented estimating the impact of Architectural Change upon system integrity as a distinct action. Computing this estimate is viewed to be an open problem and to be highly dynamic in nature since it is a function of the (changing) state of the system at the time the estimate is computed. Importantly, depending upon the context, a system may change significantly during the time it takes to perform this computation to such a degree that the estimate is no longer valid. This issue is not addressed in this thesis but is the subject of future work.

Given that the system will proceed with the change, the system model constructs relevant to the change type must change and each system integrity characteristic relevant to the change type must be assured. The UML models represent this in a “don’t care” order. Depending upon the context, a particular ordering may be preferred (for feasibility, efficiency, or other reasons). This issue is not addressed in this thesis but is the subject of future work.

In the MetaclassTalk implementation, because executability is expressed as message passing among (pure) objects (where everything is a runtime object including classes), it became clear that a constraint could be represented by a metaobject which implemented a standardized protocol for creating a system signature 'fragment' specific to that constraint. It also became clear that system signature fragments could be composed as a system signature that represented the fragments of each global and local property and the overall signature of a system. This is not viewed to change the domain design per se but instead is considered to be a utilization of the model in a manner that only became apparent when the model was encoded with the model execution environment.

5. Summary, Conclusion, and Future Work

This thesis explains what software evolution means in the context of software components that interact to implement services. Using domain analysis and design techniques, a domain model is defined of dynamic system reconfiguration due to user-driven or computing environment-imposed discrete change events. The domain model explicitly specifies the relationships that exist among changing global or local properties, changing behavioral or structural signatures, and intra- or inter-component change.

5.1. Summary

Based on the complete feature model of the domain analysis (See Figure 45), Figure 117 summarizes the primary feature interactions of dynamic system reconfiguration.

Figure 117. Primary feature interactions of dynamic reconfiguration

With respect to the common and variable causal flow presented in Chapter 3, the process of dynamic reconfiguration progresses as follows:

- Sensing *User-Driven Change* and/or *Computing Environment Change*;

- Interpreting and then representing the particular *Origin of Change* as *Global Properties* or *Local Properties* with an associated *Reconciliation Policy* (this includes reconciliation with existing *System Properties*);
- Determining what feasible subsets of *Type of Change* (if any) satisfy the *Condition Change Criteria*;
- If necessary, realigning *Dependencies* and therefore possibly also *External Interactions* and *Internal Behavior*; and
- Ensuring *System Integrity Characteristics* when enacting change to help maintain overall system consistency.

5.2. Conclusion

Informed by related research, a case study is used to identify the kinds of dynamic change required and the system integrity characteristics that must be ensured. The complete description of the case study is given in Appendix A - Financial Analysis System, the analysis of volatile environmental information based on systems dynamic interoperation. In addition, Appendix B presents the types of change using a pattern-oriented approach citing the case study as an example when applicable.

Driven by the case study and the pattern descriptions, the use case of the general context of dynamic reconfiguration specifies the dimensions of change as alternative change groups (See Figure 22). Each separate domain model is directly traceable to the use case and therefore its role is easily understood in satisfying the requirements of the domain model as a whole. In satisfying this use case, the domain model is validated as a definition of a comprehensive dynamic reconfiguration capability.

The domain model has the capacity to specify critical aspects of runtime change and was demonstrated as an executable specification (using Alloy) and as an executable representation (using MetaclassTalk). The model is therefore powerful and convincing within its defined context. As a cogent description of what it means to dynamically change interacting software components, it contributes to resolving the challenge of dynamic system reconfiguration through achieving the following objectives:

- Existing software architecture, infrastructure and patterns are analyzed in order to understand the different dimensions of dynamic system reconfiguration; a discussion is given about this in Chapter 2 – Related Research Review;
- Model-based constructs are specified that enable the representation of distinct dimensions of dynamic change; using appropriate notations, Chapter 3 – Domain Analysis and Chapter 4 – Domain Design present model-based constructs of these dimensions;
- A model-based framework is specified that enables modeling combined dimensions of dynamic reconfiguration; using appropriate composition techniques, Chapters 3 and 4 present models that combine dimensions of change; Chapter 4 uses the common and variable causal flow of dynamic reconfiguration as a composition framework;
- A demonstration is given of realizing executable representations of different dimensions dynamic change; using Alloy as a modeling language and the Alloy Analyzer as a run-time environment, Chapter 3 demonstrates an executable specification of the domain analysis; using MetaclassTalk as a run-time environment

to encode the domain model, Chapter 4 demonstrates an executable representation of the domain design within an application-specific context;

- The domain model is compatible with existing modeling standards and approaches (in particular Model-Driven Engineering (MDE) and Service-Oriented Architecture (SOA) based technology); with respect to MDE, models are treated as first class entities and the domain design is expressed using UML to be compatible with existing modeling standards and approaches; with respect to SOA, the domain model applies within a service-oriented context because it supports the composition and reconfiguration of required and provided service interfaces, their interaction contracts (as protocols), and their implementation (as components); and
- The domain model provides a model-based foundation for future work to extend these modeling constructs; using the common and variable causal flow of dynamic reconfiguration, the domain model can be extended at the feature, Alloy, UML, or MetaclassTalk levels when addressing system specific life-cycle, fault-tolerance, formal method, context-aware, or other scenarios.

Through meeting these objectives, the following contributions are made:

- Definition of a domain model of dynamic reconfiguration through the identification of relevant terms and concepts that define interacting components, context of change, and dynamic reconfiguration;
- Validation of the domain analysis;
- Definition of a UML representation of the distinct dimensions of dynamic reconfiguration;

- Definition of a model-based dynamic reconfiguration composition framework that integrates the distinct modeling dimensions of the UML representation; and
- Demonstration of the applicability of the composition framework through a representative case study.

The domain model of dynamic system reconfiguration is a useful conceptual framework for applying systematic techniques to engineer constrained software solutions within this open problem space. It is a significant artifact in the following roles:

- In the role of a software architecture, it specifies a comprehensive dynamic reconfiguration capability in a manner that can be used to forward engineer dynamic systems;
- In the role of a metamodel encoded by a component framework, it defines the steps of dynamic change that must be implemented by the framework and the 'plugin points' required for realization of the different aspects of the dynamic reconfiguration of a component-based software system; and
- In the role of a reference model, it can be used to assess how dynamic an existing implementation is, the different levels of compliance of dynamic change, and how interoperable systems are at different levels of dynamism.

The domain model restricts system evolution to a feasible number of change types and system consistency to the management of a feasible number of integrity characteristics. Because it defines dimensions of change that can be individually addressed, it is suitable for simulating dynamic change and for the integration of appropriate techniques that can apply within a specific context. The open problem of system evolution is constrained to a series of sub-problems that can be solved separately using specific methods or techniques

relevant to the particular sub-problem. Using the domain model for context, a number of important implications become clear.

The conditions for evolution establish under what circumstances a system adapts. This is represented within the domain model as the distinct actions ‘Detect Origin of Change’ and ‘Represent Change through Global/Local Properties’. A key challenge is the determination of under what conditions a system undergoes adaptation. In the context of the domain model, this is the problem of the interpretation of continuous system evolution as discrete change events that can be represented as global and/or local constraints. The system model itself limits what can be expressed as constraints (or what kind of constraints can be acted upon) by the types of change and the rate these types can be enacted.

The circumstances when evolution is cost effective determine whether the change is worthwhile enacting. This is represented within the domain model as the distinct actions ‘Reconcile Change Properties with Existing System Properties’ and ‘Estimate Impact of Change upon System Integrity’. Given that system properties can be formalized as constraints, a constraint solver can implement ‘Reconcile Change Properties with Existing System Properties’. If there is no resolution, a key challenge is the reformulation of change constraints in a manner that ensures there is a resolution. Otherwise, the change can be rejected. ‘Estimate Impact of Change upon System Integrity’ can be addressed by dataflow dependency analysis (or like) techniques that enable an assessment to be made about what the impact will be when a particular change is made. Since the domain model explicitly defines the different kinds of dependencies, dependency analysis techniques can be developed and applied within a manageable context.

The types of change determine how a system evolves given that the conditions for change have been met. This is represented within the domain model as the distinct action 'Enact Change by Realigning System Constructs'. The domain model explicitly restricted system evolution to six types of change. Because each change type is explicitly related to the particular system model constructs it realigns, the model defines a restricted and reduced context for each change type that better enables the application of appropriate software engineering methods or techniques.

The management of system evolution ensures consistency constraints are not violated. This is represented within the domain model as the distinct action 'Ensure System Integrity Characteristics'. The domain model explicitly reduced the problem of ensuring overall system consistency to the management of seven system integrity characteristics. Because the integrity characteristics are explicitly related to the types of change and specific system model constructs, the model defines a restricted and reduced context for each aspect of managing system integrity that better enables the application of appropriate software engineering methods or techniques.

5.3. Future Work

Given the contributions of this thesis, the following is future work to be undertaken:

- Determining, through simulation, statistics, heuristics or other means, under what conditions a system undergoes adaptation and how continuous system evolution is interpreted as discrete change events that can be represented as global or local properties, including the different forms through which system constraints are expressed;

- Investigating the use of a SAT (satisfiability) solver for reconciling change properties with system properties including the reformulation of change constraints in a manner that ensures there is a resolution;
- Investigating dataflow dependency analysis (and like) techniques for assessing what the impact will be when a particular change is made including the problem that a system may change significantly during the time it takes to perform this computation to such a degree that the estimate is no longer valid;
- Investigating the preferred ordering of the application of change types and associated system integrity characteristics (for feasibility, efficiency, or other reasons);
- Investigating the cross-coupling effects that can occur because of the additive composition of change group models, including investigating giving precedence to certain kinds of change and the serialization of the application of change types across change groups while maintaining global consistency;
- Investigating different levels of control in the context of decentralized software evolution (in support of load-balancing, change rejection and roll-back, fault-tolerance, state splitting and merging, and the system's life cycle); and
- Refining the domain model and the simulator (including their further validation and automatic transformation, dynamically adjusting the set of mixins, and the use of 'traits') when implementing this capability in an industrial context.

A key aspect enabling adaptive computing is the level of dynamism supported by the computing environment in which a system is running. To be fully dynamic, emergent runtime properties must influence not just what is computed but how a system computes what is computed. Open-ended facilities that enable a system to extend itself, based upon

self-representation, are viewed as mandatory characteristics of any system that can adapt to emergent properties. If a reflective substrate is utilized, it must support metaobject representations that evolve in phase with an adaptive system's changing computing environment.

A major future challenge is the investigation of context-aware component-based software systems that can continually dynamically reconfiguration themselves to stay in phase with their changing computing environments [104].

Appendix A – Financial Analysis System Case Study

The theme of this example is the analysis of volatile environmental information based on systems dynamic interoperation.

A.1. Introduction

A Financial Analysis System (FAS) is a system of financial relationships and cash flows. This kind of system is considered a complex modeling domain in which adaptability is required because of the volatile nature of the world of finance. Within this context effective investment, operating, and financing business decisions are ultimately expressed as a linkage between cash flows and the creation of economic value. Valuation itself is in essence a future-oriented process that encompasses projecting current economic performance over a time span applicable to the nature of a business activity and its market segment. To do this, there is often a distinction made between short-term and long-term prediction and any analytic outputs may be indicator and/or magnitude based information. It is noted that valuation analysis is best accomplished in conjunction with a cash flow framework in order to derive ranges of valuation [60, 61, 105].

Figure 118. Broad Context of Financial Analysis

Cash flow is typically represented by net investment, operating cash inflow, economic life and terminal value (using time adjusted measures). Methods of cash flow analysis include relatively simple payback or Return on Investment (ROI) calculations or more complex economic investment measures such as net present value, present value payback, profitability index, internal rate of return, or annualized net present value.

Valuation is concerned with whether shareholder value is being created and is primarily based on the present value analysis of future cash flows. Definitions of different value types include economic, market, book, liquidation, break-up, reproduction, collateral, assessed, appraised, going concern, and shareholder value.

This application-specific example encompasses the challenge of the continual reconciliation of user-defined needs and computing environment requirements. It is noted that a FAS must provide effective modeling techniques that facilitate the integration of prior knowledge and adaptation to changing market conditions. In this context, data-driven change, that minimizes human intervention and bias, must be systematically integrated with human-driven information, that is the result of naturally adaptive and

perceptive processes. The challenge may be summarized as the need for techniques for incorporating prior knowledge and the need for techniques for enhancing or augmenting this prior knowledge through systematic evolution.

The following example shows the coordinated evolution of two FASs (FAS-A and FAS-B). This evolution is represented by the dynamic integration of their respective Scenario Analysis Sub-systems (SASs) in order to provide an improved capability for their respective clients. Each FAS specializes in maintaining knowledge and providing predictive scenarios about the energy market sector. FAS-A is oriented to a client base that is concerned with longer-term indicator based information with respect to bond and preferred stock values. FAS-B is oriented to a client base that is concerned with shorter-term indicator and magnitude based information with respect to preferred stock and common stock values.

The following sections compose Appendix A: Section A.2 - A description of the original systems; Section A.3 - A summary of their planned evolution; and Section A.4 - A description of how each system is dynamically reconfigured.

A.2. Description of Original Systems

This section describes FAS-A and FAS-B. For each system, the following is presented:

- User context description of the main features required by the investor user community;
- A pattern definition template from [70] is adopted to describe the architectural style that guides the overall design of the system. The architecture style description is meant to be generic but within the context of financial analysis systems;

- A list of system use cases and run-time component descriptions that show Use Case Maps [62] and required and provided services; and
- A coarse-grained description of each system component.

The following are examples of global properties:

- ◆ FAS-GP1: All control flow audit records must be synchronized and ordered;
- ◆ FAS-GP2: Subset of state elements of subset of components may only be updated in a synchronized fashion;
- ◆ FAS-GP3: Subset of operations of a subset of system components must together provide needed behavior within limited time constraints;
- ◆ FAS-GP4: Preferred (global) ordering of component's operations invocation;
- ◆ FAS-GP5: Use Case Maps (UCMs) will describe high level causal flow that, which represents the control style of the system as a whole;
- ◆ FAS-GP6: Streamed updates to financial conditions information store must not take longer than X time units from time generated; and
- ◆ FAS-GP7: On-line data source systems, intermediate analysis systems, and KB components form a pipeline (they form a single chain and not a ring).

The following are examples of local properties:

- ◆ FAS-LP1: If updates to knowledge representations collide, human provided information must take precedence;
- ◆ FAS-LP2: Only control flow initiator can and must create audit record that traces control flow activities;
- ◆ FAS-LP3: Cardinality constraint that a provided service has at most one required service bound to it and vice-versa;

- ◆ FAS-GP4: Immutability constraint that the values of a certain subset of its state elements may not change (i.e. not allowed to change values of financial conditions information store);
- ◆ FAS-GP5: Coupling/cohesion constraint that the operations of a component must be continually re-factored to ensure common behavior is factored out as distinct operations;
- ◆ FAS-GP6: Cardinality constraint that a component has an upper bound on the number of service invocations that may be spawned in response to remote service requests;
- ◆ FAS-GP7: Must ensure longer or shorter term valuations not invalidated by current financial conditions (i.e. analysis performed within Y time units); and
- ◆ FAS-GP8: Must ensure local client requests are given priority over remote requests.

A.2.1 FAS-A Description

This sub-section provides summary descriptions of user context, architectural style and constraints, a list of use cases, and the run-time components of FAS-A.

A.2.1.1 FAS-A User Context

Clients of FAS-A are interested in the energy market sector, bond values, preferred stock values, indicator based information outcomes, and long-term predictions.

A.2.1.2 FAS-A Architectural Style

To build a system that meets their clients' needs, the designers of FAS-A adopted the 'Off-line Financial Analysis' architectural style described below.

Architectural Style: Off-line Financial Analysis

Problem: The ability to select appropriate financial analysis capability for longer-term economic predictions.

Context: A variety of financial analysis methods may be required when performing valuation based on changing financial conditions.

Forces: The ability to flexibly compose financial analysis capability allows more precise valuation scenarios to be computed on-demand for longer term indicators based on a current view of changing financial conditions.

Solution: Adopt off-line financial analysis architectural style as shown generically in the diagram below.

Figure 119. Off-Line Financial Analysis Architectural Style

Consequences: The off-line financial analysis approach allows valuation strategies to be formulated and applied dynamically as a function of a current view of financial conditions. However, the timeliness of the results of this analysis may affect their validity since financial conditions may have changed and invalidated them in the mean time.

A.2.1.3 FAS-A Financial Analysis Use Cases

Here is a list of use cases for FAS-A:

UC-A1. Establish prior knowledge about the energy market sector based on existing models.

UC-A2. Augment prior models with knowledge that is implicit to current financial conditions.

UC-A3. Infer long-term bond values based on cash flow projections and valuation assessment.

UC-A4. Infer long-term preferred stock values based on cash flow projections and valuation assessment.

UC-A5. Reconcile financial analysis scenarios to generate long-term key indicator information.

A.2.1.4 Run Time Components with High Level Use Case Maps

The following Use Case Maps (UCMs) describe high level causal flow, which represent the control style of the system as a whole.

Figure 120. FAS-A Use Case Maps

A.2.1.5 Run Time Components View showing Required/Provided Services

Each run-time component is described below by its required and/or provided services (and the service protocol implemented by each service), internal operations, and state elements.

Figure 121. FAS-A Provided and Required Services

A.2.1.5.1 Financial Portfolio Manager

Description: The *Financial Portfolio Manager* is a mechanism for a user community that is interested in bond and preferred stock investment opportunities as informed by longer-term financial indicator based information predictions about the energy market sector. Provides needed bond and preferred stock portfolio management capability.

Dependencies

Global Properties: FAS-GP1, FAS-GP3, FAS-GP4, and FAS-GP5.

Local Properties: FAS-LP2, FAS-LP3, FAS-LP5, and FAS-LP6.

Structural Signature

Component to component dependencies: FAS-A -> SAS-A.

Required service to provided service dependencies: P0 -> P1.

Behavioral Signature

Service to service protocol dependencies: P0 -> T1.

Operation to required service dependencies: LongTermKeyIndicatorsInformation() -> ServiceRequiredVia(P0:T1).

Operation to provided service dependencies: None shown.

Operation to operation dependencies: None shown.

Operation to state element dependencies: DiscoverInvestmentOpportunities() ->

PreferredStockPortfolio and DiscoverInvestmentOpportunities() -> BondPortfolio.

Operation to composite component service dependencies: None shown.

External Interactions

Required Services: P0 implements service protocol T1: Long Term Key Indicators Information.

Provided Services: None shown.

Internal Behavior

Operations:

```
DiscoverInvestmentOpportunities()
    ServiceRequiredVia(P0:T1): LongTermKeyIndicatorsInformation();
    Action(MakeInvestmentDecisions);
    Update(PreferredStockPortfolio, BondPortfolio).
```

State Elements: PreferredStockPortfolio and BondPortfolio.

A.2.1.5.2 Scenario Analysis Manager

Description: The Scenario Analysis Manager is responsible for generating longer-term indicator-based bond and preferred stock values predictions. These predictions are based

on current market conditions available through the energy sector knowledge base component. Specialized financial analysis components are utilized as necessary. The longer-term predicted values are then made available to clients. The energy sector knowledge representations are updated accordingly.

Dependencies

Global Properties: FAS-GP1, FAS-GP2, FAS-GP4, and FAS-GP5.

Local Properties: FAS-LP2, FAS-LP3, FAS-LP5, FAS-LP6, and FAS-LP7.

Structural Signature

Component to component dependencies: SAS-A -> FAS-A, SAS-A -> Energy Sector KB, SAS-A -> Cash Flow Projections, and SAS-A -> Valuation Assessment.

Required service to provided service dependencies: P1 -> P0, P2 -> P3, P4 -> P5, P8 -> P9, and P10 -> P11.

Behavioral Signature

Service to service protocol dependencies: P1 -> T1, P2 -> T2, P4 -> T3, P8 -> T5, and P10 -> T6.

Operation to required service dependencies: PriorKnowledgeOfEnergySector() ->

ServiceRequiredVia(P2:T2), LongTermCashFlowProjections() ->

ServiceRequiredVia(P10:T6), LongTermValuationAssessment() ->

ServiceRequiredVia(P8:T5), and AugmentPriorKnowledgeOfEnergySector() ->

ServiceRequiredVia(P4:T3).

Operation to provided service dependencies: LongTermKeyIndicatorsInformation() ->

ServiceProvidedVia(P1:T1).

Operation to operation dependencies: GenerateLongTermValues() ->

GenerateLongTermBondValues(), GenerateLongTermValues() ->

GenerateLongTermPreferredStockValues (), and GenerateLongTermValues() ->

AugmentPriorKnowledgeOfEnergySector ().

Operation to state element dependencies: LongTermKeyIndicatorsInformation() ->

LongTermPreferredStockValues, LongTermKeyIndicatorsInformation() ->

LongTermBondValues, AugmentPriorKnowledgeOfEnergySector() ->

LongTermBondValues, AugmentPriorKnowledgeOfEnergySector() ->

LongTermPreferredStockValues, GenerateLongTermBondValues() ->

LongTermBondValues, and GenerateLongTermPreferredStockValues() ->

LongTermPreferredStockValues.

Operation to composite component service dependencies: None shown.

External Interactions

Required Services: P2 implements service protocol T2: Prior Knowledge of Energy

Sector, P4 implements service protocol T3: Augment Prior Knowledge of Energy Sector,

P8 implements service protocol T5: Long Term Valuation Assessment, and P10

implements service protocol T6: Long Term Cash Flow Projections.

Provided Services: P1 implements service protocol T1: Long Term Key Indicators

Information.

Internal Behavior

Operations:

LongTermKeyIndicatorsInformation()

Result(LongTermPreferredStockValues, LongTermBondValues).

GenerateLongTermValues()

```

ServiceRequiredVia (P2:T2) : PriorKnowledgeOfEnergySector ();
ServiceRequiredVia (P10:T6) : LongTermCashFlowProjections ();
ServiceRequiredVia (P8:T5) : LongTermValuationAssessment ();
GenerateLongTermBondValues ();
GenerateLongTermPreferredStockValues ();
AugmentPriorKnowledgeOfEnergySector ().

AugmentPriorKnowledgeOfEnergySector ()

    Read (LongTermBondValues, LongTermPreferredStockValues);
    Action (InferKnowledge);
    ServiceRequiredVia (P4:T3) : AugmentPriorKnowledgeOfEnergySector ().

GenerateLongTermBondValues ()

    Action (GenerateBondValues);
    Update (LongTermBondValues).

GenerateLongTermPreferredStockValues ()

    Action (GeneratePreferredStockValues);
    Update (LongTermPreferredStockValues).

```

State Elements: LongTermPreferredStockValues and LongTermBondValues.

A.2.1.5.3 On Line Energy Sector Conditions Generator

Description: On-Line Energy Sector Conditions Generator is responsible for generating raw information about energy sector market conditions. This data is retained over time and becomes the basis for follow-on analysis methods.

Dependencies

Global Properties: FAS-GP1, FAS-GP3, FAS-GP4, FAS-GP5, and GP-FAS6.

Local Properties: FAS-LP2, FAS-LP3, FAS-LP5, and FAS-LP6.

Structural Signature

Component to component dependencies: On-Line Energy Sector Conditions -> Energy Sector KB.

Required service to provided service dependencies: P6 -> P7.

Behavioral Signature

Service to service protocol dependencies: P6 -> T4.

Operation to required service dependencies: CurrentEnergySectorMarketConditions() -> ServiceRequiredVia(P6:T4).

Operation to provided service dependencies: None shown.

Operation to operation dependencies: None shown.

Operation to state element dependencies:

GenerateCurrentEnergySectorFinancialConditions() ->

FinancialConditionsDataInputStream

Operation to composite component service dependencies: None shown.

External Interactions

Required Services: P6 implements service protocol T4: Current Energy Sector Market Conditions.

Provided Services: None shown.

Internal Behavior

Operations:

```
GenerateCurrentEnergySectorFinancialConditions ()
    Read (FinancialConditionsDataInputStream) ;
    ServiceRequiredVia (P6:T4) : CurrentEnergySectorMarketConditions () .
```

State Elements: FinancialConditionsDataInputStream.

A.2.1.5.4 Energy Sector Knowledge Base

Description: *Energy Sector Knowledge Base* is responsible for managing raw data about the energy sector that is retained over time as immutable information. In addition, responsible for retaining energy sector knowledge representations inferred from this raw information. The knowledge representations may change and their composition must ultimately be governed by human-driven information.

Dependencies

Global Properties: FAS-GP1, FAS-GP3, FAS-GP4, FAS-GP5, and FAS-GP6.

Local Properties: FAS-LP1, FAS-LP3, FAS-LP4, FAS-LP5, and FAS-LP6.

Structural Signature

Component to component dependencies: Energy Sector KB -> On-Line Energy Sector Conditions and Energy Sector KB -> SAS-A.

Required service to provided service dependencies: P7 -> P6, P3 -> P2, and P5 -> P4.

Behavioral Signature

Service to service protocol dependencies: P7 -> T4, P3 -> T2, and P5 -> T3.

Operation to required service dependencies: None shown.

Operation to provided service dependencies: CurrentEnergySectorMarketConditions() ->

ServiceProvidedVia(P7:T4), PriorKnowledgeOfEnergySector () ->

ServiceProvidedVia(P3:T2), and AugmentPriorKnowledgeOfEnergySector () ->

ServiceProvidedVia(P5:T3).

Operation to operation dependencies: CurrentEnergySectorMarketConditions() ->

ComposeKnowledgeRepresentations() and ComposeKnowledgeRepresentations() ->

AugmentPriorKnowledgeOfEnergySector().

Operation to state element dependencies: `PriorKnowledgeOfEnergySector()` ->

`FinancialConditionsData, PriorKnowledgeOfEnergySector()` ->

`EnergySectorKnowledgeRepresentations, AugmentPriorKnowledgeOfEnergySector()` ->

`EnergySectorKnowledgeRepresentations, CurrentEnergySectorMarketConditions()` ->

`FinancialConditionsData, and ComposeKnowledgeRepresentations()` ->

`FinancialConditionsData.`

Operation to composite component service dependencies: None shown.

External Interactions

Required Services: None shown.

Provided Services: P7 implements service protocol T4: Current Energy Sector Market

Conditions, P3 implements service protocol T2: Prior Knowledge of Energy Sector, and

P5 implements service protocol T3: Augment Prior Knowledge of Energy Sector.

Internal Behavior

Operations:

`CurrentEnergySectorMarketConditions()`

`Update(FinancialConditionsData);`

`ComposeKnowledgeRepresentations().`

`ComposeKnowledgeRepresentations()`

`Read(FinancialConditionsData);`

`Action(InferKnowledge);`

`AugmentPriorKnowledgeOfEnergySector().`

`PriorKnowledgeOfEnergySector()`

`Return(FinancialConditionsData,`

`EnergySectorKnowledgeRepresentations).`

`AugmentPriorKnowledgeOfEnergySector()`

`Update(EnergySectorKnowledgeRepresentations).`

State Elements: FinancialConditionsData and EnergySectorKnowledgeRepresentations.

A.2.1.5.5 Valuation Assessment Manager

Description: *Valuation Assessment Manager* is responsible for projecting current economic performance over a time-span applicable the business within the energy market sector. This projection may be formulated in conjunction with available cash flow information in order to derive ranges of valuation based on the present value analysis of future cash flows.

Dependencies

Global Properties: FAS-GP1, FAS-GP2, FAS-GP4, and FAS-GP5.

Local Properties: FAS-LP3, FAS-LP5, and FAS-LP6.

Structural Signature

Component to component dependencies: Valuation Assessment Manager -> SAS-A.

Required service to provided service dependencies: P9 -> P8.

Behavioral Signature

Service to service protocol dependencies: P9 -> T5.

Operation to required service dependencies: None shown.

Operation to provided service dependencies: LongTermValuationAssessment() ->

ServiceProvidedVia(P9:T5).

Operation to operation dependencies: LongTermValuationAssessment() ->

PresentValueAnalysisOfFutureCashFlows().

Operation to state element dependencies: PresentValueAnalysisOfFutureCashFlows() ->

PresentValueAnalysisData.

Operation to composite component service dependencies: None shown.

External Interactions

Required Services: None shown.

Provided Services: P9 implements service protocol T5: Long Term Valuation Assessment.

Internal Behavior

Operations:

```
LongTermValuationAssessment()
    PresentValueAnalysisOfFutureCashFlows();
    Action(ComputeLongTermAssessment).
PresentValueAnalysisOfFutureCashFlows()
    Read(PresentValueAnalysisData);
    Action(PerformPresentValueAnalysis).
```

State Elements: PresentValueAnalysisData.

A.2.1.5.6 Cash Flow Projections Manager

Description: *Cash Flow Projections Manager* is responsible for providing future cash flows financial scenarios based on appropriate methods of cash flow analysis and time adjusted measures. Simpler methods include simple payback or Return on Investment (ROI) calculations. More complex economic investment measures include net present value, present value payback, profitability index, internal rate of return, or annualized new present value.

Dependencies

Global Properties: FAS-GP1, FAS-GP2, FAS-GP4, and FAS-GP5.

Local Properties: FAS-LP3, FAS-LP5, and FAS-LP6.

Structural Signature

Component to component dependencies: Cash Flow Projections -> SAS-A.

Required service to provided service dependencies: P11 -> P10.

Behavioral Signature

Service to service protocol dependencies: P11 -> T6.

Operation to required service dependencies: None shown.

Operation to provided service dependencies: LongTermCashFlowProjections() ->

ServiceProvidedVia(P11:T6).

Operation to operation dependencies: LongTermCashFlowProjections() -> ROI(),

LongTermCashFlowProjections() -> PayBack(), and LongTermCashFlowProjections() ->

NetPresentValue().

Operation to state element dependencies: ROI() -> CashFlowProjectionsData, PayBack ()

-> CashFlowProjectionsData, and NetPresentValue () -> CashFlowProjectionsData.

Operation to composite component service dependencies: None shown.

External Interactions

Required Services: None shown.

Provided Services: P11 implements service protocol T6: Long Term Cash Flow

Projections.

Internal Behavior

Operations:

```
LongTermCashFlowProjections ()
```

```
    ROI ();
```

```
    PayBack ();
```

```
    NetPresentValue ();
```

```
    Action (ComputeCashFlowProjections).
```

```
ROI ()
```

```
    Read (CashFlowProjectionsData);
```

```

        Action (ComputeROI) .

PayBack ()

        Read (CashFlowProjectionsData) ;

        Action (ComputePayBack) .

NetPresentValue ()

        Read (CashFlowProjectionsData) ;

        Action (ComputeNetPresentValue) .

```

State Elements: CashFlowProjectionsData.

A.2.2 FAS-B Description

This sub-section provides summary descriptions of user context, architectural style and constraints, a list of use cases, and the run-time components of FAS-B.

A.2.2.1 FAS-B User Context

Clients of FAS-B are interested in the energy market sector, preferred stock values, common stock values, indicator based information outcomes, magnitude based information outcomes, and short-term predictions.

A.2.2.2 FAS-B Architectural Style

To build a system that meets their clients' needs, the designers of FAS-B adopted the '*On-line Financial Analysis*' architectural style described below.

Architectural Style: On-line Financial Analysis.

Problem: The ability to select appropriate financial analysis capability for shorter-term economic predictions.

Context: A variety of financial analysis methods may be required when performing valuation based on changing financial conditions.

Forces: The ability to deploy financial analysis capability in-line with financial conditions data sources allows more timely valuation scenarios to be computed for shorter term indicators and magnitude predictions for changing financial conditions.

Solution: Adopt on-line financial analysis architectural style as shown generically in the diagram below.

Figure 122. FAS-B On-Line Financial Analysis Architectural Style

Consequences: The on-line financial analysis approach allows valuation strategies to be formulated and applied as a function of current financial conditions. However, the timeliness of the results of this approach is typically based on fixed configurations of analytic capability and is therefore a trade-off with the possible need for more flexible analytic strategies. The lack of flexibility may affect the validity of results because financial conditions may change and invalidate the original analysis strategies over time.

A.2.2.3 FAS-B Financial Analysis Use Cases

Here is a list of use cases for FAS-B:

UC-B1. Establish prior knowledge about the energy market sector based on existing models.

UC-B2. Augment prior models with knowledge that is implicit to current financial conditions.

UC-B3. Infer short-term preferred stock values based on on-line cash flow projections and valuation assessment.

UC-B4. Infer short-term common stock values based on on-line cash flow projections and valuation assessment.

UC-B5. Reconcile financial analysis scenarios to generate short-term key indicator and magnitude based information.

A.2.2.4 Run Time Components with High Level Use Case Maps

The following Use Case Maps (UCMs) describe high level causal flow, which represent the control style of the system as a whole.

Figure 123. FAS-B Use Case Maps

A.2.2.5 Run Time Components View showing Required/Provided Services

Each run-time component is represented by required and/or provided services (and the service protocol implemented by each service), internal operations, and state elements.

A.2.2.5.1 Financial Portfolio Manager

Description: *Financial Portfolio Manager* is a mechanism for a user community that is interested in common and preferred stock investment opportunities as informed by shorter-term financial indicator and magnitude based information predictions about the energy market sector. Provides needed common and preferred stock portfolio management capability.

Dependencies

Global Properties: FAS-GP1, FAS-GP3, FAS-GP4, and FAS-GP5.

Local Properties: FAS-LP2, FAS-LP3, FAS-LP5, and FAS-LP6.

Structural Signature

Component to component dependencies: FAS-B -> SAS-B.

Required service to provided service dependencies: P12 -> P13.

Behavioral Signature

Service to service protocol dependencies: P12 -> T7.

Operation to required service dependencies:

ShortTermKeyIndicatorsAndMagnitudeBasedInformation () ->

ServiceRequiredVia(P12:T7).

Operation to provided service dependencies: None shown.

Operation to operation dependencies: None shown.

Operation to state element dependencies: DiscoverInvestmentOpportunities() ->

PreferredStockPortfolio and DiscoverInvestmentOpportunities() ->

CommonStockPortfolio.

Operation to composite component service dependencies: None shown.

External Interactions

Required Services: P12 implements service protocol T7: Short Term Key Indicators and Magnitude Based Information.

Provided Services: None shown.

Internal Behavior

Operations:

DiscoverInvestmentOpportunities()

ServiceRequiredVia(P12:T7):

ShortTermKeyIndicatorsAndMagnitudeBasedInformation();

Action(MakeInvestmentDecisions);

Update(PreferredStockPortfolio, CommonStockPortfolio).

State Elements: PreferredStockPortfolio and CommonStockPortfolio.

A.2.2.5.2 Scenario Analysis Manager

Description: *Scenario Analysis Manager* is responsible for generating shorter-term indicator and magnitude based bond and preferred stock values predictions. These predictions are based on current market conditions available through the energy sector knowledge base component. The shorter-term predicted values are then made available to clients. The energy sector knowledge representations are updated accordingly.

Dependencies

Global Properties: FAS-GP1, FAS-GP3, FAS-GP4, and FAS-GP5.

Local Properties: FAS-LP2, FAS-LP3, FAS-LP5, FAS-LP6, and FAS-LP7.

Structural Signature

Component to component dependencies: SAS-B -> FAS-B and SAS-B -> Energy Sector KB.

Required service to provided service dependencies: P13 -> P12, P14 -> P15, and P16 -> P17.

Behavioral Signature

Service to service protocol dependencies: P14 -> T8, P16 -> T9, and P13 -> T7.

Operation to required service dependencies: PriorKnowledgeOfEnergySector () ->

ServiceRequiredVia(P14:T8) and AugmentPriorKnowledgeOfEnergySector() ->

ServiceRequiredVia(P16:T9).

Operation to provided service dependencies:

ShortTermKeyIndicatorsAndMagnitudeBasedInformation() ->

ServiceProvidedVia(P13:T7).

Operation to operation dependencies: GenerateShortTermValues() ->

GenerateShortTermCommonStockValues(), GenerateShortTermValues() ->

GenerateShortTermPreferredStockValues(), and GenerateShortTermValues() ->

AugmentPriorKnowledgeOfEnergySector().

Operation to state element dependencies:

ShortTermKeyIndicatorsAndMagnitudeBasedInformation() ->

ShortTermPreferredStockValues,

ShortTermKeyIndicatorsAndMagnitudeBasedInformation() ->

ShortTermCommonStockValues, AugmentPriorKnowledgeOfEnergySector() ->

ShortTermPreferredStockValues, AugmentPriorKnowledgeOfEnergySector() ->

ShortTermCommonStockValues, GenerateShortTermCommonStockValues() ->

ShortTermCommonStockValues, and GenerateShortTermPreferredStockValues() ->

ShortTermPreferredStockValues.

Operation to composite component service dependencies: None shown.

External Interactions

Required Services: P14 implements service protocol T8: Prior Knowledge of Energy Sector and P16 implements service protocol T9: Augment Prior Knowledge of Energy Sector.

Provided Services: P13 implements service protocol T7: Short Term Key Indicators and Magnitude Based Information.

Internal Behavior

Operations:

```
ShortTermKeyIndicatorsAndMagnitudeBasedInformation()
```

```
    Return (ShortTermPreferredStockValues,
```

```
    ShortTermCommonStockValues).
```

```
GenerateShortTermValues()
```

```
    ServiceRequiredVia (P14:T8): PriorKnowledgeOfEnergySector();
```

```
    GenerateShortTermCommonStockValues();
```

```
    GenerateShortTermPreferredStockValues();
```

```
    AugmentPriorKnowledgeOfEnergySector().
```

```
AugmentPriorKnowledgeOfEnergySector()
```

```
    Read (ShortTermPreferredStockValues, ShortTermCommonStockValues);
```

```

    Action(InferKnowledge);

    ServiceRequiredVia(P16:T9):

AugmentPriorKnowledgeOfEnergySector().

GenerateShortTermCommonStockValues()

    Action(GenerateCommonStockValues);

    Update(ShortTermCommonStockValues).

GenerateShortTermPreferredStockValues()

    Action(GeneratePreferredStockValues);

    Update(ShortTermPreferredStockValues).

```

State Elements: ShortTermPreferredStockValues and ShortTermCommonStockValues.

A.2.2.5.3 On Line Energy Sector Conditions Generator

Description: On-Line Energy Sector Conditions Generator is responsible for generating raw information about energy sector market conditions. This data will be retained over time and will become the basis for follow-on analysis methods.

Dependencies

Global Properties: FAS-GP1, FAS-GP3, FAS-GP4, FAS-GP5, FAS-GP6, and FAS-GP7.

Local Properties: FAS-LP2, FAS-LP3, FAS-LP5, and FAS-LP6.

Structural Signature

Component to component dependencies: On-Line Energy Sector Conditions -> Cash Flow Projections.

Required service to provided service dependencies: P22 -> P23.

Behavioral Signature

Service to service protocol dependencies: P22 -> T12.

Operation to required service dependencies: CurrentEnergySectorMarketConditions() ->

ServiceRequiredVia(P22:T12).

Operation to provided service dependencies: None shown.

Operation to operation dependencies: None shown.

Operation to state element dependencies:

GenerateCurrentEnergySectorFinancialConditions() ->

FinancialConditionsDataInputStream.

Operation to composite component service dependencies: None shown.

External Interactions

Required Services: P22 implements service protocol T12: Current Energy Sector Market Conditions.

Provided Services: None shown.

Internal Behavior

Operations:

```
GenerateCurrentEnergySectorFinancialConditions()
    Read(FinancialConditionsDataInputStream);
    ServiceRequiredVia(P22:T12),
CurrentEnergySectorMarketConditions().
```

State Elements: FinancialConditionsDataInputStream.

A.2.2.5.4 Energy Sector Knowledge Base

Description: *Energy Sector Knowledge Base* is responsible for managing raw data about the energy sector that is retained over time as immutable information. In addition, responsible for retaining energy sector knowledge representations inferred from this raw information. The knowledge representations may change and their composition must ultimately be governed by human-driven information.

Dependencies

Global Properties: FAS-GP1, FAS-GP3, FAS-GP4, FAS-GP5, FAS-GP6, and FAS-GP7.

Local Properties: FAS-LP1, FAS-LP3, FAS-LP4, FAS-LP5, and FAS-LP6.

Structural Signature

Component to component dependencies: Energy Sector KB -> Valuation Assessment and Energy Sector KB -> SAS-B.

Required service to provided service dependencies: P19 -> P18, P15 -> P14, and P17 -> P16.

Behavioral Signature

Service to service protocol dependencies: P19 -> T10, P15 -> T8, and P17 -> T9.

Operation to required service dependencies: None shown.

Operation to provided service dependencies: PriorKnowledgeOfEnergySector() ->

ServiceProvidedVia(P15:T8), AugmentPriorKnowledgeOfEnergySector() ->

ServiceProvidedVia(P17:T9), and ShortTermValuationAssessment() ->

ServiceProvidedVia(P19:T10).

Operation to operation dependencies: ShortTermValuationAssessment() ->

ComposeKnowledgeRepresentations() and ComposeKnowledgeRepresentations() ->

AugmentPriorKnowledgeOfEnergySector().

Operation to state element dependencies: PriorKnowledgeOfEnergySector() ->

FinancialConditionsData, PriorKnowledgeOfEnergySector() ->

EnergySectorKnowledgeRepresentations, AugmentPriorKnowledgeOfEnergySector() ->

EnergySectorKnowledgeRepresentations, and ShortTermValuationAssessment() ->

FinancialConditionsData, and ComposeKnowledgeRepresentations() ->

FinancialConditionsData.

Operation to composite component service dependencies: None shown.

External Interactions

Required Services: None shown.

Provided Services: P15 implements service protocol T8: Prior Knowledge of Energy Sector, P17 implements service protocol T9: Augment Prior Knowledge of Energy Sector, and P19 implements service protocol T10: Short Term Valuation Assessment.

Internal Behavior

Operations:

```
ShortTermValuationAssessment ()
    Update (FinancialConditionsData);
    ComposeKnowledgeRepresentations () .
ComposeKnowledgeRepresentations ()
    Read (FinancialConditionsData);
    Action (InferKnowledge);
    AugmentPriorKnowledgeOfEnergySector () .
PriorKnowledgeOfEnergySector ()
    Return (FinancialConditionsData,
EnergySectorKnowledgeRepresentations) .
AugmentPriorKnowledgeOfEnergySector ()
    Update (EnergySectorKnowledgeRepresentations) .
```

State Elements: FinancialConditionsData and EnergySectorKnowledgeRepresentations.

A.2.2.5.5 Valuation Assessment Manager

Description: *Valuation Assessment Manager* is responsible for projecting current economic performance over a time-span applicable the business within the energy market sector. This projection may be formulated in conjunction with available cash flow

information in order to derive ranges of valuation based on the present value analysis of future cash flows.

Dependencies

Global Properties: FAS-GP1, FAS-GP3, FAS-GP4, FAS-GP5, FAS-GP6, and FAS-GP7.

Local Properties: FAS-LP2, FAS-LP3, FAS-LP5, and FAS-LP6.

Structural Signature

Component to component dependencies: Valuation Assessment Manager -> Cash Flow Projections and Valuation Assessment Manager -> Energy Sector KB.

Required service to provided service dependencies: P21 -> P20 and P18 -> P19.

Behavioral Signature

Service to service protocol dependencies: P18 -> T10 and P21 -> T11.

Operation to required service dependencies: ShortTermValuationAssessment() -> ServiceRequiredVia(P18:T10).

Operation to provided service dependencies: ShortTermCashFlowProjections() -> ServiceProvidedVia(P21:T11).

Operation to operation dependencies: ShortTermCashFlowProjections() -> ShortTermValuationAssessment() and ShortTermValuationAssessment() -> PresentValueAnalysisOfFutureCashFlows().

Operation to state element dependencies: PresentValueAnalysisOfFutureCashFlows() -> PresentValueAnalysisData.

Operation to composite component service dependencies: None shown.

External Interactions

Required Services: P18 implements service protocol T10: Short Term Valuation Assessment.

Provided Services: P21 implements service protocol T11: Short Term Cash Flow Projections.

Internal Behavior

Operations:

```
ShortTermCashFlowProjections()
    ShortTermValuationAssessment().
ShortTermValuationAssessment()
    PresentValueAnalysisOfFutureCashFlows();
    Action(ComputeShortTermAssessment);
    ServiceRequiredVia(P18:T10); ShortTermValuationAssessment().
PresentValueAnalysisOfFutureCashFlows()
    Read(PresentValueAnalysisData);
    Action(PerformPresentValueAnalysis).
```

State Elements: PresentValueAnalysisData.

A.2.2.5.6 Cash Flow Projections Manager

Description: *Cash Flow Projections Manager* is responsible for providing future cash flows financial scenarios based on appropriate methods of cash flow analysis and time adjusted measures. Simpler methods include simple payback or Return on Investment (ROI) calculations. More complex economic investment measures include net present value, present value payback, profitability index, internal rate of return, or annualized new present value.

Dependencies

Global Properties: FAS-GP1, FAS-GP3, FAS-GP4, FAS-GP5, FAS-GP6, and FAS-GP7.

Local Properties: FAS-LP2, FAS-LP3, FAS-LP5, and FAS-LP6.

Structural Signature

Component to component dependencies: Cash Flow Projections -> On-Line Energy

Sector Conditions and Cash Flow Projections -> Valuation Assessment.

Required service to provided service dependencies: P23 -> P22 and P20 -> P21.

Behavioral Signature

Service to service protocol dependencies: P20 -> T11 and P23 -> T12.

Operation to required service dependencies: ShortTermCashFlowProjections() ->

ServiceRequiredVia(P20:T11).

Operation to provided service dependencies: CurrentEnergySectorMarketConditions() ->

ServiceProvidedVia(P23:T12).

Operation to operation dependencies: CurrentEnergySectorMarketConditions() ->

ShortTermCashFlowProjections(), ShortTermCashFlowProjections() -> ROI(),

ShortTermCashFlowProjections() -> PayBack() and ShortTermCashFlowProjections() ->

NetPresentValue().

Operation to state element dependencies: ROI() -> CashFlowProjectionsData, PayBack()

-> CashFlowProjectionsData and NetPresentValue() -> CashFlowProjectionsData.

Operation to composite component service dependencies: None shown.

External Interactions

Required Services: P20 implements service protocol T11: Short Term Cash Flow Projections.

Provided Services: P23 implements service protocol T12: Current Energy Sector Market Conditions.

Internal Behavior

Operations:

```

CurrentEnergySectorMarketConditions ()
    ShortTermCashFlowProjections () .

ShortTermCashFlowProjections ()
    ROI () ;
    PayBack () ;
    NetPresentValue () ;
    Action (ComputeCashFlowProjections) ;
    ServiceRequiredVia (P20:T11), ShortTermCashFlowProjections () .

ROI ()
    Read (CashFlowProjectionsData) ;
    Action (ComputeROI) .

PayBack ()
    Read (CashFlowProjectionsData) ;
    Action (ComputePayBack) .

NetPresentValue ()
    Read (CashFlowProjectionsData) ;
    Action (ComputeNetPresentValue) .

```

State Elements: CashFlowProjectionsData.

A.3. Planned Evolution of FAS-A and FAS-B

The following is a description of the planned evolution of these two systems:

- FAS-A and FAS-B together provide an improved capability for all clients. FAS-A's strengths are based on supporting sub-system capability that provides services for long-term valuation assessments and cash flow projections for the energy sector. FAS-B's strengths are based on supporting sub-system capability that provides

services for short-term valuation assessments and cash flow projections for the same sector. FAS-A + FAS-B together provide a combined capability that can leverage the facilities from all supporting sub-systems.

- System evolution is localized to SAS-a and SAS-b and is governed by the respective architectural constraints of each system.
- In the case of FAS-A, SAS-a evolves in place by updating its external interaction facility to support the required interfaces and new types of information that are exchanged. In addition, SAS-a changes its internal behavior in order to support scenario analysis that takes into account the new information being exchanged. The evolution of SAS-a is constrained to ensure all changes were backward compatible to all existing clients; when the evolution was complete the clients of FAS-A are the beneficiaries of improved scenario analysis results.
- In the case of FAS-B, SAS-b' is substituted for SAS-b. The new SAS-b' is implemented with an external interaction facility to support the required interfaces and new types of information that are exchanged and internal behavior to support scenario analysis that takes into account the new information being exchanged. The substitution of SAS-b' for SAS-b is constrained to ensure all changes were backward compatible to all existing clients; when the evolution is complete the clients of FAS-B are the beneficiaries of improved scenario analysis results.
- Given an assessment of the combined architectural constraints of both systems, the in-place evolution of SAS-a proceeds. FAS-A is now serving its clients as before but with an SAS that was ready to externally interact with a peer subsystem of FAS-B.

- Concurrently, also given an assessment of the combined architectural constraints of both systems, SAS-b' is implemented and then substituted for SAS-b. FAS-B is now be serving its clients as before but with an SAS that was ready to externally interact with a peer subsystem of FAS-A.
- The architectural constraints of each system are reconciled with respect to one another. For example it might be established that a remote request is be delegated by a SAS to a supporting subsystem at a lower priority than a local request.
- To enact the combined capabilities of both systems, an architectural connector is established between the changed SAS-a and the new SAS-b' and the internal architectural configurations of each system are updated. The two systems are now be ready to start interacting.
- The SAS of each FAS now produces new kinds of analytic results based on the new information that is now available to each system. The clients of each system are the recipients of this new scenario analysis when this information was released to them.

A.4. Dynamic Reconfiguration of FAS-A and FAS-B

This subsection presents the new representations of FAS-A and FAS-B, and describes how FAS-A and FAS-B evolved.

A.4.1 New Representations of FAS-A and FAS-B

This section is composed of the following subsections: Updated FAS-A financial analysis use cases, FAS-A SAS component evolution showing new use case map, Updated FAS-B financial analysis use cases, FAS-B SAS component evolution showing

new use case map and Run time components view showing new required/provided services.

A.4.1.1 FAS-A Financial Analysis Use Cases

Here is the new use case for FAS-A:

A6. Validate long-term preferred stock values predictions using short-term preferred stock values predictions.

A.4.1.2 FAS-A SAS Component Evolution Showing New Use Case Map

FAS-A SAS Component Evolution Showing New Use Case Map

Figure 125. FAS-A Evolved Use Case Maps

A.4.1.3 FAS-B Financial Analysis Use Cases

Here is the new use case for FAS-B:

B6. Validate short-term preferred stock values predictions using long-term preferred stock values predictions.

A.4.1.4 FAS-B SAS Component Evolution Showing New Use Case Map

FAS-B SAS Component Evolution Showing New Use Case Map

b5. determine current knowledge about market.	b13. determine long-term predicted values.
b6. provide current financial conditions and knowledge about market.	b14. provide long-term predicted values.
b7. update preferred stock and common stock value predictions.	b15. validate short-term predictions using long-term predicted values.
	b8. provide knowledge information about market sector.
	b9. update knowledge information about market sector.

Figure 126. FAS-B Evolved Use Case Maps

A.4.1.5 Run Time Components View showing New Required/Provided Services

Each run-time component is represented by required and/or provided services (and the service protocol implemented by each service), internal operations, and state elements.

Changed SAS-a and New SAS-b' Run Time Components Showing New Required/Provided Port Bindings

Figure 127. Evolved Required and Provided Services

A.4.1.5.1 Changed Scenario Analysis Manager (System A)

Changed Dependencies

Global Properties: FAS-GP5.

Local Properties: FAS-LP8.

Structural Signature

Component to component dependencies: SAS-A -> SAS-B.

Required service to provided service dependencies: P44 -> P45 and P56 -> P57.

Behavioral Signature

Service to service protocol dependencies: P44 -> T23 and P56 -> T29.

Operation to required service dependencies: ShortTermKeyPreferredStockValues() -> ServiceRequiredVia(P44:T23).

Operation to provided service dependencies: LongTermKeyPreferredStockValues() -> ServiceProvidedVia(P56:T29).

Operation to operation dependencies: LongTermPreferredStockValues() ->

LongTermKeyIndicatorsInformation() and GenerateLongTermPreferredStockValues()-> ValidateLongTermValuesBasedOnShortTermValues().

Operation to state element dependencies:

ValidateLongTermValuesBasedOnShortTermValues() ->

LongTermPreferredStockValues.

Operation to composite component service dependencies: None Shown.

Changed External Interactions

Required Services: P44 implements service protocol T23: Short Term Preferred Stock Values.

Provided Services: P56 implements service protocol T29: Long Term Preferred Stock Values.

Changed Internal Behavior

Operations:

LongTermPreferredStockValues()

LongTermKeyIndicatorsInformation().

```

GenerateLongTermPreferredStockValues ()

    ValidateLongTermValuesBasedOnShortTermValues () ;

    Action (GeneratePreferredStockValues) ;

    Update (LongTermPreferredStockValues) .

ValidateLongTermValuesBasedOnShortTermValues ()

    ServiceRequiredVia (P44:T23) : ShortTermPreferredStockValues () ;

    Read (LongTermPreferredStockValues) ;

    Action (ValidateLongTermValues) .

```

State Elements: LongTermPreferredStockValues.

A.4.1.5.2 New Scenario Analysis Manager (System B)

Changed Dependencies

Global Properties: FAS-GP5.

Local Properties: FAS-LP8.

Structural Signature

Component to component dependencies: SAS-B -> SAS-A.

Required service to provided service dependencies: P45 -> P44 and P57 -> P56.

Behavioral Signature

Service to service protocol dependencies: P45 -> T23 and P57 -> T29.

Operation to required service dependencies: LongTermPreferredStockValues() ->

ServiceRequiredVia(P57:T29).

Operation to provided service dependencies: ShortTermPreferredStockValues() ->

ServiceProvidedVia(P45:T23).

Operation to operation dependencies: ShortTermPreferredStockValues() ->

ShortTermKeyIndicatorsAndMagnitudeBasedInformation() and

GenerateShortTermPreferredStockValues() ->

ValidateLongTermValuesBasedOnShortTermValues().

Operation to state element dependencies:

ValidateShortTermValuesBasedOnLongTermValues() ->

ShortTermPreferredStockValues.

Operation to composite component service dependencies: None Shown.

Changed External Interactions

Required Services: P57 implements service protocol T29: Long Term Preferred Stock Values.

Provided Services: P45 implements service protocol T23: Short Term Preferred Stock Values.

Changed Internal Behavior

Operations:

ShortTermPreferredStockValues()

ShortTermKeyIndicatorsAndMagnitudeBasedInformation().

GenerateShortTermPreferredStockValues()

ValidateShortTermValuesBasedOnLongTermValues()

Action(GeneratePreferredStockValues);

Update(ShortTermPreferredStockValues).

ValidateShortTermValuesBasedOnLongTermValues()

ServiceRequiredVia(P44:T23): LongTermPreferredStockValues();

Read(ShortTermPreferredStockValues);

Action(ValidateShortTermValues).

State Elements: ShortTermPreferredStockValues.

A.4.2 How FAS-A and FAS-B Evolve

This section describes how FAS-A and FAS-B are dynamically reconfigured.

A.4.2.1 How FAS-A is Dynamically Reconfigured

The dynamic reconfiguration of FAS-A is localized to SAS-a. It evolves in place to realize the new FAS-A and FAS-B scenarios. Its evolution is described below.

A.4.2.1.1. Summary of Kinds of Change

The kinds of change that are required to support the new UCMs are the following:

- New architectural constraints of FAS-A and FAS-B;
- New communication path linking the two systems;
- New service protocols Short Term Preferred Stock Values (T23) and Long Term Preferred Stock Values (T29);
- New required and provided services; and
- Evolved internal behavior to support:
 - P44 implements service protocol T23: Short Term Preferred Stock Values.
 - P56 implements service protocol T29: Long Term Preferred Stock Values.

A.4.2.1.2. Summary of Effects of Change

ValidateLongTermValuesBasedOnShortTermValues() is a new operation with a dependency on *LongTermPreferredStockValues* and *ServiceRequiredVia(P44:T23)*:

ValidateLongTermValuesBasedOnShortTermValues()

ServiceRequiredVia(P44:T23): ShortTermPreferredStockValues();

Read(LongTermPreferredStockValues);

Action(ValidateLongTermValues)

LongTermPreferredStockValues() is a new operation with a dependency on the existing operation *LongTermKeyIndicatorsInformation()*:

LongTermPreferredStockValues()

LongTermKeyIndicatorsInformation()

GenerateLongTermPreferredStockValues() is a existing operation with a dependency on the new operation *ValidateLongTermValuesBasedOnShortTermValues()*:

GenerateLongTermPreferredStockValues()

ValidateLongTermValuesBasedOnShortTermValues();

Action(GeneratePreferredStockValues);

Update(LongTermPreferredStockValues)

LongTermPreferredStockValues change because they are now validated using shorter-term preferred stock values provided by SAS-b.

A.4.2.1.3. Summary of Changed Dependencies

New component to component dependency: SAS-A -> SAS-B.

New required service to provided service dependencies: P44 -> P45 and P56 -> P57.

New service to service protocol dependencies: P44 -> T23 and P56 -> T29.

New operation to required service dependency: ShortTermKeyPreferredStockValues() -> ServiceRequiredVia(P44:T23).

New operation to provided service dependency: LongTermKeyPreferredStockValues() -> ServiceProvidedVia(P56:T29).

New operation to operation dependencies: LongTermPreferredStockValues() -> LongTermKeyIndicatorsInformation() and GenerateLongTermPreferredStockValues()-> ValidateLongTermValuesBasedOnShortTermValues().

New operation to state element dependency:

ValidateLongTermValuesBasedOnShortTermValues() ->

LongTermPreferredStockValues.

A.4.2.1.4. System Integrity Implications

To conform to FAS-GP1, global consistency management ensures the existing mechanism that is used to synchronize and order control flow audit records also synchronizes and orders the new audit records created when the new UCM is enacted.

To conform to FAS-GP3, based on the control flow between FAS-A and FAS-B, global consistency management ensures any system interactions that do not occur within prescribed time constraints were invalidated.

To conform to FAS-GP4, global consistency management ensures particular invocation patterns when certain sequences were preferred (for performance, privacy, or other reasons).

To conform to FAS-GP5, the control style properties of the system are validated with respect to the impact of the new UCMs (SAS-a as a client/controller and as a server/controllee).

Local consistency management ensures conformance to FAS-LP2 whenever SAS-a is a control flow initiator. This is based on the control style of the system and in this case occurs whenever *Generate Long Term Values()* or *Augment Prior Knowledge Of Energy Sector()* are invoked. This ensures audit records trace control flow whenever a new UCM is enacted.

Local consistency management ensures conformance to FAS-LP3 for the new provided service *Long Term Preferred Stock Values (P56:T29)* of SAS-a.

Local consistency management ensures conformance to FAS-LP5 for the new behavioral implementation of SAS-a. The new operation reuses existing operations, which are re-factored to ensure common behavior is factored out as distinct operations.

Local consistency management ensures conformance to FAS-LP6 depending on the number of remote service connections.

Local consistency management ensures conformance to FAS-LP7 given the new interactions with FAS-B. Because of new remote interactions with FAS-B, SAS-a behavior sequences may be invalidated because they not longer occur within the Y time units time tolerance. SAS-a must invalidate any end-to-end financial analysis (which includes the new UCM) that exceeds the Y time units time tolerance.

Local consistency management ensures conformance to FAS-LP8 with local users given precedence over FAS-B interactions. Under limited resource conditions, Financial Portfolio Manager (System A) is given priority. SAS-a ensures FAS-A client service requests are handled at a higher priority than SAS-b service requests.

Active reference management ensures the integrity of required to provided service dependencies.

Dependent operation management ensures the integrity of operation dependencies.

Constrained operation management ensures the integrity of operation state element dependencies.

A.4.2.2 How FAS-B is Dynamically Reconfigured

The dynamic reconfiguration of FAS-B is localized to SAS-b. SAS-b is replaced by SAS-b' whose substitutable implementation (which obeys co-variance and contra-variance rules) supports the new UCMs. Its evolution is described below.

A.4.2.2.1. Summary of Kinds of Change

The kinds of change that are required to support the new UCMs are the following:

- New architectural constraints of FAS-A and FAS-B;
- New communication path linking the two systems; and
- Deployment of a new component that is substitutable for the original SAS-b whose implementation supports new required internal behavior, new required and provided services, and new types of information exchanged.

A.4.2.2.2. Summary of Effects of Change

A new operation is introduced, which is constrained by its read relationship with the state element *ShortTermPreferredStockValues*:

ValidateShortTermValuesBasedOnLongTermValues()

ServiceRequiredVia(P44:T23): LongTermPreferredStockValues();

Read(ShortTermPreferredStockValues);

Action(ValidateShortTermValues)

ShortTermPreferredStockValues() is a new operation with a dependency on the existing operation *ShortTermKeyIndicatorsAndMagnitudeBasedInformation()*:

ShortTermPreferredStockValues()

ShortTermKeyIndicatorsAndMagnitudeBasedInformation()

GenerateShortTermPreferredStockValues() is a existing operation with a dependency on the new operation *ValidateShortTermValuesBasedOnLongTermValues()*:

GenerateShortTermPreferredStockValues()

ValidateShortTermValuesBasedOnLongTermValues()

Action(GeneratePreferredStockValues);

Update(ShortTermPreferredStockValues)

A Financial Portfolio Manager (System B) may have an established connection by binding to the provided service *Short Term Key Indicators and Magnitude Based Information (P13:T7)* and subsequently invoking *Short Term Key Indicators And Magnitude Based Information()* before SAS-b evolved. Any active (or new) client component references are tracked and then migrated to refer to the new implementation when it was deployed (for example through the use of a special front-end component; or other applicable hot swapping techniques) or terminated.

The state elements *Short Term Preferred Stock Values* and *Short Term Common Stock Values* could be transferred before the new component is swapped in order to keep previous computations. Alternatively, because these are shorter-term predictions, it may be feasible to simply compute new values on-demand based on current financial condition information when the component is deployed since the older values may no longer be valid.

ShortTermPreferredStockValues change after SAS-b evolves because this information is now validated using longer-term preferred stock values as provided by SAS-a.

A.4.2.2.3. Summary of Changed Dependencies

New component to component dependency: SAS-B -> SAS-A.

New required service to provided service dependencies: P45 -> P44 and P57 -> P56.

New service to service protocol dependencies: P45 -> T23 and P57 -> T29.

New operation to required service dependency: LongTermPreferredStockValues() -> ServiceRequiredVia(P57:T29).

New operation to provided service dependency: ShortTermPreferredStockValues() -> ServiceProvidedVia(P45:T23).

New operation to operation dependencies: ShortTermPreferredStockValues() ->

ShortTermKeyIndicatorsAndMagnitudeBasedInformation() and

GenerateShortTermPreferredStockValues() ->

ValidateShortTermValuesBasedOnLongTermValues()

New operation to state element dependency:

ValidateShortTermValuesBasedOnLongTermValues() ->

ShortTermPreferredStockValues

A.4.2.2.4. System Integrity Implications

To conform to FAS-GP1, global consistency management ensures the existing mechanism that is used to synchronize and order control flow audit records also synchronizes and orders the new audit records created when the new UCM is enacted.

To conform to FAS-GP3, based on the control flow between FAS-A and FAS-B, global consistency management ensures any inter-system interactions that do not occur within prescribed time constraints were invalidated.

To conform to FAS-GP4, Global consistency management ensures conformance to particular invocation patterns when particular sequences are preferred (for performance, privacy, or other reasons).

To conform to FAS-GP5, the control style properties of the system must be validated with respect to the impact of the new UCMs (SAS-b as a client/controller and as a server/controlled).

Local consistency management ensures conformance to FAS-LP2 whenever SAS-b was a control flow initiator. This is based on the control style of the system and in this case occurs whenever *Generate Short Term Values()* or *Augment Prior Knowledge Of*

Energy Sector() are invoked. SAS-b must ensure it creates an audit record that traces control flow activities whenever the new UCM is enacted.

Local consistency management ensures conformance to FAS-LP3 with respect to the new provided service *Short Term Preferred Stock Values* (P45:T23) of SAS-b.

Local consistency management ensures conformance to FAS-LP5 with respect to the new behavioral implement of SAS-b. This means any new operations reuse existing behavior and any existing operations are re-factored to reuse new common behavior.

Local consistency management ensures conformance to FAS-LP6 depending on the number of remote service connections.

Local consistency management ensures conformance to FAS-LP7 given the new interactions with FAS-A. Because of new remote interactions with FAS-A certain SAS-b behavior sequences may be invalidated because they not longer occur within the Y time units time tolerance. SAS-b must invalidate any end-to-end financial analysis (which includes the new UCM) that exceeds the Y time units time tolerance.

Local consistency management ensures conformance to FAS-LP8 with respect to service provided service to local users versus FAS-A interactions. Under limited resource conditions, Financial Portfolio Manager (System B) is given priority. SAS-b must now ensure FAS-B client service requests are handled at a higher priority than SAS-a service requests.

Active reference management ensures the integrity of required to provided service dependencies.

State management ensures the integrity of state elements during component substitution.

A.5. Summary and Conclusion

The theme of this application-specific case study was the analysis of volatile environmental information based on systems dynamic interoperation. The financial analysis domain investigated represents a system of financial relationships and cash flows which is considered a complex modeling application in which adaptability is required because of the volatile nature of the world of finance.

The case study presented the coordinated evolution of two financial analysis systems through the dynamic integration of their respective scenario analysis sub-systems in order to provide an improved capability for their respective clients. Each system maintained knowledge and provided predictive scenarios about the energy market sector. One system specialized in longer-term prediction while the other system specialized in shorter-term scenarios. An improved capability was enabled through the use of longer-term predictions to validate shorter-term scenarios and vice versa.

First, a description of each original system (pre-evolution) was given. This description included the main features the investor community are interested in, the architectural style that guided the overall design of the system, the list of system use cases, and run-time component descriptions using Use Case Maps and required and provided services.

Second, a description of the planned evolution of these two systems to enable the use of longer-term predictions to validate shorter-term scenarios and vice versa was presented. Evolution was restricted to the scenario analysis subsystems of each financial analysis system and was backward compatible to their existing implementations.

Third, the dynamic reconfiguration of each system was described. First, the changed representations of each system were presented by describing updated financial analysis

use cases, new use case maps, new required and provided services, and updated detailed component descriptions of the scenario analysis subsystems of each financial analysis system. Secondly, a description of how each system was dynamically reconfigured, while maintaining system integrity.

The kinds of system dynamic reconfiguration included component evolution to support:

- New architectural properties;
- New communication path linking the two systems;
- New service protocols;
- New required and provided services;
- Deployment of a new component that is substitutable for an original component whose implementation supports new required internal behavior, new required and provided services, and new types of information exchanged; and
- Changed internal behavior.

Appendix B – Architectural Patterns for Dynamic Reconfiguration

Informed by [59, 70, 106], the following template is adopted to specify a set of architectural patterns for enabling dynamic reconfiguration:

Name: A name that conveys the essence of the pattern succinctly and communicates the pattern easily.

Motivation: A scenario that illustrates the system evolution problem and how the pattern solves the problem.

Problem: Description of the recurring problem addressed.

Forces: Description of what considerations need to be balanced.

Context: Description of where the solution is appropriate.

Solution: Description of the solution that can be tailored to a specific instance of the problem.

Example: Example of pattern from different domains.

Consequences: Description of how the pattern supports its objectives, of trade-offs and results of using the pattern or of what aspects of the system it can vary independently.

Related Patterns: Description of other patterns that are closely related to this one or that this one should use.

Table 11 outlines the dynamic reconfiguration patterns discussed in this section. It lists each pattern's name with the problem that the pattern solves. These patterns collaborate to provide the necessary aspects of system evolution to enable dynamic reconfiguration. This collaboration is further described in 'How these Patterns Interact with One Another' at the end of this section.

Pattern Name	Intent
Internal Change	Evolve a system by changing a component's internal implementation.
Substitution	Evolve a system by changing one component for another.
Interface Change	Evolve a system by changing the interfaces of a component.
Protocol Change	Evolve a system by changing the interactions among components.
Topology Change	Evolve a system by changing the structure of its components.
Architectural Change	Evolve a system by changing its global or local properties.

Table 11. Pattern Catalog

B.1 Catalog of Patterns for Dynamic Reconfiguration

B.1.1. Change of System Architectural Constraints

Name: Architectural Change

Motivation: Being able to change whole configurations of components (the architecture) of a system allows for macro-level system change due to the need for the architectural change of a system. For example, within the context of a financial analysis system, there may be a need to change the overall architecture of the system in order to expose different points of interoperability to best integrate the financial system with other financial systems as a federation. As an example, two different financial analysis systems may become more integrated over time in a strategic alliance where the analytic strengths of each system are combined.

Problem: The architectural properties of a system no longer (optimally) reflect the mandatory or optional functional or non-functional architectural specification of the system because this specification has changed over time.

Forces:

- System responds to run time change stimuli;
- System characterizes change as global or local properties (including their reconciliation);

- Given default settings, default dependency, illegal feature combination, construction rules and optimization strategies specifications, it is applicable and feasible for the system to enact the required architectural change; and
- System governs change to ensure overall system consistency.

Context: The properties of a system no longer optimally meet its architectural specification because they do not reflect the correct global or local system properties that satisfy this specification.

Solution: Evolve the global or local properties of the system as implied by the evolved specification of the overall system formulated as a set of constraints. The Type of Change satisfying Condition Change Criteria is Architectural Change. The features that are dynamically reconfigured during Architectural Change are Global Properties and Local Properties of System Properties and Behavioral Signature and Structural Signature of System Signature. The System Integrity Characteristics governing Architectural Change, which ensure overall system consistency, are Global Consistency and Local Consistency.

Example: In the Financial Analysis System case study, the evolution of the SAS-a and SAS-b' components to enable dynamic system interoperability.

Consequences:

- Positive implication: Architectural Change of component will satisfy new requirements;
- Negative implication: Architectural Change could impact component's interaction with other components;
- Negative implication: No opportunity for off-line verification/validation of components' new architectural properties;

- Negative implication: Further Architectural change may be required if it is determined original evolution was not optimal.
- Negative implication: Complexity associated with on-line interaction with Protocol Change and Topology Change if required.

Related Patterns:

- Architectural Change uses Topology Change to affect required structural change to a component-based system.
- Architectural Change uses Protocol Change to affect required behavioral change to a component-based system.

B.1.2. Component Addition and/or Component Removal

Name: Topology Change

Motivation: Being able to add or remove components and component communication paths of a system allows components to be decommissioned when no longer needed or for new components to be integrated that augment an existing system with required new functionality. For example, within the context of a financial analysis system, a component that checks deal pricing may no longer be required and is removed from the system to lessen the load on processing resources. As another example, a new component that implements a new capability, such as the ability to generate cash flow projections, is integrated within a system to provide a new feature required by system stakeholders.

Problem: The existing structural makeup of a system no longer (optimally) satisfies its mandatory or optional functional or non-functional structural requirements because these requirements, or the runtime environment, have changed over time.

Forces:

- System responds to run time change stimuli;
- System characterizes change as global or local properties (including their reconciliation);
- Given default settings, default dependency, illegal feature combination, construction rules and optimization strategies specifications, it is applicable and feasible for system to enact the required topology change; and
- System governs change in a manner to ensure overall system consistency.

Context: The structure of a system no longer optimally meets its components' interconnection obligations because it does not implement the correct system topology that satisfies these interconnection obligations.

Solution: Add or remove deployed components or component communication paths to change the topology of a system as necessary when new components must be deployed, components must be decommissioned, new connections must be established, or existing connections must be terminated. The Type of Change satisfying Condition Change Criteria is Topology Change. The features that are dynamically reconfigured during Topology Change are Structural Signature of System Signature and Communication Path and/or Component. The System Integrity Characteristics governing Topology change, which ensure overall system consistency, are Active Component References, Global Consistency, and Local Consistency.

Example: In the Financial Analysis System case study, the new Communication Path linking the SAS-a and SAS-b' components.

Consequences:

- Positive implication: Topology Change of component will satisfy new requirements;

- Negative implication: Topology Change could impact component's interaction with other components;
- Negative implication: No opportunity for off-line verification/validation of components' new structural interconnections;
- Negative implication: Further Topology Change may be required if it is determined original evolution was not optimal.
- Negative implication: Complexity associated with on-line interaction with Protocol Change and Substitution if required.

Related Patterns:

- Topology Change uses Protocol Change to affect necessary change to a system's component's interactions in support of the structural evolution of a system.
- Topology Change uses Substitution to change one component for another in support of the structural evolution of a system.

B.1.3. Substituting One Component for another

Name: Substitution

Motivation: Being able to substitute one component for another enables a system to be reconfigured in response to changing requirements based on substituting components within a greater system. During substitution, cooperative components, when possible, continue to function in an unaffected manner. For example, within the context of a financial analysis system, the availability requirements of a system component may change from regular business hours to 24/7 operations. In this case the functional requirements may stay the same but a different component may have to be substituted because it was designed for continual operation with minimal downtime. As another

example, a substitution may be required because a new version of a component implements more efficient algorithms for computing financial derivatives.

Problem: A deployed component may no longer (optimally) meets its mandatory or optional functional or non-functional requirements because these requirements or the environment in which it is running have changed since it was deployed.

Forces:

- System is able to respond to run time change stimuli;
- System is able to characterize change as global or local properties (including their reconciliation);
- Given default settings, default dependency, illegal feature combination, construction rules and optimization strategies specifications, it is applicable and feasible for system to enact the required substitution; and
- System is able to govern change in a manner to help to ensure overall system consistency.

Context: The implementation of a deployed component is not optimally meeting its requirements and a determination is made that the cost of dynamically evolving the component exceeds the cost of substituting a new component in its place that does satisfies the new requirements.

Solution: Substitute one component for another when a deployed component no longer meets its mandatory or optional functional or non-functional requirements and an existing component (not yet deployed) does and it is cost effective to do so. The Type of Change satisfying Condition Change Criteria is Substitution. The features that are dynamically reconfigured during Substitution are Structural Signature of System Signature,

Component and/or Connection, and State Element. The System Integrity Characteristics governing Substitution, which ensure overall system consistency, are Component State, Active Component References, Global Consistency and Local Consistency.

Example: In the Financial Analysis System case study, the SAS-b' component substituted for SAS-b component.

Consequences:

- Positive implication: Substituted component will satisfy new requirements;
- Positive implication: Opportunity for off-line verification/validation of component's implementation;
- Positive implication: Complexity associated with possible interaction with Internal Change is more manageable because it is off-line.
- Negative implication: Substitution may interrupt component's interaction with other components;
- Negative implication: Further substitution may be required if it is determined original evolution was not optimal.

Related Patterns:

- Substitution relies on off-line Internal Change of the component to be substituted.

B.1.4. Protocol Change among Components in Place

Name: Protocol Change

Motivation: Protocol Change among components in place helps to ensure control-driven and data-driven change can be accommodated by cooperative software components. For example, within the context of a financial analysis system, new sources of information or raw data may become available (possibly in new formats) over time. Financial system

components that are responsible for generating valuation and risk numbers, stress testing investment positions, or other financial analysis activities must be able to change the information they exchange with other components as required in order to effectively process this new information.

Problem: Deployed components no longer (optimally) meet their mandatory or optional functional or non-functional interaction requirements because control-driven or data-driven system interaction patterns have changed over time.

Forces:

- System is able to respond to run time change stimuli;
- System is able to characterize change as global or local properties (including their reconciliation);
- Given default settings, default dependency, illegal feature combination, construction rules and optimization strategies specifications, it is applicable and feasible for system to enact the required protocol change; and
- System is able to govern change in a manner to help to ensure overall system consistency.

Context: A deployed component no longer optimally meets its interaction obligations because it does not implement the correct protocol that supports the required control flow and/or data flow interaction patterns.

Solution: Evolve the protocol among deployed components when control flow or data flow patterns must change among these components. The Type of Change satisfying Condition Change Criteria is Protocol Change. The features that are dynamically reconfigured during Protocol Change are Behavioral Signature of System Signature and

Service Protocol. The System Integrity Characteristics governing Protocol Change, which ensure overall system consistency are Global Consistency and Local Consistency.

Example: In the Financial Analysis System case study, the SAS-a component evolved to support the new protocol with SAS-b'.

Consequences:

- Positive implication: Protocol Change of component will satisfy new requirements;
- Negative implication: Protocol Change impacts component's interaction with other components;
- Negative implication: No opportunity for off-line verification/validation of components' new interaction protocols;
- Negative implication: Further Protocol change may be required if it is determined original evolution was not optimal.
- Negative implication: Complexity associated with on-line interaction with Topology Change and Interface Change if required.

Related Patterns:

- Protocol Change uses Topology Change to affect necessary change to a system's structure in support of the protocol change of a system.
- Protocol Change uses Interface Change to affect necessary change to component interfaces in support of the protocol change of a system.

B.1.5. Interface Change of a Component in Place

Name: Interface Change

Motivation: Being able to change the interaction of a component in place helps to ensure the component may continue to inter-operate when required interoperability

patterns change. In this case, the function of a component does not necessarily change but its interface elements do to ensure it could continue to cooperate with other components. For example, within the context of a financial analysis system, when new components are deployed over time, such as those that provide more specialized services for certain kinds of financial instruments, existing components may need to change how they interact in order to inter-operate with these new components. In a cascading affect, other existing components may then have to change how they interact in order for them to continue to inter-operate.

Problem: A deployed component no longer (optimally) meets its mandatory or optional functional or non-functional interoperability requirements with other deployed components because component exposed behavior patterns have changed over time.

Forces:

- System is able to respond to run time change stimuli;
- System is able to characterize change as global or local properties (including their reconciliation);
- Given default settings, default dependency, illegal feature combination, construction rules and optimization strategies specifications, it is applicable and feasible for system to enact the required interface change; and
- System is able to govern change in a manner to help to ensure overall system consistency.

Context: A deployed component no longer optimally meets its interoperability obligations because it does not expose the correct interfaces to expose the needed behavior to support these obligations.

Solution: Change the interface elements of a deployed component to ensure it may inter-operate with other deployed components as system interaction patterns change. The Type of Change satisfying Condition Change Criteria is Interface Change. The features that are dynamically reconfigured during Interface Change are Behavioral Signature of System Signature, and Required Service and/or Provided Service. The System Integrity Characteristics governing Interface Change, which ensure overall system consistency, are Global Consistency and Local Consistency.

Example: In the Financial Analysis System case study, the SAS-a component exposes a new provided and required service.

Consequences:

- Positive implication: Interface Change of component will satisfy new requirements;
- Positive implication: Interface Change minimally impacts component's interaction with other components;
- Negative implication: No opportunity for off-line verification/validation of component's new interfaces;
- Negative implication: Further Interface Change may be required if it is determined original evolution was not optimal.
- Negative implication: Complexity associated with on-line interaction with Internal Change if required.

Related Patterns:

- Interface Change uses Internal Change to change the internal implementation of a component as required to support its changing interfaces.

B.1.6. Internal Change of a Component in Place

Name: Internal Change

Motivation: Being able to change the internal implementation of a component in place enables a system to be dynamically reconfigured when the functional or non-functional requirements of a component change. For example, within the context of a financial analysis system, the way a component can perform scenario analysis or compute cash flow projections may have to change as a consequence of legal or policy changes that govern the financial system as a whole. As another example, the features of interest provided by a component to financial analysts, or other users, may have to change or may become outdated based on changing market trends.

Problem: A deployed component may no longer (optimally) meets its mandatory or optional functional or non-functional requirements because these requirements or the environment in which it is running have changed since it was deployed.

Forces:

- System is able to respond to run time change stimuli;
- System is able to characterize change as global or local properties (including their reconciliation);
- Given default settings, default dependency, illegal feature combination, construction rules and optimization strategies specifications, it is applicable and feasible for system to enact the required internal change; and
- System is able to govern change in a manner to help to ensure overall system consistency.

Context: The implementation of a deployed component is no longer optimally meeting its requirements and it is cost effective to evolve the component in place rather than substituting a new component for the existing component. For example, substitution may not be possible because it may be too costly to manage active component references or to transfer necessary state when substituting a new component.

Solution: Evolve the internal implementation of a deployed component when functional or non-functional requirements of the component change over time and it is cost effective to do so. The component must evolve in a manner that minimizes the impact of change on any other components that interact with it. The Type of Change satisfying Condition Change Criteria is Internal Change. The features that are dynamically reconfigured during Internal Change are Behavioral Signature of System Signature and Operation, State Element and Service Invocation. The System Integrity Characteristics governing Internal Change, which ensure overall system consistency are Composite Components, Dependent Operations, Constrained Operations, Component State, Global Consistency and Local Consistency.

Example: In the Financial Analysis System case study, the internal evolution of the SAS-a component to satisfy the new system interoperation obligations.

Consequences:

- Positive implication: Component will evolve its internal implementation to satisfy new requirements;
- Positive implication: No on-line interaction required with other patterns of dynamic reconfiguration.

- Negative implication: Component evolution may interrupt component's interaction with other components;
- Negative implication: Since substitution was not applied no opportunity for off-line verification/validation of component's implementation;
- Negative implication: Component's new implementation must be verified/validated on-line;
- Negative implication: Further internal change may be required if it is determined original evolution was not optimal.

Related Patterns:

- GOF [59] and other design patterns.

B.2. How these Patterns Interact with One Another

Figure 128 shows the primary relationships among the architectural patterns for enabling dynamic reconfiguration as a pattern interaction diagram:

Figure 128. Pattern Interaction Diagram

The primary interaction patterns shown demonstrate that different subsets of change types may be required depending upon the condition change criteria of a particular system context. Different combinations of change types are applied to accommodate the required change. Which subset is feasible, or most practical, depends on how the system is able to characterize change when responding to change stimuli and the system's default settings, default dependency, illegal feature combination, construction rules and optimization strategies specifications.

The primary interaction patterns also show an ordering of change types; the most complex dynamic reconfigurations require the application of all types of change to fully enable the required evolution of a system.

Based on the pattern interaction diagram, the following table enumerates different possible combinations of change types. These combinations are classified into change groupings. Each grouping is described in section 3.5.

Change Grouping	Change Sequences
Pure Architectural Change	<ul style="list-style-type: none"> • Architectural Change.
Pure Behavioral Change	<ul style="list-style-type: none"> • Architectural Change -> Protocol Change; • Architectural Change -> Protocol Change -> Interface Change; or • Architectural Change -> Protocol Change -> Interface Change -> Internal Change.
Behavior-Driven Structural Change	<ul style="list-style-type: none"> • Architectural Change -> Protocol Change -> Topology Change; or • Architectural Change -> Protocol Change -> Topology Change -> Substitution.
Comprehensive Behavior-Driven Change	<ul style="list-style-type: none"> • Pure Behavioral Change + Behavior-Driven Structural Change.
Pure Structural Change	<ul style="list-style-type: none"> • Architectural Change -> Topology Change; or • Architectural Change -> Topology Change -> Substitution.
Structure-Driven Behavioral Change	<ul style="list-style-type: none"> • Architectural Change -> Topology Change -> Protocol Change. • Architectural Change -> Topology Change -> Protocol Change -> Interface Change; or • Architectural Change -> Topology Change -> Protocol Change -> Interface Change -> Internal Change.
Comprehensive Structure-Driven Change	<ul style="list-style-type: none"> • Pure Structural Change + Structure-Driven Behavioral Change.
Pure Behavioral and Structural Change	<ul style="list-style-type: none"> • Pure Behavioral Change + Pure Structural Change.
Comprehensive Change	<ul style="list-style-type: none"> • Comprehensive Behavior-Driven Change + Comprehensive Structure-Driven Change.

Table 12. Change Groups and Change Sequence

References

1. Walsh, D., F. Bordeleau, and B. Selic. *Change Types of Dynamic System Reconfiguration*. in *13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*. 2006. Potsdam, Germany: IEEE.
2. Oreizy, P., *Issues in Modeling and Analyzing Dynamic Software Architectures*. 1999, Information and Computer Science, University of California, Irvine: Irvine, California.
3. Oreizy, P., et al., *An Architecture-Based Approach to Self-Adaptive Software*. IEEE Intelligent Systems, 1999: p. 54-62.
4. Selic, B., *Private communication about the long standing problem of dynamic reconfiguration*. 2003: Ottawa.
5. Jarke, M., *Meta models for requirements engineering*. 1996.
6. Karsai, G. and J. Sztipanovits, *A Model-Based Approach to Self-Adaptive Software*. IEEE Intelligent Systems, 1999: p. 46-53.
7. Colman, A. and J. Han. *On the autonomy of software entities and modes of organisation*. in *First International Workshop on Coordination and Organization (COORG 2005)*. 2005. Namur, Belgium.
8. Coleman, A. and J. Han. *Operational Management Contracts for Adaptive Software Organisation*. in *Proceedings of the 2005 Australian Software Engineering Conference (ASWEC '05)*. 2005.
9. Coleman, A. and J. Han, *Using Association Aspects to Implement Organisational Contracts*. Electronic Notes in Theoretical Computer Science, 2006. 3(150): p. 37-53.
10. Coleman, A., et al. *Adaptive Application-Specific Middleware*. in *Proceedings of Middleware for Service-Oriented Computing 2006*. 2006. Melbourne, Australia: ACM.
11. Minsky, N.H., et al. *Building Reconfiguration Primitives into the Law of a System*. in *3rd IEEE International Conference on Configurable Distributed Systems (ICCDs '96)*. 1996: IEEE.
12. Lopes, A., M. Wermelinger, and J.L. Fiadeiro. *A Compositional Approach to Connector Construction*. in *5th International Workshop on Algebraic Development Techniques*. 2001: Springer-Verlag.
13. Wermelinger, M., *Specification of Software Architecture Reconfiguration*, in *Departamento de Informatica*. 1999, Universidade Nove de Lisboa: Lisbon, Portugal.
14. Wermelinger, M., A. Lopes, and J.L. Fiadeiro. *A Graph Based Architectural (Re)configuration Language*. in *ACM ESE/FSE 2001*. 2001. Vienna, Austria: ACM.
15. Kramer, J. and J. Magee, *Dynamic Configuration for Distributed Systems*. IEEE Transactions on Software Engineering, 1985. SE-11(4): p. 424-435.
16. Wermelinger, M. *A Hierarchic Architecture Model for Dynamic Reconfiguration*. in *IEEE 2nd International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE '97)*. 1997: IEEE.

17. Wermelinger, M. *Towards a Chemical Model for Software Architecture Reconfiguration*. in *4th International Conference on Configuring Distributed Systems*. 1998: IEEE Computer Society.
18. Medvidovic, N. and R.N. Taylor, *A Classification and Comparison Framework for Software Architecture Description Languages*. IEEE Transactions on Software Engineering, 2000. **26**(1): p. 70-93.
19. Kramer, J. and J. Magee. *Analysing Dynamic Change in Software Architectures: A case study*. in *IEEE 4th International Conference on Configurable Distributed Systems*. 1998. Annapolis, Maryland: IEEE.
20. Magee, J. and J. Kramer. *Self Organising Software Architectures*. in *ACM SIGSOFT '96*. 1996. San Francisco, CA: ACM.
21. Georgiadis, I., J. Magee, and J. Kramer. *Self-Organising Software Architectures for Distributed Systems*. in *ACM WOSS '02*. 2002. Charleston, SC: ACM.
22. Magee, J. and J. Kramer. *Dynamic Structure in Software Architectures*. in *ACM SIGSOFT '96*. 1996. San Francisco, CA: ACM.
23. Jackson, D., *Alloy: A Lightweight Object Modelling Notation*. 2001, Massachusetts Institute of Technology: Cambridge, Massachusetts. p. 32.
24. Jackson, D., I. Schechter, and I. Shlyakhter, *Alcoa: The Alloy Constraint Analyzer*. 1999, Massachusetts Institute of Technology: Cambridge, Massachusetts.
25. Birman, K., A. Schiper, and P. Stephenson, *Lightweight Causal and Atomic Group Multicast*. ACM Transactions on Computer Systems, 1991. **9**(3): p. 271-314.
26. Bialek, R.P. *The Architecture of a Dynamically Updateable, Component-based System*. in *26th Annual International Computer Software and Applications Conference (COMPSAC '02)*. 2002: IEEE Computer Society.
27. Warren, I. and I. Sommerville. *A Model for Dynamic Configuration which Preserves Application Integrity*. in *IEEE 3rd International Conference on Configurable Distributed Systems (ICCDs '96)*. 1996: IEEE.
28. Guo, J. *Software Components Adaptive Integration*. in *Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS '01)*. 2001: IEEE Computer Society.
29. Appavoo, J., et al., *Enabling automatic behavior in systems software with hot swapping*. IBM Systems Journal, 2003. **42**(1): p. 60-76.
30. Yellin, D.M., *Competitive algorithms for the dynamic selection of component implementations*. IBM Systems Journal, 2003. **42**(3): p. 85-97.
31. Whisnant, K., Z.T. Kalbarczyk, and R.K. Iyer, *A system model for dynamically reconfigurable software*. IBM Systems Journal, 2003. **42**(1): p. 45-59.
32. Batista, T.V. and N.d.I.R. Rodriguez. *Dynamic Reconfiguration of Component-Based Applications*. in *International Symposium on Software Engineering for Parallel and Distributed Systems*. 2000.
33. de Paula, V.C., G.R.R. Justo, and P.R.F. Cunha. *Specifying and Verifying Reconfigurable Software Architectures*. in *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*. 2000.

34. Truyen, E., et al. *A Dynamic Customization Model for Distributed Component-Based Systems*. in *IEEE 21st International Conference on Distributed Computing Systems Workshops (ICDCSW '01)*. 2001: IEEE.
35. Walsh, D., F. Bordeleau, and B. Selic. *A Domain Model for Dynamic System Reconfiguration*. in *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems*. 2005. Montego Bay, Jamaica: Springer.
36. Walsh, D., F. Bordeleau, and B. Selic, *Domain analysis of dynamic system reconfiguration*. Accepted for Publication in *Software and System Modeling*, Springer-Verlag, 2006.
37. Walsh, D., F. Bordeleau, and B. Selic. *A Constrained Executable Model of Dynamic System Reconfiguration*. in *The 40th Hawaii International Conference on System Sciences*. 2007. Hawaii: IEEE.
38. Czarnecki, K. and U.W. Eisennecker, *Generative Programming: Methods, Tools, and Applications*. 2000, New York, N.Y.: Addison-Wesley.
39. Gomaa, H., *Designing Software Product Lines with UML From Use Cases to Pattern-Based Software Architectures*. Object Technology Series, ed. G. Booch, I. Jacobson, and J. Rumbaugh. 2004, New York, N.Y.: Addison-Wesley.
40. Duffy, D.J., *Domain Architectures Models and Architectures for UML Applications*. 2004, Hoboken, N.J.: John Wiley and Sons.
41. Evans, E., *Domain-Driven Design: Tackling Complexity in the Heart of Software*. 2004, New York, N.Y.: Addison-Wesley.
42. Software Engineering Institute, C.M., *Domain Engineering and Domain Analysis Software Technology Roadmap*. 2006, Carnegie Mellon, University.
43. Oreizy, P., N. Medvidovic, and R.N. Taylor. *Architecture-Based Runtime Software Evolution*. in *Proceedings of the International Conference on Software Engineering*. 1998.
44. Szyperski, C., *Component Software: Beyond Object-Oriented Programming Second Edition*. Component Software Series. 2002, New York, N. Y.: Addison-Wesley.
45. Deployment-and-Configuration-Draft-Adopted-Specification, *Deployment and Configuration Draft Adopted Specification*. 2003, OMG: Needham, MA.
46. Lopes, A., M. Wermelinger, and J.L. Fiadeiro, *Higher-Order Architectural Connectors*. *ACM Transactions on Software Engineering and Methodology*, 2003. 12(1): p. 64-104.
47. Jackson, D., *Micromodels of Software: Lightweight Modelling and Analysis with Alloy*. 2002, MIT Lab for Computer Science: Cambridge, Mass. p. 1-58.
48. Vaziri, M. and D. Jackson, *Some Shortcomings of OCL, the Object Constraint Language of UML*. 1999, MIT Laboratory for Computer Science: Cambridge, Mass. p. 1-17.
49. Marriott, K. and P.J. Stuckey, *Programming with Constraints An Introduction*. 1998, Cambridge, Mass.: MIT Press.
50. Fowler, M., *UML Distilled Third Edition A Brief Guide to the Standard Object Modeling Language*. Object Technology Series. 2004: Addison-Wesley.
51. Warmer, J. and A. Kleppe, *The Object Constraint Language Second Edition Getting Your Models Ready for MDA*. Object Technology Series. 2003, New York, N.Y.: Addison-Wesley.

52. Parnas, D., *On the Criteria To Be Used in Decomposing Systems Into Modules*. Communications of the ACM, 1972. **15**(12): p. 1053-1058.
53. Shaw, M. and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. 1996, Upper Saddle River, New Jersey: Prentice Hall.
54. Bass, L., P. Clements, and R. Kazman, *Software Architecture in Practice*. SEI Series in Software Engineering. 1998, New York, New York: Addison-Wesley.
55. D'Souza, D.F. and A.C. Wills, *Object, Components, and Frameworks in UML The Catalysis Approach*. Object Technology Series, ed. Addison-Wesley. 1999, New York, New York: Addison-Wesley.
56. Atkinson, C., et al., *Component-based Product Line Engineering with UML*. Component Software Series, ed. C. Szyperski. 2002, New York, New York: Addison-Wesley.
57. Martin, R.C., *Agile Software Development: Principles, Patterns, and Practices*. Alan Apt Series. 2003, Upper Saddle River, N.J.: Prentice Hall.
58. Frankel, D., *Model Driven Architecture Applying MDA to Enterprise Computing*, ed. O. Press. 2003, Indianapolis, Indiana: Wiley Publishing.
59. Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. 1995, Reading, MA.: Addison-Wesley.
60. Lee, M.A. and M.H. Smith. *Handling Uncertainty in Finance Applications Using Soft Computing*. in *3rd International Symposium on Uncertainty Modelling and Analysis (ISUMA '95)*. 1995: IEEE Press.
61. Helfert, E.A., *Financial Analysis Tools and Techniques A Guide for Managers*. 2001, New York, N.Y.: McGraw-Hill.
62. Buhr, R.J.A. and R.S. Casselman, *Use Case Maps for Object-Oriented Systems*. 1996, New York, NY: Prentice Hall.
63. Kiczales, G., J. des Rivieres, and D.G. Bobrow, *The Art of the Metaobject Protocol*. 1991, Cambridge, MA: MIT Press.
64. Wirfs-Brock, R. and J. Schwartz. *Tutorial #75: The Art of Writing Use Cases*. in *Object-Oriented Programming, Systems, Languages, and Applications*. 2000. Minneapolis, Minnesota: ACM Press.
65. Cockburn, A., *Writing Effective Use Cases*. The Agile Software Development Series, ed. A. Cockburn and J. Highsmith. 2001, New York, N.Y.: Addison-Wesley.
66. Deployment-and-Configuration-of-Component-based-Distributed-Applications-RFP, *Deployment and Configuration of Component-based Distributed Applications - Request for Proposal*. 2002, OMG.
67. Online-Upgrades-RFP, *Online Upgrades - Request For Proposal*. 2001, OMG.
68. Online-Upgrades-Draft-Adopted-Specification, *Online Upgrades - Draft Adopted Specification*. 2002, OMG.
69. Jackson, D., *Dependable Software by Design*, in *Scientific American*. 2006.
70. Wirfs-Brock, R. and A. McKean, *Object Design Roles, Responsibilities, and Collaborations*. 2003, New York, New York: Addison Wesley.
71. Gottlob, G., M. Schrefi, and B. Rock, *Extending Object-Oriented Systems with Roles*. ACM Transactions on Information Systems, 1996. **14**(3): p. 268-296.

72. Kendall, E.A. *Role Model Designs and Implementations with Aspect-Oriented Programming*. in *Object-Oriented Programming Systems, Languages, and Applications*. 1999: ACM Press.
73. Liskov, B. *Data Abstraction and Hierarchy*. in *Object-Oriented Programming Systems, Languages, and Applications*. 1987: ACM Press.
74. Clarke, S., *Extending Standard UML with Model Composition Semantics*. 2001, Department of Computer Science, Trinity College.: Dublin, Ireland.
75. Ossher, H. and P. Tarr, *Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software*. Communications of the ACM, 2001. 44(10): p. 43-50.
76. Clarke, S. *Subject-Oriented Design: Towards Improved Alignment of Requirements, Design, and Code*. in *Object-Oriented Programming Systems, Languages, and Applications*. 1999: ACM Press.
77. Clarke, S. and R.J. Walker. *Composition Patterns: An Approach to Designing Reusable Aspects*. in *International Conference on Software Engineering*. 2001: IEEE.
78. Clarke, S., *Composition of Object-Oriented Software Design Models*, in *School of Computer Applications*. 2001, Dublin City University: Dublin.
79. Clarke, S. and R.J. Walker. *Towards a Standard Design Language for AOSD*. in *AOSD 2002*. 2002. Enschede, The Netherlands: ACM.
80. Riehle, D. and T. Grosse. *Role Model Based Framework Design and Integration*. in *Object-Oriented Programming Systems, Languages, and Applications*. 1998: ACM Press.
81. Riehle, D., *Framework Design A Role Modeling Approach*. 1999: Zurich.
82. Bergmans, L. and M. Aksit, *Composing Crosscutting Concerns Using Composition Filters*. Communications of the ACM, 2001. 44(10): p. 51-57.
83. Harrison, W. and H. Ossher. *Subject-Oriented Programming (a critique of pure objects)*. in *Object-Oriented Programming, Systems, Languages, and Applications Conference*. 1993: ACM Press.
84. Ossher, H. *Subject-Oriented Composition Rules*. in *Object-Oriented Programming Systems, Languages, and Applications*. 1995: ACM Press.
85. Tarr, P., H. Ossher, and W. Harrison. *N Degrees of Separation: Multi-Dimensional Separation of Concerns*. in *International Conference on Software Engineering '99*. 1999. Los Angeles, CA.: ACM Press.
86. Bouraqadi-Saadani, N., T. Ledoux, and F. Rivard. *Safe Metaclass Programming*. in *Object-Oriented Programming Systems, Languages, and Applications*. 1998. Vancouver, B.C.: ACM Press.
87. Yokote, Y. *The Apertos Reflective Operating System: The Concept and its Implementation*. in *Object-Oriented Programming Systems, Languages, and Applications*. 1992. Vancouver, B.C.: ACM Press.
88. Cointe, P. *Metaclasses are First Class: the ObjVlisp Model*. in *Object-Oriented Programming Systems, Languages, and Applications*. 1987: ACM Press.
89. Foote, B. and R.E. Johnson. *Reflective Facilities in Smalltalk-80*. in *Object-Oriented Programming Systems, Languages, and Applications*. 1989: ACM Press.

90. Forman, I.R. and S.H. Danforth, *A New Dimension in Object-Oriented Programming Putting Metaclasses to Work*. 1999, New York, N.Y.: Addison-Wesley.
91. Hinkle, B. *Reflective Programming in Smalltalk-80*. in *Object-Oriented Programming Systems, Languages, and Applications*. 1994: ACM Press.
92. Maes, P. *Concepts and Experiments in Computational Reflection*. in *Object-Oriented Programming Systems, Languages, and Applications*. 1987: ACM Press.
93. Mulet, P., J. Malenfant, and P. Cointe. *Towards a Methodology for Explicit Composition of MetaObjects*. in *Object-Oriented Programming Systems, Languages, and Applications*. 1995: ACM Press.
94. Briot, J.-P. and P. Cointe. *Programming with Explicit Metaclasses in Smalltalk-80*. in *Object-Oriented Programming Systems, Languages, and Applications*. 1989: ACM Press.
95. Abelson, H., G.J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*. 1985, Cambridge, MA.: MIT Press.
96. Bouraqadi-Saadani, N. and T. Ledoux, *Supporting AOP Using Reflection*, in *Aspect-Oriented Software Development*. 2005, Addison-Wesley: New York, New York. p. 261-279.
97. Bouraqadi-Saadani, N. and T. Ledoux, *Aspect-Oriented Programming Using Reflection*. 2002, Ecole des Mines de Douai and Ecole des Mines de Nantes: Douai Cedex and Nantes Cedex - France. p. 1-16.
98. Bouraqadi-Saadani, N., *Safe Metaclass Composition Using Mixin-Based Inheritance Power Point Slides*. 2003, Ecole des Mines: Douai, France.
99. Bouraqadi-Saadani, N., *MetaclassTalk: A Testbed for Exploring Programming Paradigms Power Point Slides*. 2003, Computer Science Laboratory - Ecole des Mines: Douai, France.
100. Bouraqadi-Saadani, N., *MetaclassTalk & Metaclass Composition Power Point Slides*. 2003, Computer Science Laboratory - Ecoles des Mines: Douai, France.
101. Bouraqadi-Saadani, N., *AOP using Reflection in Smalltalk: The MetaclassTalk Experiment Power Point Slides*. 2003, Computer Science Laboratory - Ecole des Mines de Douai: Douai, France.
102. Bouraqadi-Saadani, N. *Metaclass Composition Using Mix-in-Based Inheritance*. in *11th Smalltalk ESUG Conference*. 2003. Bled, Slovenia.
103. Scharli, N., et al. *Traits: Composable units of behavior*. in *ECOOP 2003*. 2003: LNCS 2743, Springer Verlag.
104. Walsh, D. and F. Bordeleau. *Towards Open Context Environments*. in *MICON '01*. 2001. Ottawa: In Proceedings of MICON'2001.
105. Bannock, G. and W. Manser, *The Penguin International Dictionary of Finance Fourth Edition*. 2003, New York, N.Y.: Penguin Books.
106. Yoder, J. and J. Barcalow, *Architectural Patterns for Enabling Application Security*, in *Pattern Languages of Program Design Four*, P.-B. Editors, Editor. 1998, Addison-Wesley.