

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

**ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600**

UMI[®]

NOTE TO USERS

This reproduction is the best copy available.

UMI[®]

DISSERTATION

RUN-TIME CONFIGURATION
MANAGEMENT SPECIFICATION
FOR DISTRIBUTED SYSTEMS

Submitted by

Geri Georg

Computer Science Department

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Fall 2001

UMI Number: 3038635

Copyright 2001 by
Georg, Geri Gafford

All rights reserved.

UMI[®]

UMI Microform 3038635

Copyright 2002 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Copyright by Geri C. Georg

All Rights Reserved

COLORADO STATE UNIVERSITY

September 28, 2001

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER
OUR SUPERVISION BY GERI G. GEORG ENTITLED RUN-TIME
CONFIGURATION MANAGEMENT SPECIFICATION FOR DISTRIBUTED
SYSTEMS BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work

Harikaran Iyer

DB

Stephen B. Keider

James M. R.

Adviser

Stephen B. Keider

Department Head

ABSTRACT OF DISSERTATION

RUN-TIME CONFIGURATION MANAGEMENT SPECIFICATION FOR DISTRIBUTED SYSTEMS

Two methods are introduced; one that identifies key run-time configuration management requirements in complex distributed systems, and another that specifies these key requirements, validates their adequate representation in a specification, and verifies specification internal consistency. The methods are created using experiential observations of the development of distributed systems. They are generalized to work with distributed systems in other problem domains. Techniques used in industry are extended and combined with techniques proposed by academia to create specifications that can be validated for adequate representation of key run-time configuration management requirements as well as for internal consistency. New methods and notations combine industrial and academic methods to adequately capture flexibility and extensibility requirements in the specification of software run-time configuration management for complex distributed systems. The methods focus on software items involved in run-time configuration management.

Specification methods used in industry are not expressive enough to precisely specify key software run-time configuration management requirements of loosely coupled distributed systems. This research analyzes a system that has required large amounts of engineering resources to maintain and enhance, in part as a consequence of not meeting its original run-time configuration management flexibility and extensibility

requirements. Methods are developed to adequately capture key software run-time configuration managements of this system, to validate their existence in a specification, and to verify specification internal consistency. These methods are then generalized and demonstrated on a distributed system in another problem domain. The methods are developed to be acceptable for use in an industrial software development setting. Methods are scalable to large systems since they were developed using a large system as the case study. They do not require extraordinary skills to use and are flexible to fit into the existing development paradigm of an organization.

Geri Gafford Georg
Computer Science Department
Colorado State University
Fort Collins, CO 80523
Fall 2001

ACKNOWLEDGEMENTS

I would like to thank my committee for their guidance and coaching during my research and while writing my dissertation. My thanks to Dr. Hari Iyer for agreeing to serve as my outside committee member and for his practical approach to many of the abstract topics we discussed. I would like to thank Dr. Stephen Seidman for working with me during my graduate studies and for his insight into the architectural issues I encountered. Thanks to Dr. Robert France for engaging and driving precise modeling during this research and for sharing his extensive expertise. I thank Dr. James Bieman for being my advisor and for his support and guidance during my graduate studies. I am grateful for his expertise in software engineering and methods, and his unerring ability to ask insightful questions that clarify my thoughts.

I would also like to thank my colleagues and supervisors at Agilent Laboratories for their support of my graduate studies over many years. In particular I would like to thank Dean Thompson for his guidance and expertise in the development of large scale distributed systems, and Bill Shreve for allowing me to take a leave of absence to finish my research and dissertation. I would like to thank the faculty and staff of the Computer Science Department for their support and encouragement.

Finally, I am deeply grateful to my family, whose continual support, encouragement, and insightful discussions made this effort possible.

Geri Georg
September 28, 2001

CHAPTER DESCRIPTIONS

Abstract - A brief description of the research and its results.

Chapter 1 – Introduction: Problem and background information, description of the initial and second distributed systems (ATM network monitoring system and bioinformatics systems, respectively), problems with existing industrial and academic specification, validation, and verification techniques, and characterization of the research solution.

Chapter 2 – Related Work: Previous work in the general area of configuration management, run-time configuration management, requirements elicitation, and specification techniques. The three specification methods developed in academia that are studied in this research are discussed in detail. These techniques are property refinement, fuzzy sets, and soft goals. The three validation and verification methods developed in academia that are studied in this research are also discussed in detail. These techniques are composability reachability analysis, abstraction, and multiple view analysis. Other specification and validation and verification methods that have been developed both in academia and for industrial use are also described.

Chapter 3 – Research Methodology: A description of the inductive research method used in the research, its generalization, and evaluation criteria.

Chapter 4 – Identifying Key Run-Time Configuration Management Requirements in Distributed Systems: A methodology to identify key requirements (the

KR method) and demonstration of its use and ability to identify key issues in the ATM network monitoring system and the bioinformatics system.

Chapter 5 – Evaluating Specification Techniques Using the ATM Network Monitoring System: Several academia-developed specification techniques (property refinement, fuzzy sets, soft goals, Alloy, and the temporal logics RTL, TILCO-X, and interval logic) are applied to the ATM/IP network monitoring system software run-time configuration management requirements. Technique results and applicability are discussed. Specification techniques currently in use in industry are also applied (sets and UML/OCL). From these results, a method and notation are proposed for run-time configuration management specification of ATM system run-time software configuration management requirements (the specification portion of the CMSpecV&V method).

Chapter 6 – ATM Monitoring System Software Run-Time Configuration Management Specification – Initial Version: The ATM network monitoring system software run-time configuration management specification as developed from the method developed in Chapter 5.

Chapter 7 – Evaluating Validation and Verification Techniques Using the ATM Network Monitoring System: Validation and verification of the ATM network monitoring system software run-time configuration management specification. Four techniques developed in academia are used (problem abstraction, composability reachability analysis, multiple view analysis, and the Alloy automated tool), and their results and applicability are discussed. Problems with the specification presented in Chapter 6 are discussed and resolved. From these results, a method (the validation and

verification portion of the CMSpecV&V method) is proposed for the validation and verification of specifications developed using the technique proposed in Chapter 5.

Chapter 8 – ATM Monitoring System Software Run-Time Configuration Management Specification – Version 2: The corrected specification of the ATM network monitoring system is presented.

Chapter 9 – Applying the Proposed Specification Method to a Bioinformatics System: Use of the specification portion of the CMSpecV&V method developed in Chapter 5 on a bioinformatics system, and the resulting software run-time configuration management specification.

Chapter 10 – Applying the Proposed Validation and Verification Method to the Configuration Management Specification of the Bioinformatics System: Use of the validation and verification portion of the CMSpecV&V method developed in Chapter 7 on the bioinformatics system specification and results of the method.

Chapter 11 – Research Interpretations: Implications of the research and general observations made during development and use of the CMSpecV&V method.

Chapter 12 – Conclusions: Key results of the research, additional contributions, and proposals for future research.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	V
CHAPTER DESCRIPTIONS.....	VI
TABLE OF CONTENTS	IX
CHAPTER 1 – INTRODUCTION.....	1
1. PROBLEM STATEMENT	1
2. SCOPE OF THE PROBLEM	3
2.1 <i>Run-Time Configuration Management: Operations that Structure Items</i>	3
2.2 <i>Run-Time Configuration Management: Items that must be Structured</i>	5
3. DESCRIPTIONS OF AN ATM NETWORK MONITORING SYSTEM AND A BIOINFORMATICS SYSTEM	9
3.1 <i>ATM Technology, Networks, and Monitoring Systems</i>	9
3.2 <i>Bioinformatics Systems</i>	13
4. PROBLEMS WITH CURRENT SPECIFICATION METHODS USED IN INDUSTRY	20
5. PROBLEMS WITH SPECIFICATION METHODS PROPOSED BY ACADEMIA	22
5.1 <i>Specification problems:</i>	22
5.2 <i>Verification/validation problems:</i>	25
5.3 <i>Scaling problems:</i>	26
6. CHARACTERIZATION OF THE DESIRED SOLUTION	27
6.1 <i>Methods Developed by the research</i>	29
6.1.1 <i>KR Method Steps</i>	30
6.1.2 <i>Specification Portion of CMSpecV&V Method</i>	30
6.1.3 <i>Validation and Verification Portion of CMSpecV&V Method</i>	32
CHAPTER 2 – RELATED WORK	34
1. INTRODUCTION	34
2. CONFIGURATION MANAGEMENT.....	35
2.1 <i>Run-time Configuration Management</i>	35
3. REQUIREMENTS ELICITATION	38
4. SPECIFICATION METHODS.....	40
4.1 <i>Run-time Configuration Specification and Design Techniques</i>	40
4.2 <i>Other Distributed Systems Specification and Design Techniques</i>	41
4.3 <i>Potentially Adaptable Specification Techniques</i>	42
4.3.1 <i>Property Refinement Methods</i>	42
4.3.2 <i>Fuzzy sets</i>	44
4.3.3 <i>Soft Goals</i>	46
5. VALIDATION AND VERIFICATION METHODS	49
5.1 <i>Problem Abstraction</i>	50

5.2 Composability Reachability Analysis	50
5.3 Multiple View Analysis	51
6. ADDITIONAL SPECIFICATION METHODS	52
6.1 Other Specification Methods.....	53
7. ADDITIONAL VALIDATION/VERIFICATION METHODS	57
CHAPTER 3 – RESEARCH METHODS	58
1. RESEARCH METHODS USED	58
2. SPECIFIC USE OF THE RESEARCH METHOD	63
2.1 Technique selection criteria.....	64
2.2 Idea generating criteria	66
2.3 Technique evaluation criteria	66
CHAPTER 4 - IDENTIFYING KEY RUN-TIME CONFIGURATION MANAGEMENT REQUIREMENTS IN DISTRIBUTED SYSTEMS.....	68
1. INTRODUCTION	68
2. CONFIGURABLE ELEMENTS AND STRUCTURING CAPABILITIES	70
3. APPLYING THE KR METHOD TO THE ATM/IP NETWORK MONITORING SYSTEM	72
3.1 Results	80
4. APPLYING THE KR METHOD TO THE BIOINFORMATICS SYSTEM.....	80
4.1 Results	86
5. CONCLUSIONS.....	86
CHAPTER 5 – EVALUATING SPECIFICATION TECHNIQUES USING THE ATM NETWORK MONITORING SYSTEM.....	91
1. INTRODUCTION	91
2. ATM MONITORING SYSTEM CONFIGURATION ITEMS AND THEIR CONFIGURATION ACTION NEEDS	93
3. THE PROPERTY REFINEMENT SPECIFICATION TECHNIQUE	97
3.1 Properties.....	98
3.1.1 Incremental Property.....	98
3.1.2 Automatic Property.....	99
3.2 Conclusions.....	99
3.3 Property Refinement Evaluation.....	100
4. THE FUZZY SETS SPECIFICATION TECHNIQUE.....	101
4.1 Requirements Relationships.....	103
4.2 Observations	105
4.3 Conclusions.....	106
4.4 Fuzzy Sets Evaluation	107
5. THE SOFT GOALS SPECIFICATION TECHNIQUE	108
5.1 Conclusions.....	111
5.2 Softgoals Evaluation	112
6. SPECIFICATION USING SETS.....	114
6.1 Set Modeling Evaluation.....	118
7. DEVELOPING TYPE MODELS USING CONCEPTS, UML TYPES, AND OCL.....	119
7.1 Type Model.....	121
7.2 Observations	123

7.3 <i>Type Model Evaluation</i>	131
8. USING ALLOY TO COMPLEMENT THE UML SPECIFICATION	132
8.1 <i>ATM monitoring system Alloy model development</i>	133
8.2 <i>Alloy model testing</i>	133
8.3 <i>Observations about Alloy</i>	135
8.4 <i>Alloy Evaluation</i>	136
9. TEMPORAL ISSUES	138
9.1 <i>Temporal Logic</i>	139
9.2 <i>Temporal Logic (RTL) Evaluation</i>	146
10. SUMMARY	147
10.1 <i>Property refinement</i>	147
10.2 <i>Fuzzy sets</i>	148
10.3 <i>Soft goals</i>	148
10.4 <i>Set Modeling</i>	148
10.5 <i>UML/OCL</i>	149
10.6 <i>Alloy</i>	150
10.7 <i>Temporal Logic</i>	150
11. PROPOSED SPECIFICATION METHOD AND NOTATION	151
11.1 <i>A Recommended Specification Method</i>	151
11.2 <i>Recommended Specification Notation</i>	152
APPENDIX 5A –COMPLETE UML/OCL RUN-TIME CONFIGURATION MANAGEMENT SPECIFICATION FOR THE ATM NETWORK MONITORING SYSTEM	154
1. UML TYPE MODELS	154
2. OCL DEFINITIONS	159
CHAPTER 6 - ATM MONITORING SYSTEM SOFTWARE RUN-TIME CONFIGURATION MANAGEMENT SPECIFICATION – INITIAL VERSION	165
1. THE <i>TYPE</i> MODEL FIGURES	165
2. THE ALLOY MODEL	170
CHAPTER 7 – EVALUATING VALIDATION AND VERIFICATION TECHNIQUES USING THE ATM NETWORK MONITORING SYSTEM..	183
1. INTRODUCTION	183
2. THE MULTIPLE VIEW ANALYSIS TECHNIQUE	184
2.1 <i>Consistency Check Results:</i>	192
2.2 <i>Observations</i>	194
2.3 <i>Multiple View Analysis Evaluation</i>	195
3. THE PROBLEM ABSTRACTION TECHNIQUE	196
3.1 <i>Observations</i>	197
3.2 <i>Problem Abstraction Evaluation</i>	197
4. THE COMPOSABILITY REACHABILITY ANALYSIS TECHNIQUE	199
4.1 <i>Observations</i>	200
4.2 <i>Composability Reachability Analysis Evaluation</i>	201
5. AUTOMATED TOOLS BASED ON SPECIFICATION NOTATION	203

5.1 Automated Alloy Tool Evaluation.....	205
6. RECOMMENDED SPECIFICATION VERIFICATION METHOD	207
APPENDIX 7A - MULTIPLE ARCHITECTURAL VIEW CONSISTENCY CHECKS.....	208
1. FUNCTIONAL/INTERACTION VIEWS.....	208
2. INTERACTION/DEPENDABILITY VIEWS.....	210
3. INTERACTION/SCALABILITY VIEWS	211
4. INTERACTION/INCREMENTAL VIEWS	212
5. INTERACTION/AUTOMATIC VIEWS.....	212
6. FUNCTIONAL/SCALABILITY VIEWS	213
7. FUNCTIONAL/INCREMENTAL VIEWS	213
8. FUNCTIONAL/AUTOMATIC VIEWS.....	214
9. FUNCTIONAL/DEPENDABILITY VIEWS.....	215
10. DEPENDABILITY /SCALABILITY VIEWS	215
11. DEPENDABILITY /INCREMENTAL VIEWS	216
12. DEPENDABILITY /AUTOMATIC VIEWS.....	217
13. SCALABILITY /INCREMENTAL VIEWS.....	217
14. SCALABILITY /AUTOMATIC VIEWS	218
15. INCREMENTAL /AUTOMATIC VIEWS.....	219
APPENDIX 7B – USING PROBLEM ABSTRACTION TO GENERATE TEST CASES FOR THE ATM MONITORING SYSTEM	220
APPENDIX 7C – USING COMPOSABILITY REACHABILITY TO VALIDATE THE ATM MONITORING SYSTEM.....	224
Observations	229
CHAPTER 8 - ATM MONITORING SYSTEM SOFTWARE RUN-TIME CONFIGURATION MANAGEMENT SPECIFICATION – VERSION 2	233
1. THE TYPE MODEL FIGURES	233
2. THE ALLOY MODEL	234
APPENDIX 8A – REVISED COMPLETE UML/OCL SPECIFICATION	252
1. TYPE MODEL DIAGRAMS	252
2. OCL DEFINITIONS	253
CHAPTER 9 – APPLYING THE PROPOSED SPECIFICATION METHOD TO A BIOINFORMATICS SYSTEM.....	265
1. INTRODUCTION	265
2. STEP 1: BIOINFORMATICS SYSTEM FOCUS AND CONFIGURATION MANAGEMENT ISSUES.....	267
3. STEP 2: PROPERTY REFINEMENT.....	267
3.1 Incremental Operation Property.....	268
3.2 Persistence Property.....	269
3.3 Conclusions.....	270
4. STEP 3: SOFT GOALS	271
4.1 Conclusions.....	273

5. STEP 4: <i>TYPE MODEL</i>	274
5.1 <i>General description of the type model</i>	274
5.2 <i>Type Model Figures</i>	276
6. STEP 5: USING ALLOY TO COMPLEMENT THE TYPE MODEL	276
6.1 <i>Bioinformatics run-time configuration management specification Alloy model</i>	282
7. STEP 6: TEMPORAL ISSUES	293
8. SUMMARY	296
8.1 <i>System Focus</i>	296
8.2 <i>Property refinement</i>	296
8.3 <i>Soft goals</i>	297
8.5 <i>Type Model</i>	297
8.6 <i>Alloy</i>	297
8.7 <i>Temporal Logic</i>	298
9. CONCLUSIONS	299
CHAPTER 10 – APPLYING THE PROPOSED VALIDATION AND VERIFICATION METHOD TO THE CONFIGURATION MANAGEMENT SPECIFICATION OF THE BIOINFORMATICS SYSTEM	301
1. INTRODUCTION	301
2. MULTIPLE VIEW ANALYSIS	302
2.1 <i>Consistency Checks:</i>	311
2.1.1 <i>Functional/Interaction Views</i>	311
2.1.2 <i>Interaction/Dependability Views</i>	311
2.1.3 <i>Interaction/Scalability Views</i>	312
2.1.4 <i>Functional/Scalability Views</i>	312
2.1.5 <i>Functional/Dependability Views</i>	312
2.1.6 <i>Dependability /Scalability Views</i>	313
2.2 <i>Consistency Check Results:</i>	314
2.3 <i>Observations</i>	314
3. ALLOY AUTOMATED TOOL ANALYSIS	315
4. CONCLUSIONS	318
CHAPTER 11 – RESEARCH INTERPRETATIONS	321
1. INTRODUCTION	321
2. RESEARCH IMPLICATIONS	321
3. GENERAL OBSERVATIONS	324
CHAPTER 12 – CONCLUSIONS	327
1. INTRODUCTION	327
2. KEY RESULTS	327
2.1 <i>Key System Requirements Identification</i>	327
2.2 <i>Specification Methodology and Notation</i>	330
2.3 <i>Specification Validation and Verification</i>	332
2.4 <i>Generalizations</i>	332
3. CONTRIBUTIONS	333
4. FUTURE WORK	335

REFERENCES..... 337

CHAPTER 1 - INTRODUCTION

1. PROBLEM STATEMENT

This dissertation describes methods for the derivation and specification of run-time configuration management system attributes. One method identifies key run-time configuration management requirements in complex distributed systems. The dissertation also describes a method to specify these key requirements, validate their adequate representation in a specification, and verify specification internal consistency with respect to the key requirements. The methodologies are created using experiential observations of the development of distributed systems. They are generalized to work with distributed systems in other problem domains. Techniques used in industry are extended with techniques proposed by academia and new techniques to create specifications that can be validated for adequate representation of key run-time configuration management requirements as well as for internal consistency. The research develops new methods and notations that combine industrial and academic methods to adequately capture flexibility and extensibility requirements in the specification of software run-time configuration management for complex distributed systems. The research focuses on software item specification since run-time configuration management of software components is often an issue for loosely coupled distributed systems.

Specification methods used in industry are not expressive enough to precisely specify key run-time configuration management requirements of loosely coupled

distributed systems.¹ This research examines a system that has required large amounts of engineering resources to maintain and enhance as a consequence of not meeting its original flexibility and extensibility requirements. The system is an ATM/IP network monitoring system. It is a distributed system that contains measurement components as well as computational components.² The methods developed using the ATM/IP network monitoring system are demonstrated on a distributed system in another problem domain, that of bioinformatics.

Run-time configuration management flexibility requirements mean that it is possible to logically or physically move portions of a distributed system, e.g. to logically move a set of resources such as a user interface, analysis routines, and physical instrumentation to create a “virtual instrument” for the duration of a measurement, or to physically move instrumentation or software from one computer to another. Extensibility requirements in such a system mean that it is possible to create new virtual systems (or subsystems, or instruments) by logically grouping resources (including software), or to add another computing system or an instrument to a computer. Flexibility and extensibility requirements are generally applicable to all distributed systems, although specific requirements may vary between problem domains.

The rest of this chapter is organized as follows. Section 2 describes the run-time configuration management characteristics of a loosely coupled distributed system. The discussion includes all types of information and resources that may need to be configured while a system is in operation. However, the remainder of the research and the methods developed by the research are focused on run-time configuration

¹ Section 4 of this chapter discusses these and other problems with specification methods used in industrial software development.

² The system is fully described in Section 3 of this chapter.

management issues associated with software items. Section 3 describes the initial distributed system used in this research, an ATM/IP hybrid network monitoring system. The second distributed system used in this research, a bioinformatics system is also described in Section 3. Section 4 describes some of the challenges with specification methods currently in use in industry when applied to the run-time configuration management needs of a loosely coupled distributed system, and Section 5 describes some of the challenges of applying research specification methods to those same needs. Section 6 describes the characterization of the desired solution to these problems.

2. SCOPE OF THE PROBLEM

Run-time configuration management requirements are one kind of system-wide behavior constraint on distributed systems. It is difficult to define these requirements because each system has different run-time configuration management needs depending on its problem domain, size, and customer usage patterns. Examples of run-time configuration management requirements are given below.

2.1 Run-Time Configuration Management: Operations that Structure Items

Run-time configuration management includes the ability to structure a system. The structuring operations must be identified along with items in the system that will be structured. The structure operation list below includes examples of items that can be structured using the operations. In a loosely coupled distributed system, *structuring operations* that need to be addressed are the following:

- Defining the constituents of a desired system structure and ensuring that this desired structure obeys overall static and dynamic system constraints (e.g. “one instance of a location service, version 2.3, needs to exist in the system and three

instances of analysis applications need to exist in the system”). System structure may include physical hardware, software, network connections, logical groupings of hardware and/or software, etc.

- Registering items in the system structure (e.g. “let other computers know that version 2.3 of a location service is available on computer XYZ”, or that “another instrument has been added to the system”).
- Locating items in the system (e.g. “find a computer running version 2.3 of the location service”, or “find a communication path to database QRS”).
- Migrating items in the system. *Migration* may need to be interpreted differently depending on the problem domain. For example, in the bioinformatics domain it may make sense to think about migrating a data source from one public database to another. However, in the context of agents [1, 62, 89], *migration* refers to the movement of an executing agent from one computer to another, along with associated context and data.³
- Binding items in the system (e.g. “bind the resources of computer XYZ to version 2.3 of the location service”, or “bind the output of instrument A to the input stream of analysis program B”).
- Deploying items around the system (e.g. “copy version 2.3 of a location service to computer XYZ”). The deployment capability physically moves items around the system. Depending on the problem domain, it may make sense to interpret the term *deployment* to include not only executable components of the system, but

³ An agent is a software entity capable of autonomous operation, including decision making and learning [62].

also items such as user information. Deployable items may consist of software items (i.e. source code, object code, objects, services, and applications), functionality (since functionality ultimately consists of executable code), user information, capabilities, and roles. Deployment differs from migration in that migration moves items already running (e.g. executing code) or in use in the system (e.g. a data source), whereas deployment moves items that are not yet executing (e.g. an application executable file), or in use.

- Activating and deactivating items in the system (e.g. “start or re-start a location service on computer XYZ”, “stop execution of a location service on computer XYZ”, or “trigger instrument Q to begin taking measurements”).
- Instantiating items in the system (e.g. “create version 2.3 of a location service on computer XYZ”). Instantiation can be similar to activation. The difference lies in whether the item being created also begins running by the act of creation or whether another action is needed to actually start it. Similarly, the item may be stopped (deactivated), but not destroyed. It can still exist (be instantiated), and then it must be restarted with an activation operation. Instantiation makes sense for executable items in the system such as object code, objects, services, applications, and agents. Instantiation can also be used to create a logical grouping of resources, or a virtual instrument.

2.2 Run-Time Configuration Management: Items that must be Structured

The types of *items* that need to be considered for run-time configuration management in a loosely coupled distributed system vary from system to system. This variation, along with variations in the types of structuring needs, lead to differences in

the run-time configuration management capabilities required by a system. Variations in the number and kinds of data sources can also influence run-time configuration management requirements.

Single data source distributed systems consist of users, computers, and the connections between them. Data in such a system comes from a single kind of source such as a database, instruments, or users themselves. Multiple data source distributed systems consist of users, computers (and their associated resources), multiple kinds of data sources (often including external databases and/or instrumentation), and the connections between them. These additional data sources may either be physically separate instruments, embedded devices such as sensors directly connected to other controllers in the system (both of which generate measurement data), or other sources of data such as geographically remote computing systems or databases. Quite often, multiple data source distributed systems contain a mixture of measurement and geographically remote data sources.

The *items* that may need to be configured in loosely coupled distributed systems and examples of their structuring needs are listed below:⁴

- Communication paths; e.g. a pair of computers may physically be connected via a local area network (LAN) and an infrared (IR) connection. Configuring the communication path may consist of *registering* a choice to use the LAN path over the IR path unless network traffic is very high.

⁴ Some of the items in the list above are also referenced in [2, 5, 9, 62, 69].

- Software (this can include source and object code, as well as directly executable code such as an application); an example of structuring software is *registering* the version number of an application that resides on a computer.
- Agents; examples of structuring agents are *registering* the functionality an agent needs from a host computer and then choosing to *migrate* (i.e. move the agent code and its execution state) to a particular machine based on the functionality it can provide to the agent.
- Objects; an example of structuring object is *locating* objects of a particular class that are instantiated on several different computers and then choosing one particular object to access.
- Data (this can include data resulting from measurements or computation, or data that resides in data storage warehouses or databases); an example of structuring data is *registering* the location of replicated data.
- Computing resources (these resources include the physical resources such as processor capabilities, memory, disk storage space, network bandwidth, etc); a structuring example is *registering* the location of processing, memory, and storage resources.
- Functionality (these items include analysis functionality, management functionality, security, etc); structuring examples of functionality include *locating* particular analysis routines that may exist on multiple machines or *activating* user management or system backup functions.

- User information; examples of structuring user information are *locating* a login name or network domain.
- Capabilities; an example of structuring capabilities is *locating* the right of access to particular data or applications
- Roles (roles are not necessarily associated with particular people, they can be considered equivalent to the actors in a use case and may therefore include system actors as well); examples of structuring roles are *locating* an administrator, novice user, or a data source role.
- Instrumentation resources; these are separate instruments (e.g. a voltmeter) or embedded devices (e.g. a “smart sensor” containing not only a sensing device but also an embedded local control device). An example of their structuring capability is *registering* their location and the types of measurements they can obtain.
- Measurements themselves can be subject to configuration. An example is a phase noise measurement; phase noise is the remaining instability in frequency stable oscillators that can cause close-in signals to be unobservable in signal analyzers [81]. This kind of measurement is complicated to perform, requiring not only measuring a signal, but also performing analytical analysis on the result in order to create a final output. An example of structuring this kind of item is to *register* where it is available.
- *Items* based on the logical or hierarchical grouping of other *items* in the system; an example of logical structuring items is *defining* a particular set of analytic, instrumentation, and storage resources as one *item*.

- Geographically remote databases that supply data for analysis in the system; a structuring examples is *binding* a remote database as a data source.
- The semantic meaning of data obtained from remote sources; an example of structuring this item is *defining* semantic meaning.

Items that need to be considered for structuring are often obvious, but sometimes more obscure items should be considered as well. More obscure items are often direct results of the characteristics of the problem domain for which the system is being developed. For example, in the ATM network monitoring system an item that needs to be considered for run-time configuration management is a *configuration* (a *configuration* is a description of the versions of software, the machines on which they should execute, the portions of the network that should be monitored, the types of analyses that should be used, etc). A *configuration* provides a logical grouping of the state of the system and its monitoring functions. An example of a new item in the bioinformatics system that needs to be configured is a research thread. (A research thread is a plan for research, searches, and analyses as well as a trail of previously executed research along with results. A research thread shows all research paths, including non-profitable paths, that culminate in new knowledge and usually in published results.)

3. DESCRIPTIONS OF AN ATM NETWORK MONITORING SYSTEM AND A BIOINFORMATICS SYSTEM

3.1 ATM Technology, Networks, and Monitoring Systems

Asynchronous Transfer Mode (ATM) technology can be used to build high speed communication systems for local area networks and wide area networks [66].

ATM technology divides voice, data, image, and video into small packets and interleaves their transmission across an ATM link. The network bandwidth is divided into periodic slots that are assigned to calls. Slots are allocated on demand, so no bandwidth is used unless information is transmitted. ATM requires that a connection be established between two end points using a signaling protocol prior to any data transfer. Packets are then routed to destinations with minimal information required in their headers. The International Telegraph and Telephone Consultative Committee (CCITT) has agreed upon the use of fixed sized slots/cells of 48 bytes of data and 5 bytes of header.

Logical connections between two endpoints in an ATM network are called virtual connections. A virtual channel is the concatenation of the virtual links between two switching points. Traffic usage parameters associated with virtual channels include loss rate, peak rate, bandwidth, and quality of service. Virtual connections can be static (set up by manually loading routing information into the switches) or dynamic (the connection is established by signaling and the route is determined by the network equipment).

ATM networks are often part of an overall hybrid network offered by network operators or communications services providers. A hybrid network consists of a high speed ATM network that is connected at various points on its periphery to other network mediums, such as Internet Protocol (IP) networks. As in other kinds of communications networks, the quality of service between points on the network must be reliable. In addition, the network traffic itself contains information which is also of potential value to the communications provider, since it can be mined for things like billing information or fraudulent usage.

The Agilent Technologies *accessATM/IP Network Monitoring System* is an extension to a previous network monitoring system (*accessSS7*) that allows monitoring of a hybrid ATM/IP network for many different kinds of information. One of the capabilities provided by this system is monitoring quality of service measurements (e.g. delay, lost packets, peak bandwidth, etc.) between various points of the network. Since the central core of an ATM network has link speeds that are very high (up to OC192 speeds, or approximately 10Gb), real time monitoring within the core ATM network it is not feasible. Thus, the *accessATM/IP Monitoring System* uses hardware probes placed just outside the core ATM network, but before the IP portion of the hybrid network, where speeds are of the order OC3, or approximately 155Mb. These probes are connected to local area networks (LANs) at various local monitoring sites, which also contain site server computers and routers. Data is consolidated at the geographical level at these local sites. The local sites are then connected via wide area networks (WAN) to a central server that is connected via a local area network to user workstations. Operators can access the monitored information that has been collected throughout the monitoring network from these workstations, as well as direct the operation of the overall system. A simplified version of this entire system is shown in Figure 1.⁵

The ATM/IP network monitoring system consists of probes that are physically attached to the network at various points. The probes capture packet headers and strip off the relevant information. This information is analyzed (looking for patterns) at site servers. The site servers are connected to a central server computer via LAN or WAN (depending on their geographical locations). Final analysis and decision making occur at

⁵ Graphic developed with the help of Graham Pollock and Dean Thompson of Agilent Laboratories, Agilent Technologies.

the center site. The network hardware probes must infer information across the possibly millions of virtual circuits that may be in use within the ATM network, so there is a huge amount of data being processed at the lowest level of the monitoring system. Due to the huge amount of data in this system, a primary goal is to perform as much processing as possible at the source of the data in order to cut down on the amount of information that has to pass among computers at different geographical locations in the system. The probe data can be mined for many types of information, such as quality of service, connection information for billing purposes, and also patterns that indicate fraudulent use of the network.

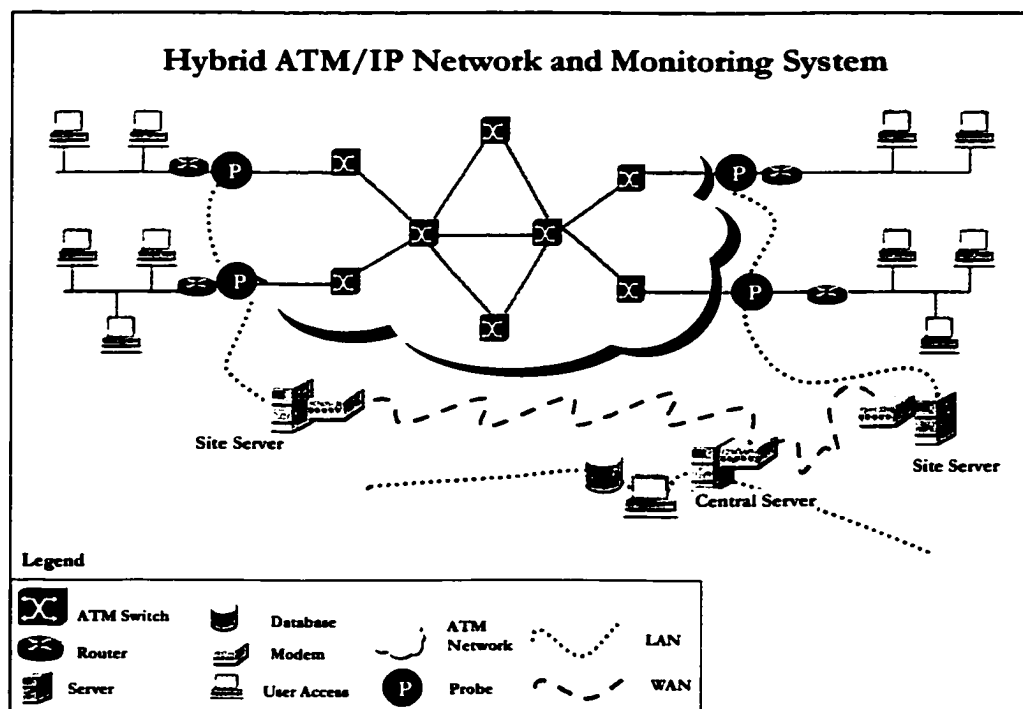


Figure 1-1. A simplified hybrid ATM/IP network and its monitoring system. Hardware monitoring probes must be located outside the core ATM portion of the network, due to the high speed of the ATM network itself (up to OC192 speeds, approximately 10Gb). (Network speed at the probes is approximately 155Mb, or OC3 speeds.) Probes are connected locally to site server computers that drive the analysis of the header information they obtain, and these local monitoring networks are further connected via wide area networks (WANs) to a central server. Workstations at the central site of the monitoring system drive the observations of the probes and the analysis of header information at local sites. The monitoring system may be comprised of tens of thousands of probes with each local site having on the order of tens of probes. There is one probe per link being monitored in the system.

The system is very dynamic; it can be changed from one that monitors network performance to one that searches for fraudulent use by changing the recognition patterns used in the system. This transformation requires changing applications that are running on the various computers. Since the computing and probe system is also huge (typically thousands of probes and hundreds of computers are involved over large geographical areas), the task of getting the correct portion of the correct application to the correct computer presents a significant challenge. A related problem is getting the correct services/applications operating on the correct computers, in the correct sequence. For example, if a probe is to begin searching for a new pattern, the local computer must already be running the new correlation and analysis program before it can receive any of the new data from the probe and produce correct results.

Another interesting aspect of this system is that since it is using many networks (e.g. LAN and WAN) that are geographically dispersed, portions of the system may be operating or not, solely depending on network status. New probes/computers may also be added at any time, further complicating the situation. Whenever the monitoring system topology changes, it must determine its configuration, and also determine what code should be running at each location, get this code to the correct location, and execute it.

3.2 Bioinformatics Systems

In biology and medicine, computing and information systems are increasingly used to capture and analyze data. Bioinformatics is the term generally applied to this field of research and application, and its application is widely used in the field of molecular biology. Molecular biology is concerned with the study of DNA sequences,

the genes associated with them, the end products of the genetic codes (proteins), and their ongoing interactions during the life of cells. One of the sources of data being obtained for many different organisms is the sequence of base pairs in DNA. The genetic material of some organisms has been completely determined, while that of others is still in process. Sequence data is typically stored as ASCII strings, representing the first letter of base pairs (C, G, A, and T). It is available in several public databases for researchers to use. However, the volume of data in these databases is very large; the GenBank public database held approximately 746 million base pairs as of the end of 1997. Automated sequencing techniques are expected to increase database size at least 10X every three years, and some experts predict that the sequence data will actually grow exponentially [88].

Sequence information is also being gathered for the protein products of genetic transcription and translation. This information, in the form of ASCII representations of the amino acid sequences is also available to researchers in public databases. Examples are the Swiss-Prot database that contains 22 million amino acids, and the Protein Information Resource database that contains 32 million amino acids. In general, the sequences in the public genetic and protein databases, along with the annotations included for each sequence, account for many gigabytes of data. Finding relevant information in this mass of data is a daunting task for a researcher.

Sequence data is important to a researcher because it allows experimentation to determine what proteins are present/absent as part of a disease. Experiments using proteins are more straightforward in their conclusions, but it is very difficult to determine exactly what proteins are present/absent in cells. A slightly easier experimentation approach, albeit one with less conclusive results is to search for

particular genetic sequences in tissue. The theory is that if a particular genetic sequence is present, then most likely it will be *expressed*, that is translated into its associate protein, in the cell. Due to the myriad of regulation processes that exist as part of genetic translation, this is not always the case, but it is still a reasonable assumption for the development of a hypothesis regarding a disease.

As part of the research process of a disease, a researcher needs to determine what genes are related to each other as part of a metabolic pathway. These related genes produce proteins that are involved in the pathway (subject to translation regulations present in the cell), and the presence or absence of these proteins, as well as the amount of these proteins affect the extent to which the pathway is present in cells. Using the relative amounts of the genetic material in the cell as indicators of the amounts of the proteins they encode, researchers are able to hypothesize which genes are involved in particular metabolic pathways. Additional genetic expression experiments or other experimentation techniques may be used to verify these hypotheses.

One genetic expression experimentation technique is the DNA micro-array. This technique involves placing known sequences of bases (on the order of tens or hundreds of bases)⁶ in a single strand on a glass plate (typically in hundreds of thousands of locations on the plate), then exposing these known sequences to the

⁶ Although gene length varies widely, the human gene that encodes one blood clotting proteins is approximately 186 thousand bases pairs in length, but only about 9 thousand bases are present in the transcribed mRNA [15]. Since the base sequences that can be placed on a glass slide are much shorter, choosing these shorter sequences is critical in determining whether particular genes are actually present in the target unknown sequences. Bases placed on the glass plate must be specific enough to increase the likelihood that a particular gene is actually represented in the target. Thus, the sequences of many pieces of particular genes which are particular to the gene are used on an array. Designers of these *probes* rely heavily on the sequence information available in the public genetic databases to help choose these shorter sequences.

unknown makeup of genetic material in other cells (again in single strands).⁷ Generally two types of cells are used; diseased cells, and normal cells. The genetic sequences in one type of cells is tagged with one color of fluorescent dye, and the sequences in the other type is tagged with a different color of fluorescent dye (typically red and green). The unknown sequences are then placed in contact with the known sequences on the micro-array. Since the base pairs always bind in a particular way to form double strands, the unknown sequences will only bind with the sequences on the array if they are *complementary*, that is, A binds with T, and C with G. Once the genetic sequences have bound, the array is scanned with a fluorescent scanner, and the amount of red (only one kind of cell *hybridized* to the known sequence on the array at a particular position), green (only the other kind of cell hybridized), yellow (both kinds of cells hybridized), and black (neither kind of cell hybridized) color at each position on the slide is determined. The amount of genetic material from the different sequences on the array that is present in each of the unknown sequences can be determined from this intensity information.⁸

Researchers use the expression data from micro-array experiments to formulate further hypotheses regarding the genes involved in metabolic pathways or diseases. Analyses such as clustering are often performed on the raw micro-array data to suggest

⁷ Usually the sequences are derived from *messenger RNA* (mRNA) which is the result of *transcribing* a section of DNA from a gene. DNA contains many base pairs that either do not code for any proteins or are redundant. In addition, many of the cell's regulatory processes take place around DNA transcription, so looking at mRNA is a better basis for determining which proteins are actually being produced in a cell. mRNA is translated into proteins in the cell by ribosomes and *transfer RNA* (tRNA). Proteins thus produced fold into particular 3-dimensional configurations based on their amino acid sequences as well as other thermodynamic considerations and ionic makeup. They are then transported to the locations in the cells where they will operate. Many become enzymes that help the cell with its chemical reactions; others may become part of the actual structure of the cells. For an in-depth discussion of complete function of life in a cell, see [15].

⁸ Note that a large amount of data cleaning and normalization are needed to ensure that the results of the micro-array experiment are useful and correct. For example, low intensity levels of red and green may not be significant, and claiming up or down regulation of genes in these cases may not be correct. On the other hand, low levels may be significant, and then the background *noise* due to experimental procedures must somehow be extracted in order to determine actual intensities.

particular genes that may be related in a disease or pathway. In the process of formulating these hypotheses and designing further expression experiments, researchers must access other available sources and types of data. For example, the sequence databases may give a researcher an insight into a complete gene that has been up- or down-regulated in the expression experiment. Searching in the protein databases might produce the encoded protein name, or structural configuration. Searching the various medical literature databases may reveal the results of other researchers' work relating to the gene, protein, pathway involvement, or related disease. As mentioned above, the sizes of these databases is easily on the order of gigabytes of information. In addition, they are geographically distributed all over the world, and while many are publicly accessible, some are proprietary to particular organizations.

There are many tools available to researchers to search databases, most notably tools that look for possible sequence matches with experimental results and known genetic or protein sequences. These tools are typically accessed via the web and provide the ability to perform queries into particular databases. (Some companies offer tools that serve as front ends to multiple search and analysis tools, and therefore to multiple databases. They may return results the next day, or continue to look for changing/added data that changes the query results over time. In general, results are either returned via e-mail, or the researcher is notified via e-mail that results are available from a the tool web site.) BLAST and Fasta are two analysis tools that look for regions of similarity between known genetic sequences in the databases and one supplied by a researcher. If a large area of similarity is found, the researcher may be able to determine the official gene name of the sequence and use this to drive queries into protein sequence databases or the medical literature databases. It should be noted that many of the public databases

are not curated, so incorrect or partial information is often found by researchers, who must then use alternate methods to reach conclusions. In addition, some genes have multiple names due to being "discovered" by multiple parties at the same time, and are represented by different names in different databases. Information in the databases is constantly being updated and added, so obtaining relevant data can be very time consuming.

Bioinformatics systems are increasingly being used to aid researchers in their search for knowledge in this overwhelming arena of heterogeneous, geographically distributed, and dynamic data. Within a research organization, scientists typically have access to personal workstations that may be tied to an analysis computation system for their internal genomic data. These workstations also allow access to the internet, and thus to the interfaces into public databases. Sometimes the analysis system can provide ties to public databases, and sometimes researchers use public tools available via the web to perform all their analyses, and do not use an organizational analysis system at all. The results of micro-array expression experiments may be kept locally (in a private database, or often in a spreadsheet), or in an organizational database. Security issues in a bioinformatics system not include those that might be expected regarding the protection of data from outside organizations, but also internal issues regarding data protections across groups in the organization and sometimes within a group. Not only raw data, but also any annotations (e.g. hypotheses regarding its meaning, analysis information such as clusters, or results of queries to further the research) may need to be protected. A simplified diagram of a bioinformatics system is shown in Figure 2.

A comprehensive bioinformatics system must be able to support researchers in many different aspects of their research. Examples are laboratory information

management, image analysis of micro-array results, data integration with both public and private genomic and other medical databases, and data mining from existing experimental results (possibly from public databases). The latter includes data warehousing, exploration, integration with heterogeneous, geographically dispersed databases, statistical analysis, and visualization. For detailed examples of these areas of functionality, see [4] and [60].

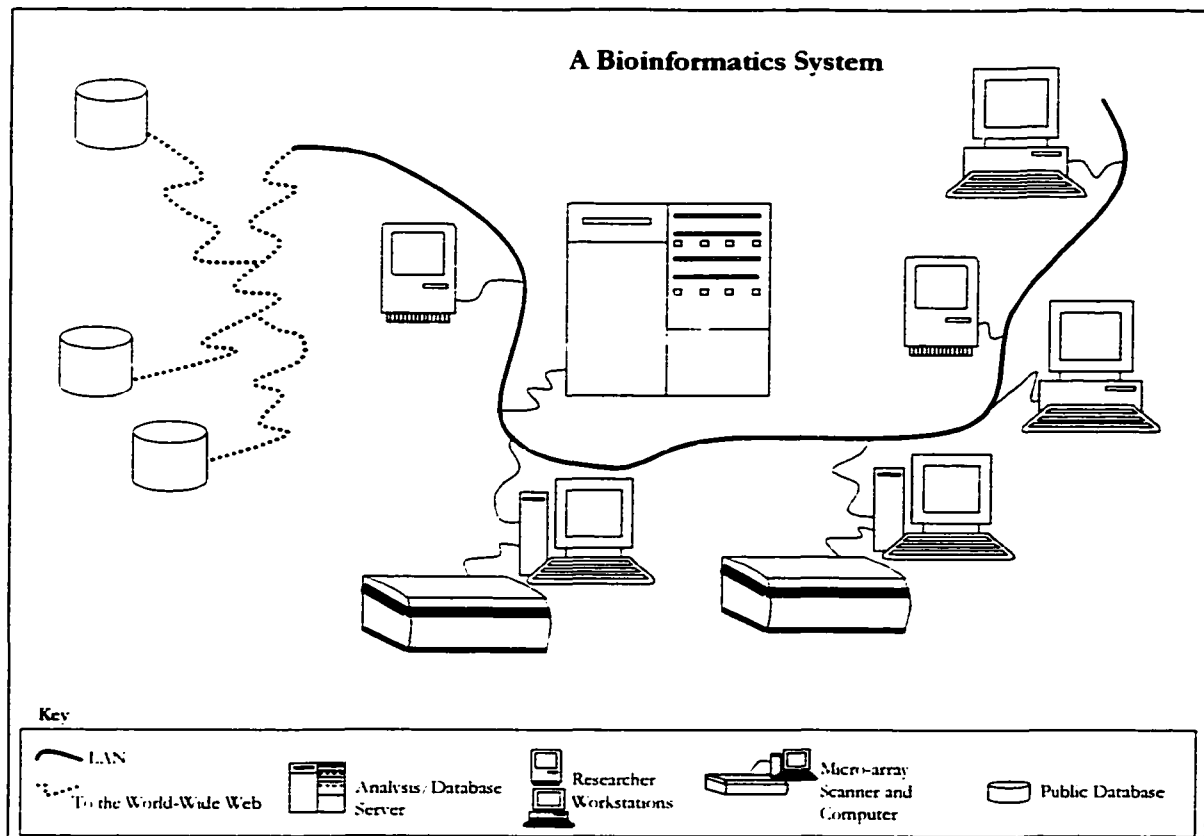


Figure 1-2. A simplified bioinformatics system. Within a research organization, researchers have access to micro-array scanners (which are generally connected to some sort of PC). Results from micro-array experiments are produced in terms of scanner images, which are then cleaned and normalized. These results can then be stored and analyzed on a researcher's local workstation or on an organizational computing resource. Researchers also access the world-wide web to gain access to other public databases.

4. PROBLEMS WITH CURRENT SPECIFICATION METHODS USED IN INDUSTRY

The specifications of many run-time configuration management requirements in large distributed systems are expressed in very general and imprecise terms. Specification documents used in industry are often only textual, especially for “non-functional” requirements such as run-time configuration management. Sometimes block diagrams or UML object models are created that serve as requirements or architectural specifications. These documents help bridge the gap between requirements and system design. Although formal specifications are used for requirements specification as well as design specifications in safety-critical applications, they are rarely found in other industrial applications. The result is imprecise specifications of key non-functional requirements, such as run-time configuration management.

It is difficult to check imprecise specifications for internal consistency and for adequate representation of key non-functional requirements. Imprecise specifications also make it difficult for designers and implementers to deduce correct behavior at the component and inter-component level, and to determine if correct behavior is missing in the design or implementation. Another problem is whether the implemented functionality will lead to the desired system-wide constraints on run-time configuration management.

Part of the problem lies in a specification process that often overlooks non-functional requirements such as run-time configuration management. Another part of the problem is the tool set used to specify requirements. An English language word processor is not a precise tool. Similarly, Unified Modeling Language (UML) object

models (without accompanying Object Constraint Language/OCL expressions) can only specify classes, relations, attributes, and operation signatures. Constraints on relations and operations are required to specify non-functional requirements. (Constraints are a minimal requirement; other constructs may be necessary depending on the non-functional requirement being specified.)

An example of the difficulty in interpreting imprecise non-functional requirements occurs in the network monitoring system.⁹ This system is so large and loosely coupled that run-time configuration changes must occur automatically. However, automatic operation must be tempered with the ability to take direction from network administrators regarding how much automation is allowed. If an imprecise description is used as a specification, designers and implementers may choose many different ways to implement both automatic operation and user-imposed bounds on that operation. If the specification is not precise, extra care must be taken especially during system design to ensure that all related subsystems and components adopt the same design for automatic operation and its user-imposed bounds. In a large and complicated system, this can be very difficult.

Imprecise specifications also lead to difficulty in verifying that the desired system-wide configuration management behavior exists in the system finally delivered to a customer. A specification must be complete and also include possible uses of the specified capability. For example, automatic run-time configuration management could be restricted to run-time changes in the physical network that is being monitored (i.e. the addition or deletion of network routers), the monitoring network itself (i.e. the addition

⁹ Although the example of automatic operation is discussed in terms of the ATM/IP hybrid network monitoring system, the research described in this dissertation is generalized to other distributed systems. The automatic operation example is used throughout this chapter for consistency.

or deletion of local site computers or central site workstations), or it could also include run-time changes to the monitoring software, such as changes to the patterns that are being sought and analyzed. Depending on the possible uses of automatic operation, different portions of the system software need to implement functionality that enable it.

A detailed and precise specification allows deeper better analysis of system wide constraints on behavior (for example in unusual or error conditions), as well as the ability to check for consistency across the specification [27, 28].

5. PROBLEMS WITH SPECIFICATION METHODS PROPOSED BY ACADEMIA

Problems with specification methods proposed by academic research can be categorized as problems associated with producing a specification, problems associated with verifying/validating a specification, and the problem of scalability, which is associated with both categories.

5.1 Specification problems:

- Few system level non-functional properties are well defined and studied: most research deals with properties such as correctness, reliability [8, 19, 83], and performance [48, 57] (usually in terms of system response time). As a result, existing techniques cannot adequately specify properties such as run-time configuration management. The gap between existing techniques and run-time configuration management requirements becomes most noticeable when part of the technique consists of refining or decomposing a high-level property into properties that are specified at a component level (see below).

- Many specification methods refine or decompose non-functional properties to the component level, and use adding or conjunction to "sum" the properties at the component level to arrive at an overall system level property. While this method works for well-studied properties such as response time, it is much harder to use for less studied properties such as run-time configuration management.

For example, consider an overall system wide constraint on behavior that activities within the system must occur within a certain time bound. Components can be specified as having particular time bound requirements, then the bounds of all sequential components in an execution path can be added to arrive at an overall system time bound for any execution path through the system. The maximum time from the parallel execution paths then defines the overall system time response bound. Refining the time bound constraint therefore leads to a statement that the maximum time of all parallel paths must be within the time bound, and that the time of a path is the sum of the times of all its sequential components. Figure 1-3 shows an example of this constraint.

Refining the meaning of run-time configuration management capabilities is much more difficult, and is also domain and problem specific. Consider the automatic operation of run-time configuration management in the network monitoring system. One of the first problems with specifying this capability is refining the definition of automatic run-time configuration management operation; it can refer only to changes in the physical

monitoring system or be extended to include changes in pattern searching and analysis software.

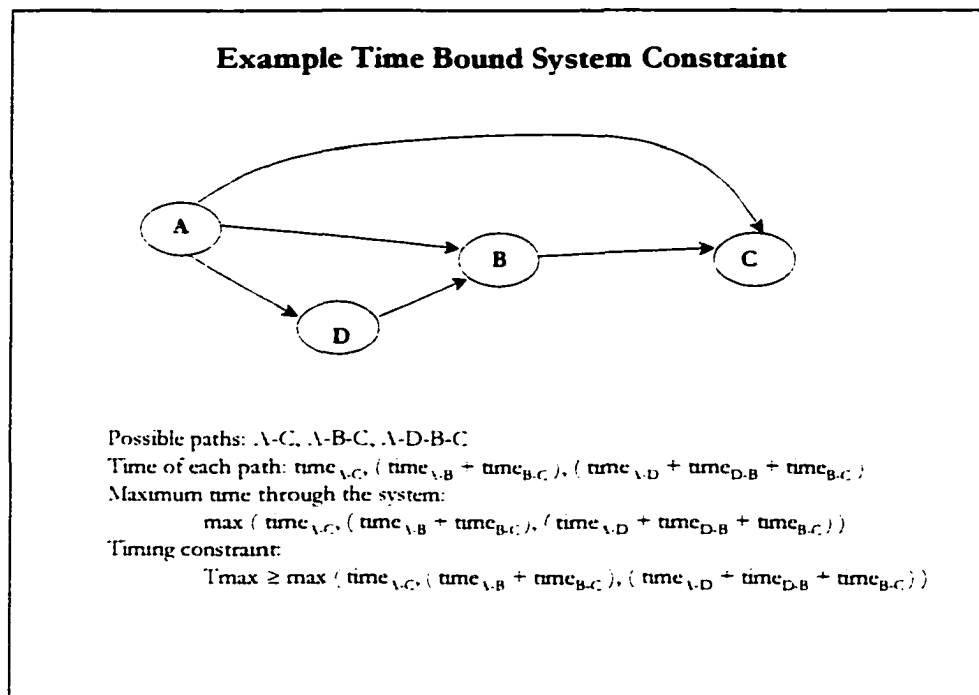


Figure 1-3. An example of a system-wide constraint on time bounds. There are three execution paths shown in the figure. The first is through A and B, the second through A, B, and C, and the third through A, D, B, and C. The maximum time it takes to traverse through the portion of the system shown in the figure is the maximum time taken by the longest path (in terms of time). The time taken by any path is the sum of the times taken by each of its constituent parts (i.e. A, B, C, and D).

Another problem is defining constraints on automatic operation. This is especially a problem if configuration changes to monitoring software are included in automatic operation. The system must not continually attempt to change software and never monitor the network, so some constraint on how often the system attempts software changes must be present. These problems revolve around the fact that run-time configuration management has not been studied to the same depth as behaviors such as system response time; refinements related to run-time configuration management are not well understood nor agreed upon.

5.2 Verification/validation problems:

- Composing component behavior to obtain an overall view of run-time configuration management behavior is difficult. Individual component run-time configuration management capabilities (e.g. entities that allow automatic operation) cannot simply be “added” together along an execution path in order to obtain an overall view of any particular run-time configuration management capability of a system. Component aspects of run-time configuration management interact in different ways that have not been well studied. Most system properties studied to date are verified by adding component properties, or by conjugating component properties [40, 65]. This is clearly not adequate for run-time configuration management system capabilities - a different way of combining component capabilities is needed.

As an example, again consider automatic operation in the network monitoring system. If a component implements methods that allow automatic operation and another component does not, “adding” the components’ behavior does not indicate whether or not automatic operation exists. In addition, a component may implement methods that allow certain automatic operations, but not others. A different component may implement methods that allow different automatic operation. Both components allow automatic operation, but simple conjunction of their capabilities probably does not imply that the overall automatic operation behavior is adequately implemented.

- Many specification and verification techniques resulting from academic research assume the use of formalisms (e.g. Z specifications or temporal calculus) in both a specification [13, 19, 40, 61, 83] and in the process of verifying system-wide behaviors (e.g. theorem provers) [13, 83]. These techniques do not often scale well to industrial use and to large systems [18, 26, 31, 32, 37, 41, 74, 91]. However, in some problem domains these techniques are useful [27, 28]. The main problem with trying to use these methods on a large system is that they require too much time, personnel, specialized skills, or computer resources to formally specify a complete system or to prove its behavior with respect to a particular system wide constraint on behavior. Thus, formal methods must be able to be used in a partial fashion. This includes the ability to specify parts of a system vertically (i.e. from abstract levels down to component levels for a particular capability) or horizontally (i.e. at an abstract level but across the entire system), with corresponding verification. Many methods do not lend themselves to partial use. Methods that can be used partially, as well as the tradeoffs gained or lost in using them in this fashion are also not well understood.

5.3 Scaling problems:

- Examples of potential scaling problems can be seen in the run-time configuration management capabilities needed by an ATM network monitoring system. Existing specification methods usually involve very little decomposition/refinement (as in the system response time example; refinement consists of specifying time bounds on individual components), whereas automatic operation of run-time configuration changes clearly needs multiple

levels of refinement. (E.g. what configuration changes can be accomplished on what areas of the system, and what functionality must be included in components to allow these configuration changes.) Thus, scaling the methodology of refinement can be an issue.

- Another scalability problem arises in verifying that the capability exists across the entire system. Many current methods use theorem proving or state validation to show that a system property is properly specified. Such methods are not feasible for most commercial systems. In addition to the sheer size of the problem (in terms of the number of components involved) there are far too many states to traverse in a complicated distributed system. The problem must be constrained or abstracted so that it can become tractable to use these methods of verification. (See [10, 65].)

6. CHARACTERIZATION OF THE DESIRED SOLUTION

The research described in this dissertation develops methods to identify, specify, validate, and verify software run-time configuration management requirements in complex distributed systems. An ATM/IP hybrid network monitoring system is used as an exploration tool in the research. This system is large and provides a good test case for the methods developed in the research. Methods are generalized and demonstrated to work on a system in another problem domain, that of bioinformatics.

The methods developed in this research are adequate to completely specify key software run-time configuration management requirements, to validate their existence in the specification, and to be generalized to use with systems in other domains. The outcomes of the research are:

- A Key Requirements (KR) method to identify the key run-time configuration management requirements of a loosely coupled distributed system.
- A method to specify key software run-time configuration management requirements, developed using the methods from academia as well as from industry. (These methods are property refinement, fuzzy sets, soft goals, UML/OCL, Alloy, and temporal logics; see detailed descriptions of property refinement, fuzzy sets, and soft goals in Chapter 2).
- Actual specifications of key software run-time configuration management requirements.
- A method to validate and verify a specification for key run-time configuration management specifications and internal consistency, developed from the methods of problem abstraction, composability reachability analysis, multiple view analysis, and automated tools (see detailed descriptions of problem abstraction, composability reachability analysis, and multiple view analysis in Chapter 2).
- An evaluation of the generalizability of the methods in other problem domains.

In general, the methods must be acceptable for use in industry. This means that they should not require extraordinary skills and tools. The methods must also fit in the existing development paradigm of an organization. Any method that is not able to conform to these conditions has a low likelihood of acceptance. It is not possible to assure that the specification, validation, and verification methods will be acceptable in every case due to differences among development organizations. However, the likelihood of acceptance of these methods is documented based on my own experience

at Hewlett Packard Company and Agilent Technologies, and attempts to use them in the research environment at Agilent Laboratories. These criteria are discussed in Chapter 3 as part of evaluation criteria used in this research.

The contributions of this research are:

- Development of a methodology (using heuristics) to identify key run-time configuration management issues of loosely coupled distributed systems that is general enough to be useful in different problem domains.
- A specification method expressive enough to define flexible and extensible software run-time configuration management requirements.
- Validation and verification methods that are complementary to the specification method.
- Methods generally acceptable for industrial software development. These methods are scalable to large systems. The methods do not require extraordinary skills or tools to use and are also flexible enough to fit into existing organizational software development paradigms.

6.1 Methods Developed by the research

The two methods described in this dissertation are the Key Requirements (KR) method and the Configuration Management Specification, Validation, and Verification (CMSpecV&V) method. These methods are described in the steps outlined below. The KR method is fully described in Chapter 4, the specification portion of the CMSpecV&V method is described in Chapter 5, and the associated validation and verification method is described in Chapter 7.

6.1.1 KR Method Steps

Figure 1.4 shows the information obtained from each step and how it used.

1. Identify primary domain requirements.
2. Identify implications of primary domain requirements and corresponding constraints on other domain requirements as well as additional new requirements.
3. Identify key user roles of the system. Identify key tasks for each of these roles.
4. Identify requirements of key user roles.
5. Synthesize and derive key run-time configuration management requirements from the primary domain requirements and their implications, and additional requirements imposed by user roles and tasks.
6. Identify potential items that need to be configured.
7. Associate potential items that need to be configured with the type of structuring capability needed, to create a preliminary configuration matrix.
8. Use the key run-time configuration management requirements to prune the preliminary configuration matrix to include only the few key items that need to be configured in the system.

6.1.2 Specification Portion of CMSpecV&V Method

Figure 1.5 shows the information created at each step and how it is used.

1. Identify the system focus and related run-time configuration management issues.

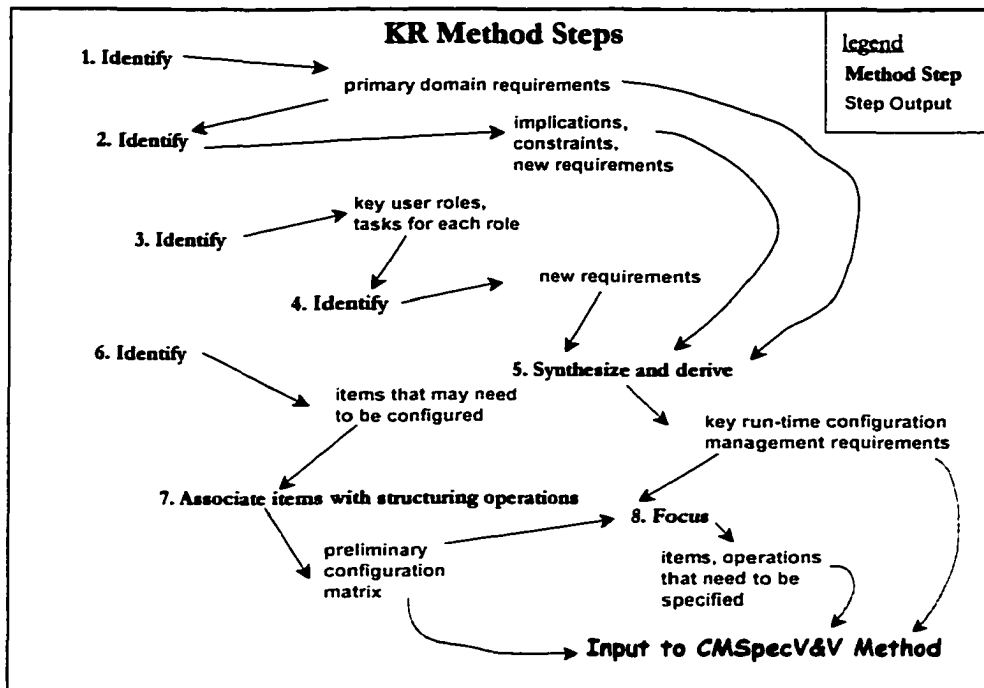


Figure 1.4 Information obtained from each step of the KR method and how it is used in subsequent steps. The output of this method is a small set of key run-time configuration management requirements, along with the items that need to be configured and their associated configuration operations.

2. Use property refinement to obtain more precise function-related statements that embody vague system-wide run-time configuration management behavioral constraints.
3. Use soft goals to identify and further refine run-time configuration management functionality and identify links (e.g. satisficing and argumentation goals) between requirements.
4. Create a *type* model without OCL constraints, using the information gained in previous steps. In conjunction with this step,
5. Create an Alloy model to complement the type model and extend it using information from previous steps. The Alloy model is a complete model of

the run-time configuration management specification of the system. The Alloy model includes constraints and operation behavior specification.

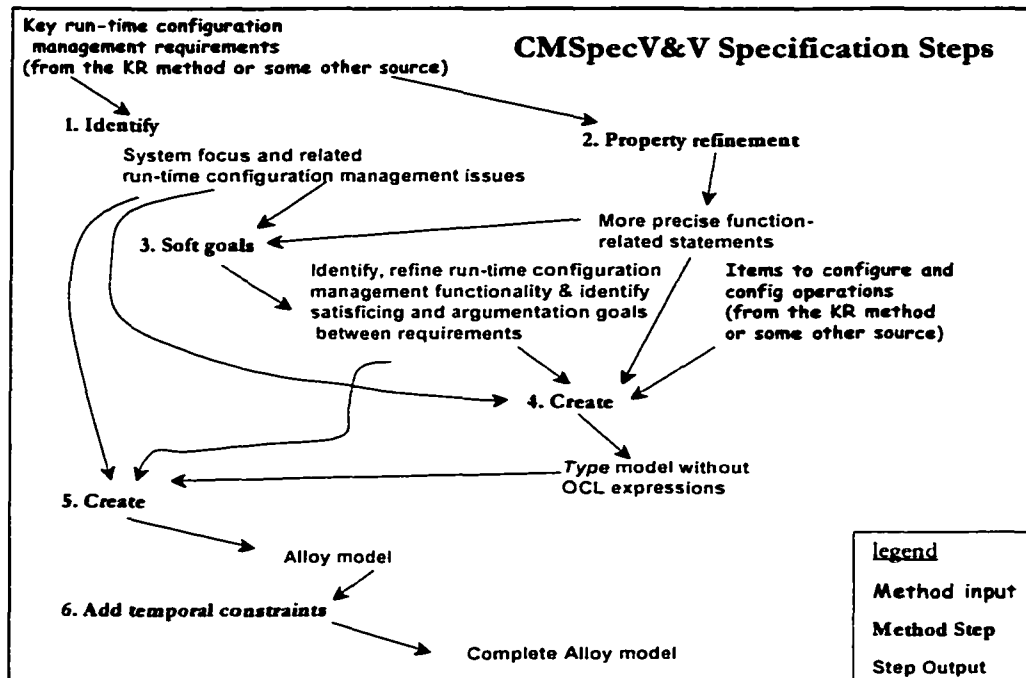


Figure 1.5 Information obtained at each step of the specification portion of the CMSpecV&V method and how that information is used. Input comes from the KR method or some other requirements elicitation method. Output of this method is a type model without OCL constraints and a complete Alloy model that includes temporal constraints. (Temporal constraints are included as comments in the Alloy model.)

6. Add temporal constraints to the Alloy model using RTL-style comments if they are needed.

6.1.3 Validation and Verification Portion of CMSpecV&V Method

Figure 1.6 shows the information created at each step and how it is used.

1. Use modified multiple view analysis to validate whether key requirements are reflected in the specification.
2. Use the Alloy tool to verify internal consistency of invariants, operations, and overall system state.

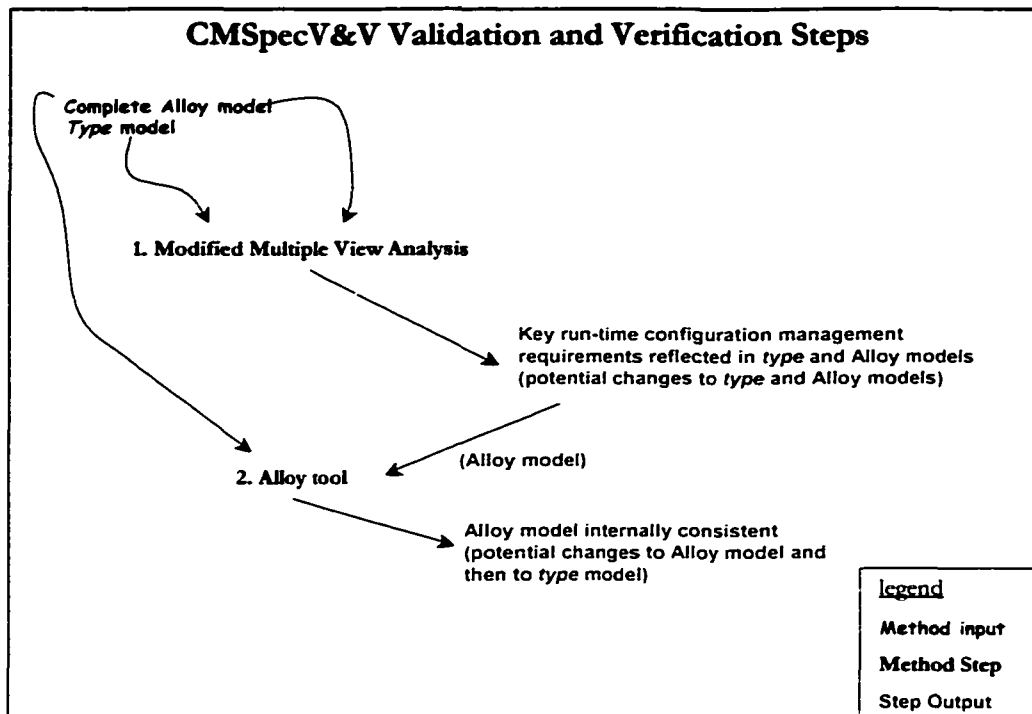


Figure 1.6 Information obtained at each step of the validation and verification portion of the CMSpecV&V method and how that information is used. Input comes from the specification portion of the CMSpecV&V method.

CHAPTER 2 - RELATED WORK

1. INTRODUCTION

This chapter outlines work related to the specification and validation of run-time configuration management requirements. This includes work on general configuration management, requirements elicitation techniques and several specification and validation techniques. Configuration management and run-time configuration management work is discussed in Section 2. Requirements elicitation work is presented in Section 3. Requirements elicitation is a topic especially related to the discussion of the key requirements identification method (KR method) in Chapter 4. (This method determines the key run-time configuration management requirements of a loosely coupled distributed system.) Section 4 presents a discussion of specification techniques (three of these techniques are evaluated by adapting them to the ATM network monitoring problem in Chapter 5; they are discussed in detail in Section 4 of this chapter). Section 5 presents a discussion of validation and verification techniques (three of these techniques are evaluated by adapting them to the ATM network monitoring problem in Chapter 7; they are discussed in detail in Section 5 of this chapter). Following these discussions, an overview of alternative specification methods is presented in Section 6 and a discussion of alternate validation and verification methods is presented in Section 7.

2. CONFIGURATION MANAGEMENT

Configuration management involves structuring systems during development as well as after they have been constructed and deployed. Configuration management is applicable to both the software and hardware in systems, although once the hardware in a system is nominally working, software tends to change more rapidly than the corresponding hardware, during both the development and post-deployment stages. Software is therefore most often the target of configuration management tools. Development-phase software configuration management is well understood and tools support automatic management of software modules during development (e.g. RCS, [85, 86]). Such tools support module identification, module version control, and development environment control. None of these established tools address the configuration needs of systems that are already deployed, and need to be dynamically changed while they are running.

2.1 Run-time Configuration Management

Run-time configuration management solutions tend to focus on managing networked computing systems. Thornton [84], developed a system to describe and manage the software available across a set of workstations in a network. This description is called a *configuration*. A *configuration* consists of a declarative description of the correct state using parameters and a relational database that contains instances of the parameters. (One example of a *configuration* is that all workstations should have mount points on their disks; the specific mount points for each machine are stored in the relational database. Another example is that all machines should have a particular software program on their disks, and the database stores which version is needed on

each type of machine.) Algorithms are then used to verify that the overall workstation network conforms to these *configurations*, and also to repair situations where an actual workstation does not conform to the description.

Coatta and Neufeld [14] propose a method to describe the correct state of a distributed system, and to bring the system into compliance with that description, using *composite objects*. *Composite objects* are created from other objects in the system. Descriptions of composite objects include their particular instances, general predicates defining their structure or behavior, constraints on their states, and methods to repair them if they diverge from the stated constraints. For example, a *composite object* describing a tree can be created, with a constraint that there are no cycles in the tree. The tool described by Coatta and Neufeld, RCMS, will monitor the tree's state once it has been created, and if the non-cyclic constraint is violated, then RCMS will invoke a repair mechanism (contained in the composite tree definition) to remove the cycle and restore the tree to a state compliant with its constraints.

Berghoff, et al. [7] uses agents in a generalized technique for run-time configuration management. Agents move around a distributed system and perform configuration changes. Multiple agents may attempt conflicting changes, so issues such as resource locking and security must be included in the configuration operations. PRISMA, a prior system of Berghoff et al., also allowed generalized dynamic configuration management, but employed centralized control. One global management component attempted to coordinate configuration changes across the entire distributed system. This centralized solution did not scale well, since it became an algorithmic as well as network bottleneck in a large distributed system.

Thomas et al. [82], Kramer and Magee [45], Hiltunen [30], and Oreizy and Taylor [64] propose other run-time configuration management schemes. Thomas et al. [82] uses a visualization of configuration based on warping¹⁰ to provide cues that show when a configuration is not in compliance with a defined good state. This allows a system to be checked against a pre-defined configuration that is known to be good. Changes can also be made to bring the system back into alignment with the known good state if it is out of compliance.

The other schemes pre-check a configuration change to make sure it will yield a correct system. Kramer and Magee [45] use labeled transition systems to model components. Labeled transition systems also model proposed configuration changes, which can then be checked for validity. Only changes that yield correct configurations are allowed to proceed. Similarly, Hiltunen [30] identifies module relationships, which in turn are used to define allowable re-configurations. Oreizy and Taylor [64] apply relationships between system components to verify proposed configuration changes. Unlike the other methods that operate on system components, their method operates at a higher level of abstraction using an architectural description.

All of the techniques described above are methods to accomplish run-time configuration management. They are not specification techniques, and therefore are not directly applicable to the subject of this research, the specification of software run-time configuration management requirements.

¹⁰ Warping visually animates a three-dimensional object as users grasp, move, scale, and rotate it. Thomas et al. create three-dimensional objects that represent physical aspects of system configuration (e.g. process performance). If the physical property changes as the system runs (e.g. process performance decreases due to increased load), the object representing that property is warped. The more warped an object becomes, the more extreme its behavior has become from the normal.

3. REQUIREMENTS ELICITATION

The work discussed above concentrates on methods to provide run-time configuration management. This work does not discuss the identification of key run-time configuration management properties actually crucial to the system being developed. A set of work related to this latter problem is that of requirements elicitation. Techniques used to elicit requirements from users of a system being developed usually produce several views of a system's behavior, including its states, control flow, and data flow. Some of the commonly used notations include finite state machines, state charts, Petri nets, and decision tables/trees.¹¹ These views and notations focus on a system's functional behavior. Additional requirements documents often include less precise textual descriptions of the "non-functional" aspects of the system (e.g. reliability or manageability). Unfortunately, these textual descriptions are not precise enough to drive system specification. In order to elicit more precise requirements regarding non-functional behavior such as run-time configuration management, requirements engineering approaches are needed that provide a more complete view of a system's requirements.

Martins [54] uses principles of Activity theory to view system actions along multiple dimensions: community, rules, subject, object, tools, division of labor, and results. An overall system model is then created for the major *activities* of the system. This approach leads to requirements that take into account some of the context in which a system will operate, thus yielding more complete requirements. Jarke [39] uses domain abstractions and objective-driven user requirements to create meta-models directed at

¹¹ See [17] for detailed discussion of the various notations and their products.

accurate requirements elicitation. The use of meta-models also allows problem domain models to be reused from application to application.

Richards and Menzies [71] use a knowledge engineering basis to build a set of requirements from multiple viewpoints. Although a viewpoint is defined as a style, area of concern, space work-plan, and work record, the project described deals only with viewpoints that represent experts providing requirements knowledge. Conflicting requirements provided by different experts are resolved to yield a consistent set of requirements.

Herlea [29] and Morris et al. [56] and Zave and Jackson [92] all describe the need for user involvement during the process of requirements elicitation as well as various methods to achieve useful user participation via meetings and communication. Pautler et al. [67] touches on the use of domain knowledge to place constraints on the requirements of a simulation system.

The techniques described above are all aimed at eliciting the requirements of a complete system. They are also generally aimed at the requirements from a user's point of view. The problem of software run-time configuration management is a partial view of the system, and one not necessarily deeply connected with its functionality. In addition, the requirements specification that is needed includes many requirements that are considered internal to the system; no user accesses them directly, but taken together they create the overall system behavior that the user requires.

4. SPECIFICATION METHODS

4.1 Run-time Configuration Specification and Design Techniques

Only a few papers specify the capabilities needed for run-time configuration management of distributed systems and their specification. Zimmermann and Drobniak [93] propose one comprehensive method. In this approach, a specification consists of:

- an interface specification, including interaction behavior (e.g. sequencing rules)
- a role scheme that defines the roles in the system (used to regulate the use of configuration operations)
- component specifications that include:
 1. interfaces of the component
 2. binding properties (which can be either explicit or implicit) such as binding to a particular machine or to one with specific memory availability
 3. configuration constraints, which are assumed to be true for all time in the system

Instantiation and termination rules are explicitly defined in the specification, and the technique includes the provision for other application-specific reconfiguration rules that may be needed. This technique in some cases goes beyond the needs of the run-time configuration management specification of the ATM network monitoring system and in some cases is inadequate. The method is inadequate in terms of the interface specifications; the run-time configuration management specification must be able to specify what configuration operations do, not just their interfaces. In addition, it does not appear that the method can handle the run-time configuration management of

physical entities such as network connections, network probes, computers, etc. The method potentially goes beyond the needs of the ATM monitoring system in the area of roles. It is not clear that roles are necessary in the use of configuration operations for this system. On the other hand, in the bioinformatics system, security is a greater concern, so roles may be appropriate in its run-time configuration management specification.

Lim [49] describes a design technique in which a set of finite state machines model the behavior of each process in a distributed system, and the Cartesian product of these machines is used to model the entire system. Constraints on states and transitions are given using first order predicate logic and machines are hierarchically composed to enhance scalability. The process of reconfiguration is then moving from one overall system state to another. A reconfiguration graph is constructed to model this transition, and it can be checked for legal paths from the current configuration to the proposed new one, as well as to check a legal path to ensure that it contains no invalid transitions among the more basic finite state machines that define the overall system. This method is a design technique, and not adaptable to specification requirements. In a large system with many potential non-deterministic state machines, it is also not clear that the method will scale.

4.2 Other Distributed Systems Specification and Design Techniques

Other work related to the specification and design of distributed systems deals with formal ways to specify the functional needs of systems, rather than system-wide constraints on behavior such as run-time configuration management. Little work has actually been done in the area of specification of system-level requirements other than

correctness [16], reliability and dependability [8, 19, 75, 83], performance [48, 57], and transaction services [33, 90]. Since these system-level properties behave quite differently from run-time configuration management, especially in their refinement and decomposition into requirements that can be specified at a component level, these specification methods cannot be generally used to specify run-time configuration management needs of a distributed system.

4.3 Potentially Adaptable Specification Techniques

Three specification methods have been chosen to study for their adaptability to the task of run-time configuration management requirements specification. These are methods that deal with system level “non-functional” properties, and methods that are described in enough detail by their primary authors that they can be used on problems other than those described in the literature. These specification methods are 1) property refinement (developed by Valerie Issarny et. al. [33, 75, 90]), 2) fuzzy sets (developed by Xiaoqing Liu and John Yen [50]), and 3) soft goals (developed by John Mylopoulos et. al. [58]). Each of these methods is discussed in more detail below, along with example specifications from the authors' work.

4.3.1 Property Refinement Methods

Zarras and Issarny [90], Saridakis and Issarny [75], and Issarny, Bidan, and Saridakis [33] use property refinement to obtain first order logic statements which act as requirements on system components. Two system-level properties are described in these papers, transaction services ([33, 90]) and dependability ([75]).

As an example, the overall behavior of a transaction service is refined into four properties: atomicity, consistency, isolation, and durability. These properties are further

refined until first order logic statements can be written for individual software components. Atomicity is described as the principle of all or nothing: *either the execution of the transaction results in a state transition to the final state (for the transaction; the state goes from σ to σ' on transaction t) or the system appears to have never left the initial state.* This can be captured by the formula shown in Figure 2-1.

$$\text{atomic}(t) \equiv ((\text{pre}(t) = \sigma) \wedge (\sigma \rightarrow, \sigma')) \Rightarrow ((\text{post}(t) = \sigma) \vee (\text{post}(t) = \sigma'))$$

Figure 2-1. Predicate statement of the atomic transaction property.

The atomicity formula can be refined until it becomes a set of actions that are specifications for system components.

An example of a specification, sufficiently refined to represent component design specifications, is *exactly once failure semantics*. In this protocol, a tuple can be imported only once in the absence of failures (i.e. there are no duplicates) and never otherwise. In this example, A_1 and A_2 are activities, and v is the value exchanged between them for coordination purposes. These semantics are shown in Figure 2-2 as the *ExactlyOnce* predicate.

$$\begin{aligned} \text{ExactlyOnce}(A_1, A_2, v) \equiv & \text{export}(A_1, A_2, v) \wedge ((\neg \text{failure}(A_1, A_2, v) \Rightarrow \\ & \text{import}(A_1, A_2, v)) \wedge (\text{failure}(A_1, A_2, v) \Rightarrow \neg \text{import}^*(A_1, A_2, v))) \end{aligned}$$

Figure 2-2. Predicate statement of the exactly once failure property. In this predicate, *export* (A_1, A_2, v) means that v is sent to A_2 from A_1 , *import* (A_1, A_2, v) means that A_2 gets v only once, *import** (A_1, A_2, v) means that A_2 gets v at least once, and *failure* (A_1, A_2, v) means that a failure has occurred during the coordination between A_1 and A_2 .

The non-functional property of dependability can be precisely defined using property refinement. Dependability is refined to abstract properties, or system states, of

safety, availability, reliability, detection, and failure masking. Concrete properties are then defined and specified as system actions. For example, a system action to make objects dependable can occur by using replication to increase the likelihood of an object being in an available state. The system action to accomplish this replication can be based on active, semi-active, or passive actions. One predicate for passive replication is shown in Figure 2-3 below.

$$\text{Passive}(\alpha) \equiv \exists \gamma, \beta : \text{Replication}(\{\alpha, \beta\}) \wedge \text{StableStorage}(\alpha, \gamma) \wedge \text{Restore}(\alpha, \beta, \gamma)$$

Figure 2-3. Predicate statement for passive replication. This statement states that an object α is passively replicated if there exists a stable storage γ and another object β such that β is a copy of α and α is stored in γ and part of retrieving α from γ includes updating β to maintain the fact that it is a copy of α .

Refining dependability properties is fairly subjective according to the authors' experience. The guiding rule is that one property refines another if its logic implies the logic of the original property. Thus, refinement yields stronger expressions on the "left-hand side" of a logic statement, like the format of a grammar. The result of this entire process is that abstract properties are transformed into concrete constraints that are used as software specifications.

4.3.2 Fuzzy sets

Another specification approach is the use of fuzzy sets, described by Liu and Yen [50]. Liu and Yen approach system-wide behaviors (which they call imprecise requirements) using fuzzy subsets that are analyzed across related requirements. First, they state requirements in terms of satisfaction functions using fuzzy subsets. For example, consider a development process p to create a conference room scheduling

system, and an input i . Some possible requirements can be formed using fuzzy subsets SMALL, LOW, and HIGH, as shown in Figure 2-4.

R₁: DevCost (p) should be LOW

R₂: Resource (p) should be SMALL

R₃: AVG_i (Utilization (Schedule (i, Product (p)))) should be HIGH

Figure 2-4. Requirements statements using fuzzy subsets. The *should be* operator is vague, but it can be interpreted as *minimize* if the associated fuzzy set is LOW, or SMALL, and *maximize* if the fuzzy set is HIGH.

Once requirements are stated, relationships between them are defined. Relationships between requirements can be *irrelevant*, *conflicting*, *cooperative*, *mutually exclusive*, *completely conflicting*, or *completely cooperative*. Defining these relationships allows analysis to proceed across the requirements. This analysis is performed by mapping an attribute value onto a satisfaction degree, using a linear function. The amount of tradeoff a customer is willing to give up is also taken into account. This leads to the ability to determine somewhat quantitative relationships between requirements. An example of the relationship graph for requirements R₁, R₂, and R₃ of the room scheduling system is shown in Figure 2-5.

Aggregation can be used to combine requirements. For example, a fuzzy conjunction can be described as the minimum function (MIN), fuzzy disjunction as the maximum function (MAX), and fuzzy compromise as an average function (AVERAGE). Priorities across requirements are used to make tradeoffs. The examples in the authors' paper are relatively small, so whether this method is reasonable for describing and analyzing complicated and large systems remains an open problem.

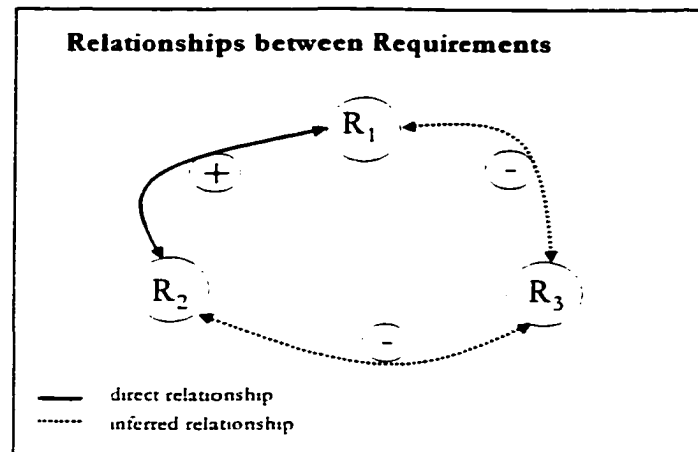


Figure 2-5. Direct and inferred relationships between requirements R_1 , R_2 , and R_3 . In this diagram, (simplified from [50]) requirements R_1 and R_2 have a direct cooperative relationship. Relationships can also be inferred between requirements that do not have direct relationships; these are shown in the diagram as dotted lines. Thus, although there are no direct relationships between R_3 and the other requirements, relationships can be inferred between them (these are based on the relationships between other requirements which are not shown in this simple diagram). In this case both R_1 and R_2 have conflicting relationships with R_3 .

4.3.3 Soft Goals

Mylopoulos, Chung, and Nixon [58, 59] describe a method similar to fuzzy sets, where non-functional goals are described and relationships between them are defined and analyzed to create an optimal system design. The method proceeds by refining non-functional goals into more precise goals with additional relationships between them. A graph is constructed using the refinement process and nodes are labeled once they can be no longer refined using an algorithm that helps find an optimal design. Analysis is used to decide which paths to follow in a design in order to reach leaf design decision nodes that correspond to a system that meets the requirements. Goals are divided into three types:

- non-functional requirement goals (e.g. dependability, accuracy, security, development cost, performance, etc.),

- satisficing goals (goals that help satisfy the non-functional requirement goals, e.g. preventative, curative, and precautionary goals for satisfying an accuracy goal), and
- argumentation goals (goals that provide evidence that another goal or refinement has been met or cannot be met)

Each goal in a system has a *sort* or *type*, as well as *parameters* or *targets*.

Examples of goals, types and parameters are shown in Table 2-1.

Table 2-1. Examples of soft goal statements. Goal statements do not explicitly state the relationships between non-functional goals, related satisficing goals, and argumentation goals. The goal refinement tree that is eventually created shows the relationships between each kind of goal.

goal statement	type/sort	parameters/ targets	discussion
<i>accuracy</i> non-functional goal			
Accuracy [attributes (Employee)]	<i>accuracy</i>	all attributes of the data type <i>Employee</i>	accuracy is needed for all employee-related attributes
<i>validation</i> satisficing goal			
Validation [attributes (Employee)]	<i>validation</i>	all attributes of the data type <i>Employee</i>	validating all employee-related attributes will enhance accuracy
<i>argumentation</i> goal			
FormalClaim [$\exists e: \text{ValidatedBy } [e, \text{attr}(\text{Employee})] \wedge \text{EmployeeStatus}(e, \text{SecretaryClassI})$]	<i>formal claim</i>	all attributes of the data type <i>Employee</i>	validating employee-related attributes by a Class I Secretary is a piece of evidence that validation will occur correctly

Relationships between goals and their refinements are:

- conjunction,
- disjunction,
- necessary but not sufficient,
- partially conflicting,
- equal, and
- unknown

Continued refinement of goals with unknown relationships is required to determine the correct relationship. Once goals are refined enough to have a known relationship, that relationship can be promoted back up the tree. This is accomplished by traversing the tree upward and applying labeling rules to the node relationships.

Methods used to refine goals include:

- decomposition (usually using conjunction of refinements),
- creating a set of satisficing goals that may have different link types to the parent goal, and
- providing argumentation goals for evidence

A goal refinement tree is constructed by refining all non-functional requirements as much as possible, adding the link types (dependency, satisficing, or argumentative) between them, and identifying the relationships between them. An example of such a tree is shown in Figure 2-6.

The goal refinement tree can be traversed using a labeling algorithm to determine the degree to which each non-functional requirement is addressed by a particular set of refinements/design decisions. Labeling occurs from the bottom up, combining refinements upward to decide the degree to which the original goal is satisfied.

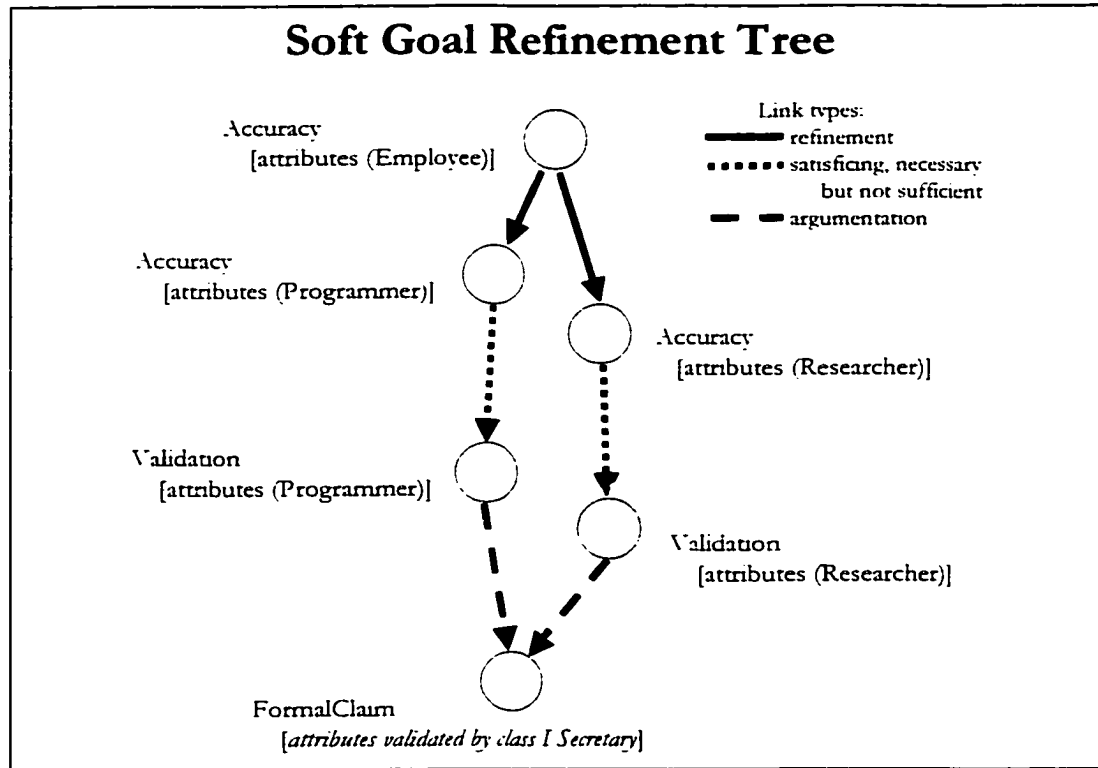


Figure 2-6. Example of a goal refinement tree showing different kinds of goals and their links/relationships.

5. VALIDATION AND VERIFICATION METHODS

Validation demonstrates that the key run-time configuration management properties are present in the specification, while verification demonstrates that the specification is internally consistent. (In addition, it is necessary to demonstrate that the final system exhibits the required run-time configuration management properties. However, this type of verification is not a part of the research described in this document, and is not discussed further.) Unfortunately most academic research on

validation and verification are highly theoretical, deal with complete formal specifications, or have been used on very small systems. There are some methods that might be useful on larger, industrial systems. Three verification/validation methods chosen for further study are: 1) problem abstraction (described by Carver [8]), 2) component reachability analysis (described by Cheung and Kramer [10, 11]), and 3) multiple view abstraction (described by Issarny, Saridakis, and Zarras [33]).

5.1 Problem Abstraction

Carver [8], describes a validation technique that consists of creating abstract sequencing constraints that are refined into observable events with the constraints of *always*, *possibly*, and *never* attached to them. These concrete event sequences guide the selection of test sequences. Carver claims that, since the constraints are abstract, this technique can be used for validation. It is not clear whether or not this method can be used in the validation of run-time configuration management properties, since the complexity and number of observable events may not lend themselves to a small set of test sequences. In addition, since some run-time configuration management capabilities may hinge on primarily internal functions, whether these can be tested using externally visible events remains an open question.

5.2 Composability Reachability Analysis

Cheung and Kramer [10, 11] describe a method of system verification using compositional reachability analysis. They create a model of the system that includes all states and all transitions where the same actions label similar activities. This allows only “relevant” actions to be used and all other properties to be abstracted out of the system. Abstraction can be used to reduce the number of states that must be considered during

analysis. Two commonly used techniques are partial ordering (to avoid multiple combinations of paths that use the same states) and intermediate simplification of subsystems. The second technique is known as composability reachability analysis (CRA). Only globally visible actions can ultimately be verified with previous CRA methods. The analysis described by Cheung and Kramer starts at the lowest level of the system and reasons about execution traces at this level. The behavior is then abstracted to higher levels of the system such that if the lower level trace forces the system into an undefined state, the derived abstraction will also force the system into the undefined state. The analysis is based on the property being analyzed. By deriving an abstracted image automaton and composing it with the processes to which it applies, the result is a globally observable error condition if such a condition can exist in the lower levels of the system.

The composability reachability analysis method is applied in the context of synchronized communicating processes and a suggested extension will include behavior constraints imposed by neighboring processes to further minimize the number of states and complexity of the system that must be analyzed. The authors describe the property of safety in their examples, using an undefined state, π , as the unacceptable state to which safety violations lead the system. If this method is used to analyze a different system level property, such as a CM capability, the undefined or unacceptable state(s) may be different than the error state π discussed by Cheung and Kramer.

5.3 Multiple View Analysis

Issarny, Saridakis, and Zarras [33] promote the idea of different views of an architecture that can be checked across one another for consistency and correctness.

The authors describe four views of a system: functional (e.g. the operations available in the system), interaction (the interaction protocols across components), dependability (the structure and synchronization protocols with respect to fault tolerant mechanisms), and efficiency (system structures that increase efficiency). These views are aggregated to obtain a view of the entire system in two ways; the view of architectural structure (a straightforward consistency check of the structural components across all views), and the view of architectural consistency that is defined as the composition of the views that lead to correct behavior.

This composition includes ensuring that the synchronization of the functional operation is consistent with the synchronization of the underlying interaction protocol, the synchronization in the dependability view is consistent with the synchronization of the interaction protocols, the dependability properties are consistent across the types of fault tolerant mechanisms used, and the dependability properties are consistent with the functionality operations. It appears that the efficiency and dependability views are the views that need to be defined differently for each system-level behavior property that will be analyzed. A key problem is deciding which views need to be defined for configuration management of a distributed system and how to check their types and protocols for consistency.

6. ADDITIONAL SPECIFICATION METHODS

Specifying "non-functional" system requirements of system-level properties is very difficult using the traditional techniques most often seen in industrial software development. (See [17] for a survey of such techniques.) There are a few exceptions, such as system response time defined in terms of timing constraints, which have been

adequately specified using techniques such as finite state machines, state charts, and Petri nets. The resulting specifications from each of these techniques can be checked for overall consistency and adherence to the original requirements. However, this ability is very dependent on the size of the specification; large distributed systems usually experience state explosion as they advance in their execution, making state checking verification techniques infeasible.

Techniques such as decision trees and decision tables require that each high level system property be refined until it can be transformed into functionality required by an object or component. At this point the decision techniques can be used. However, verification of the high level property in the complete system specification must take the form of ensuring that the tables are properly filled out, or the trees properly constructed. This can easily take on the form of a mostly manual effort, with a great deal of human knowledge and guesswork involved in the process.

6.1 Other Specification Methods

There are many formal methods and notations to specify system functionality. In general these methods all work quite well for operational behavior, but are inadequate for system level "non-functional" properties. One interesting possible exception is TRIO [12] that specifies supervisory and control systems. TRIO can specify some system level non-functional properties, particularly fault tolerance and static configurability. In the case of static configurability, refinement of a the system level property generates specification predicates. These predicates are part of a class related to the system level property of static configurability. TRIO allows all of the predicates that logically implement a system level property to be viewed as a single group. Functional

object classes can then be checked to see if relevant aspects of the system level property class exist in them. This technique can only be used if the system level property has been refined to the level where it can be transformed in predicates.

Architecture description languages are another technique that can often be used to specify complicated systems. I have attempted to use various languages to specify system level properties of distributed embedded measurement systems with varying degrees of success. Darwin [52] and ASDL [78] fall short of being able to adequately specify non-functional system level properties [25].

Darwin is a language developed by Magee, et. al. [52] as a configuration language to describe system structure orthogonally to its functionality. Radestock and Eisenbach [70] also developed a π -calculus that can be used in conjunction with Darwin to add semantics to the structural interfaces specified in Darwin. A Darwin specification clearly shows the communication and structure of a system, but is not capable of showing non-functional system level properties at either the overall system level or at a component level.

ASDL (Architectural Style Description Language) is a formalism to model both architectures and architectural styles, developed by Seidman and Rice [78]. Each element of an ASDL description (templates, settings, and units) is associated with a Z schema that specifies both structural features as well as the semantics of interfaces between elements. Template component schemas do not contain functional information, unless appropriate Z predicates are added. ASDL can specify component interface semantics of a system but cannot specify non-functional system level properties at the overall system level.

SDL (Specification and Description Language) [22, 80] is a standard language for specification and description of reactive systems. It was originally developed for telecommunications applications and has both high usage and good tool support in this domain. It has a formal semantic basis as well as object oriented features and the ability to execute specifications (which aids in the verification process). SDL can be used to describe systems composed of executing processes that respond to signals. Thus, the structure of the system is described by the processes in it, and the dynamic behavior is described by the way it responds to signals. SDL is capable of specifying some non-functional properties, such as fault tolerance and availability, but these cannot be specified easily at the system level. Multiple process diagrams must be pieced together in order to see the overall system behavior.

UML (Unified Modeling Language) [21, 63] is also a specification notation. UML includes different kinds of diagrams that can specify different aspects of a system. Class diagrams show the structure of the system and its static relationships. Collaboration diagrams show the dynamic relationships that exist in the system. Each collaboration diagram shows the interaction of classes of objects for a single scenario (e.g. adding a new physical instrument to the instrument assets in a measurement system). A dynamic system that changes frequently requires many collaboration diagrams that must be viewed together to interpret the actual behavior of the system. A combination of class and collaboration diagrams can specify some system level properties. Instrument resource utilization management in a distributed embedded measurement system is one example. However, as noted above, the behavior at a system level cannot be specified in a single diagram; multiple class and collaboration diagrams are needed.

Specification is an active research topic, and new techniques are constantly being introduced. Some newer methods include those in the following list. These methods are not appropriate for this research since they either were developed with a specific system level property in mind (such as reliability), and are not readily generalizable to other system level properties like the run-time configuration management properties needed by a system, or they are not described in enough detail to allow their use in the inductive approach of this research.

- Decomposition using risk analysis in both a horizontal and vertical fashion [55]. This method aims to specify dependability and does not seem to be generalizable to other system-level non-functional system properties.
- Dynamic descriptions of system level properties. These specify performance design, statistical guarantees [42], and attribute contracts [73]. Dynamic descriptions may be generalizable to other non-functional properties, but the technique is not described in enough detail to use on the ATM monitoring system.
- Policy specifications of an execution environment. This method was developed in the context of actor systems [3, 44] and does not seem to be adaptable to run-time configuration management requirements.
- Levels of abstraction [68] using causality relations and information models [42]. These methods are not described in enough detail to use with the CM capabilities of the ATM monitoring system.

7. ADDITIONAL VALIDATION/VERIFICATION METHODS

Alternative verification techniques described in the literature fall into the four basic categories shown below. These techniques were not chosen for use in this research because they are generally too dependent on formalisms or do not lend themselves to scaling up to very large systems.

- Theorem verification is described in [83] using LOTOS to specify a system and a theorem prover called LOLA to verify it. A second system described by the same author [83] uses the LARCH theorem prover for verification. Both descriptions only address small portions of an example system. Temporal logic is used in one specification to help address the dynamic, distributed aspects of the system.
- Construction techniques include following control flows and states using formal languages [3] and preservation of causality relations [68].
- Modeling techniques include composition [51], and checking states [10]. In general, checking states is difficult due to state explosion.
- Finally, testing includes using measurements to determine if desired system behaviors are present [55]. This technique requires a complete system implementation.

1. RESEARCH METHODS USED

The scientific method consists of three iterative steps. These are: 1) propose a model based on existing experimental observation, 2) check predictions made by the model against further observations, and 3) adjust the model as required. This method can be viewed as formalized learning from experience [87].

The scientific method is used in this research to develop a model that solves the problem of creating, validating, and verifying run-time configuration management specifications. The model consists of a methodology to develop, validate, and verify internal consistency of these specifications and a notation to communicate them. The model is created using experiential observations of the development of distributed systems. These observations were made over several years and projects at Hewlett Packard Company and Agilent Technologies. This model is called the Configuration Management Specification Validation and Verification (CMSpecV&V) model.¹²

The scientific method has been studied and formalized by Kuhn, Popper, Lakator, Laudén, and Sarkar, among others [23, 43, 46, 76, 77, 87]. Sarkar [76] defines a research method as consisting of: 1) an objective component and 2) a normative component. The objective portion of a research method also consists of two parts. The first relates to logical problems addressed by the research method, such as the

¹² The research method described in this chapter was also used to develop the Key Requirements (KR) method that is introduced in Chapter 4.

likelihood that a model validated by the method really is validated. The second part of the objective component defines key terms. The normative portion of a research method consists of two parts as well. The first is the conditions of acceptance of a model, and when a model should be pursued or adopted. The other part of the normative component defines rational decisions made using the research method.

The research method used in this research can be defined using Sarkar's objective and normative components. First, the research method is described.

The research method described in this dissertation creates a model using a specific case and then generalizes that model. An hypothesis is made about the general model and tested. The scientific method is used at two levels in this research. The first use of the scientific method creates an overall model (the CMSpecV&V model) to solve the problem of creating, validating, and verifying run-time configuration management specifications. Multiple competing models to create specifications, validate, and verify them are investigated. These models are accepted or refuted (perhaps partially) based on their ability to solve a particular instance of the problem they address. This analysis is performed in the context of an initial distributed system. The most promising aspects of these models are integrated into the overall CMSpecV&V model based on the results of this analysis. The hypothesis of the research is that the CMSpecV&V model is adequate to create, validate, and verify run-time configuration management specifications of any distributed system. This hypothesis is tested in the second use of the scientific method. The CMSpecV&V methodology and notation are applied to a different distributed system (one in a completely different problem domain, and one with different characteristics in terms of its focus).

The logical portion of the objective component of this research method has three aspects. These aspects are discussed under the assumption that a model has been validated by the research method. These logical aspects are: 1) whether the model is adequate for the original distributed system, 2) whether the model is useful for all distributed systems, and 3) whether the specifications developed from the model lead to software systems that meet their run-time configuration management requirements.

A well understood, existing distributed system is used to develop the model. Model results can then be checked against known results in the existing system. The first logical aspect of the objective component is addressed if the known results are consistent with the model results.

The second logical aspect is addressed by choosing a distributed system for hypothesis testing that is quite different from the distributed system originally used to develop the model. It is more likely that a model is useful across a wide range of distributed systems if it works for systems that are different in several aspects.

The initial system used in this research is an existing network monitoring system that is large in terms of the number of its physical computers, monitoring probes, and network connections. Portions of the overall system may be inaccessible at any time due to the large number of physical network connections. Physical configuration is therefore a primary focus of this system. The system is also large in terms of the amount of data that must be handled. The data is extremely dynamic (changing over sub-seconds) and has a short time-value.

The second system is a bioinformatics system that is small in terms of the number of its physical computers and network connections. This system deals with very

large amounts of data, and while it is dynamic data, the data changes over days, weeks, or months. The data in the second system is more geographically distributed than in the network monitoring system, and it has a long time-value. The focus of the bioinformatics system is on data.¹³

The two systems chosen for this research are therefore similar in that they are both distributed geographically (part of the definition of a distributed system) and they both deal with large amounts of data. They differ in problem domain (network monitoring versus bioinformatics), size (hundreds of computers versus a handful), system focus (physical configuration versus data), in the time-value of data (short versus long), and in the time associated with dynamic data (sub-seconds versus months). If the CMSpecV&V model works for both of these systems, it is likely that the model will work for other distributed systems that have characteristics between these two systems. (A possible exception is a distributed system that deals with a small amount of data, although in software engineering it is more often the case that dealing with small scale is easier than dealing with large scale. [20])

The third logic aspect of the objective component is whether a specification developed using the CMSpecV&V model can be used to create a system that meets its run-time configuration management requirements. Further testing must be undertaken to address this aspect. This testing is not a part of the research described in this document, but is a topic for future research.¹⁴

The second portion of the objective component is a definition of key terms used in the research method. These key terms are: generalizing (reasoning from the specific

¹³ See Chapter 1 of this dissertation for complete descriptions of the two systems.

¹⁴ See Chapter 12 for a discussion of future research.

to the general to create a model), the scientific method (three steps of proposing a model, making predictions from the model, testing these predictions and adjusting the model as needed), technique evaluation (comparing the results of a technique against a set of criteria), technique selection (choosing techniques to evaluate based on a set of criteria), specification validation (adequate representation of a requirement in a specification), and specification verification (internal specification consistency).

The first part of the normative component is the conditions of acceptance, pursuit, or adoption of a model developed using the research method. Conditions for pursuing a model (or technique) are based on its ability to meet or partially meet the technique selection criteria. Conditions for accepting/adopting a model are 1) that it meets all the technique evaluation criteria, 2) it is able to work for a system that is different in multiple dimensions from the original system, and 3) that the two methodology parts of the model (creating a specification and validating/verifying it) are complementary.

The second part of the normative component is the definition of rational decisions made regarding models developed using the method. There are three kinds of decisions that can be made about models. One decision is whether to pursue a model (or technique) for integration into the CMSpecV&V model. Rational decisions are based on which techniques come closest to meeting the technique selection criteria (portions of techniques that meet portions of the selection criteria may also be pursued). A second decision is whether to further pursue models developed by this research method. A rational decision to further pursue a model is based upon its ability to work for a second, different distributed system. A caveat to this decision is that the initial and second systems are different enough to increase the likelihood that the model will work for

other distributed systems. A third decision is whether to adopt models developed by this research method. A rational decision to adopt a model must be based upon further testing with distributed systems that differ in multiple dimensions from the systems used to create and validate the model. A rational decision for adoption must also be based on the model's ability to create a specification that leads to a system that meets its run-time configuration management requirements.

2. SPECIFIC USE OF THE RESEARCH METHOD

A CMSpecV&V model that works for the initial distributed system must be created before a general CMSpecV&V model that works for other distributed systems can be created. Techniques must be chosen for analysis and possible inclusion in the CMSpecV&V model. The criteria for selecting these techniques are influenced by two evaluation criteria. The first evaluation criterion is that the CMSpecV&V model be able to create, validate, and verify all run-time configuration management requirements of the initial distributed system. This evaluation criterion leads to a technique selection criterion that the technique be able to create, validate, or verify a specification related to non-functional system requirements.¹⁵ The second evaluation criterion is that the CMSpecV&V model is acceptable to industrial users. This is most likely if the new model uses or complements existing techniques and notations currently in use in industry.

Several research and industrial techniques are discussed in this research. Selection criteria for these techniques are developed in light of evaluation criteria. The *selection* criteria are listed below. Selected techniques are analyzed and evaluated

¹⁵ See Chapter 1 for a discussion of non-functional system requirements and run-time configuration management.

according to the *evaluation* criteria listed below. In some cases portions of the techniques that best meet the evaluation criteria are combined and extended to create the CMSpecV&V model developed in this research.

Occasionally no techniques are found that meet the selection criteria. This is the case for key run-time configuration management requirements identification. In this case ideas from existing requirements elicitation methods are extended to develop the KR methodology presented in Chapter 4. The only technique evaluation criteria that can be used on this methodology are informal validation and the ability to fit into industry. There is no formal way to verify that this methodology does find key run-time configuration management requirements of any distributed system. However, the first distributed system (used to develop the methodology) is an existing, commercially available system. Interviews with its architects provide an informal, but reliable check of the methodology results. The system and its predecessor system have been available for several years, so experience and engineering maintenance information is available (through the system architects) regarding key system requirements. This is not the case for the second distributed system, which is a new research system. Informal information is again available, however, in the form of user interviews and studies conducted internally at Agilent Laboratories.

2.1 Technique selection criteria

Technique selection criteria are shown in two sections. The first lists criteria for specification techniques and the second lists criteria for validation and verification techniques. Selection criteria used to develop the KR method are listed separately as idea-generating criteria.

The criteria to pursue analysis of a *specification* technique are: 1) the technique is able to specify system level “non-functional” properties, 2) techniques are described in enough detail by their primary authors to be used on problems other than those described in the literature, and 3) techniques (or notations) are currently in use in industry. The first two of these criteria are given greater weight since experience in Hewlett Packard and Agilent Technologies has shown that existing specification methodologies and notations are not sufficient to adequately specify the run-time configuration management requirements of distributed systems.¹⁶

The criteria to pursue analysis of a *validation/verification* technique are: 1) the technique is capable of showing that key run-time configuration management requirements are present in the specification (validating the specification), or that the specification is internally consistent (verifying the specification), 2) the technique does not rely on a complete, formal specification, 3) the technique appears to be adaptable to large systems, and 4) the technique is described in enough detail that it can be used on problems other than those described in the literature. (These criteria do not address the need to demonstrate that a system created from the specification exhibits the required run-time configuration management properties. This is a separate research topic, and is discussed as such in Chapter 12 of this document.)

No techniques are found that specifically address the problem of identifying key run-time configuration management requirements. The selection criteria are therefore changed into idea-generating criteria.

¹⁶ See Chapter 1 for a discussion of specification techniques used in industry.

2.2 Idea generating criteria

Run-time configuration management requirements may be key if they are: 1) discussed in existing design documentation of the network monitoring system, 2) present in academic research literature, or 3) come from personal industrial experience with other distributed systems in Hewlett Packard Company and Agilent Technologies.

Ideas used to create the KR method come from: 1) requirements elicitation techniques described in the literature and 2) personal industrial experience.

2.3 Technique evaluation criteria

Techniques are evaluated according to the following criteria: 1) reliable but potentially informal validation of results with system architects or system user feedback, 2) the ability to deal with all requirements of run-time configuration management of the distributed systems used in this research, 3) the applicability of the technique to specification versus design or implementation, 4) the ability of specification techniques to directly support validation and verification, and 5) the ability of the technique to fit into industry.

The fifth criterion is further comprised of the following: 1) automation opportunities, 2) existing tool support, 3) skill set needed by practitioners, 4) flexibility to fit into existing development practices, and 5) the ability to use the technique in a partial or incremental fashion. The criteria related to fitting into industry are based on personal experience of tool and process changes in software-only and measurement systems development groups in Hewlett Packard Company and Agilent Technologies. (The use of these criteria is consistent with viewing the scientific method as one of learning from experience. Experience has shown that methodologies and notations that do not meet

these criteria are usually not accepted in Hewlett Packard Company or Agilent Technologies. The criteria therefore are useful in deciding which techniques have a greater likelihood of being accepted in these companies, and perhaps in others in industry.)

Results of applying the evaluation criteria are discussed in Chapters 4, 5, and 7 of this dissertation, as various techniques are investigated and analyzed in the context of the network monitoring system. Results of the evaluation criteria are also discussed in Chapters 9 and 10 as the overall CMSpecV&V model is applied in the context of the bioinformatics system. These results are used to make rational decisions regarding the overall CMSpecV&V model and are the basis of conclusions to the research as well as future research topics.

CHAPTER 4 - IDENTIFYING KEY RUN-TIME CONFIGURATION MANAGEMENT REQUIREMENTS IN DISTRIBUTED SYSTEMS

1. INTRODUCTION

A preliminary critical problem of specifying run-time configuration management is deciding what properties actually need to be specified. In the most general sense, configuration management is concerned with the structuring of the system; locating, migrating, registering, deploying, binding, activating, and de-activating elements of the system. More precisely, run-time configuration management is the act of using these structuring capabilities to move a system in some particular *state* or *configuration* to another.

The difficulty comes in deciding which structuring needs exist, and for what elements of the system. Correct implementations of the appropriate structuring capabilities for the proper elements is crucial to the overall success of the system, since they directly impact not only the flexibility and extensibility of the system for end users, but also the cost and time to complete future system upgrades.

This chapter demonstrates a method to identify the key run-time configuration management requirements that must be implemented in order for a complex distributed system to meet the expectations of its users. The method also identifies the system elements that need to be configured and the configuration operations needed to

configure these elements.¹⁷ This Key Requirements, or KR, method identifies the key run-time configuration management needs of a distributed system with various kinds of data sources, including measurement data. (See Figure 1.4 in Chapter 1 for an overall view of the KR method.) Once the key configuration management requirements, system elements that need to be configured, and element structuring operations are identified, the techniques described in Chapter 5 can document the run-time configuration management specification. The KR method is thus a first step in the detailed requirements elicitation and specification process of a system's run-time configuration management needs [24].

The KR method was originally developed using the ATM/IP monitoring system as a study system. The method was then generalized and applied to the bioinformatics system. This step demonstrates the ability of the KR method to generalize to systems in other problem domains.

The KR method identified one key run-time configuration management need that was not supported by the initial ATM/IP monitoring system, that of scalable, automatic configuration. The KR method identified a set of run-time configuration management requirements in the bioinformatics system that matched those articulated by biological scientific researchers for incremental analysis and the ability to deal with changed data.

The chapter is structured as follows. Section 2 presents a two-dimensional view of the potential structuring capabilities and the items in a distributed system that can be structured. This *configuration matrix* contains the structuring capabilities and items

¹⁷ The KR method identifies key items that need to be configured and thus need to be included in a run-time configuration management specification. The remainder of the research described in this dissertation focuses on specifying run-time configuration management of software items only.

discussed in Chapter 1 of this dissertation. Sections 3 and 4 present the application of the KR method to the ATM/IP monitoring system and the bioinformatics system. Section 5 compares the identified configuration management needs of the two systems, and Section 6 presents conclusions about the method, its application, and areas where future work is needed.

2. CONFIGURABLE ELEMENTS AND STRUCTURING CAPABILITIES

Run-time configuration management includes the ability to structure a system. Chapter 1 presents a discussion of the structuring capabilities and potential items that need to be structured in a distributed system. Note that these lists contain items and structuring capabilities that are applicable for computer-only distributed systems as well as for distributed system that include multiple data sources.

In many distributed systems, there are an enormous number of items that can be configured, but only a critical few can be actually configured in any given system. To focus on key configuration management needs, problem domain and user role knowledge must be mapped onto the set of configuration management possibilities. Configuration management possibilities can be mapped into a two-dimensional space consisting of the possible structuring capabilities of the system crossed with the items that could be configured. The result is a densely populated two-dimensional *configuration matrix*. Empty values in the matrix result from items with no corresponding structuring capability that makes sense.

Table 4-1 below shows all the *items* and structuring possibilities discussed in Chapter 1. Structuring capabilities that do not make sense for a particular *item* correspond to an empty element of the matrix. These elements are highlighted in Table

4-1. For example, there is no entry in the *configuration matrix* for the deployment of a communication path since this does not make sense. Nor does it make sense to migrate a communication path, so this element is also missing from the matrix.

Table 4-1. Two-dimensional *configuration matrix* showing all possible combinations of *items* that could be configured and the structuring capabilities that make sense for each *item*. Empty elements (highlighted) of the matrix indicate instances of a structuring capability that does not make sense for the *item*.

Structuring Possibilities → Items ↓	Register	Locate	Migrate	Deploy	Activate, deactivate	Bind	Instantiate	Define
Communication Paths	X	X			X	X		X
Code	X	X		X	X	X	X	X
Agents	X	X	X	X	X	X	X	X
Objects	X	X		X	X	X	X	X
Data	X	X			X	X		X
Computing Resources	X	X			X	X		
Functionality	X	X		X	X	X		X
User Information	X	X			X	X		X
Capabilities	X	X			X	X		X
Roles	X	X			X	X		X
Instruments	X	X			X	X		X
Measurements	X	X			X	X	X	X
Logical Groups	X	X			X	X	X	X
Remote Data	X	X				X		X
Semantic Meaning	X	X				X		X

This *configuration matrix* represents a theoretical, ideal view of configuration management in a complex distributed system. In order to actually create any system, pragmatism must be applied, and the *configuration matrix* must become sparsely populated with the critical few items that need to be structured in a few ways in order to create a successful system. If all items and all capabilities are attempted, it is extremely unlikely

that any system which satisfies its intended users will be developed in any reasonable amount of time, using any reasonable amount of development resources.

One of the problems in identifying key items and how they need to be configured is that determining the critical few combinations usually varies widely from system to system, depending on the problem domain and the users of the system. In addition, in systems whose data comes from multiple sources, there can be configuration management needs related to these data sources. Ad-hoc methods of configuration management requirements elicitation may or may not take into account all these considerations. Thus, the resulting identified configuration management needs may or may not address key aspects of the system.

Extensive knowledge of the problem domain as well as knowledge of the user roles that will interact with the system can focus a configuration management specification and design effort in a repeatable manner. Identification of key configuration management needs to focus design effort can be accomplished using the KR method.

3. APPLYING THE KR METHOD TO THE ATM/IP NETWORK MONITORING SYSTEM

Step 1: Identify the primary domain requirements. (These requirements may not be specifically related to configuration management. Rather, they are overall problem domain key requirements. There should be no more than 3 or 4 of them, and the fewer the better. The primary domain requirements are used in steps 2 and 5 of the KR method.)

Two primary characteristics of the monitoring system are:

- large scale

The scalability aspect of this problem domain can be seen in two ways; the physical size of the monitoring system and the measured data volume and rate of production. The first aspect is most obvious. There are a very large number of geographically dispersed probes and computers that are collecting and analyzing data. The second aspect is seen at the lowest levels of the system, the probes, where the volume and speed of data seen greatly restrict the amount of analysis that occurs as well as the amount of data that can be saved for further analysis.

- dynamic environment

The dynamics of the system can also be viewed in two ways; the physical resources of the monitoring system and the physical configuration of the hybrid ATM/IP network being monitored. The physical resources consist of the ever changing instrument and computing resources that constitute the monitoring system itself; this aspect is complicated by the fact that portions of the monitoring network itself are likely to come and go seemingly at random due to network failures. The physical configuration includes the dynamics of the monitored network itself, where virtual connections are continually being created, used, and changed in order to transfer data. Note that the data itself is static rather than dynamic, with a limited time value. Beyond very short time periods, the data is not of any value.

Step 2: Identify implications and constraints of primary problem domain requirements as well as additional new requirements. (Additional requirements can stem from the primary problem domain characteristics. Primary domain requirements may

also constrain other requirements. This step identifies additional requirements and constraints that are used in step 5 of the KR method.)

Implications of the primary domain requirement of scale constrain other aspects of the system. For example, due to scale, a large number of different networked computers and probes need to communicate. There is a high probability that the system will not all be functioning at any given point in time: some network connection will fail, causing some computer or probe to fail to communicate properly with other parts of the system. Hardware failures in probes and computers are also more likely with a large system. In addition, these failures can occur without warning, at any time. This failure implication in turn provides additional requirements (e.g. more need for synchronization at critical points, more fault tolerance/autonomy built into the system, more independent operation across parts, or increments, of the system rather than across the entire system at once) and also constrains other domain requirements (i.e. completely accurate monitoring data of the ATM/IP network cannot be obtained since parts of the system are always down).

The second aspect of scale in the ATM/IP system, the volume and speed of data at the probe level in the system, constrains accuracy as well as persistence in the system. Only a quick view of patterns in the header information is possible, rather than a complete analysis across vast areas in the monitored network.

The second primary characteristic, that of a dynamic environment, implies that the actual resources involved in monitoring the network (and the network itself) change frequently. Because of frequent changes and the large scale requirement, the exact “current configuration” of the monitoring system cannot be accurately determined at any

point in time. In addition, since the virtual connections used by the ATM/IP network itself change frequently, the system being monitored also has a configuration that changes over time. Quick analysis techniques must be used to determine that configuration at any particular point in time in order to support the measurements being taken by probes. This situation calls for the additional requirement of being able to move the monitoring system analysis forward on partial knowledge.

Step 3: Identify key user roles for the system. Identify key tasks for each of these roles. (In any distributed system there are several types of users, with varying backgrounds and tasks to accomplish. The roles and tasks identified in this step are used to identify additional system requirements in step 4.)

In the ATM/IP monitoring system, the types of users¹⁸ interacting with it for configuration purposes are:

- On-site experts who are knowledgeable about the particular computer/probe configuration present in the system: Representative tasks are configuring computers, probes, network topology, and communication paths in the monitoring system as part of deployment as well as making changes in these items after deployment.
- Daily operators of the system who are not very knowledgeable about the complete hardware configuration: A representative task is to configure new recognition patterns for analysis
- Monitoring software designers who are knowledgeable about the problem domain and the complete design of the monitoring software: Representative

¹⁸ User roles originally developed by Dean Thompson, Agilent Laboratories, Agilent Technologies.

tasks include configuring static and dynamic structuring and defining dependencies of software components

Step 4: Identify the requirements of the key user roles. (The key user roles and tasks may impose additional system requirements. These additional requirements may not be directly related to run-time configuration management, but they may constrain configuration management operation. These requirements are used in step 5 of the KR method.)

The role of on-site expert adds a requirement on the monitoring software designers to quantify and use in-depth software knowledge, such as permissible component combinations, component resource requirements, versioning rules, etc. so that correct deployment configurations can be designed by the on-site expert. The role of daily operator adds requirements for fault tolerance. Both the role of on-site expert and operator have additional requirements in the area of trust; these users often do not trust that the system is performing properly, especially if it is performing automatic changes. Improper automatic operation can be catastrophic to the network, so until they trust the system operates properly they insist on making changes manually. The system therefore needs to be able to be driven from their input, and to show that it is operating correctly. *Trust* is a common issue in any network management system; automatic configuration/problem solving is often highly distrusted by on-site experts and operators.¹⁹

Step 5: Use the primary domain requirements (from step 1) and their implications (from step 2) and additional requirements imposed by user roles and tasks

¹⁹ Insight into this issue developed with the help of Rick Whitner of Agilent Laboratories, Agilent Technologies.

(from step 4) to identify the key run-time configuration management requirements in the system. (This step synthesizes data from previous steps and derives run-time configuration management issues included in the information from previous steps. Information from this step is used in step 8 of the KR method and is also input to the specification portion of the *CMSpecV&V* method described in Chapter 5.)

The scale of the network monitoring system and its dynamic environment point to the notion of incrementally moving toward a desired configuration. Because of the system size, this transformation needs to be automatic. However, trust issues raised by the user roles of operator and on-site expert indicate manual operation or at least limited automation. These conflicting needs must be reconciled. (Note also that the specific run-time configuration management needs of a factory designer may also differ from those of an on-site expert: the factory designer needs to be able to specify the allowable combinations and ordinality and resource needs of the software components, while the on-site expert needs to be able to create configurations that use this information for validation purposes.) This synthesis of information about the system leads to the following key configuration management needs of the hybrid ATM/IP network monitoring system.

The primary run-time configuration management capabilities needed by this system are the need to move automatically and incrementally toward a desired configuration based upon partial knowledge of the existing inventory of the system. Automation must be blended with the ability to handle user input and provide feedback as to the correct operation of this movement in order to address user trust issues.

Step 6: Identify the potential *items* that need to be configured. (Configurable *items* primarily depend on the particular problem domain. The complete list of *items* in Table 4-1 can be pruned based on the problem domain. In addition, new *items* may be suggested by the problem domain. The *items* are used in step 7 of the KR method.)

For the network monitoring system, the notion of a *configuration* is a new *item* that must be considered during the process of identifying key configuration management needs. A *configuration* item comes about due to the scale characteristics of the problem domain. There is one *configuration* designated as the *desired configuration* at any point in time. This is the *configuration* toward which the system is moving. Instrument resources, namely probes, are also *items* added by the problem domain. Another new *item* is that of *measured items* e.g. monitored links, recognition patterns, and filters.

Other *items* that need to be considered are either identical to those in the generic list presented in Chapter 1 (e.g. communication paths) or are modified versions of those *items* (e.g. the functional capabilities needed by this domain and set of user roles).

Step 7: Associate the potential *items* that need to be configured with the type of structuring capability that is needed to create a preliminary *configuration matrix*. (In this step, the potential *items* that could be configured determine which structuring capabilities need to be configured. The preliminary configuration matrix is used in step 8.)

For this system, the structuring capabilities are slightly modified from the generic set shown in Chapter 1, as a result of the problem domain. This modification takes the form of the need to create/define a *configuration* and the need to cause a state transition.

Next, the list of potential *items* that need to be configured is compared with the possible structuring capabilities, to determine which *items* can be structured in which ways. For example, the idea of deploying a network topology does not make sense, so this structuring capability is not included for the network topology item. But the idea of deploying a *configuration* does make sense; each part of the system must have access to the current *desired configuration* in order to move toward it. The set of potential configuration management capabilities needed for each *item* is represented in the *configuration matrix* shown in Table 4-2.

Table 4-2. Two-dimensional preliminary *configuration matrix* showing all possible combinations of *items* that could be configured and the structuring capabilities that make sense for each *item* in the ATM/IP hybrid network monitoring system. Empty elements of the matrix indicate instances of structuring capabilities that do not make sense for the *item* associated with the matrix element.

Structuring Possibilities → Items ↓	Register	Locate	Migrate	Deploy	Activate, deactivate	Bind	Instantiate	Define	State transition
Configurations		X		X	X	X	X	X	X
Software	X	X	X (agents)	X	X	X	X		X
Communication/ data paths	X	X			X	X		X	X
Network topology	X	X							
Functionality	X	X			X	X			X
Measured items	X	X		X (patterns / filters)					
User information	X	X			X	X		X	X
Instrument, computing resources	X	X			X	X			

Step 8: Use the key run-time configuration management requirements (from step 5) to prune the preliminary *configuration matrix* (from step 7) to include only the few key items that need to be configured in the system. (This step may be necessary if the preliminary configuration matrix is large. In reality, only a few key items and key

configuration management operations can be included in a delivered system. This step uses the key run-time configuration management requirements to focus the configuration management effort. Information from this step is used as input to the specification portion of the CMSpecV&V method described in Chapter 5.)

Many of the potential configurable items in the system shown in the *configuration matrix* in Table 4-2 are included in the notion of a *configuration*. These include software, communication/data paths, network topology, measured items, instrument/computing resources, and potentially user information. Thus the run-time configuration management needs for these items are bundled into those of a *configuration*.

3.1 Results

The results of the KR method for this system are validated by experience with the initial release of the Agilent Technologies ATM network management system. In the initial system the physical configuration was not incremental, nor fully automated. Customer feedback consistently raised these usability issues related to the lack of automated physical configuration. Due to the scale of the system this usability issue demanded an approach based on partial knowledge of the physical structure of the system. As a consequence, subsequent releases of the monitoring system have addressed the problems of automatic configuration in this large scale monitoring system.

4. APPLYING THE KR METHOD TO THE BIOINFORMATICS SYSTEM

Step 1: Identify the primary domain requirements.

Four primary requirements exist for the bioinformatics system:

- large scale

In this system, large scale primarily pertains to the volume of data which must be analyzed or queried in some way to obtain knowledge and information.

- dynamic environment

There are two aspects to this requirement; the data involved in the system and the sources of that data. The first aspect deals with the fact that the data is constantly changing; new data is being produced and put into databases and "old" data is being corrected or augmented. The second aspect deals with the fact that researchers tend to access many different data sources for the same or different items. Any system must deal with the need to obtain data from many sources. However, in the bioinformatics system, there are relatively few data sources compared with the volume of the data that must be processed and the rate at which that data changes.

- flexible uses of data sources and analyses

A bioinformatics system must be flexible enough to allow researchers to look at the same problem in different ways, combining, changing, and analyzing huge amounts of data to gain new insights. This requirement is related to that of the dynamic environment; researchers need to re-analyze previous results based on newly available information.

- heterogeneity of data sources and semantics

In the bioinformatics system, data comes from many different sources; measurement as well as different textual databases. In addition, the data received from each of these sources is heterogeneous in nature; both in the format and

meaning of the data. A bioinformatics system must be able to deal with the widely differing data from widely differing sources that it is being used to analyze.

Step 2: Identify implications of primary domain requirements and the corresponding constraints on other domain requirements as well as additional new requirements.

The primary requirement of large scale carries with it an implication that the system probably cannot deal with all the data at one time. This leads to a requirement that an incremental approach to dealing with and analyzing the data must be used. A constraint of the large scale requirement is that analyses cannot be very “real time” (there is too much changing data to deal with), so they are probably not completely accurate or precise.

The requirement of a dynamic environment has an implication that the system will have to re-do work once new data or changed data is present. This means a new requirement to somehow persist partial “calculations” in order to re-do analyses or to be able to start old analyses from scratch with the new or changed data. It also means the system must be able to reach at least some conclusions on an analysis with what is known to be partial data. Unlike the hybrid ATM/IP network monitoring system, data is dynamic, and it also has long-term value.

A requirement of flexibility in the bioinformatics system implies that the system must be able to re-do work with different data or to use different analysis types; it must also be able to extend previous work. The additional requirements of this implication are the need to provide the ability to change portions of an analysis “on-the-fly”; (e.g. the

data, data sources, analyses, etc.). One constraint this adds is related to persistence – the system may never really be done with any data or partial results, or analysis techniques it has previously used.

Finally, the need to meet the heterogeneity present in the system implies that the system must be distributed, due to multiple geographic sources, and also that it must be able to handle multiple semantic meanings of data. More than likely the system will have to transform different kinds of data to obtain compatible formats that can be used in analyses. A constraint of the geographic dispersion is that fault tolerance must be supplied to deal with situations where data sources cannot be reached.

Step 3: Identify key user roles for the system, based on the boundary setting that has been done for the system. Identify key tasks for each of these roles.

For the bioinformatics system, there are three main user roles. These roles and the primary tasks they must accomplish are:

- The primary researcher, whose main task of a researcher is to obtain relevant data from the data sources in order to develop hypotheses, and to plan and carry out experiments to further these hypotheses. Often the generation of a paper is the final outcome of this sequence.
- Collaborator(s), whose main task is to obtain relevant information from the primary researcher, as well as contribute pertinent information from the collaborator's own research.
- Data sources themselves, where primary tasks are to store data, updating, changing, and/or correcting it as new information is available.

Step 4: Identify the requirements of the key user roles.

Some of the requirements of the primary researcher are:

- flexibility (the system must be easy to (re)configure in terms of data used, analyses performed, and the path of logic the researcher is using)
- security (data must be protected, as must “un-cooked” results)
- automated maintenance (code, data, and functionality must be available where they are needed and when they are needed, check pointing the system, backing up the system, re-starting analyses, and knowing when data has changed or when new relevant data is available)
- the ability to keep track of the path the research has followed (both successful branches and unsuccessful side tracks).

For the collaborator, some of the requirements are that:

- latest results and “uncooked” theories need to be available on demand as well as when they are next available.

Finally, for the data sources, requirements include the need to handle:

- heterogeneous data semantics and formats
- geographical distribution
- high volume of data
- changing data as well as new data

Step 5: Use the primary domain requirements (from step 1) and their implications (from step 2), and additional requirements imposed by the user roles and tasks (from step 4) to identify the key configuration management requirements in the system.

The scale of the bioinformatics system in terms of its data and the dynamic nature of that data, point to the notion of incrementally analyzing data as well as being able to deal with changing data. The user requirement of flexibility, in terms of data and analyses, points to the need to be able to configure incremental analysis. Following this process to create a synthesis of all the information about the system leads to the following key configuration management needs of the bioinformatics system.

The key configuration management need of this system is to be able to define how the system deals with new/changed data. Related to this need is being able to configure how incrementally the system operates, the amount of persistence in the system (for partial analyses, data sources, and research tracks), and when to re-start analyses. Another key need is to be able to configure the security protection of data and “uncooked results”.

Step 6: Identify the potential *items* that need to be configured.

For this system, a new *item* can be identified from the problem domain and user requirements. Specifically this is the idea of a research track or path. The domain and user roles cause the generic *item* functionality to be defined as incremental operation, persistence, analyses, precision, timing, and security. The domain also imposes interpretations on the other generic *items* listed in Chapter 1. An example is the notion

of “measured items”; in the bioinformatics domain, measured items relates to data residing in a database.

Step 7: Associate the potential *items* that need to be configured with the type of structuring capability that is needed to create a preliminary *configuration matrix*.

The bioinformatics application does not add additional structuring capabilities to the generic list from Chapter 1.

The list of potential *items* that need to be configured is then compared with the possible structuring capabilities, resulting in a *configuration matrix* of which *items* could be structured in which ways. As in the hybrid ATM/IP network monitoring system, applying this thought process to each potential *item* and structuring capability leads to a set of potential configuration management capabilities needed for each item, represented in the preliminary *configuration matrix* shown in Table 4-3.

4.1 Results

The results of applying the KR method to the bioinformatics system are more informally validated than those of the ATM/IP network monitoring system since the bioinformatics system is a research vehicle still in the definition phases. However, the key configuration requirements identified by the KR method have been articulated by biological scientific researchers in user studies.

5. CONCLUSIONS

In an ad-hoc approach to determining key configuration management needs, the results are largely dependent on the individual(s) who define those needs; their experience with similar systems, the problem domain, and users of the system. An ad-

hoc approach does not necessarily take multiple views of the system into account and can miss important requirements, especially if they are not obvious.

Table 4-3. Two-dimensional preliminary *configuration matrix* for the bioinformatics system showing all possible combinations of *items* that could be configured and the structuring capabilities that make sense for each *item* in this problem domain. Empty elements of the matrix indicate instances of structuring capabilities that do not make sense for the particular *item*.

Structuring Possibilities → Items ↓	Register	Locate	Migrate	Deploy	Activate, deactivate	Bind	Instantiate	Define
Communication, data paths	X	X			X	X		X
Software	X	X	X (agents)	X	X	X	X	
Data format, semantics	X	X			X	X		X
Functionality	X	X			X			
User information	X	X			X	X		X
Measured items	X	X				X		
Computing, data-source resources	X	X			X	X		
Research path	X	X			X	X	X	X

By contrast, the KR method is a logical composition of steps that can be followed by system architects. It is based on the use of multiple aspects of a system, which makes it more likely that the important configuration management needs of the system being developed will be correctly identified.

In addition, the results of each step can be documented, and thus the entire process of defining key configuration management needs is traceable. A documented process is useful when system development personnel change, and can be crucial when various aspects of the system requirements change (e.g. the addition of a new type of user role with different requirements than have been taken into consideration during the

initial determination of configuration management needs). Then, the new requirements can be compared to the original ones.

Finally, the outcome of the KR method results in a small set of key configuration management needs of the final system, and therefore specification of these needs can proceed. Obviously these requirements will have to be hierarchically decomposed to the point that they can become functional specifications for the various components of the system. This movement from abstract requirements to system component specifications and the final specification notation is the basis of on-going research.

The KR method was developed using the ATM network monitoring system as a specific system example. It was successful in identifying one of the key run-time configuration management requirements that was largely missing from the original network monitoring system, as indicated through engineering maintenance information from system architects. The KR method was generalized and successfully applied to the bioinformatics system. The method correctly identified the key run-time configuration management requirements of this system as indicated through user studies conducted internally at Agilent Laboratories.

The KR method is evaluated according to the Technique Evaluation Criteria described in Chapter 3 as follows:

- 1) Reliable but potentially informal validation of results with system architect or system user feedback: This feedback exists for use of the KR method in both the network monitoring system and the bioinformatics system.

- 2) The ability to deal with all requirements of run-time configuration management of the two distributed systems used in this research: This criteria is not applicable to the KR method.
- 3) The applicability of the technique to specification versus design or implementation: The KR method is applicable to requirements elucidation and specification.
- 4) The ability of the technique to fit into industry: This criterion is comprised of the following:
 - 1) Automation opportunities: None exist with the KR method. Like most requirements elucidation methodologies, it is entirely manual, although supporting documentation and tracing could be automated.
 - 2) Existing tool support: There are no tools to help support the KR method. This is an area for future research.
 - 3) Skill set needed by practitioners: No specialized skills are needed to apply the KR method.
 - 4) Flexibility to fit into existing development practices: The KR method can be used along with other requirements elucidation techniques, or simply as a mental method during initial requirements specification.
 - 5) The ability to use the technique in a partial or incremental fashion: The KR method can be used in a partial fashion as is shown by

its use on a single aspect of a distributed system, that of run-time configuration management. It can be used incrementally, in that if more information becomes available in terms of the problem domain requirements, user roles, or items that need to be configured the subsequent steps of the method can be re-applied to determine if any of the key run-time configuration management requirements need to change.

CHAPTER 5 – EVALUATING SPECIFICATION TECHNIQUES USING THE ATM NETWORK MONITORING SYSTEM

1. INTRODUCTION

Chapter 4 presented a method to identify the key configuration management needs of the ATM network monitoring system. The key configuration need of this system is to move automatically and incrementally toward a desired *configuration*. This move must be done in a trusted manner (so users believe that the system is operating correctly), based on partial knowledge of the existing *inventory*. A *configuration* consists of items such as software, data and communication paths, network topology, identification, and history. An *inventory* consists of the actual elements of a *configuration* that are present in the system as well as associated information about them (e.g. states, ordinality, etc.). The configuration management actions of all items must be specified in order to specify run-time configuration management for the monitoring system. Specification of the actual functionality of the items that comprise the network monitoring system is an orthogonal issue, and is dependent on the desired functionality of the monitoring system. This monitoring functionality is not considered in this research.

Section 2 of this chapter discusses the specific items that need to be configured in the ATM monitoring system. This section also narrows the scope of items considered for specification to those dealing with software. This is due to the volume of items that need to be considered in a full configuration management specification.

Sections 3, 4, and 5 contain discussions of the three specification methods proposed for study: property refinement, fuzzy sets, and soft goals. Each technique is evaluated as part of the discussion. None of the methods is sufficient by itself to fully specify the run-time configuration management requirements of the network monitoring system.

A combination of some elements of these methods and some additional techniques do seem to be an effective way to specify key software run-time configuration management requirements in the network monitoring system. Existing techniques and notations must be used as much as possible in the specification methodology so that the resulting technique is readily accepted in industry. Therefore, techniques such as invariants and notations such as the Unified Modeling Languages (UML) are used where possible. (The specification methodology has also been developed with the idea of validation in mind. Constructs such as invariants can make a specification clear and can be used in automated validation tools. The development of such tools is not within the scope of this research, but is a topic for future research.²⁰) These additional techniques are presented in Sections 6-9.

A conceptual understanding of run-time configuration management using sets is shown in Section 6. UML models with OCL constraints are discussed in Section 7. Alloy is used to complement these UML models, and is discussed in Section 8. Temporal issues are discussed in Section 9, and Section 10 presents conclusions regarding the use of the various methods and notations to develop a specification for

²⁰ Future research topics are discussed in Chapter 12.

run-time configuration management of software items in the ATM network monitoring system. Section 11 presents a recommended specification method and notation.

2. ATM MONITORING SYSTEM CONFIGURATION ITEMS AND THEIR CONFIGURATION ACTION NEEDS

There are two basic configuration items in the ATM/IP network monitoring system. These are a *configuration* and an *inventory*. A *configuration* is a convenient packaging of all the *items* that need to be dynamically configured while the system is running. Associated information needed to ensure that configuration changes are allowed is also included in a *configuration*. An *inventory* contains current information about the *items* that actually exist in the system at any point in time. One *configuration*, the *desired configuration*, is the *configuration* toward which the system is moving.

A *configuration* consists of the following items:

- Software, including source code, object code, objects, services, and applications: each of these kinds of software must be manipulated independently. Attributes that allow manipulation are:
 - Version (e.g. version 2.3, or version Jan-2001-B.A6021, or a more complicated data record)
 - Platform (e.g. byte swapped little endian, and C compiler 7.B.02)
 - Dependencies (e.g. application Z needs 1 location service and 3 probe objects to run)
 - Ordinality requirements (e.g. up to 2 location services can be running on one machine at the same time)

- Location requirements (e.g. service Q needs to be running on all site servers)
- Resource requirements (e.g. object K requires instrument A to provide measurement data and needs 1M RAM for storage)
- Desired bindings (e.g. bind application P to site computer R, probe B, and network connection D)

Some attributes must be set by the monitoring software designer (e.g. source code version, dependencies, and ordinality requirements), while others need to be manipulated by an on-site expert (e.g. desired bindings)

- Data and communication paths with attributes such as speed or cost; requirements for these attributes (e.g. speed) must be set by monitoring software designers while desired attributes (e.g. cost) need to be manipulated by the on-site expert.
- Measured items such as:
 - Network links to be monitored
 - Patterns to look for in packet headers
 - Filters to apply
 - Notification/alerts

These items can be defined or created at the monitoring site, but interoperability rules need to be developed by monitoring software designers.

- Network topology includes the monitoring system hierarchy of probes, computers, and network connections; these items must be defined at the site

- A *configuration* identification, which must be unique in time and space
- A *configuration* history of some sort including time and action

The items in an *inventory* are similar to those in a *configuration*, except they are the actual *items* present in the monitoring system. The ideas of a *history* and an *identifier* from a *configuration* are not part of the definition of an *inventory*. The networking and data and communication paths are in the *inventory* as resources being used by the monitoring system, and the measured items of an *inventory* are those currently being monitored.

Key actions that must be able to take place on a *configuration* are:

- Create a configuration
- Locate a configuration
- Activate a configuration
- Deactivate a *configuration*

There are also two basic configuration management actions that need to be able to take place across the system, in order to effect configuration changes. These are:

- Compare a desired *configuration* with an *inventory*:
 - Get an *inventory*
 - Get the desired *configuration*
 - Decide what is different
- Make changes to the system based on the comparison between the *inventory* and the desired *configuration*.

- Only make changes when the affected item is at a state in its execution where changes are allowed. Only made changes that are allowed
- Deploy *items* as necessary
- Locate *items*
- Activate *items*
- Register *items* if newly activated
- Deactivate *items*

Configuration management actions must take place in the context of allowing user direction and showing correct operation. In addition, they must be able to take place in an incremental fashion.

The discussion above indicates that a large number of *items* must be considered in a complete run-time configuration management specification. This includes all the *configuration items* and all the actual *inventory items*, around 30 different *items* in addition to the unique attributes of each item and their *configuration management actions*. Therefore the scope of the research described in this chapter has been further narrowed to the software-specific *items*. These were chosen for full specification research since the run-time configuration management of software *items* is of general interest in most distributed systems.

The specification methods developed in this chapter are not expected to need changes in order to be used for the bioinformatics system. The results described in this chapter should be easily extensible to other items in the ATM monitoring system, although a demonstration of this generality is outside the scope of this dissertation. Only

those *items* associated with software are considered in all subsequent discussion. The software *items* in the ATM network monitoring system are source code, object code, objects, services, and applications.

3. THE PROPERTY REFINEMENT SPECIFICATION TECHNIQUE

The purpose of property refinement is to translate system-wide behavior statements (often called non-functional behavior) into functionality that can be specified and implemented in software components. It has been successfully used to translate the system-wide notion of reliability into specific transaction functionality. This refinement includes the concept of a transaction as well as its related actions. Similarly, property refinement can be used to translate the system-wide run-time configuration management behavior statement developed in Chapter 4 to specifiable functionality.

The key configuration management need of the ATM network monitoring system is to move automatically and incrementally toward a desired configuration. Movement must be in a trusted manner, based on a partial knowledge of the existing inventory. Property Refinement can be used to refine what is meant by automatic and incremental operation. To do this, property refinement must deal with both *configuration items* and the *actions* that need to be taken on them. *Items* of interest in the ATM monitoring system are a *configuration* and an *inventory*, both of which can further be refined into their constituent *items*. *Configuration management actions* that need to be taken on these items are to *compare an inventory* and a desired *configuration* and to *automatically make changes* to the system to bring it into alignment with the desired configuration. Movement to a new configuration must also allow user input and demonstrate correct behavior.

3.1 Properties

The *incremental* property and the *automatic* property need to be refined for this system. Each is discussed below.

3.1.1 Incremental Property

In general, the *incremental* property constrains *configuration actions* taken on *items* to occur at certain times (*ReadyTime*) and over certain spaces (*ActionSpace*). The *items* involved in this system are the software *items* in *configurations* and *inventories*. *ReadyTime* implies that certain states have been reached in the running system, and *ActionSpace* implies that *configuration actions* will occur over particular physical or logical areas.

ActionSpace is a geographical locality in the hybrid ATM/IP network monitoring system. This means that software configuration managers are located at local sites, and they are responsible for all *configuration items* at their site.

ReadyTime means that for each of the software *configuration items*, there must be points in their execution when the *item's* configuration can be changed.

All *configuration actions* must finish within some acceptable amount of time. (This is needed so that if a failure occurs, such as a network failure, the system can attempt to recover rather than wait forever for a configuration action to finish.)

Configuration actions must also be able to take user input and demonstrate correct behavior. These requirements must be balanced with automatic operation since the system is so large; accepting user input and demonstrating correct behavior of every change in a huge system is not feasible.

The definition of *incremental* in this system is:

For all software *items* in an *ActionSpace*, when *ReadyTime* is acceptable, a desired action takes place on the *items*, in an acceptable timeframe.

3.1.2 Automatic Property

The *automatic* property has to do with taking desired, acceptable *configuration actions* on *items*, without external intervention. The lack of external intervention implies that all the information needed to decide to take an action is present, and all the operations necessary to be able to take the action exist.

In the monitoring system, the *configuration management actions* are to get an inventory, then to compare the inventory and a desired configuration, then to change items to move toward the desired configuration. Since user trust is an issue, this automation must be tempered by the amount of automation acceptable to users.

The definition of *automatic* in this system is:

Be able to get all the information needed to make a decision about a configuration change and be able to make the configuration change if automation is acceptable to the user.

3.2 Conclusions

Property refinement can be used to develop statements about a property over all the *items* and *configuration management actions* in the system. As soon as it is possible to say “for all” or “for every” then there is a way to track whether the property obligations at the component level have been fulfilled.

If precise specifications exist for each type of software item and configuration management action (including what each action does; its semantics), then an automated

tool can be built to help provide verification. A tool can look for the configuration management action signature and semantics required by all members of each kind of software item and perform existence and consistency checks. Even if such tools do not exist, property statements can be used to provide gross system checking.

3.3 Property Refinement Evaluation

The property refinement technique is evaluated using the technique evaluation criteria discussed in Chapter 3:

- 1) Validation of results using system architect or user feedback: There is no feedback available to validate the results of property refinement.
- 2) Ability to specify all necessary requirements: Property refinement can be used to develop predicates regarding functionality. However, if properties are not refined to this level there is not enough detail to specify functionality.
- 3) Applicability to specification: Property refinement is very suited to specification, especially at abstract levels, since it explores the meaning of properties and helps identify their refinements.
- 4) Ability to directly support validation/verification: Property refinement can be used to informally validate the existence of behaviors across a system since it yields statements regarding functionality of some or all parts of the system.
- 5) Ability to fit into industry:
 - 1) Automation potential: none
 - 2) Existing tools: none

- 3) Skill set needed by practitioners: Skill in predicate logic is needed for complete refinement. No particular skills are needed for more abstract use of property refinement.
- 4) Flexibility to fit into existing development processes: This technique can easily fit into specification and design activities since it is principally a thought-ordering technique.
- 5) Ability to use partially or incrementally: The technique works well for partial or incremental use. In this example, it was only used to partially refine incremental and automatic properties. It is also possible to continue to refine these statements until predicate logic statements are derived that completely specify the functionality needed to implement the properties.

4. THE FUZZY SETS SPECIFICATION TECHNIQUE

Fuzzy set analysis is based on a single system operation. For this analysis, the operation is to move the system to a new configuration: *config (sys)*. Operation requirements are constrained by fuzzy sets (e.g. HIGH and LOW). *Config* operation requirements and their fuzzy set constraints are shown in Table 5-1. Relationships between requirements are identified. These relationships can be non-existent (*irrelevant*), *cooperating*, *conflicting*, or some degree thereof. Tradeoffs between requirements are made based on the relationships between them.

Normally relations between requirements are not completely cooperating or completely conflicting, but rather reflect a degree of cooperation or conflict. However, without a complete design or some empirical data, the exact relationship between any

two requirements is difficult to ascertain. Therefore I have added a relationship type “harder”, which indicates a diminished degree of cooperation. Table 5-2 shows the types of relationships used in the discussion below.

Table 5-1. *Config* (sys) requirements and their constraining fuzzy sets.

Requirement Identifier	Requirement and Fuzzy Set Constraint	Explanation
Incremental	incremental (config (sys)) should be HIGH	<i>config</i> needs to operate incrementally
Automatic	automatic (config (sys)) should be HIGH	<i>config</i> needs to operate automatically
User Input	user-input-driven (config (sys)) should be HIGH	<i>config</i> needs to take user input
Demonstrate	demonstrate-correct (config (sys)) should be HIGH	<i>config</i> needs to demonstrate correct behavior
Time	time (config (sys)) should be LOW	<i>config</i> should take very little time
Scale	handle-large-scale (config (sys)) should be HIGH	<i>config</i> needs to handle large scale systems

Table 5-2. Possible relationships between two requirements.²¹

Relationship Symbol	Ordinal Rank	Relationship description
+	2	<i>cooperating</i> , fulfilling one requirement helps fulfill the other
•	1	<i>harder</i> , fulfilling one requirement makes it harder to fulfill the other
0	0	<i>irrelevant</i> , fulfilling one requirement has no effect on fulfilling the other
-	-1	<i>conflicting</i> , fulfilling one requirement prohibits fulfilling the other

²¹ Symbols or an ordinal scale can describe relationships. Note that the numerical scale in the table is an ordinal scale, not a ratio scale (the latter would indicate a complete mathematical understanding of the relationship between requirements).

A complete mathematical description of the relationship between any two requirements is unknown during specification. Therefore the effect of the relationship cannot be precisely computed for either of the requirements, and depending on which requirement is studied, the other may positively or negatively influence it. For example, if *requirement A* = *requirement B*, we can state the full relationship with this equation. When no precise mathematical relationship exists, we can only make the general observations regarding the two requirements. In the ATM system, for example, having to handle user input makes demonstrating correct behavior harder, but having to demonstrate correct behavior has no effect on the need to handle user input. (See Figure 5-6, paths from $U \rightarrow D$ and from $D \rightarrow U$.)

4.1 Requirements Relationships

The relationships between the requirements listed in Table 5-1 are shown in Figure 5-1. The relationships are different depending on which end of a relation is the focus. The relationship symbol closest to a requirement symbol shows how that requirement is influenced by the requirement at the other end of the relationship. For example, the incremental requirement (I) is made harder by the need to deal with large scale(S), while the scale requirement (S) is helped by incremental operation(I).

Figure 5-1 is complicated by the fact that each relationship can be viewed in two ways. For example, incremental operation helps the system deal with larger scale, so a cooperating symbol is placed near the scale requirement in the figure on the line from it to the incremental requirement. On the other hand, larger scale makes it harder to perform tasks incrementally (since there are more tasks to do, or they cover a larger scope). This causes a dot to be placed on the line between the two requirements near

the incremental requirement. The complete set of relationships between these requirements are:

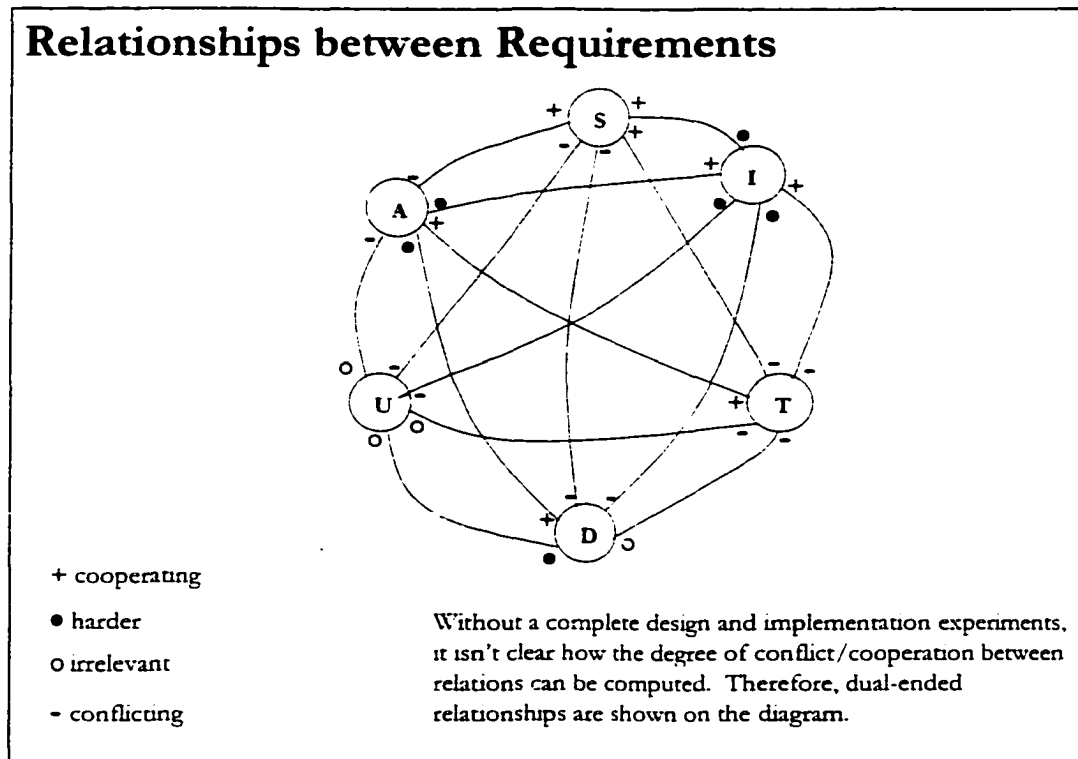


Figure 5-1. Relationships between requirements of key configuration management elements of the ATM network monitoring system. Relationships can be cooperating, conflicting, irrelevant, or simply harder to achieve. The harder category was added to the relationship types defined by Liu and Yen [50] due to the inability to accurately compute the degree of conflict between two requirements. The same problem led to the use of double ended relationships. Requirements shown in the figure are Scale, Incremental, Automatic, handle User input, complete in a short Time, and Demonstrate correct behavior.

- Scale is positively influenced by increased incremental action, automation, and minimized time to take actions. On the other hand, large scale is negatively impacted by handling user input and demonstrating correct behavior.
- Incremental action is positively influenced by increased automation and by minimized time to take action. It is made harder by the need to demonstrate correct behavior, handle user input and large scale.

- Automation is positively impacted by minimized time to take actions, and made harder by increased incremental action and the need to demonstrate correct behavior. Increased scale and handling user input negatively impact automation.
- Handling user input is negatively impacted by incremental action and large scale; the time to take actions, automation, and demonstration of correct behavior are irrelevant to the user input requirement.
- The need to demonstrate correct behavior is positively impacted by automation, made harder by handling user input, and negatively impacted by large scale and incremental action. The actual time to take action is irrelevant to this requirement.
- Finally, the time needed to take an action is positively impacted by automation, but negatively impacted by all the other requirements; large scale, incremental action, handling user input, and the need to demonstrate correct behavior.

4.2 Observations

Prioritization among requirements is needed when relationships are different depending on which requirement is viewed. Fuzzy sets can help set prioritization based on which requirements positively influence higher priority requirements. For example, **S** is known to be a key problem domain requirement, so it needs to have highest priority. **A** and **I** are next in priority since they positively influence **S**. Thus, the highest priority is to handle large scale, next is to have HIGH incremental and automatic operation. Since **I** conflicts with **T**, and **I** has been chosen to be HIGH, **T** cannot also be HIGH. Similarly, **U** and **D** conflict to some extent with requirements already chosen to be HIGH. However they are necessary to deal with the system configuration management

trust issue, so they have to be included to some extent. One solution is to bound them. Similarly, since most things conflict with **T**, an acceptable upper bound on **T** could be used to prevent problems without further minimization effort.

Aggregation, as described by Liu and Yen [50], can be used to some extent to show other relationships. For example, if minimum acceptable lower bounds are established for **U** and **D**, then:

$$T = \max (\min (I), \min (U), \min (D))$$

$$A = \min (U)$$

4.3 Conclusions

It is possible that understanding requirement relationships may help prioritize requirements, but since the relationships are not computable during specification this may be of limited value. Defining lower bounds on **U** and **D** may help with the process of developing overall specification predicates for those operations, but again this is not clear. The whole issue of deriving a true mathematical relationship between requirements is not addressed by Liu and Yen. Perhaps empirical data could be used if enough existed for systems similar to the one being specified, or perhaps if the design were completely developed these mathematical relationships could be determined. The fuzzy set technique is only useful to abstractly prioritize and understand tradeoffs between requirements unless there is a more complete understanding of the mathematical relationship between those requirements.

4.4 Fuzzy Sets Evaluation

The fuzzy sets technique is evaluated using the technique evaluation criteria discussed in Chapter 3:

- 1) Validation of results using system architect or user feedback: There is no feedback available to validate the results of fuzzy sets.
- 2) Ability to specify all necessary requirements: Fuzzy sets cannot completely specify all the software run-time configuration management requirements of the ATM monitoring system. Fuzzy sets only analyze relationships between requirements for a single system operation.
- 3) Applicability to specification: Fuzzy sets are not suited to specification, since mathematical relationships between requirements are not generally known during this phase of system development.
- 4) Ability to directly support validation/verification: none
- 5) Ability to fit into industry:
 - 1) Automation potential: none
 - 2) Existing tools: none
 - 3) Skill set needed by practitioners: No particular skills are needed.
 - 4) Flexibility to fit into existing development processes: This technique can easily fit into different development activities since it is principally a thought-ordering technique.
 - 5) Ability to use partially or incrementally: The technique may be used partially, however it is not clear how to resolve conflicting

relationship information that can result from studying different system operations.

5. THE SOFT GOALS SPECIFICATION TECHNIQUE

This technique can be used to decompose requirements into sub-goals (sub-goals completely partition a higher level goal) and satisficing goals (goals that are necessary but not sufficient to complete the higher level goal; one satisficing goal may also help complete multiple higher level goals), and to establish argumentation goals (additional requirements that help prove a requirement has been met or that it cannot be met). Myopolous [59] describes the technique as useful for making design decisions.

The overall key configuration management requirement for the network monitoring system is to move automatically and incrementally to a new system configuration while preserving high user trust. The overall system therefore has three soft goals: incremental (I), automatic (A), and high user trust (U). The conjunctive relationships between these three soft goals are shown in Figure 5-2.

The soft goals themselves can be decomposed into more sub-goals, satisficing goals, and argumentation goals. Argumentation goals exist between the three main soft goals. The decomposition of the three soft goals is shown in Figure 5-3. The incremental soft goal can be decomposed into three sub-goals: defining, comparing, and changing. Each of these sub-goals can be further decomposed into satisficing and argumentation goals. (Due to space constraints, Figure 5-3 shows these sub-goals as bubbles labeled “define”, “compare”, and “change”, placed below the incremental soft goal title. Their associated satisficing and argumentation goals are shown inside the respective bubbles.)

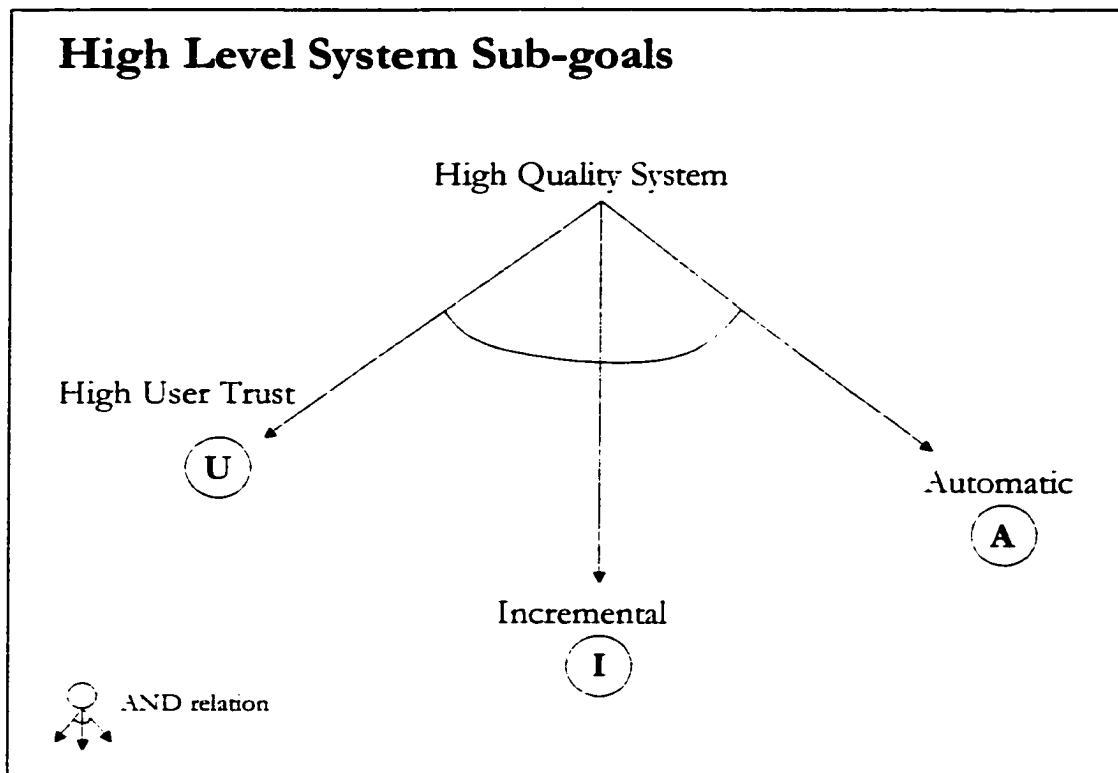


Figure 5-2. The overall soft goals of the ATM network monitoring system. These soft goals may be viewed as sub-goals of the complete system.

Figure 5-3 shows the basic decomposition of the three main soft goals in the ATM monitoring system (high user trust, incremental operation, and automatic operation). Sub-goals names are **bolded and underlined**. Argumentation goal names are *italicized* and have dotted lines; these are ideas that help prove or disprove that a goal has been met. Satisficing goals are shown in normal font with solid lines; these are goals that help the higher-level goal be met. Satisficing goals can be thought of as necessary, but not sufficient goals - many satisficing goals may be needed to allow one higher level goal to be met, and one satisficing goal may help several higher level goals be met. The bubbles outlined with dash-dot-dot lines indicate the three sub-goals of the incremental soft goal: *define*, *compare*, and *change*.

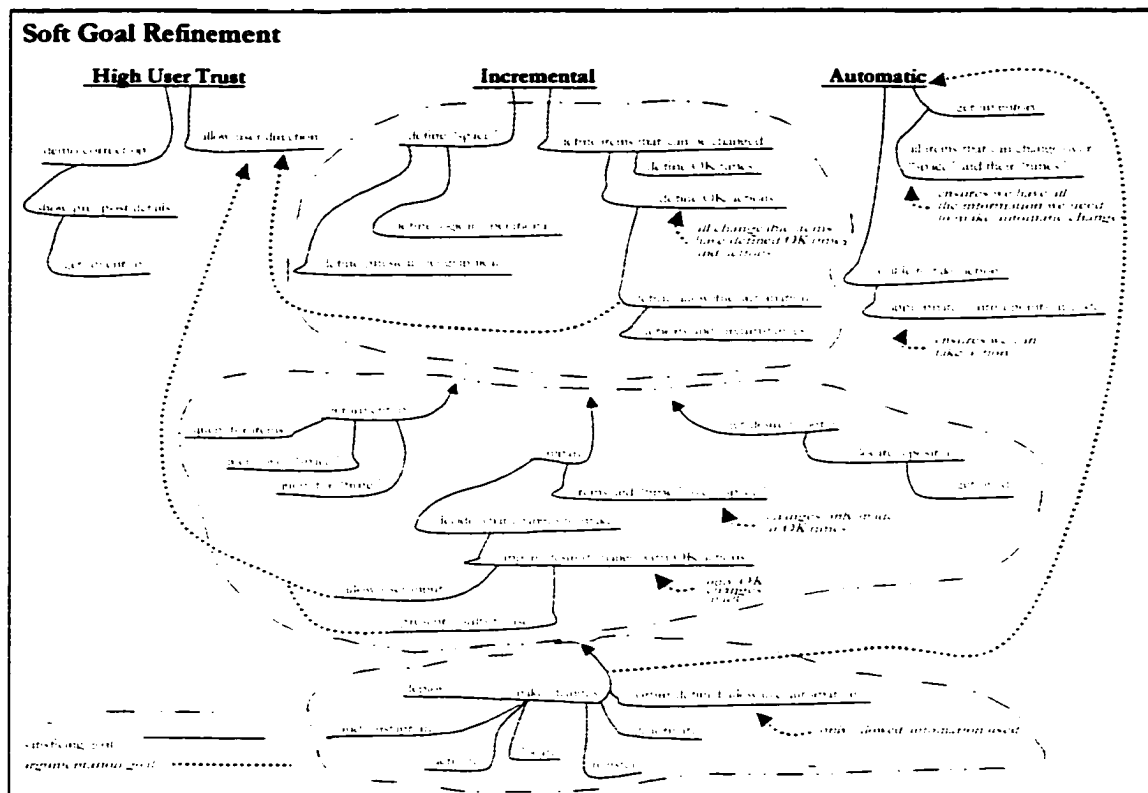


Figure 5-3. Soft Goal refinement. This figure shows the breakdown of the three major soft goals, or sub-goals of the overall system, including argumentation goals within and between major soft goals. Bubbles (outlined with a dash-dot-dot line) denote the sub-goals of the incremental soft goal. Sub-goal names are bolded and underlined. The associated satisficing and argumentation goals are shown within each bubble. Satisficing goals are shown in normal font and with solid lines and argumentation goals are shown in italicized font with dotted lines.

Figure 5-3 shows the high-level soft goal breakdown. There are also relationships between the soft goals at the high level. These relationships are shown in Figure 5-4. They are most easily identified after the diagram decomposing the high level soft goals is constructed (e.g. Figure 5-3). For example, Figure 5-3 shows that if the incremental soft goal is met, this provides argumentation goals that the high user trust soft goal is also met. The incremental soft goal is therefore *satisficing* to high user trust (as is shown in Figure 5-4). On the other hand, the incremental soft goal is not helped by high user trust. (This is shown by the fact that there are no argumentation goals from the user trust soft goal to the incremental soft goal in Figure 5-3.) Incremental operation can be more easily satisfied if the high user trust soft goal is not considered at all. Thus,

the high user trust soft goal has a *-sub* relationship to the incremental soft goal in Figure 5-4.

Figure 5-4 shows that the incremental soft goal is aided by the automatic soft goal and also aided if the high user trust soft goal is not met. The automatic soft goal is not really influenced by the incremental soft goal, but may be unattainable if the high user trust soft goal is met. The high user trust soft goal may be aided by incremental action, but it can be unattainable if the automation soft goal is completely met.

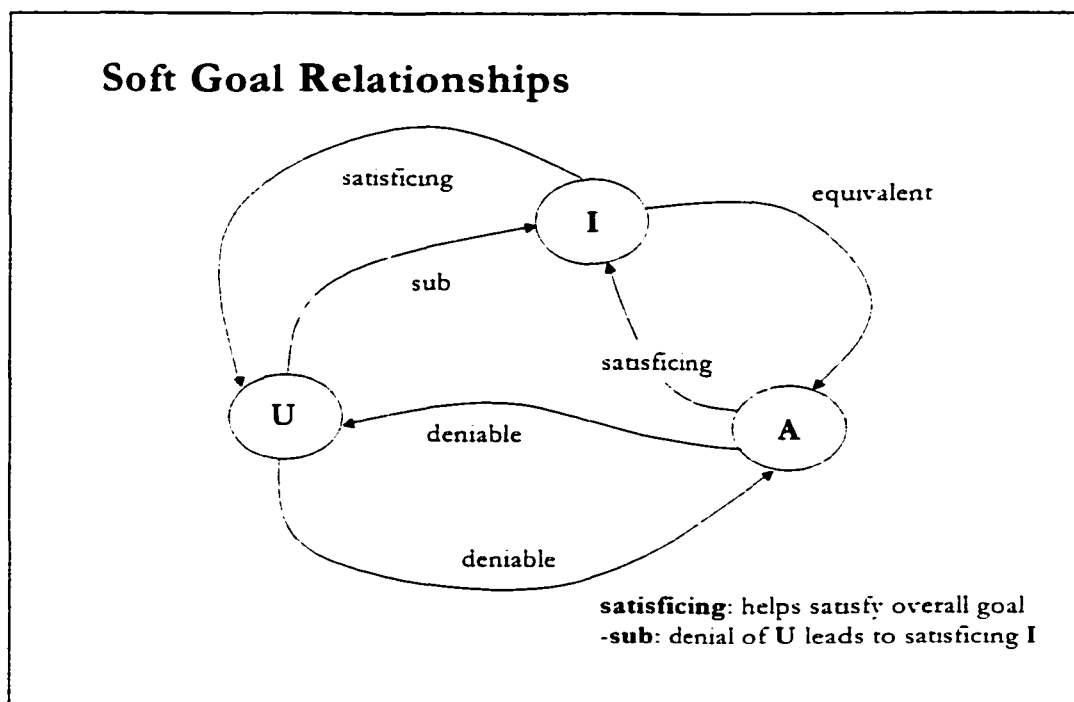


Figure 5-4. Soft goal relationships. This figure shows the relationships between major soft goals incremental operation (I), automatic operation (A), and high user trust (U).

5.1 Conclusions

Relationships among high level soft goals are most easily identified once their decomposition is defined (as in Figure 5-3). These relationships are similar to those identified by fuzzy sets across the high level requirements of the system.

The soft goal technique seems most useful during design. The number of design choices is large, and their impact on overall system requirements are least clear during design, so this is a good time to use the soft goal technique. There are not many choices to make during the specification phase since a specification speaks to what a system should do, not how it should do it. Therefore this technique is of limited value for system specification, except in helping to refine particular soft goals in terms of their sub-goals and satisficing goals. The soft goals technique also provides some rationale about decomposition through argumentation goals. Therefore a diagram such as that shown in Figure 5-3 is most useful for specification purposes. The other diagrams, Figures 5-2 and 5-4 do not add information to a specification.

The static and dynamic structuring operations needed by the ATM monitoring system are identified using this technique. Static structuring includes defining configurations. Dynamic structuring includes getting a configuration and getting an inventory. Configuration operations needed by software items are also identified by the technique. These include deploying, binding, instantiating, activating, and deactivating software items in the system.

5.2 Softgoals Evaluation

The soft goals technique is evaluated using the technique evaluation criteria discussed in Chapter 3:

- 1) Validation of results using system architect or user feedback: There is no feedback available to validate the results of soft goals.
- 2) Ability to specify all necessary requirements: Soft goals do not contain enough detail to completely specify the ATM monitoring system software run-

time configuration management requirements. The soft goal diagrams contain information about actions, but no information about actual software items, their attributes, and the relations between software items and other run-time configuration management concepts.

3) Applicability to specification: Even though soft goals were developed to aid in design decisions, this technique is useful in specification. It is especially useful for refining system operations.

4) Ability to directly support validation/verification: Argumentation goals provide direct validation that behaviors represented by soft goals are present in a specification.

5) Ability to fit into industry:

1) Automation potential: none

2) Existing tools: none

3) Skill set needed by practitioners: No particular skills are needed.

4) Flexibility to fit into existing development processes: This technique can easily fit into different development activities since it is principally a thought-ordering technique.

5) Ability to use partially or incrementally: The technique is very useful in an incremental and partial application. Since refinement occurs hierarchically it can be taken to any useful level during specification.

6. SPECIFICATION USING SETS

The software components of a configuration consist of source code, object code, object, services, and applications. There are static and dynamic structuring needs for each of these items. Static structuring deals with attributes of software items. Some static structuring needs must be defined as part of the monitoring software designer role²²: ordinality requirements, resource requirements, dependencies (identification and revision) and state requirements (state definitions and what transitions are acceptable and when). Some static structuring needs must be defined by on-site experts: acceptable automation, desired bindings, and the definition of a local set (i.e. *ActionSpace* as defined in the property refinement discussion of incremental operation in Section 3). Dynamic structuring needs include deployment, activation, resource binding, instantiation, registration, state transition, and de-activation. These operations must be included in software items. Finally, configuration operations include getting a desired configuration definition, getting an inventory, comparing the two and deciding what to change, and making changes. These are operations of a configuration manager.

Sets can be used to define software items along with the relations between them, as well as the attributes and operations that must be available in each item. The sets, relations, attributes, and configuration operations that must be specified are shown in Table 5-3.

²² In the ATM/IP monitoring system, the types of users interacting with it for configuration purposes are the monitoring software designers who know about the problem domain and the "ins and outs" of the software, on-site experts who are knowledgeable about the particular computer/probe configuration present in the system but not too knowledgeable about the actual "ins and outs" of the system software, and daily operators who are not completely knowledgeable about the entire hardware nor software configuration.

A special instance of a service, called a *configuration service*, is the service that performs configuration changes. The configuration service must implement several operations in order to effect configuration changes. These are: get-config (obtain the *desired configuration*), get-inventory (identify the “local” inventory), and compare-change (compare the *desired configuration* to the current inventory and decide what changes need to be made and make them).

Some sets (e.g. attributes and actions) contain information that is common across the other sets. For example, the attributes *version* and *resource requirements* exist in the source code, objects, applications, and services sets. (Different attributes are part of the object code set – *machine type* to identify on what machine the object code will run, and a *compiler* identifier to identify the compiler used to create the object code.) In addition, the configuration service item augments the attribute set with the items that belong to the local set it manages. (The configuration service item also augments the set of Actions.)

Each software item type needs its own set of attributes, state requirements, etc. This helps support the notion of emergent properties in the higher-order software items. For example, a service might have resource requirements that are not simply a sum of the resource requirements of the objects from which it was created. The service might actually experience a need for fewer resources due to synergistic operation of objects. Similarly, the definition of a STOP state and the acceptable transitions to it may be different for an application than for each of the objects from which it was created.

There is no convenient way to describe what concrete items are actually considered part of any particular set (e.g. item XYZ is a member of the source code set,

or item ABC is a member of the services set). However, the software item sets (source code, object code, objects, services, and applications) can be viewed as a starting point for a UML model of the entire system. In this case the software item sets can be viewed as abstract classes, and the designer can then use the correct abstract class as a starting point for other classes in the actual monitoring system.

Table 5-3. ATM Monitoring System Sets, Relations, Attributes, and Actions

Item to be specified	Examples
Sets	Source code, object code, objects, services, applications, local set (<i>ActionSpace</i>), configuration items, sets based on relations (e.g. created-from and depends-on)
Relations	Membership, created-from, depends-on
Attributes	Version, states, state transitions, and when transitions can be made, resource requirements and usage, ordinality requirements, desired bindings, acceptable automation for configuration changes, machine type and compiler identifiers
Actions	Deploy, locate, activate, checkpoint, state transition, deactivate, check resources and version, set bindings, set automation, get inventory, register (for software items); get a desired configuration, get an inventory, and compare and make changes (for a configuration manager, or configuration service)

Constraints on system-wide behavior must also be specified. Property refinement was used to create more precise statements regarding incremental and automatic functionality from vague behavioral concepts. However, the functionality that must be present in the system to meet these properties is not easily included in a specification comprised solely of sets, unless the functionality of each action in the set of actions is more precisely specified.

The functionality that must be specified to meet the incremental property includes:

- Local areas of the system can work independently, moving toward a new desired configuration.

Independent operation can be achieved by creating local groups of configurable items. The components of a local group must be controlled only by the configuration service of that local group. A related specification is that a local group may have only one configuration service.

- Do not move toward a new desired configuration unless the minimum time has passed since the last move began.

This can be accomplished using a minimum time between desired configuration moves in the *compare-check* operation.

- A configuration change can occur only when a configuration item is ready.

This can be accomplished by moving the runnable item to a IDLE or STOP state and then starting the change.

- Configuration changes happen within an acceptable time.

This can be accomplished with a timeout on called operations.

The following functionality must be specified to achieve automatic configuration management:

- Configuration changes must be able to take place on every runnable item in the system.

This can be accomplished through the definition and independent configuration of local sets.

- Configuration actions may be applied to all kinds of items.

This can be accomplished if each type of configuration item defines all the configuration operations.

- All the information needed to make a configuration change must be obtained.

This can be accomplished by defining an inventory to contain information necessary to make configuration change decisions.

- A configuration action must be able to be performed.

This can be accomplished by defining the configuration operations in each type of configuration item. This means that the necessary operations are available in software items to make configuration changes.

- If the information is sufficient and automation is acceptable, make the configuration change.

This can be accomplished in the *compare-change* operation by checking information in an *inventory* and the *automation* associated with a runnable item.

6.1 Set Modeling Evaluation

The set technique is evaluated using the technique evaluation criteria discussed in Chapter 3:

- 1) Validation of results using system architect or user feedback: There is no feedback available to validate the results of using sets.
- 2) Ability to specify all necessary requirements: Sets alone are insufficient to completely specify the ATM monitoring system software run-time

configuration management requirements. There is no way to specify operation functionality without additional notations.

3) Applicability to specification: Sets are a useful organization technique during specification.

4) Ability to directly support validation/verification: none

5) Ability to fit into industry:

1) Automation potential: none

2) Existing tools: none

3) Skill set needed by practitioners: No particular skills are needed.

4) Flexibility to fit into existing development processes: This technique can easily fit into different development activities since it is principally a thought-ordering technique.

5) Ability to use partially or incrementally: The technique can be used in an incremental and partial fashion by including more or less detail in sets. However, sets alone are not sufficient to specify operation functionality.

7. DEVELOPING TYPE MODELS USING CONCEPTS, UML TYPES, AND OCL

Type models can be developed using results from the methods discussed above.

A *type* model consists of *concepts*, UML types, and the relations between them. In the UML, a "...Type is used to specify a domain of objects together with operations

applicable to the objects without defining the physical implementation of those objects” ([63]). A *concept* is a domain of objects. (See [47] for a discussion of *concepts*.) A *concept* becomes a *type* once its operations are identified. Partial operation signatures and pre- and post-conditions are used to specify operations for the ATM network monitoring system. Specific operation return values are not needed at this level of abstraction, so the operation signatures do not include return values.

Problem domain ideas are shown as *concepts* rather than attributes of existing *concepts* if these ideas are shared by other *concepts* or *types*, or if it is not clear what their data type should be. An example of multiple sharing is *SrcType* in Figure 5-5. The problem domain idea is that software functionality can be categorized (e.g. as being a lookup functionality, or a pattern matching functionality). *SrcType* is used by *SourceCode* as well as by *Inventory*. *SrcType* is used by *Inventory* to keep track of how many runnable items of any particular category exist in a local set. (This is used by the configuration service to determine if additional runnable items of a particular category can be started during a move to a new desired configuration.) An example of the case where data type is not clear is the *Version* of a *VersionableItem*. The version can be designed as a number, a string, or a more complicated record. Any of these choices may be correct, so this problem domain idea is shown as a separate concept in the *type* model.

Behavior is shown in two ways. First, operation signatures (excluding return value types) are shown in the <<operations>> section of the concept in the *type* model. Second, a state transition diagram is shown in the type's <<state machine>> section. These two sections show the operations as well as their effect on the state of the object. OCL descriptions of the operations use information from the state diagram.

The Object Constraint Language (OCL) can be used to specify constraints and operation pre- and post-conditions in UML models. OCL was added to the *type* model using two approaches. The first was to place OCL constraints in the *type* model. The second approach was to employ a research tool called USE (UML-Based Specification Environment) that was developed at the University of Bremen.²³ The USE tool requires the entire model to be written in OCL. The USE tool does not implement the full OCL language, and its use was subsequently abandoned.

7.1 *Type Model*

The *type* model of the software run-time configuration management specification of the ATM network monitoring system is presented in several figures. The complete OCL text that is needed to complete the *type* model is shown in Appendix 5A.

Figures 5-5 and 5-6 show the fundamental software *types* (Figure 5-5) and their operations (Figure 5-6). Figure 5-7 shows the interactions of the software *types* with other configuration management-related *concepts* and *types* (i.e. versions, resource, and software states).

There are five basic types of items that need to be managed in the *type* model of configuration management software items. These are sources, object code, objects, services, and applications. Not all of these items may exist in the system (depending on the particular application), but in all cases there will at least be source code, object code, and some sort of executing code. All five types of software items exist in the ATM network monitoring system. These are shown as specializations of *ConfigItem* in Figure 5-5.

²³ See <http://> for a complete description of the tool, and [72].

Configuration changes are made to executing or executable items (i.e. applications, services, and objects). These items are specializations of *RunnableItem*. The *RunnableItem* type contains methods needed to perform configuration changes. An example of a configuration management action is to *deploy* an application to several computers. Deployment may simply consist of copying executable code to a machine, or it may be more complicated. For example, it may be necessary to obtain the corresponding object code for a particular machine type and compiler (see the *Platform* aggregation in Figure 5-5), link it, then put it on a machine. If the object code cannot be obtained for some reason, the corresponding source code may be needed. The source code can be compiled and linked to create the proper executable.

There are often other applications, services, or objects that must be executing in order for an application, service, or object to execute properly. For example, an application may need to use a location service to find a particular kind of object. In this case the application is dependent on the existence of a running location service. This kind of dependence leads to the concept of a dependency set, and the *dependsOn* relation. This relation is used to define the dependency set for each kind of configuration management software item in the system. The relation is first defined for the generalized *ConfigItem*. OCL statements precisely define the kinds of items that can be in a dependency set for the other specializations (see Figure 5-5).

Another fundamental type in the ATM network monitoring system is the *LocalSet*. This type allows incremental dynamic configuration across the system. It is shown in the type model. Figure 5-8, shows the definition of the loose aggregation of a *LocalSet*. It consists of *RunnableItems*, the *Machines* they run on, and associated *Registries* and *Inventories*. Figure 5-9 shows the relations between portions of a *LocalSet* and other configuration

management-related concepts (i.e. *MachineType*, *Resources*, *SrcType*, *Configurations*, and *Repositories*).

7.2 Observations

Some observations about the *type* model include:

- OCL constraints in the *type* model describe component and association specification, association constraints, and operation pre- and post- conditions. Unfortunately, the use of OCL in industry is limited. This is due to the on-going definition of OCL, and because system architects do not see the benefits of adding OCL expressions to their models compared to the effort of creating these expressions. OCL is another language that must be learned, and unlike many programming languages, there are no readily available automated tools that can be used to check the syntax and semantics of OCL expressions included in UML models.
- OCL can specify system-wide constraints on behavior. An example is independent operation of portions of the system, specified by constraints shown in Figure 5-8. The constraints state that: 1) local sets are unique, 2) machines only belong to one local set at a time, 3) inventories only represent one local set, 4) registries only represent one local set, and 5) runnable items only belong to one local set.
- Temporal constraints of the ATM network monitoring system are not shown in the UML/OCL *type* model.

- OCL/UML allows development of models containing *concepts* and *types*, relations, and operations. It has built-in data types (e.g. Integer and Set) and operations on them (e.g. mathematical operations on Integers and operations such as union and iterate on Sets). These built in data types and their associated operations can cause a subtle shift in the viewpoint of the modeler to the more concrete aspects of a system, thus potentially turning specification into design. Operations like *iterate* exacerbate this problem. The shift from abstraction to concrete aspects is less likely to occur when the modeler only uses sets to model the system (as when using Alloy), although no doubt experience can help keep the specification process at an abstract level. An example of this problem occurred when writing OCL post conditions for operations like *deploy*. The *deploy* operation must ensure that the runnable item being deployed was built for the machine type where it will be running. One way to perform such a check is to iterate through the associated object code, however there are many ways to accomplish the check. An OCL expression that uses the *iterate* operation can be interpreted as the correct design, precluding other methods that accomplish the machine type check. A more abstract expression uses the *forAll* operation on the associated object code. The general nature of OCL means that the architect of the requirements specification must initially identify the kinds of models and language constructs that will be used in a requirements specification in order to prevent the move to design. These constraints must be understood and adhered to by all participants in the specification process.
- UML/OCL does not provide operation composition. This is a problem for the ATM network monitoring system, since simple operations are composed to

create more complex ones. Operations typically perform some configuration action and then return a value indicating whether or not they were successful. A complex operation can therefore be composed of several of the simpler operations. An example of a simple operation is *deploy*. Deploy takes a version and a machine as arguments and checks argument values before putting the runnable item on a specified machine. If these events occur correctly, the client receives a correct operation return value. UML/OCL does not allow the *deploy* operation to be composed with other operations to create more complex operations such as *activate*. Instead, the *deploy* post conditions must be duplicated in the *activate* operation.

One solution is to use function composition, which is supported by Alloy. Configuration management operations can be specified in Alloy using function composition. Another approach is to integrate actions into the OCL language and add action methods to the specification of the *deploy* operation. Without one of these capabilities in OCL, post conditions of simple methods like *deploy* must be repeated in complex methods such as *activate*. This can lead to duplication errors and a verbose specification.

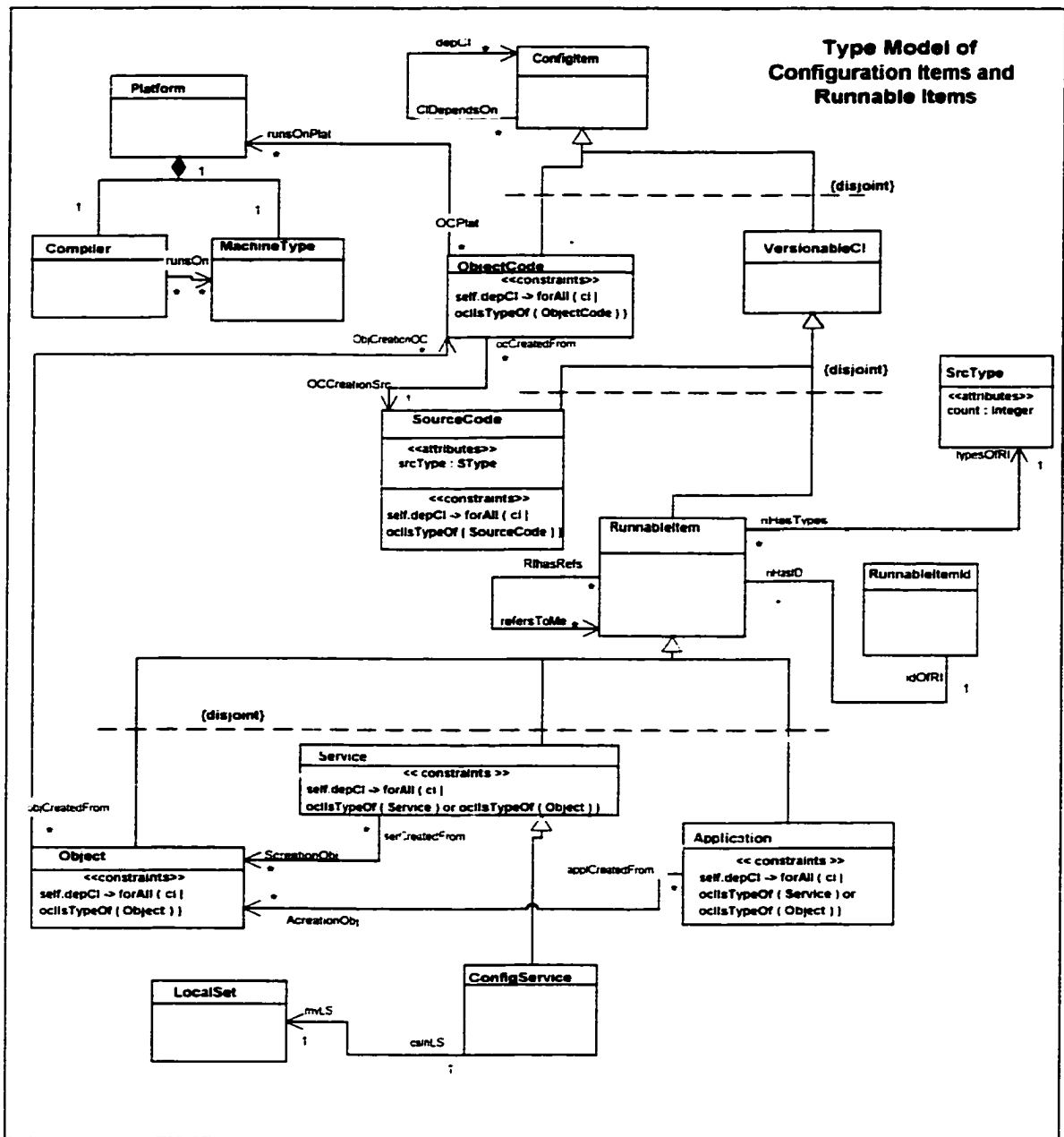


Figure 5-5. Type model of case study software configuration management items showing software items and their associations. Dependency relations define a dependency set for each software item type. Constraints on the specialized dependency relations are given for the *SourceCode*, *ObjectCode*, *Object*, *Service*, and *Application* specializations of *ConfigItem*. The dependency set of an *Object* cannot contain another *Service* or *Application*. The dependency set of a *Service* cannot contain another *Application*. An *Application*, on the other hand, can depend on a *Service*, and/or *Object*. A *ConfigService* performs actual configuration changes on an associated *LocalSet*.

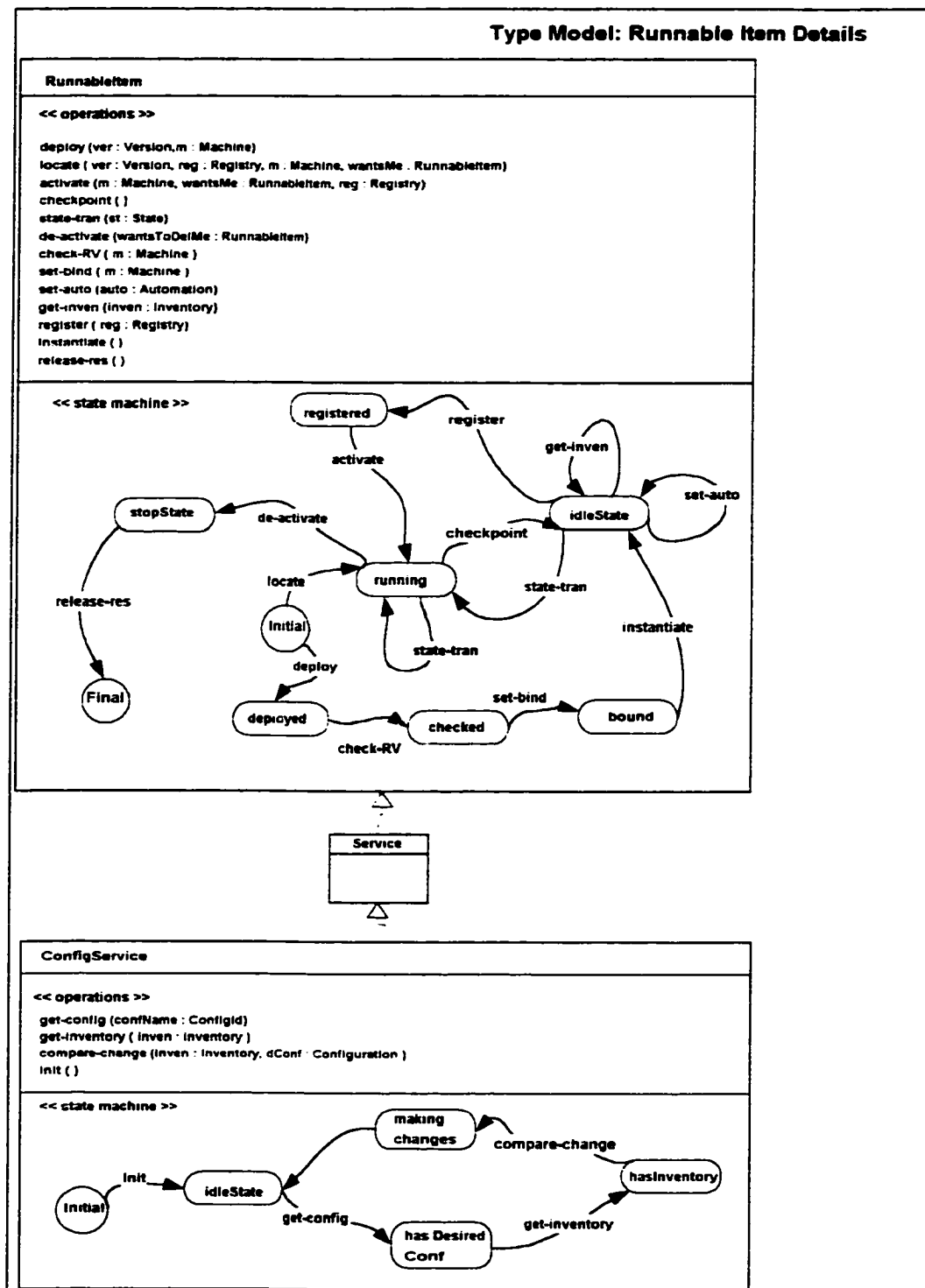


Figure 5-6. Operations and state machine details for *RunnableItem*. The **<<operations>>** section shows the calling signatures of the operations and the **<<state machine>>** section shows how each operation effects the state of the type. *ConfigService* is a specialization of *Service* and has additional operations that are needed to manage the dynamic configuration of the system.

[illegible]

VersionableCI types have a version, require resources, and have transition states. Two states are required to be in the set of transition states, a stop and an idle state. Other states in the transition set are a result of the actual functionality of the software (e.g. a fraud detection application). These other states are orthogonal to the issue of software run-time configuration management and vary according to the functionality of their associated software. They are not included in the software run-time configuration management *type* model. Configuration changes are made to software *types* when they are in the idle or stop state, provided *Automation* for these states is properly set. If *Automation* is not properly set, automatic configuration changes are not allowed. The *Automation* concept is defined as an attribute in this *type* model, so it must be given a data type. This model shows it as a Boolean value whose name is *automationOK*. In addition, the *RunnableItem* specialization of *VersionableCI* uses resources, always the same or a subset of the required resources.

VersionableCI types have a version, require resources, and have transition states. Two states are required to be in the set of transition states, a stop and an idle state. Other states in the transition set are a result of the actual functionality of the software (e.g. a fraud detection application). These other states are orthogonal to the issue of software run-time configuration management and vary according to the functionality of their associated software. They are not included in the software run-time configuration management *type* model. Configuration changes are made to software *types* when they are in the idle or stop state, provided *Automation* for these states is properly set. If *Automation* is not properly set, automatic configuration changes are not allowed. The *Automation* concept is defined as an attribute in this *type* model, so it must be given a data type. This model shows it as a Boolean value whose name is *automationOK*. In addition, the *RunnableItem* specialization of *VersionableCI* uses resources, always the same or a subset of the required resources.

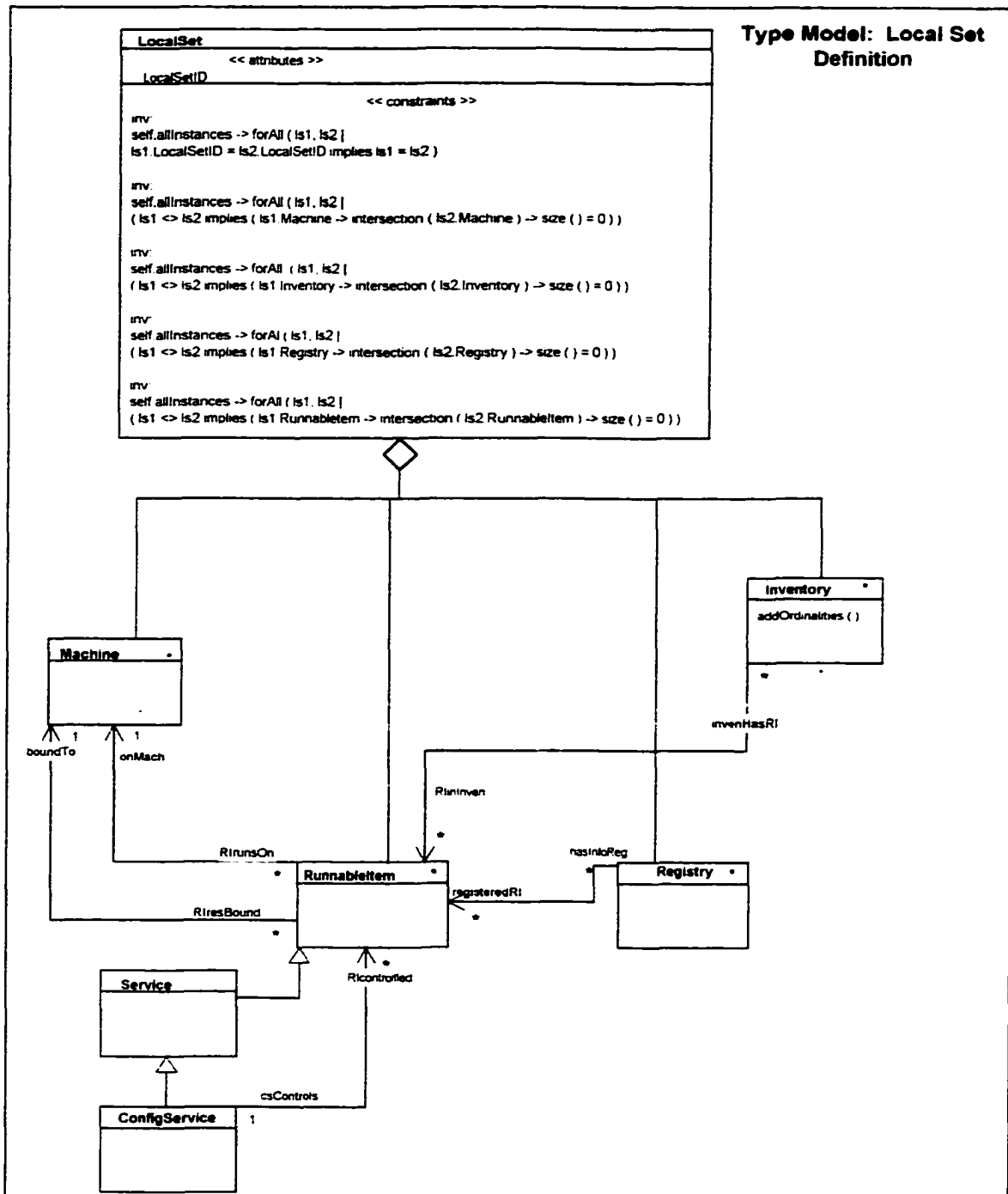


Figure 5-8. The type model of a *LocalSet*. A *LocalSet* partitions the overall system to allow independent run-time configuration management of the partitions. At any given time, machines and the runnable items executing on them can only be in one *LocalSet*. Inventories and registries are only associated with one *LocalSet*. A *LocalSet* is defined as a loose aggregation of runnable items, the physical machines on which they run, and associated registries and inventories. The notion of a local set identifier is included to ensure unique local sets.

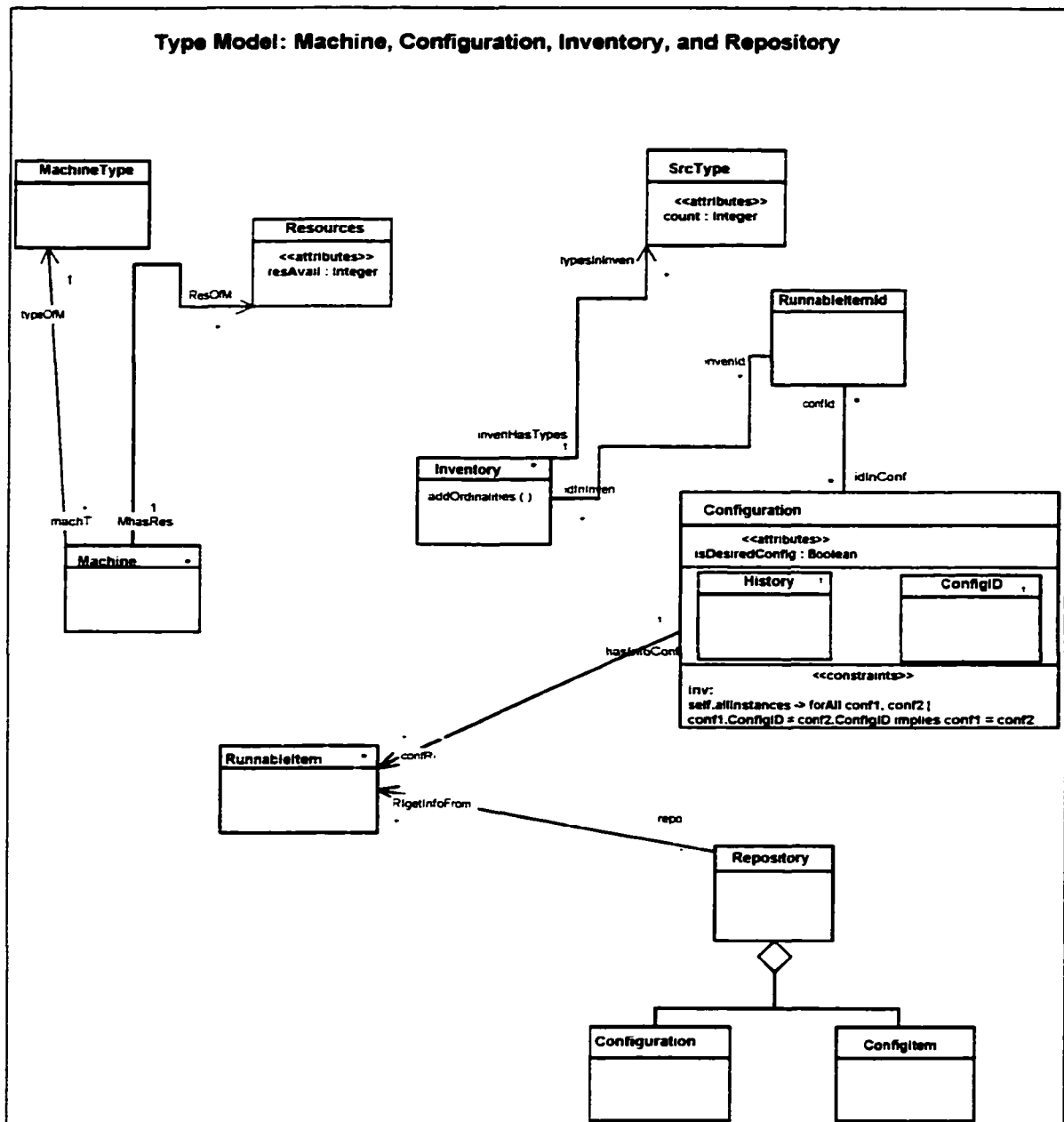


Figure 5-9. Relations among some of the concepts and types in a *LocalSet* (*Machine*, *RunnableItem*, and *Inventory*) and other configuration management-related concepts and types in the system. *RunnableItems* are found in *Repositories*, and are requested to be in the system by *Configurations*. *RunnableItems* are represented in *Inventories*, which reflect the ordinalities of various kinds of runnable items (*SrcType*; e.g. a trader service or a fraud application). *Machines* have types (*MachineType*; e.g. byte swapping or little endian) as well as *Resources*.

Yet another approach is to use a state machine to define the pre- and post-conditions of the operations in OCL. This is the method used in Figure 5-6 and Appendix 5A. The post-conditions of simple operations, such as *deploy*, include a

resulting state. This state can be used as part of the pre-condition for more complicated operations (e.g. *activate*). A potential problem with this approach is large numbers of pre-condition states due to non-determinism in the state machine. This is not a problem for the ATM network monitoring system, it can be a problem for other loosely coupled distributed systems whose configuration management behavior cannot be as easily sequenced.

7.3 Type Model Evaluation

The *type* model technique is evaluated using the technique evaluation criteria discussed in Chapter 3:

- 1) Validation of results using system architect or user feedback: There is no feedback available to validate the results of *type* models.
- 2) Ability to specify all necessary requirements: *Type* models that include OCL expressions are sufficient to specify software run-time configuration management requirements of the monitoring system, except for temporal constraints.
- 3) Applicability to specification: *Type* models are applicable to specification.
- 4) Ability to directly support validation/verification: OCL constraints and operation definitions can provide both validation that behaviors are present in a specification and verification that the specification is internally consistent.
- 5) Ability to fit into industry:
 - 1) Automation potential: none

- 2) Existing tools: primarily drawing tools; no generally available OCL tools
- 3) Skill set needed by practitioners: Skills in UML modeling and especially in OCL are required.
- 4) Flexibility to fit into existing development processes: This technique can easily fit into specification and design activities provided the kinds of models and the OCL language constructs are restricted to those applicable to specification. Other models and language constructs of UML and OCL can be used during design activities.
- 5) Ability to use partially or incrementally: The technique can be used in an incremental and partial fashion by including more or less detail.

8. USING ALLOY TO COMPLEMENT THE UML SPECIFICATION

Alloy is a modeling language. It is supported by a tool designed to allow users to test assertions, invariants, and operations in an Alloy model. An Alloy model consists of several parts. The first part is a list of sets that are fundamental to the model. The second part is state, which consists of sets derived from the domain sets and relations between sets. The third part of an Alloy model contains definitions of relations included in the model, as well as invariants and assertions regarding sets and their relationships. Finally, the last part of an Alloy model contains operations that are performed on sets. This structure is shown in Figure 5-10, which contains a portion of the Alloy model of the ATM network monitoring system.

It is possible to partially define a system using Alloy and then test the invariants, assertions, and operations that have been defined using the associated Alloy tool. The

run-time configuration model can therefore be defined and “tested” independently of the rest of the functionality of the ATM monitoring system.²⁴

8.1 ATM monitoring system Alloy model development

A model of the configuration management aspects of the ATM monitoring system has been developed using Alloy. The model is shown in Chapter 6.

8.2 Alloy model testing

An Alloy model can be tested using its associated tool. The model must be compiled prior to running any analyses. Compilation checks for syntactic correctness and creates an expression that can be solved during analysis. (See [36] for further details on the results of compiling a model.) The Alloy model presented in Chapter 6 compiles properly. Further testing of this model is discussed in Chapter 7.

²⁴ The testing tool creates a logical formula that represents all the model elements. Testing an invariant or state corresponds to searching for an instance of the model that satisfies the formula within a given scope (the number of elements of each primitive type). (The default scope is three; this can be increased with the penalty of extra processing time.) Testing does not prove that a model is correct, but can show whether there are any instances where it is consistent or counter examples where assertions (theorems) are not valid. Instances demonstrate consistency for invariants and operations. Jackson ([36]) argues that if an instance exists, it usually exists in a small scope. So an assertion is probably true if no counterexample is found using a small scope. Similarly, an invariant is probably inconsistent if no instance can be found in a small scope. The Alloy tool is further discussed in Chapter 8.

model ATM-monitoring-CM-example

```

{
domain { Machine, LocalSetId, Platform, Compiler, MachineType, ConfigItem, OpAttempted, ReturnValue, Version }

state
{
    partition SourceCode, ObjectCode, Object, Service, Application : ConfigItem
    NotOCCode : ConfigItem
    RunnableItem : ConfigItem
    ConfigService : Service
    partition deployed, located, bound : OpAttempted
    OK : ReturnValue
    Error : ReturnValue
    controlledBy (~controls) : RunnableItem -> ConfigService !
    on (~on) : RunnableItem -> Machine !
    LocalSet (~isRl) : RunnableItem -> LocalSetId !
    LocalIndex (~mach) : Machine -> LocalSetId !
    dependsOn : ConfigItem -> ConfigItem
    createdFrom : ConfigItem -> ConfigItem
    platformC : Platform -> Compiler !
    platformMT : Platform -> MachineType !
    Feedback : OpAttempted -> ReturnValue
    hasMType : Machine -> MachineType !
    hasVersion : NotOCCode -> Version !
    developedToRunOn : ObjectCode -> Platform
}

def NotOCCode { NotOCCode = SourceCode + Object + Service + Application }
def RunnableItem { RunnableItem = Object + Service + Application }
def LocalSet { all n | n.LocalSet = n.on.LocalIndex }
def controlledBy { all n : RunnableItem | one cs : ConfigService | n.controlledBy = cs -> n.on.LocalIndex =
cs.on.LocalIndex }

def dependsOn { all ci | ( no ci1 : ConfigItem | ci.dependsOn = ci1 )
|| ( ( ci in SourceCode -> some src : SourceCode | ci.dependsOn = src && ! (src.dependsOn = ci) ) && ( ci in
ObjectCode -> some oc : ObjectCode | ci.dependsOn = oc && ! (oc.dependsOn = ci) ) && ( ci in Object -> some obj :
Object | ci.dependsOn = obj && ! (obj.dependsOn = ci) ) && ( ( ci in Service -> some ser : Service | ci.dependsOn = ser
&& ! (ser.dependsOn = ci) ) || ( ci in Service -> some obj : Object | ci.dependsOn = obj && ! (obj.dependsOn = ci) ) )
&& ( ( ci in Application -> some app : Application | ci.dependsOn = app && ! (app.dependsOn = ci) ) || ( ci in
Application -> some ser : Service | ci.dependsOn = ser && ! (ser.dependsOn = ci) ) || ( ci in Application -> some obj :
Object | ci.dependsOn = obj && ! (obj.dependsOn = ci) ) ) )
}

def createdFrom { all ci | ( ci in SourceCode -> ci.createdFrom = ci ) && ( ci in ObjectCode -> one src : SourceCode |
ci.createdFrom = src ) && ( ci in Object -> some oc : ObjectCode | ci.createdFrom = oc ) && ( ci in Service -> some obj :
Object | ci.createdFrom = obj ) && ( ci in Application -> some obj : Object | ci.createdFrom = obj ) }

inv oneCSControls { all cs1, cs2 : ConfigService | no n : RunnableItem | n in cs1.controls & cs2.controls && ! ( cs1 = cs2 )
}
inv oneCSControlsAll { all lsid : LocalSetId | one cs : ConfigService | cs in lsid.isRl && lsid.isRl = cs.controls }
inv sets { some ci1, ci2, ci3, ci4 : ConfigItem | ci1 in Service && ci2 in Object && ci3 in ObjectCode && ci4 in
SourceCode }

op deploy ( n : RunnableItem !, ver : Version !, m : Machine ! ) {
    m.hasMType in n.*createdFrom.developedToRunOn.platformMT
    n.hasVersion = ver
    n.on' = m
    deployed.Feedback' in OK
}
}

```

Figure 5-10 A portion of the Alloy model of run-time software configuration management for the ATM network monitoring system. Only a subset of the full model is shown to highlight some of the structural components of Alloy. The *domain* and *state* sections are shown, along with definition (*def*) and invariant (*inv*) statements. An operation (*op*) is also shown. Notation used in the *state* section includes *partition* (subsequent names are subsets of the name after the colon, do not share any members, and are its only subsets), colon (the first name is a subset of the second name), and exclamation mark (singleton set). Model components shown include *platformC* and *platformMT* relations (required to define the *Platform* tuple), *dependsOn* relation (initially defined as existing between *ConfigItem*s and then explicitly defined in a *def* statement), and invariants (*oneCSControls* and *oneCSControlsAll* which are used to specify independent configuration of local sets). Primed values of state components in operation definitions indicate the post state of variables that are primed. The notation **createdFrom* indicates one or more uses of the *createdFrom* relation, in order to reach the associated *ObjectCode* set (which has the relation *developedToRunOn*). This Alloy model defines a *LocalSet* based upon a partitioning of the machines on which *RunnableItems* execute. The full ATM network monitoring system software run-time configuration management Alloy model is presented in Chapter 6.

8.3 Observations about Alloy

Some observations about using Alloy to extend the *type* model include:

- *Type* model components must be duplicated in Alloy before the model can be extended. While this is time consuming, it does seem to be possible to follow a method to achieve it. An automated tool could probably be developed to aid in the initial translation from UML to Alloy.
- Portions of an Alloy model associated with specific invariants or operations can be extracted and inserted into designs or source code since Alloy is a formal language. Automated regression testing can use these formal statements. Including specification statements in designs and source code provides closer ties between specification, design, development, and maintenance. It also enables continued association between a specification and its associated implementation.
- Alloy allows relations to be declared, then to be precisely defined using the *def* notation. Figure 5-10 shows the *dependsOn* relation is initially defined as a relation between *ConfigItems*. The *def* (definition) notation is then used to precisely state the nature of this relation between applications, services, objects, object code and source code.
- One problem with using version 1.0 of Alloy is that it requires one relation for each part of a tuple. To model the *Platform* aggregation, two relations are added to the Alloy model. The relation *platformC* is included in the Alloy model to map between a *Platform* and a *Compiler*, and the relation *platformMT* is added to map between a *Platform* and the *Machine* type it represents. If Alloy supported tuples directly, these multiple relations could be replaced by a single relation.

- System-level constraints on behavior can be included in Alloy. For example, the notion that local sets can work independently, moving toward a new desired configuration can be included in the model as a set of invariants associated with local sets and configuration services.
- The temporal constraints of configuration management operations cannot be specified using Alloy.
- Alloy does not include the notion of sequencing. Thus, it is difficult to specify that particular operations need to be sequenced. For example, one version of an application may need to be stopped and then another version started, perhaps because the resources used by the first version must be released before the second version is started. It is not possible to express this kind of sequence specification in Alloy. Instead additional operations must be created that compose the post conditions of operations that need to be sequenced.
- The Alloy specification presented in Chapter 6 suffers from a structural problem; definitions (i.e. *def* statements) and invariants (i.e. *inv* statements) of state relations and subsets are included in the model after the state declaration. This makes it difficult to find constraints associated with the state. In a new version of Alloy [35], all the information about a set is kept together, thus packaging relevant information.

8.4 Alloy Evaluation

The Alloy technique is evaluated using the technique evaluation criteria discussed in Chapter 3:

- 1) Validation of results using system architect or user feedback: There is no feedback available to validate the results of Alloy models.
- 2) Ability to specify all necessary requirements: Alloy models are sufficient to specify software run-time configuration management requirements of the monitoring system, except for temporal constraints.
- 3) Applicability to specification: Alloy models are applicable to specification.
- 4) Ability to directly support validation/verification: Alloy assertions, invariants, and operation definitions help provide both validation that behaviors are present in a specification and verification of internal consistency. The Alloy tool provides these services in an automated fashion.
- 5) Ability to fit into industry:
 - 1) Automation potential: There is a potential for automated assistance in developing Alloy models from UML diagrams.
 - 2) Existing tools: There is an associated verification tool.
 - 3) Skill set needed by practitioners: Some skills in first order predicate logic is required.
 - 4) Flexibility to fit into existing development processes: This technique can easily fit into specification activities. It is only applicable to specification, so other techniques must be used to create system designs.

- 5) Ability to use partially or incrementally: The technique can be used in an incremental and partial fashion by including more or less detail. It can also be verified using the accompanying tool in an incremental fashion.

9. TEMPORAL ISSUES

There are some aspects of time that need to be specified for the ATM network monitoring system. These are not stringent since the system is so loosely coupled, but they do need to be included in the specification. A configuration change can take place on a runnable item only if *ReadyTime* is acceptable. The change must also take place within an acceptable amount of time. In addition, a local set must not attempt to move to a new desired configuration too soon.

This leads to three time-related specifications:

- A configuration service must not try to move toward a new desired configuration until a specified amount of time has passed. A configuration service might try to continually move to new configurations before previous changes are complete in the absence of this requirement. This leads to a chaotic system.
- Configuration changes occur on specific elements of the system when *ReadyTime* is acceptable. This can be handled by a specification that configuration changes only occur when a runnable item is in its STOP or IDLE states.
- Configuration changes take place within an acceptable amount of time. This can be specified as a time bound on changes made to runnable items.

Neither Alloy nor OCL embody the notion of time directly, so even these simple temporal constraints cannot be included in a specification that uses these notations.²⁵

One potential solution to this problem is to create a *concept* that represents the minimum amount of time that must pass between attempting to move toward a new desired configuration and another to represent the maximum amount of time allowed for an individual configuration change. Constraints/invariants can then be developed that use these concepts. This approach is not satisfactory because the notion of time is still not explicitly present, so the specification does not clearly convey temporal requirements.

A better approach is to use additional time-related concepts in the specification, such as temporal logic.

9.1 Temporal Logic

Temporal logic can specify time constraints. Many temporal logic systems are based on state transitions (e.g. [53]). Concurrency is represented by the interleaved execution of atomic instructions in concurrent processes. Thus concurrency is reduced to non-deterministic transition systems. This type of model does not adequately reflect the operation of the ATM monitoring system; since the system is so large, actions of its various parts truly are concurrent. In addition, again since it is so large, the local clocks on computers around the system will probably not be completely synchronized, further undermining the model of interleaved execution to represent concurrency. In addition to a model mismatch, transition-based temporal logics often add additional temporal

²⁵ UML does include sequence models. Sequence diagrams can include time bounds, however this capability is not integrated into OCL. Sequence diagrams are used for instances of classes; they are not used as specifications unless they reference roles. The minimum time between attempts to move to a new desired configuration cannot be specified with sequence diagrams since it needs to be expressed as a time from the start of the last move to a new configuration to the start of the next move to a new configuration.

operations that require higher order logic. These operations would likely be difficult to add to existing UML/OCL and Alloy validation and verification tools. This is due to mismatches in the temporal and tool concepts as well as the need for higher order logic.

RTL [38] is a temporal logic system that is based on first order logic. RTL does not use the notion of transitions or states. Instead, time passes between sets of events, and events are regarded as taking no time. The fact that events might occur simultaneously is irrelevant to RTL, whereas in transition-based temporal logic concurrent actions must be defined by a transition that places total ordering on actions of the transitions. The event based foundation of RTL thus more closely models actual concurrency in the ATM network monitoring system. In addition, since RTL is based on first order logic, it may be easier to add such a temporal logic to existing tools. (This is an area that requires further research, and is not part of the research described in this dissertation.)

Time is a basic element of the RTL logic system, and always continues forward. Other basic concepts are events and actions. Actions require bounded, non-zero system resources and have durations. Events have no system resource requirements and no duration, but do occur at a particular point in time. Events are used as markers of time. There are two types of events; start and stop events that characterize an action (defining its duration²⁶) and transition events that represent a change in a state variable. An occurrence relation is used to define the time of an event. This occurrence relation is defined as $\theta(e, i, t)$. It denotes the i^{th} occurrence of the event e that happens at time t .

²⁶ See [79] for a discussion of temporal logic and its application to data communication protocols.

RTL predicates are formed using the occurrence relation, equality/inequality predicates, universal and existential quantifiers, and first-order logical connectives.²⁷

Simple occurrence relations are sufficient to specify the timing needs of the ATM monitoring system. Let τ_{cs} be the minimum time required before beginning to move to a new *desired configuration*. (For simplicity assume that there is one value of τ_{cs} for all configuration services in the monitoring system.) Similarly, τ_{op} is defined as the maximum allowable time for a configuration operation. (There could be various values for τ_{op} , depending on the type of operation, but for simplicity assume that all operations need to complete within the same amount of time.) The timing requirements of the configuration management specification are:

- A configuration service cannot begin the move toward a new desired configuration until τ_{cs} time after the beginning of the last move toward a new desired configuration.
- At most τ_{op} time is allowed for each configuration operation to complete.

These specifications imply that the values of τ_{op} and τ_{cs} must be chosen so that they are reasonable. For example, if a configuration service needs to activate a runnable item, the *activate* operation locates all dependencies of the runnable item, or activates the dependencies if they cannot be located. *Activate* must then deploy, check resources, bind resources, instantiate, and register the runnable item. Clearly the timeout value for the overall *activate* operation must be large enough to accommodate all these actions. (Note that some of the actions may actually execute concurrently, which may or may not

²⁷ Equality/inequality predicates are =, <, >, ≤, and ≥. First-order logical connectives are ¬, ∧, ∨, and →.

decrease the time needed for *activate*. Note also that the time needed for *activate* to complete is related to the number of dependencies of the runnable item.) Likewise, the minimum time between moves to new desired configurations must be large enough that the configuration service is able to move to the current desired configuration before it attempts to move to a new desired configuration.

The RTL specification for the minimum time between moves to desired configurations is:

$$\forall i \forall t \theta(\uparrow \text{compare-change}, i, t) \rightarrow \exists t' \mid (\theta(\uparrow \text{compare-change}, i+1, t')) \wedge (t + \tau_{cs} \leq t')$$

In this occurrence relation, $\uparrow \text{compare-change}$ is the start event of a *ConfigService* *compare-change* operation, i is the i^{th} occurrence of the start of the *compare-change* operation, and $i+1$ is the $i^{\text{th}} + 1$ occurrence of the start of the *compare-change* operation. The RTL relation states that for all values of i and t , the start of the $i^{\text{th}} + 1$ occurrence of the *compare-change* operation cannot occur until at least $t + \tau_{cs}$ time after the start of the i^{th} occurrence of *compare-change*.

The RTL specification of the maximum time allowed for an operation must be added to the specification. For example, the *ConfigService* *get-inventory* operation only calls one operation, *get-inven*, on each runnable item in the local set. The *get-inventory* operation must be augmented with the following RTL specification:

$$\forall i \forall t \theta(\uparrow \text{get-inven}, i, t) \rightarrow \exists t' \mid (\theta(\downarrow \text{get-inven}, i, t')) \wedge (t < t') \wedge (t' < t + \tau_{op})$$

The notation in this occurrence relation is the same as in the above specification, except that $\downarrow \text{get-inven}$ indicates the end of the *get-inven* operation. The RTL relation states that for all values of i and t , the start of the i^{th} occurrence of the *get-inven* operation

implies that the *get-inven* operation ends at a later time (t') and that later time is no more than τ_{OP} time after the operation begins.

The *compare-change* operation needs similar specifications for the operations it calls, namely *activate*, *de-activate*, *state-tran*, and *set-auto*. These specifications are:

$$\forall i \forall t \theta (\uparrow \text{activate}, i, t) \rightarrow \exists t' \mid (\theta (\downarrow \text{activate}, i, t')) \wedge (t < t') \wedge (t' < t + \tau_{op})$$

$$\forall i \forall t \theta (\uparrow \text{de-activate}, i, t) \rightarrow \exists t' \mid (\theta (\downarrow \text{de-activate}, i, t')) \wedge (t < t') \wedge (t' < t + \tau_{op})$$

$$\forall i \forall t \theta (\uparrow \text{state-tran}, i, t) \rightarrow \exists t' \mid (\theta (\downarrow \text{state-tran}, i, t')) \wedge (t < t') \wedge (t' < t + \tau_{op})$$

$$\forall i \forall t \theta (\uparrow \text{set-auto}, i, t) \rightarrow \exists t' \mid (\theta (\downarrow \text{set-auto}, i, t')) \wedge (t < t') \wedge (t' < t + \tau_{op})$$

In each of these RTL occurrence relations, the notation is the same as in the previous examples. The specifications are inserted into the correct positions in the Alloy model of the ATM monitoring system. Note that RTL statements can be added to either OCL descriptions of these operations or the Alloy descriptions of the operations.

Symbols are not easily included in an Alloy model since it consists solely of ASCII characters. Neither are occurrence relations since Alloy has no notion of time. Therefore, RTL statements are added to the model as Alloy notation comments. Thus, $\forall$ is replaced with *all* and $\exists$ with *one*, $\wedge$ with *&&*, t with *t1*, and t' with *t2* (the last two changes are made to avoid confusion between the t' of the temporal expression and the notion of primed variables to indicate an end state in Alloy). The θ symbol is replaced with *theta*, and $\uparrow$ and $\downarrow$ with functions *begin ()* and *end ()*, respectively. Finally, τ_{CS} and τ_{op} are replaced with *tCS* and *tOP*. A portion of the Alloy specification for the *compare-change* operation, extended with temporal predicates, is shown in Figure 5-11.

Compare-Change Operation Specification with Temporal Extensions

```

op compare-change ( inven : Inventory !, dConf : Configuration !, configS : ConfigService ! ) {
  // all i : Integer, t1 : Time | theta ( begin (compare-change), i, t1 )
  // → one t2 : Time / (theta ( begin (compare-change), i + 1, t2 )) && (t1 + tCS <= t2)

  all ri : ( ( dConf.hasConfigInfo - inven.hasInvenInfo ) & RunnableItems ) |
    activate ( ri, ri.on, configS, configS.csReg, configS )

  // && all i : Integer, t1 : Time | theta ( begin (activate), i, t1 )
  // → one t2 : Time / (theta (end (activate), i, t2 )) && (t1 + < t2) && (t2 < t1 + tOP)

  all ri : ( inven.hasInvenInfo - dConf.hasConfigInfo ) | de-activate ( ri, configS, configS )

  // && all i : Integer, t1 : Time | theta ( begin (de-activate), i, t1 )
  // → one t2 : Time / (theta (end (de-activate), i, t2 )) && (t1 + < t2) && (t2 < t1 + tOP)
  ...
  compared.FeedBack' = OK
}

```

Figure 5-11. A portion of the Alloy specification for the *compare-change* operation of a configuration service, including temporal extensions. Temporal extensions are comments since Alloy cannot process them, and are noted by bold italics.

Another temporal logic based on first order logic is TILCO-X [6]. This logic is quite flexible and is much more expressive than is needed for the ATM monitoring system's simple temporal needs. The TILCO-X operators are more complex than those of RTL, so if one or the other were to be added to a language such as Alloy, RTL would probably be easier to add. Figure 5-12 shows the two temporal constraints needed by the ATM monitoring system written in the TILCO-X notation.

ATM Monitoring System Temporal Constraints in TILCO-X

1. $(\uparrow \text{change-config}) ?^0 (- \text{change-config}, \tau_{cs})$

or $\neg \text{change-config} @ (\text{change-config}, \tau_{cs})$

2. $\text{operation} \rightarrow \neg \text{operation} ? [0, \tau_{op}]$

Figure 5-12. TILCO-X temporal constraints for the ATM network monitoring system. The first statement is that the change-config operation becomes true ($\uparrow$ symbol) at most 0 times in the interval excluding (open parenthesis symbol) the last time it became true (- symbol) and the time τ_{cs} . An alternative statement is that the change-config operation is false ($\neg$ symbol) across the entire interval (@ symbol) excluding (open parenthesis symbol) the time when it last became true and the time τ_{cs} . That is, a new change-config operation cannot be started by a ConfigService until at least τ_{cs} time after the last one started. The second statement is that an operation being true implies ($\rightarrow$ symbol) that it will become false ($\neg$ symbol) sometime (? symbol) during the interval including (open bracket symbol) the current time (0 symbol) and the time τ_{op} . That is, an operation must finish within τ_{op} time from when it started.

An alternative, transition-based, higher order temporal logic [53] defines the temporal operation *until*. The formula $p \mathcal{U} q$ is defined as: the formula p holds at least until the first occurrence of q . For the temporal constraint on the movement to a new desired configuration, p can be defined as "no change will begin to the configuration of the local set" and q can be defined as "the minimum time has passed". The specification then becomes "no change will begin to the configuration of the local set" until at least the first time the "the minimum time has passed". Note that this needs to be a recurring formula; once a change begins, the minimum time needs to pass again before another change is started.

The *eventually* operator specifies the maximum time an operation can take to complete: *eventually the operation finishes* (either properly, or because it times out). Specifying a time out value is not possible using the *eventually* operator.

These temporal specifications are not used in the ATM monitoring system specification because: 1) the concurrency model for transition-based temporal logic does not accurately model the ATM monitoring system, 2) the logic is based on a higher order, and 3) recurrent and time out based requirements cannot be specified.

9.2 Temporal Logic (RTL) Evaluation

RTL is evaluated using the technique evaluation criteria discussed in Chapter 3:

- 1) Validation of results using system architect or user feedback: There is no feedback available to validate the RTL results.
- 2) Ability to specify all necessary requirements: RTL is only useful for temporal specifications.
- 3) Applicability to specification: RTL is applicable to specification.
- 4) Ability to directly support validation/verification: RTL statements provide a basis for validating the existence of a behavior in a specification and consistency verification across related portions of a specification.
- 5) Ability to fit into industry:
 - 1) Automation potential: There is a potential for automation if RTL is integrated into other tools.
 - 2) Existing tools: none
 - 3) Skill set needed by practitioners: Some skills in first order predicate logic is required.
 - 4) Flexibility to fit into existing development processes: This technique can easily fit into specification activities.

- 5) Ability to use partially or incrementally: The technique can be used in an incremental and partial fashion.

10. SUMMARY

The results of each of the techniques described in this chapter are discussed in the following paragraphs. Recommendations for a specification notation are presented in the last section. Chapter 6 presents the initial specification of software run-time configuration management in the ATM network monitoring system using this notation.

10.1 Property refinement

Property refinement can be used to define incremental and automatic operation of software items involved in the run-time configuration management of the ATM network monitoring system. These properties can be completely refined using predicate logic, although they can also be left at a more abstract level, as was done in this specification.

This technique fits into the specification process, when the goal is to further refine system requirements. The results are useful even if property refinement is not used to completely refine a property to predicate logic statements. Using property refinement at more abstract levels does not require extra skills on the part of architects, although completely refining properties does require familiarity with predicate logic. Generally available tools do not exist to support refinement using predicate logic, so the technique is most useful in industry when applied at the more abstract level.

Property refinement is included in the recommended specification method.

10.2 Fuzzy sets

Relative prioritization of requirements can be determined using fuzzy sets. This information is already available through problem domain knowledge, so the fuzzy set technique is of limited value during specification. This technique does not provide any new insight into relations between requirements since the exact mathematical relations between requirements are generally unknown during specification. Therefore fuzzy sets are not included as part of the recommended specification method.

10.3 Soft goals

Static and dynamic structuring needs can be refined into specific operations using soft goals. This technique also shows the supporting relationships between goals. The soft goals technique was developed to help provide a methodology to make design choices. It is not useful in specification as a choice mechanism, but is very useful as a refinement mechanism for operations. Although no tools exist for the technique, nor is automation likely, it still provides a useful refinement methodology. Soft goals are included in the recommended specification method.

10.4 Set Modeling

Set modeling yields abstract understanding of the important concepts and relations in a specification. It may be a useful first step in building *type* models, depending on familiarity with the problem. If architects are extremely familiar with all the concepts in the problem, this technique is of little value since it provides no new insights. Therefore, set modeling is not included as part of the recommended specification method, except as part of other techniques (i.e. Alloy modeling).

10.5 UML/OCL

The *type* model contains graphical information that must be supplemented with OCL documentation. The use of UML and OCL requires that an architect restrict the full set of models and language constructs to those most applicable to specification. For example, UML *types* are used in the ATM monitoring run-time configuration management *type* model, and language constructs such as *for-All* (instead of *iterate*) are used in OCL expressions. Specification may easily slip into design if models and language constructs are not restricted. Restrictions must be understood by all participants in the specification process, as well as by specification users.

OCL is a general language, and it can be difficult to choose the correct restrictions for specification. In addition, the language can be difficult to learn since it does have some similarity with first order predicate logic (e.g. *implies* and *for-All*) as well as with general programming languages (e.g. *if-then-else*). OCL does not have automated verification tools. If an entire model is written using the USE tool, some verification is available. However, USE does not currently support the full OCL language, and it was infeasible to completely specify run-time configuration management of the ATM monitoring system in USE. *Type* models without OCL expressions are useful to show graphical structure and associations within systems, but these models are incomplete without OCL constraints and operation descriptions. UML/OCL also needs temporal logic extensions to completely specify ATM network monitoring software run-time configuration management requirements.

UML is included as part of the recommended specification method. OCL is not included in the recommended specification method.

10.6 Alloy

An Alloy model is useful during specification. It can serve as a vehicle for completing *type* models without OCL expressions, and the accompanying tool provides some automated support for verification purposes. Since Alloy only deals with sets there is no need to constrain the language to specification-only constructs. (On the down side, this also means Alloy is not suited to design purposes, when it is important to state *how* the specification will be met.) Temporal logic needs to be added to Alloy to completely specify the ATM network monitoring system software run-time configuration management requirements, and an automated method to convert graphical *type* models into an Alloy model needs to be developed as well.

Alloy is included in the recommended specification method.

10.7 Temporal Logic

Event-driven temporal logic is needed to specify the simple timing constraints of the ATM monitoring system. RTL is sufficient for this purpose. TILCO-X is much more than is needed, may be harder to add to existing tools. A temporal logic needs to be added as an extension to specification languages such as Alloy and UML/OCL. The temporal logics studied in this research varied greatly in their understandability and ability to express the temporal needs of the ATM monitoring system. RTL requires the least extra skills on the part of users (especially in terms of its notation) while still being able to express the temporal requirements of the ATM monitoring system. It is also based on first order logic, unlike some of the other temporal logic systems. No absolute statements can be made about the correctness of expressions written in RTL since it is

undecidable. Further research is needed to determine if RTL can be added to languages such as Alloy or UML/OCL, and their associated tools.

RTL is included as part of the recommended specification method.

11. PROPOSED SPECIFICATION METHOD AND NOTATION

11.1 A Recommended Specification Method

The specification of software run-time configuration management requirements of the ATM network monitoring system can be developed using a generalized specification method as follows:²⁸

1. Identify the system focus and related run-time configuration management issues using system knowledge and key configuration management requirements. Use this step to identify potential items that need to be included in subsequent specification steps as well as in specification validation and verification.
2. Use property refinement to obtain more precise function-related statements that embody vague system-wide run-time configuration management behavioral constraints.
3. Use soft goals to identify and further refine run-time configuration management functionality and to identify links (e.g. satisficing and argumentation goals) between requirements.
4. Create a *type* model without OCL expressions, using the information gained in the previous steps. In conjunction with this step,

²⁸ See Figure 1.5 in Chapter 1 for an overall view of the specification portion of the CMSpecV&V method.

5. Create an Alloy model to complement the *type* model and extend it using information from previous steps. (The process of creating an Alloy model can suggest refinement changes to the *types* models and visa-versa.) Extensions include constraints and operation definitions.
6. Add temporal constraints to the Alloy model using RTL-style comments if they are needed.

11.2 Recommended Specification Notation

A *type* model (using concepts, UML *types*, and relations) in conjunction with an Alloy model that includes temporal logic extensions present a complete specification of the software run-time configuration management requirements of the ATM network monitoring system.

The use of Alloy over OCL for operation specification allows the specification to remain abstract in the sense that "what" needs to be done is specified since sets are used, but "how" it is to be implemented is still up to the designer since no language-specific details are used. (To accomplish the same thing using OCL requires the language and models to be restricted to those appropriate for specification.) In addition, Alloy has an associated tool that can be used to help validate and verify a specification.

The Alloy model also includes invariants that can express incremental and automatic properties. A temporal extension such as RTL is needed for Alloy to be able to express the simple time-related requirements of the ATM/IP network monitoring system. In the meantime, RTL-style constraint comments can be added to the Alloy model.

The specification of the ATM monitoring system's software run-time configuration management using this notation is shown in Chapter 6.

**APPENDIX 5A –COMPLETE UML/OCL RUN-TIME CONFIGURATION
MANAGEMENT SPECIFICATION FOR THE ATM NETWORK MONITORING
SYSTEM**

A *type* model specifies run-time software configuration management in the ATM/IP network monitoring system. The model is accompanied by a completely textual OCL specification of additional constraints and operations. The entire model is included in this appendix.

1. UML TYPE MODELS

Type models are shown below.

Figures 5A-1 and 5A-2 show the fundamental software types (Figure 5A-1) and their operations (Figure 5A-2). Figure 5A-3 shows the interactions of the software types with other configuration management-related concepts and types (i.e. versions, resource, and software states).

Another fundamental concept in the ATM network monitoring system is the LocalSet. This concept allows incremental dynamic configuration across the system. It is shown in the type model. Figure 8A-4, shows the definition of the loose aggregation of a LocalSet. It consists of RunnableItems, the Machines they run on, and the associated Registries and Inventories. Figure 5A-5, shows the relations between portions of a LocalSet and other configuration management-related concepts (i.e. MachineType, Resources, SrcType, Configurations, and Repositories).

Type Model: Runnable Item Details

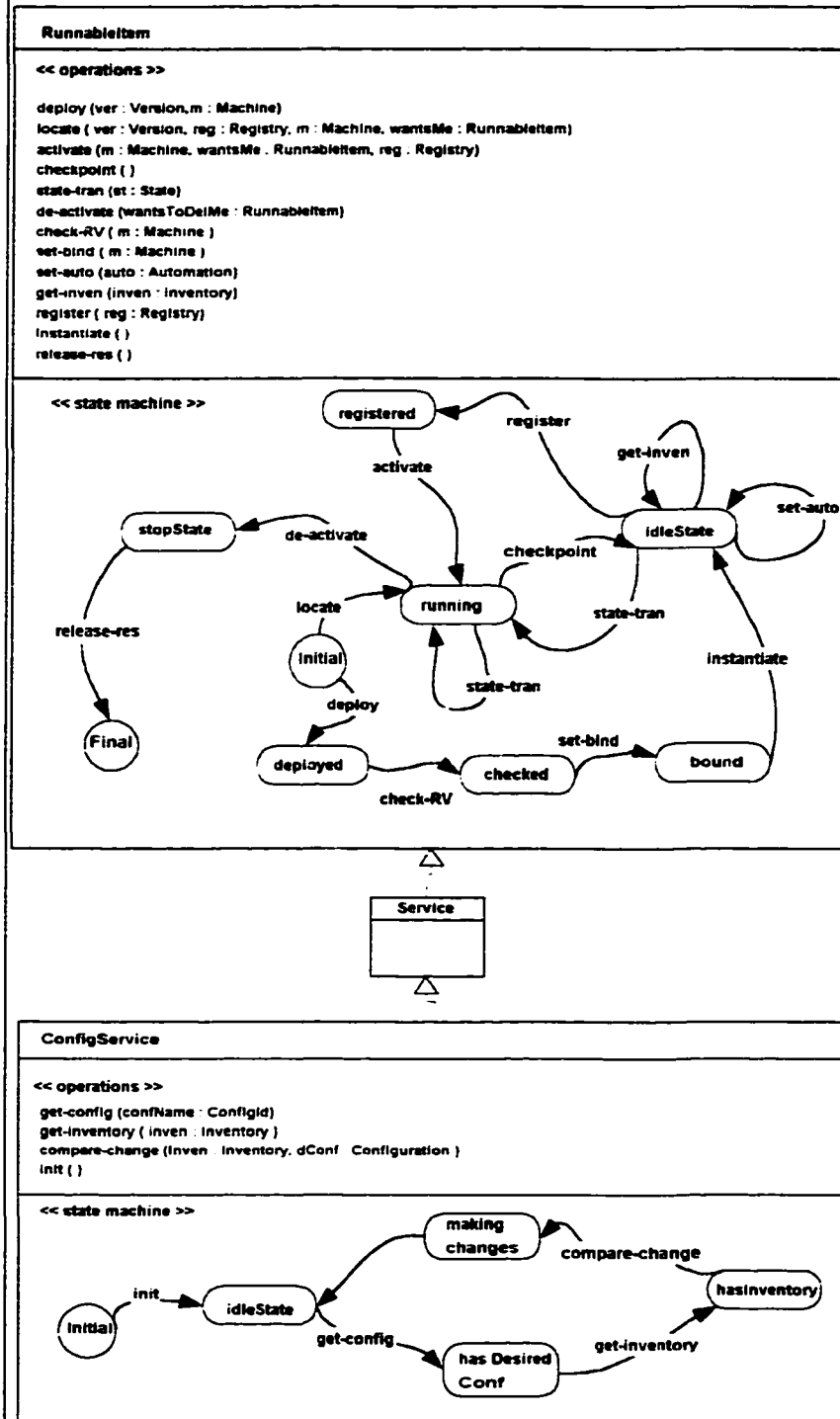


Figure 5A-2. Operations and state machine details for *RunnableItem*. The <<operations>> section shows the calling signatures of the operations and the <<state machine>> section shows how each operation effects the state of the type. *ConfigService* is a specialization of *Service* and has additional operations that are needed to manage the dynamic configuration of the system.

Type Model: RunnableItems, States, Versions, and Resources

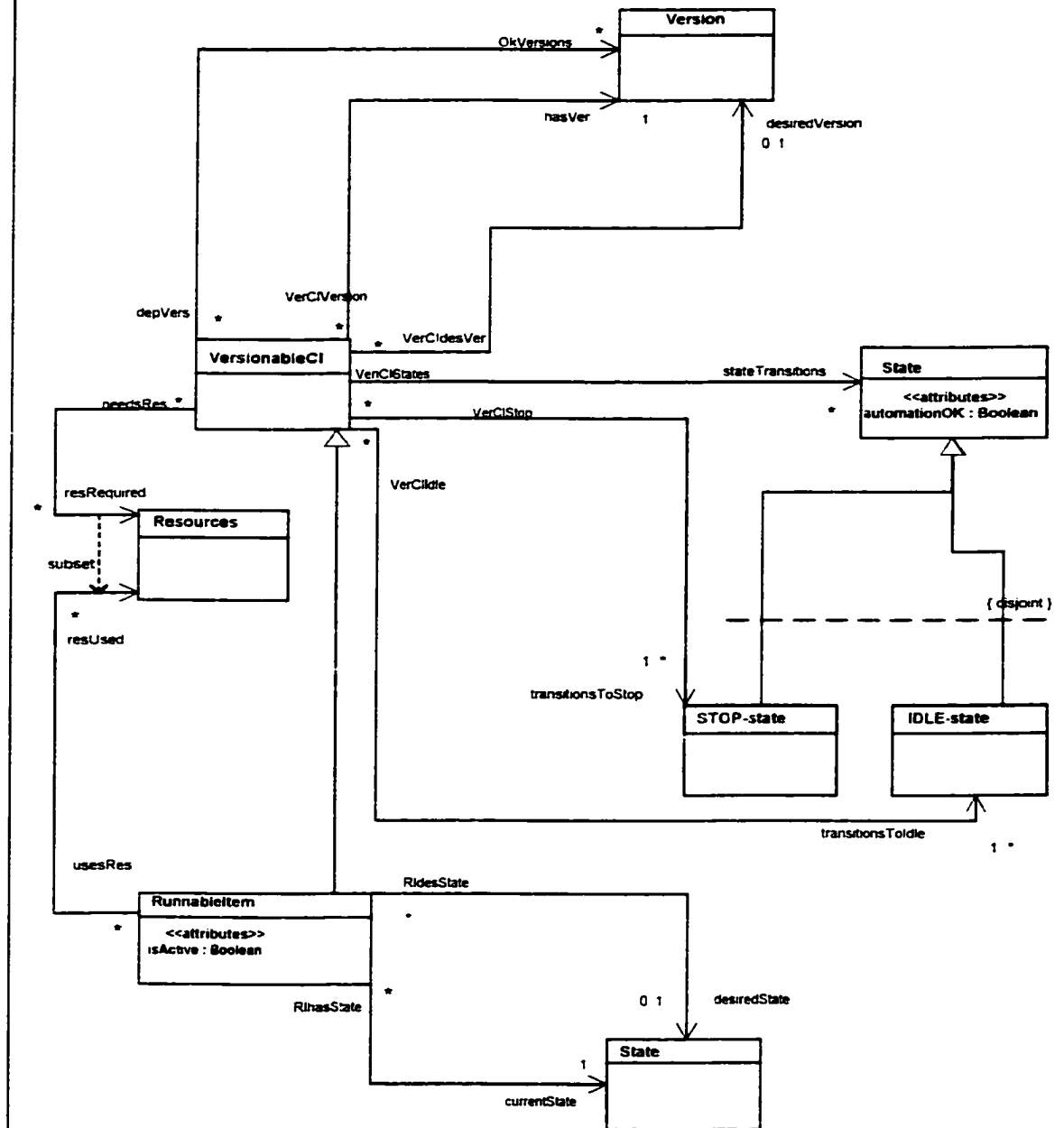


Figure 5.A-3. *VersionableCI* interactions with other configuration management-related concepts. *VersionableCI* types have a version, require resources, and have transition states in this system. Two states are required to be in the set of transition states, a stop and an idle state. Other states in the transition set are a result of the actual functionality of the software (e.g. a fraud detection application). These other states are orthogonal to the issue of software run-time configuration management and vary according to the functionality of their associated software. They are not included in the software run-time configuration management type model. Configuration changes are made to software types when they are in the idle or stop state, provided *Automation* for these states is properly set. If *Automation* is not properly set, automatic configuration changes are not allowed. The *Automation* concept is defined as an attribute in this type model, so it must be given a data type. This model shows it as a Boolean value whose name is *automationOK*. In addition, the *RunnableItem* specialization of *VersionableCI* uses resources, always the same or a subset of the required resources.

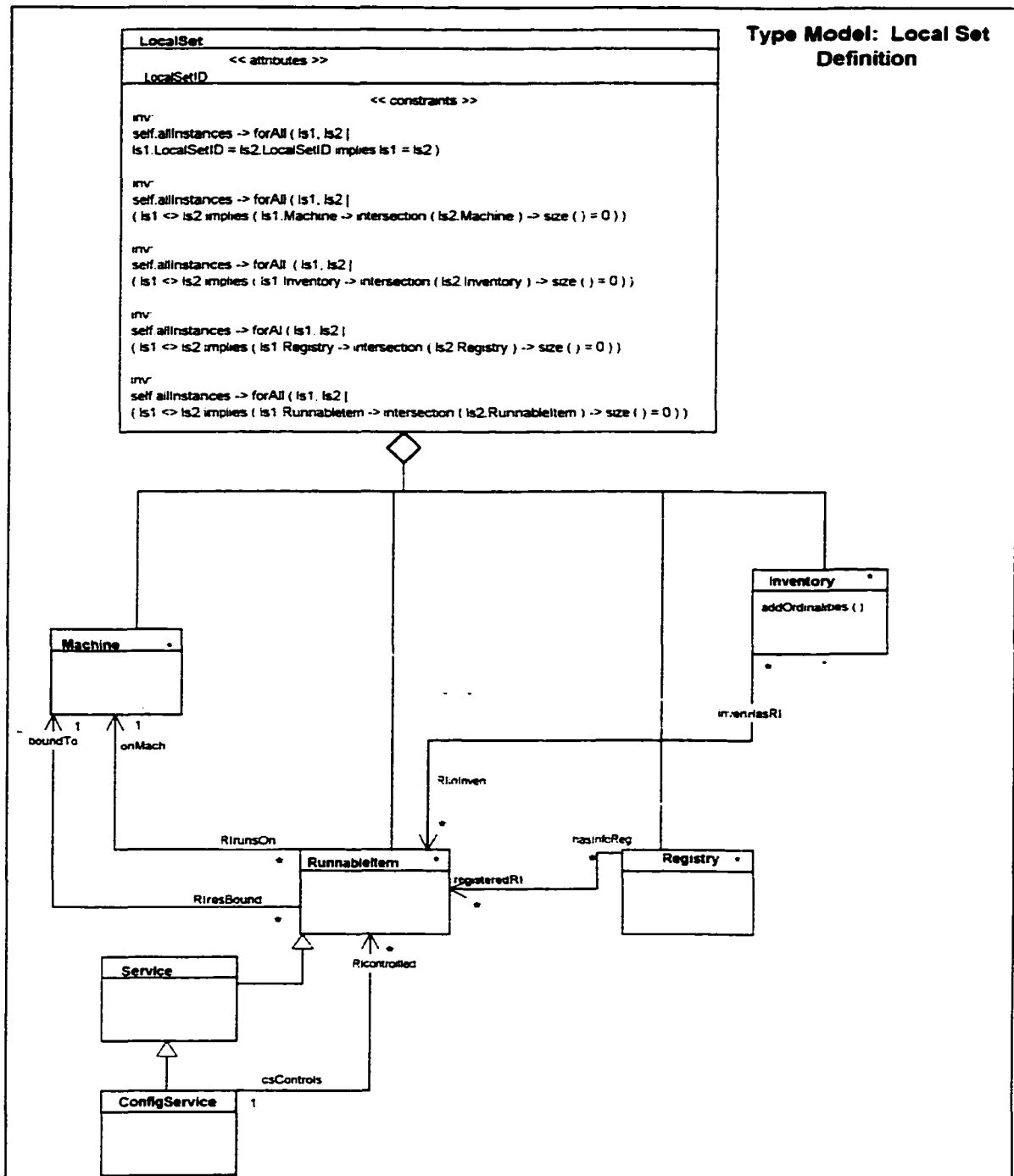


Figure 5.A-4. The *type* model of a *LocalSet*. A *LocalSet* partitions the overall system to allow independent run-time configuration management of the partitions. At any given time, machines and the runnable items executing on them can only be in one *LocalSet*. Inventories and registries are only associated with one *LocalSet*. A *LocalSet* is defined as a loose aggregation of runnable items, the physical machines on which they run and associated registries and inventories. The notion of a local set identifier is included to ensure unique local sets.

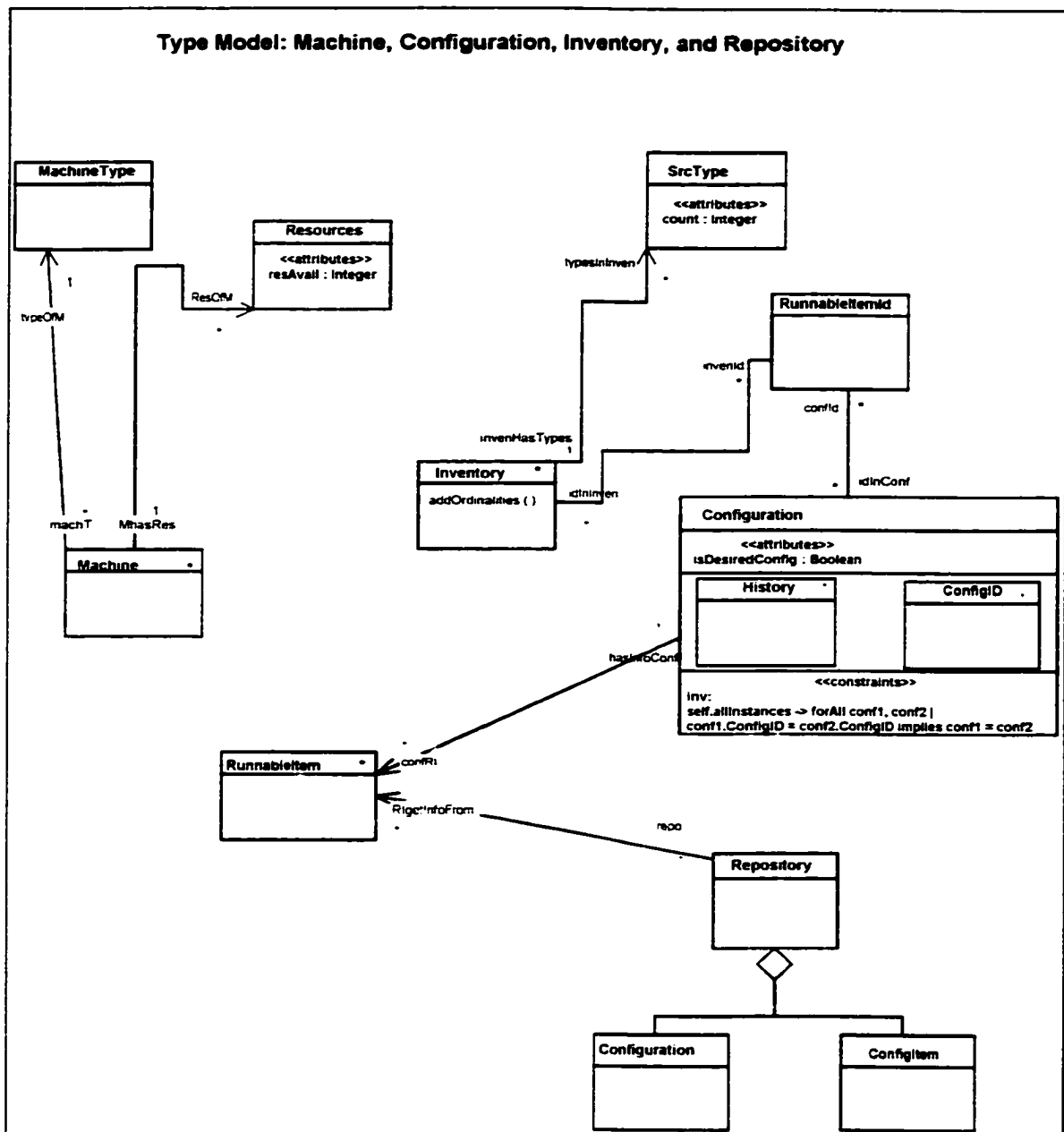


Figure 5.A-5. Relations among some of the concepts and types in a *LocalSet* (*Machine*, *RunnableItem*, and *Inventory*) and other configuration management-related concepts and types in the system. *RunnableItems* are found in *Repositories*, and are requested to be in the system by *Configurations*. *RunnableItems* are represented in *Inventories*, which reflect the ordinalities of various kinds of runnable items (*SrcType*, e.g. a trader service or a fraud application). *Machines* have types (*MachineType*, e.g. byte swapping or little endian) as well as *Resources*.

2. OCL DEFINITIONS

The OCL portion of the UML/OCL type models is shown below. Note that even though this is a requirements specification, using type models to retain abstraction, operation return value data types are required.

Pre- and post- conditions dealing with the state machine (as noted in the type model) use the notation `self.oclnState` to specify the state a type is in once the operation is complete. For example, `self.oclnState (deployed) = true` in the `deploy` operation definition of a `RunnableItem` indicates that a post-condition of the `deploy` operation is that the runnable item is left in its deployed state. These states are used in more complex operations that would ordinarily compose simpler functions, by using the previously defined post-conditions states as pre-conditions. OCL expressions are grouped under the heading of their applicable types. Type names are in a large font. Expressions are operation pre- and post- conditions or constraints/invariants. Operation pre- and post- conditions are listed before type invariants if both exist in a type. Operation and constraint/invariant names are bolded.

RunnableItem

The **deploy** operation must be overridden by each specialization: object, service, and application. Its signature is given in the generalization.

```
context RunnableItem :: deploy (ver : Version, m : Machine) : Boolean
pre: self.oclnState ( INITIAL ) = true
post: self.oclnState ( deployed ) = true and result = true
```

The **locate** operation. The post condition is that EITHER there is a registry at the beginning of the operation that already contains the runnable item in question, OR the runnable item is deployed, its resources and version checked, its resources bound, it is instantiated, and it is placed in the registry. In addition, its reference list is augmented to include the runnable item that wanted it. The result of this post condition is TRUE.

```
context RunnableItem :: locate ( ver : Version, reg : Registry, m : Machine,
                                wantsMe : RunnableItem ) : Boolean
pre: self.oclnState ( INITIAL ) = true
post : ( ( self.hasInfoReg @pre -> includes ( reg ) )
xor ( self.deploy ( ver, m ) and self.checkRV ( m ) and self.set-bind ( m ) and self.instantiate ( )
and ( reg -> including ( self ) )
and ( self.refersToMe -> including ( wantsMe )
and self.oclnState ( running ) = true
and result = true
```

The **activate** operation. The post condition states that all runnable items in the dependency list are located. In addition, the resources and version for this runnable item are checked, the resources are bound, the runnable item is instantiated and added to the registry. The result of the post condition is TRUE.

```
context RunnableItem :: activate ( m : Machine, wantsMe : RunnableItem, reg : Registry ) : Boolean
pre: self.oclnState ( registered ) = true
post : ( self.depRI -> iterate (
n : RunnableItem;
res : Boolean = true; |
n.locate ( self.hasVer, self.hasInfoReg, m, self ) ) )
and self.check-RV ( m ) and self.set-bind ( m ) and self.instantiate ( )
and ( reg -> including ( self ) )
and self.oclnState ( running ) = true
and result = true
```

The **checkpoint** operation simply places the runnable item into its IDLE state. The post condition result is TRUE.

```
context RunnableItem :: checkpoint ( ) : Boolean
pre: self.oclnState ( running ) = true
post : self.currentState = IDLE-state and self.oclnState ( idleState ) = true and result = true
```

The **state-tran** operation places the runnable item into the state passed in as an argument. The post condition result is TRUE.

```
context RunnableItem :: state-tran ( st : State ) : Boolean
pre: self.oclnState ( idleState ) = true or self.oclnState ( running ) = true
post : self.currentState = st and self.oclnState ( running ) = true and result = true
```

The **de-activate** operation. The post condition states that the reference list is "decremented" by the item that wants to deactivate this runnable item. Furthermore, if this list is empty that implies that the runnable item can move to its stop state, release all its resources, take itself out of the registries that have information about it, and attempt to deactivate all of the runnable items in its dependency set. The post condition result is TRUE.

```
context RunnableItem :: de-activate ( wantsToDelMe : RunnableItem ) : Boolean
pre: self.oclnState ( running ) = true
post : self.refersToMe -> excluding ( wantsToDelMe )
      and self.refersToMe -> isEmpty implies self.state-tran ( STOP-state ) and self.release-res ( )
      and self.hasInfoReg.allInstances -> forAll ( regis | regis.registeredRI -> excluding ( self ) )
      and self.depRI.allInstances -> forAll ( ri | ri.deactivate ( self ) )
      and self.oclnState ( stopState ) = true
      and result = true
```

The **check-RV** operation. The resource check states that the resources available on the machine at the beginning of the operation must be greater or equal to those required by the runnable item. The version check is that the set of versions that are OK for each item in the dependency set must include the version of this runnable item. The post condition is TRUE.

```
context RunnableItem :: check-RV ( m : Machine ) : Boolean
pre: self.oclnState ( deployed ) = true
post : m.ResOfM.resAvail >= self.resRequired and self.depRI -> forAll ( ri |
      ri.OkVersions -> includes ( self.hasVersion ) ) and self.oclnState ( checked ) = true and result = true
```

The **set-bind** operation decrements the available resources of the machine passed in as an argument by the resources required by the runnable item and sets the runnable item boundTo relation to the machine. The resulting resources on the machine must be greater than or equal to 0. The result of the post condition is TRUE.

```
context RunnableItem :: set-bind ( m : Machine ) : Boolean
pre: self.oclnState ( checked ) = true
post : m.ResOfM.resAvail = m.ResOfM.resAvail - self.resRequired and self.boundTo = m
      and m.ResOfM.resAvail >= 0 and self.oclnState ( bound ) = true and result = true
```

The **set-auto** operation. This iterates through all the states of the runnable item and sets each state's automationOK attribute to the Boolean passed into the operation. The post condition result is TRUE.

```
context RunnableItem :: set-auto ( auto : Boolean ) : Boolean
pre: self.oclnState ( idleState ) = true
post : self.stateTransitions.allInstances -> forAll ( st | st.automationOK = auto )
      and self.oclnState ( idleState ) = true and return = true
```

The **get-inven** operation. The runnable item is added to the information in the inventory. The result of the post condition is TRUE.

```
context RunnableItem :: get-inven ( inven : Inventory ) : Boolean
pre: self.oclnState ( idleState ) = true
post : inven -> including ( self ) and self.oclnState ( idleState ) = true and return = true
```

The **register** operation. The set of registries that have information about the runnable item is augmented to include the registry passed into the operation.

```
context RunnableItem :: register ( reg : Registry ) : Boolean
pre: self.oclnState ( idleState ) = true
post : self.hasInfoReg -> includes ( reg ) and self.oclnState ( registered ) = true and result = true
```

The **instantiate** operation puts the runnable item in its IDLE state. The result of the post condition is TRUE.

```
context RunnableItem :: instantiate ( ) : Boolean
pre: self.oclnState ( bound ) = true
post : self.currentState = IDLE-state and self.oclnState ( idleState ) = true and result = true
```

The **release-res** operation. The available resources on the machine on which the runnable item is executing is augmented by the resources in use by the runnable item. The result of the post condition is TRUE.

```

context RunnableItem :: release-res ( ) : Boolean
pre: self.oclnState ( stopState ) = true
post : self.onMach.ResOfM.resAvail = self.onMach.ResOfM.resAvail + self.resUsed
and self.oclnState ( FINAL ) = true and result = true

```

The **sameRliid** invariant. Runnable items with the same runnable item identifier share the same type of sources.

```

context RunnableItem :: inv sameRliid :
self.idOfRI.nHasID -> forAll ( ri | n.typesOfRI = self.typesOfRI )

```

ConfigService

The **get-config** operation. The post condition states that the configuration that corresponds to the configuration name is the desired configuration. The result of the post condition is TRUE.

```

context ConfigService :: get-config ( confName : ConfigId ) : Boolean
pre: self.oclnState ( idleState ) = true
post : confName.configuration.isDesiredConfig = true and self.oclnState ( hasDesiredConf ) = true
and result = true

```

The **get-inventory** operation. The post condition states that the inventory has the same local set identifier as the configuration service and that the get-inven operation is called for each runnable item in the local set. The result of the post condition is TRUE.

```

context ConfigService :: get-inventory ( inven : Inventory ) : Boolean
pre: self.oclnState ( hasDesiredConf ) = true
post : inven.LSIDofInven = self.myLSID
and self.RiinLS.allInstances -> forAll ( ri | ri.get-inven ( inven ) )
and self.oclnState ( hasInventory ) = true and result = true

```

The **compare-change** operation. Depending on the contents of the desired configuration and an inventory, runnable items are activated, deactivated, their states changed, or their automationOK attributes changed. See the Alloy model for the logic of this operation.

```

context ConfigService :: compare-change ( inven : Inventory, dConf : Configuration ) : Boolean
pre: self.oclnState ( hasInventory ) = true
post: ( dConf.confId - inven.invenId ) -> forAll ( nId :
    nId.nHasID.csControls = self implies
    nId.nHasID.activate ( nId.nHasID.onMach, self, nId.nHasID.hasInfoReg ) )
and ( inven.invenId - dConf.confId ) -> forAll ( nId :
    nId.nHasID.csControls = self and nId.nHasID.currentState.automationOK = true implies
    nId.nHasID.de-activate ( self ) )
and ( dConf.confId -> intersection ( inven.invenId ) -> forAll ( nId :
    ( ( dConf.nId.nHasID.desiredState <> inven.nId.nHasID.currentState and
    inven.nId.nHasID.currentState.automationOK = true ) implies
    inven.nId.nHasID.state-iran ( dConf.nId.nHasID.desiredState ) )
    and ( ( dConf.nId.nHasID.desiredVersion <> inven.nId.nHasID.hasVer and
    inven.nId.nHasID.currentState.automationOK = true ) implies
    inven.nId.nHasID.de-activate ( self ) and
    inven.nId.nHasID.activate ( dConf.nId.nHasID.onMach, self, inven.nId.nHasID.hasInfoReg
    ) )
    and ( ( dConf.nId.nHasID.stateTransitions.automationOK = false and
    inven.nId.nHasID.currentState.automationOK = true ) implies
    inven.nId.nHasID.set-auto ( false ) )
    and ( ( dConf.nId.nHasID.typesOfRI.count > inven.nId.nHasID.typesOfRI.count ) implies
    inven.nId.nHasID.activate ( dConf.nId.nHasID.onMach, self, inven.nId.nHasID.hasInfoReg ) )
    and ( ( dConf.nId.nHasID.typesOfRI.count < inven.nId.nHasID.typesOfRI.count and
    inven.nId.nHasID.currentState.automationOK = true )
    implies inven.nId.nHasID.de-activate ( self ) ) )
    and self.oclnState ( makingChanges ) = true and result = true

```

The **init** operation. This operation sets all the automation attributes so that automatic operation can occur. It also moves the item into its idle state. The result of the post condition is TRUE.

```

context ConfigService :: init ( ) : Boolean
pre: self.oclnState ( INITIAL ) = true
post: self.RiinLS -> forAll ( ri :
    ri.transitionsToStop.automationOK = true and ri.transitionsToIdle.automationOK = true )
and self.oclnState ( idleState ) = true and result = true

```


Object

The **deploy** operation for an Object. The post condition result starts out as TRUE. All of the object code used to create this object is checked to make sure it was created to run on the platform of the same type as the machine passed in. If any in the set was not created for the same platform, the post condition result becomes FALSE. The object's version is the one passed in and it is on the machine passed in.

```
context Object :: deploy ( ver : Version, m : Machine ) : Boolean
pre: self.oclnState ( INITIAL ) = true
post : result = true
    and self.ObjCreationOC.allInstances -> forAll ( oc |
        if oc.runsOnPlat.machineType = m.typeOfM
        then ( if result = false then result = false else result = true endif )
        else result = false endif )
    and self.hasVer = ver and self.onMach = m and self.oclnState ( deployed ) = true
```

The **notDepOnSerAppl** constraint. The configuration items in the object's dependency set can only be of the type Object.

```
context Object :: inv notDepOnSerAppl :
    self.depCI -> forAll ( ci | oclIsTypeOf ( Object ) )
```

Service

The **deploy** operation for a Service. The post condition result starts out as TRUE. All of the object code used to create this service is checked to make sure it was created to run on the platform of the same type as the machine passed in. If any in the set was not created for the same platform, the post condition result becomes FALSE. The object's version is the one passed in and it is on the machine passed in.

```
context Service :: deploy ( ver : Version, m : Machine ) : Boolean
pre: self.oclnState ( INITIAL ) = true
post : result = true
    and self.SCreationOC.allInstances -> forAll ( obj |
        obj.ObjCreationOC.allInstances -> forAll ( oc |
            if oc.runsOnPlat.machineType = m.typeOfM
            then ( if result = false then result = false else result = true endif )
            else result = false endif ) )
    and self.hasVer = ver and self.onMach = m and self.oclnState ( deployed ) = true
```

The **notDepOnAppl** constraint. The configuration items in the service's dependency set can only be of the type Service or Object.

```
context Service :: inv notDepOnAppl :
    self.depCI -> forAll ( ci | oclIsTypeOf ( Service ) or oclIsTypeOf ( Object ) )
```

Application

The **deploy** operation for an Application. The post condition result starts out as TRUE. All of the object code used to create this application is checked to make sure it was created to run on the platform of the same type as the machine passed in. If any in the set was not created for the same platform, the post condition result becomes FALSE. The object's version is the one passed in and it is on the machine passed in.

```
context Application :: deploy ( ver : Version, m : Machine ) : Boolean
pre: self.oclnState ( INITIAL ) = true
post : result = true
    and self.ACreationOC.allInstances -> forAll ( obj |
        obj.ObjCreationOC.allInstances -> forAll ( oc |
            if oc.runsOnPlat.machineType = m.typeOfM
            then ( if result = false then result = false else result = true endif )
            else result = false endif ) )
    and self.hasVer = ver and self.onMach = m and self.oclnState ( deployed ) = true
```

The **notDepOnAppl** constraint. The configuration items in the application's dependency set can only be of the type Application or Object.

```
Context Application :: inv notDepOnSer:
    self.depCI -> forAll ( ci | oclIsTypeOf ( Application ) or oclIsTypeOf ( Object ) )
```

LocalSet

The **uniqueLSID** constraint. Every local set has a unique local set identifier.

```
context LocalSet :: inv uniqueLSID:
    self.allInstances -> forAll ( ls1, ls2 | ls1.LocalSetID = ls2.LocalSetID implies ls1 = ls2 )
```

The **machInOneLS** constraint. Every machine can only be in one local set at any given point in time.

```
context LocalSet :: inv machInOneLS:
```

```

self.allInstances -> forAll ( ls1, ls2 | ls1 <> ls2 implies
    ( ls1.Machine -> intersection ( ls2.Machine ) ) -> size ( ) = 0 )
The invenInOneLS constraint. Inventories are associated with only one local set.
context LocalSet :: inv invenInOneLS:
    self.allInstances -> forAll ( ls1, ls2 | ls1 <> ls2 implies
        ( ls1.Inventory -> intersection ( ls2.Inventory ) ) -> size ( ) = 0 )
The registryInOneLS constraint. Registries are associated with only one local set.
context LocalSet :: inv registryInOneLS:
    self.allInstances -> forAll ( ls1, ls2 | ls1 <> ls2 implies
        ( ls1.Registry -> intersection ( ls2.Registry ) ) -> size ( ) = 0 )
The RlinOneLS constraint. Executing runnable items are associated with only one local set.
context LocalSet :: inv RlinOneLS:
    self.allInstances -> forAll ( ls1, ls2 | ls1 <> ls2
        implies ( ls1.RunnableItem -> intersection ( ls2.RunnableItem ) ) -> size ( ) = 0 )

```

Configuration

The **uniqueID** constraint. Every configuration has a unique identifier.

```

context Configuration :: inv uniqueID:
    self.allInstances -> forAll ( conf1, conf2 | conf1.ConfigID = conf2.ConfigID implies conf1 = conf2 )

```

Inventory

The **addOrdinalities** operation. The post condition checks all runnable items in the local set for which the inventory has information, and if any two unique runnable items are active and have the same source type this implies that the source type count for that type of item in the inventory is incremented. The post condition is TRUE. This operation is needed in the change-config operation of a configuration service so that it can count how many active runnable item types are in the local set and activate or deactivate some if the count is different than that requested in the desired configuration.

```

context Inventory :: addOrdinalities ( ) : Boolean
    post : self.LsInInven.RunnableItem -> forAll ( ri1, ri2 |
        ( ri1 != ri2 and ri1.isActive = true and ri2.isActive = true
            and ri1.srcType = ri2.srcType ) ) implies self.typesInInven = ri1.srcType and
        self.typesInInven.count = self.typesInInven.count + 1
    and return = true

```

CHAPTER 6 - ATM MONITORING SYSTEM SOFTWARE RUN-TIME CONFIGURATION MANAGEMENT SPECIFICATION - INITIAL VERSION

This chapter presents the ATM network monitoring system software run-time configuration management specification. This is presented in two sections: first the *type* model graphical figures, and second the Alloy model. As discussed in Chapter 5, the Alloy tool cannot take graphical input, so the Alloy model contains all the information from the *type* model as well as information not present in the *type* model. The Alloy model presented in this Chapter compiles and its state and operations have been verified for internal consistency using the Alloy tool.

1. THE *TYPE* MODEL FIGURES

There are four figures associated with the ATM network monitoring system run-time configuration management specification of software items. The first shows the software items and the relations between them. This is shown in Figure 6-1. The second shows the relations between software items and other configuration-related items, such as states, resources, and versions. This is shown in Figure 6-2. The definition of a local set is shown in Figure 6-3. Finally, the relation between the software items and the configuration, inventory, and registry concepts is shown, along with the relation to machines, is shown in Figure 6-4.

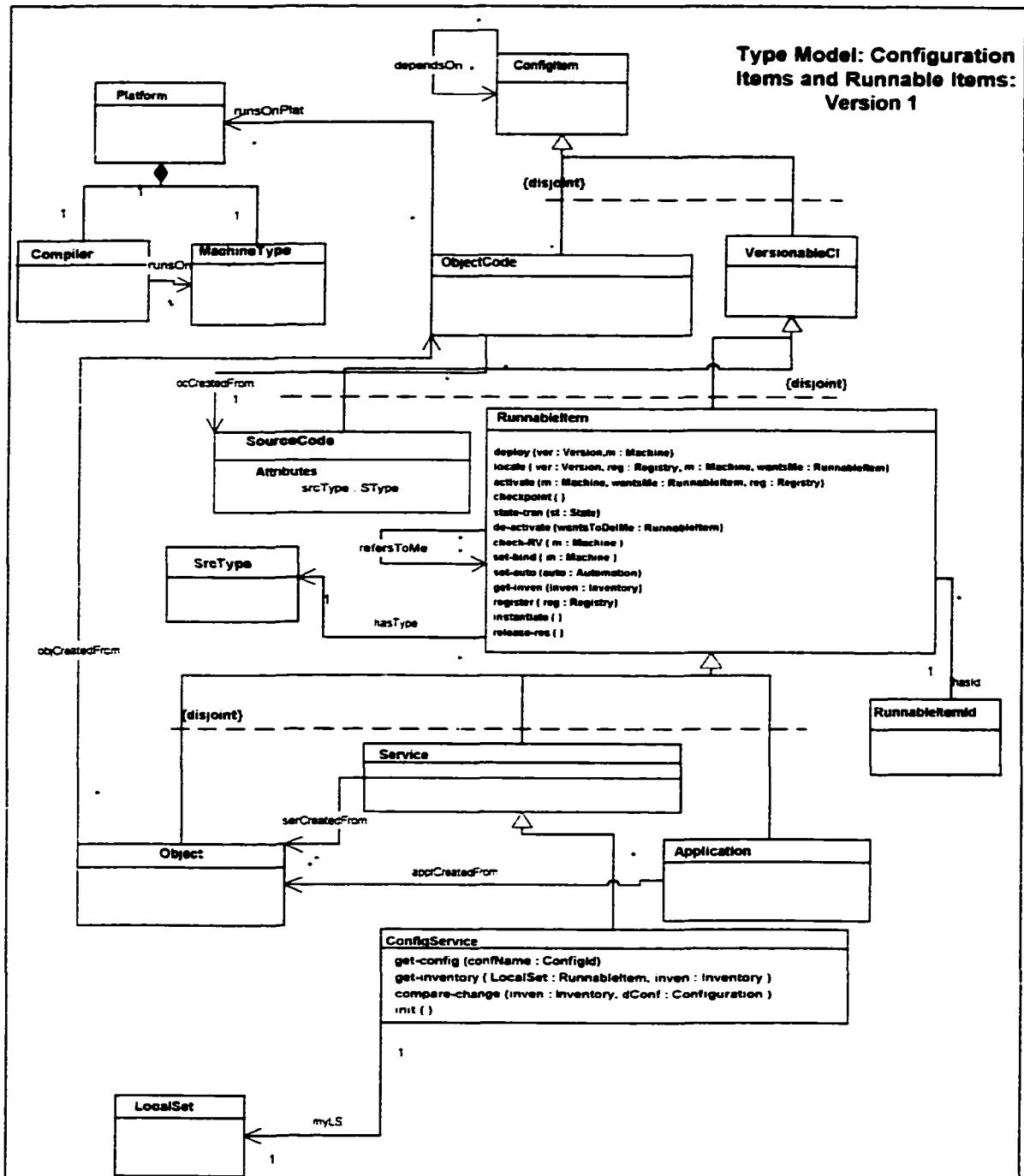


Figure 6-1. The software items and the relationships between them needed for run-time configuration management of the ATM/IP hybrid network monitoring system.

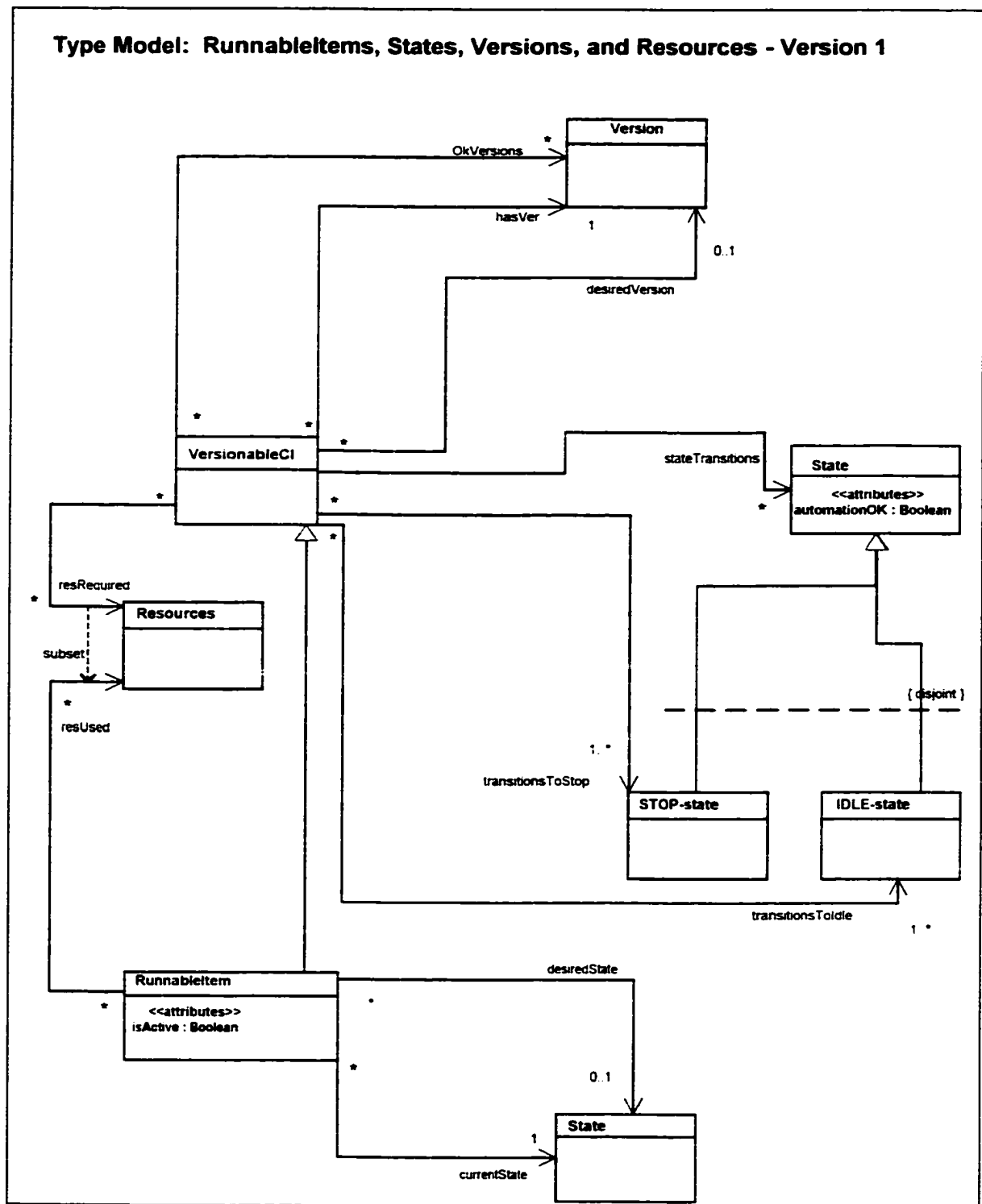


Figure 6-2. The software items and the relationships between them and versions, resources, and states.

Type Model: Local Set Definition - Version 1

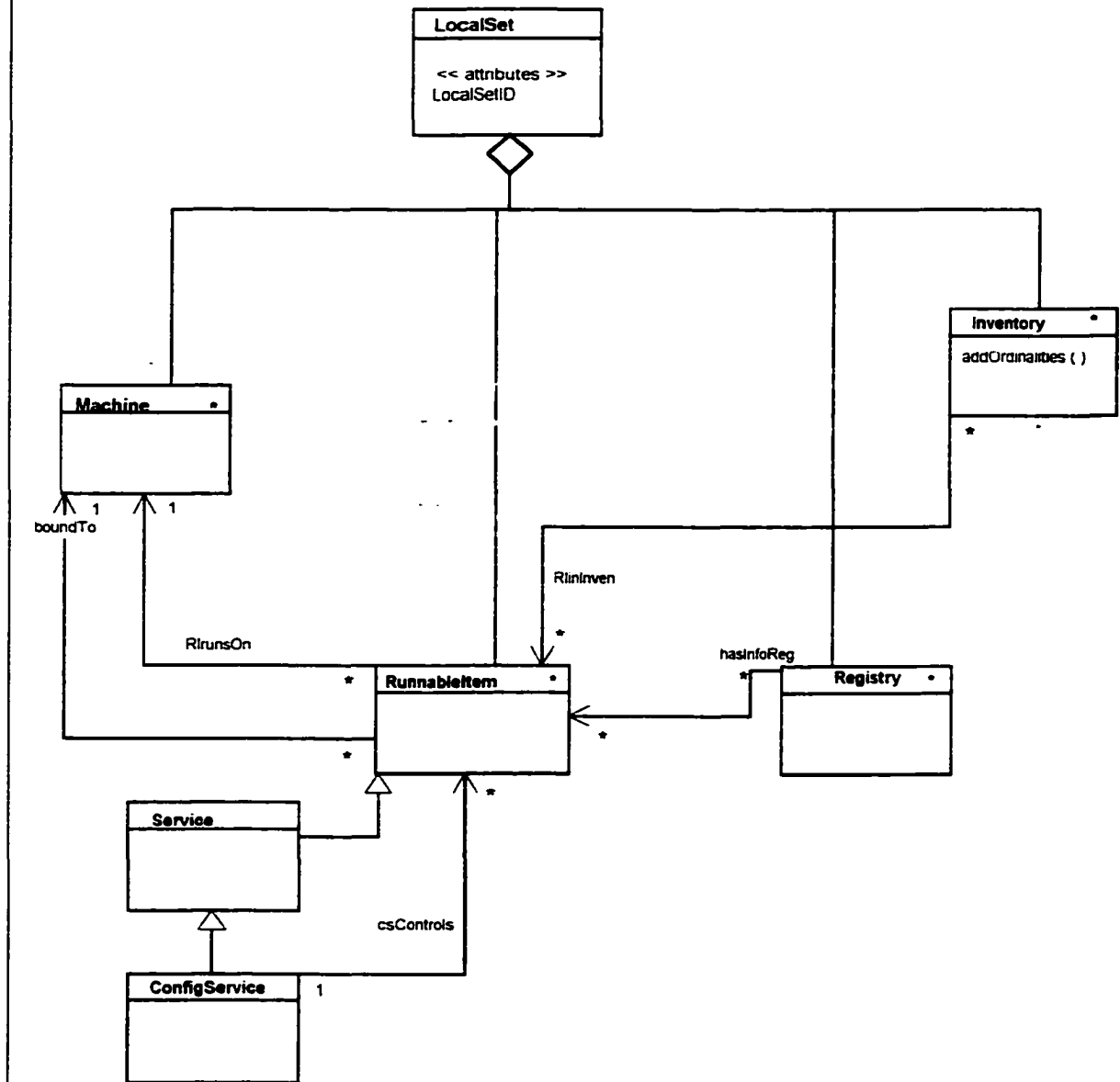


Figure 6-3. The definition of a local set.

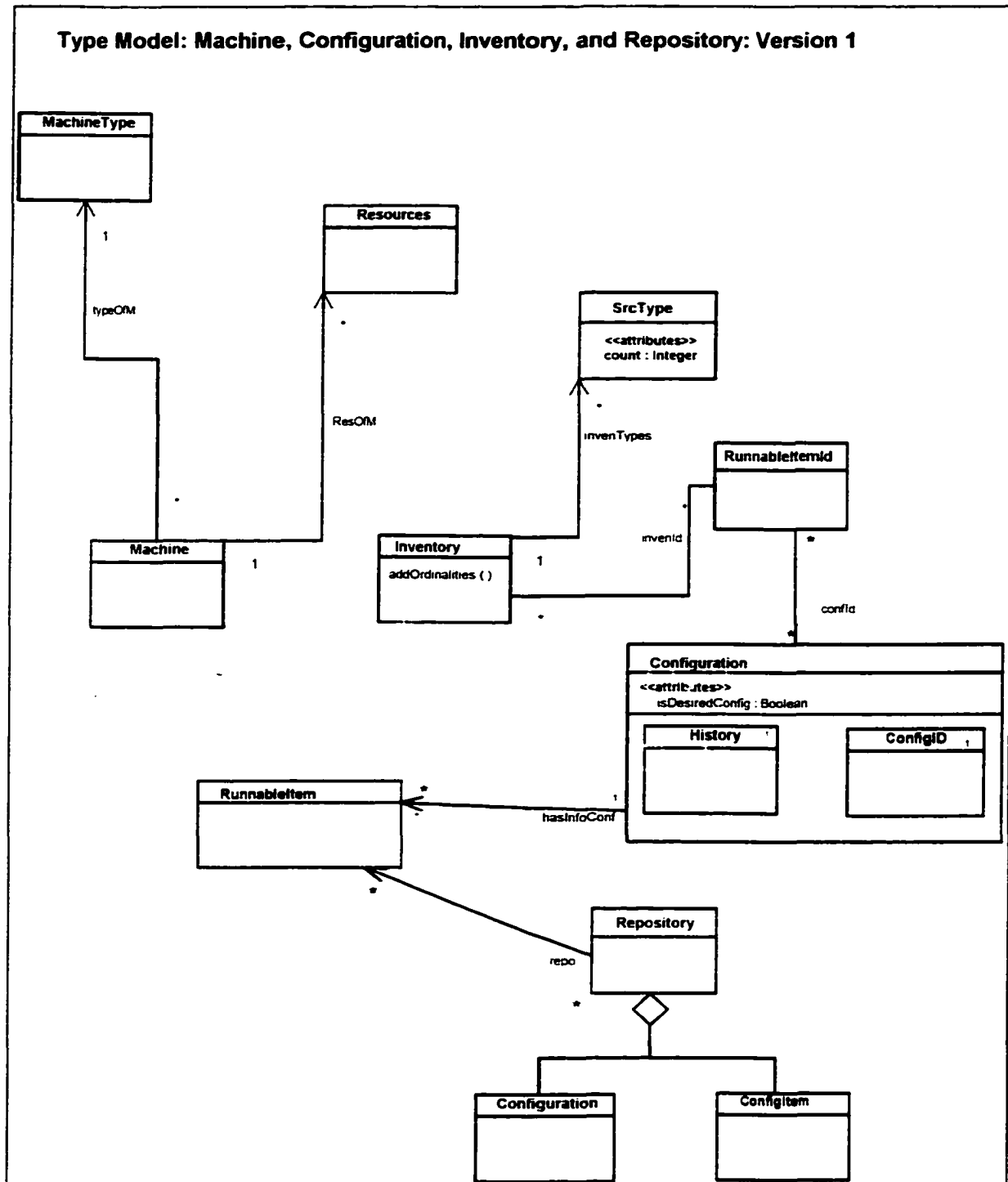


Figure 6-4. The executable software items and their relationships with the inventory, configuration, registry, and machines.

2. THE ALLOY MODEL

The Alloy model complements and extends the *type* model shown in Figures 6-1, 6-2, 6-3, and 6-4. Alloy complements the *type* model in terms of adding constraints and complete definitions to relations such as *dependsOn*. Alloy also extends the *type* model by defining the post conditions of operations and defining additional sets and relations needed to define those operations. The Alloy model also contains RTL-style temporal constraints in the form of comments (since Alloy does not include the notion of time). The complete Alloy model for the ATM network monitoring system's software run-time configuration management is shown below.

```
model ATM-network-monitoring-CM-initial-version
{
// The format of this Alloy model is:
//
//   model ATM-monitor
//   {
//       domain { ... } – the domain-level sets used in the model
//       state { ... } – subsets of the domain-level sets and relations among sets
//       def { ... } – definitions of subsets and of relations
//       inv { ... } – invariants that constrain the state of the model
//       op { ... } – definitions of the operations that act on the model
//   }

// This model was created as an attempt to link an UML meta-model showing the abstract model of
// an ATM/IP network monitoring system's dynamic configuration needs to a specification that held
// more detail than just the UML diagrams. The hope was also that this Alloy model could be used to
// test some of the specification consistency, and thus be used to help verify/validate the specification.

// Start by looking at the UML diagrams, and figuring out which classes represent primary sets for Alloy.
// In this model, all the classes that have generalizations (e.g. most of the configuration items
// of 'ConfigItem', so 'ConfigItem' is a domain set for Alloy) fall into this category. So do the class
// components of things like 'Platform' (i.e. 'Compiler' and 'MachineType') since the sub-components
// do not have any relation among themselves that can be used for definition purposes.

// The resulting domain-level sets are:
// Machine, Resources, LocalSetId, Inventory, Configuration, History, ConfigId, Repository,
// Platform, Compiler, MachineType, ConfigItem, State, Registry, Automation, and Version.

// We add Integers as a domain set; things like Resources can be defined from the set of integers.
// Using them also allows the use of equalities where one item's Resources is equal to that of another.
// Boolean is also added; this is useful for relations such as when a runnable item is either active or not.

// We also have to add OpAttempted and ReturnValue as domain sets in order to model the feedback
// mechanisms necessary to the operations of the system. These are not shown in the UML diagrams,
// they need to be specified as part of the actual operations to provide consistency in the feedback
// mechanisms of each of these operations.

// The type Boolean is used for some of the UML attributes, and SType is used to carry the types of
// sources, for use in figuring out the actual number of any given type of runnable item that is active.
// SType is also used along with RunnableItemId to identify runnable items that are of the same type.
```



```

// This is used for comparison purposes between information in an inventory about a runnable
// item and information in a desired configuration.

    domain { Machine, Resources, LocalSetId, Inventory, Configuration, History, ConfigId, Repository, Platform,
Compiler, MachineType, ConfigItem, State, Registry, Automation, Version, Integers, OpAttempted,
ReturnValue, Boolean, SType, RunnableItemId }

// Next define all the other sets and relations needed in the model. For the domain sets that correspond
// to classes with generalizations in the UML diagram, this simply means declaring subsets.

state
{
// The sub-classes of 'ConfigItem' partition this domain set.
// In addition, they are grouped in various ways to get everything but object code and runnable
// items. These groupings are defined in the definition section. For example, configuration items are
// source code, object code, objects, services, or applications. Runnable items are objects, services, or
// applications.

    partition SourceCode, ObjectCode, Object, Service, Application : ConfigItem
    VersionableCI : ConfigItem
    RunnableItem : ConfigItem

// ConfigService is a subset of the Service set.

    ConfigService : Service

// States. There are two required states that must be a subset of all the states that exist in the system for
// any configuration item (except object code, which takes its states from its corresponding source code).
// These two required states can be defined as subsets of all the states. They can't be the same however,
// so this is added as an invariant later in the model ('inv reqSt').

    STOP-state : State
    IDLE-state : State

// Operations and return values. These are needed for feedback purposes in the operation definitions.
// The type of operations is a group of disjoint sets that partition all possible operations so that a type only
// corresponds to one operation. For return values, there is a set of OK return values and a set of possible
// error
// values. The error values are constrained by an invariant below so that they are not OK ('inv errNotOK').

    partition deployed, located, instantiated, registered, bound, released, checkPt, transitioned,
inventoried, activated, de-activated, deActivated, setAuto, gotDC, checkedRes, checkedVer,
compared, deAct-a, setUB, addedOrd, initialized : OpAttempted
    OK : ReturnValue
    Error : ReturnValue

// Miscellaneous subsets useful in the Alloy model. The integer and boolean subsets are self-explanatory.
// The
// notBound subset of AllMachines is used in the 'boundTo' relation to indicate when a runnable item
// is not bound to any machine's resources. The automation values are used to determine if configuration
// changes can be made.

    positiveIntegers : Integers
    positiveNotZeroIntegers : positiveIntegers
    true : Boolean !
    false : Boolean !
    notBound : Machine !
    autoOK : Automation

// A desired configuration is a singleton subset of the configurations.

    desiredConfig : Configuration !

// Next define the relations.

// Relations needed to define local sets: 'on' and 'controlledBy' (inversely, 'controls'). The 'on' relation maps

```

```
// runnable items to the machines they are on, and 'controlledBy' maps them to the configuration
// service that controls them. The 'controls' relation is the mapping the other direction, from a
// configuration
// service to all the runnable items it controls. The 'controlledBy' relation is further
// defined below, in terms of the 'on' relation as well as the LocalSetId common to all the machines in the
// local set.
```

```
    controlledBy (~controls) : RunnableItem -> ConfigService !
    on (~riOn) : RunnableItem -> Machine !
```

```
// A LocalSet is a group of runnable items. They all are controlled by a single configuration service,
// and are all on machines that share a common LocalSetId. Inventories and Registries are also
// associated with local sets.
// This definition is shown below in the definition section of the model.
```

```
    LocalSetRI (~IsRI) : RunnableItem -> LocalSetId !
    LocalSetM (~IsM) : Machine -> LocalSetId !
    LocalSetI (~IsI) : Inventory -> LocalSetId !
    LocalSetR (~IsR) : Registry -> LocalSetId !
```

```
// Relations among the configuration items. These are from the UML diagram, the 'dependsOn' and
// 'createdFrom' relations. They have additional definitions shown in the definition section.
```

```
    dependsOn : ConfigItem -> ConfigItem
    createdFrom : ConfigItem -> ConfigItem
```

```
// Relations among the configuration items and their attributes. These are from the UML diagram:
// 'requires', 'uses', 'transitionsTo', 'developedToRunOn', 'hasState', 'hasAutomation',
// 'hasOrdinality', and 'hasVersion'.
// Note that the 'hasAutomation' relation is between all the states a runnable item can move
// to and the level of acceptable automation for moving among them. The 'transitionsTo' relation
// needs an invariant to ensure that the required states are part of this set of states the configuration
// item can be in (part of 'inv reqSI'). The uses and requires relations need an invariant to ensure
// a runnable item does not use more resources than it requires ('inv resourceUsage' below).
// The srcType relation is added to aid in the identification of current ordinality of runnable items,
// needed by the compare-change operation of the ConfigService. The idea is that the createdFrom
// relation can be used to get to the source code actually used to create a runnable item, then the
// srcType relation can be used to compare the types of the sources. If the set of types is the same,
// then the runnable items are the same type, and their ordinality can be 'added'. Since Alloy only
// supports sets, the sets will actually be unioned.
// The relations riType and riHasId are used to identify runnable items that are the same in that their
// sources create the same kind of runnable item, but they may differ in terms of their current state,
// ordinality, or version. This situation occurs when a runnable item is represented in an inventory
// and the configuration service has to determine if it is in the proper state, has the proper version, or
// has the proper ordinality as required by a desired configuration. Just comparing the complete
// runnable item to see if it is the same won't work since the relations may not be the same between
// the two descriptions of the runnable item. Using an identifier that is the same, based on the source
// type (SType) that is also the same allows such comparisons to be made.
```

```
    requires [Resources] : VersionableCI -> Integers
    uses [Resources] : RunnableItem -> Integers
    transitionsTo : VersionableCI -> State
    developedToRunOn : ObjectCode -> Platform
    hasState : RunnableItem -> State !
    hasAutomation : State -> Automation !
    hasVersion : VersionableCI -> Version !
    hasOrdinality : RunnableItem -> Integers !
    isActive : RunnableItem -> Boolean !
    srcType : SourceCode -> SType
    riType : RunnableItem -> SType
    riHasId (~idOfRI) : RunnableItem -> RunnableItemId !
```

```
// More relations between the configuration items and attributes. These are not part of the UML
// diagrams but are needed to specify the operations. In particular, for changing configurations,
// things like the desired ordinality, the maximum allowed ordinality, desired version, desired state,
// desired automation, and a set of compatible versions are needed.
```

```

// The set of compatible versions for a single
// configuration item is defined by the relation 'okVersions'. This relation is used to define a set
// of compatible configuration items. (That definition is given below.)

desiredOrdinality : RunnableItem -> Integers !
allowedOrdinality : RunnableItem -> Integers !
desiredVersion : VersionableCI -> Version !
desiredState : RunnableItem -> State !
desiredAutomation : State -> Automation !
okVersions : VersionableCI -> Version

// Relations among sets of configuration items and other classes. These are from the UML diagram:
// 'hasRegInfo' to map from a registry to the runnable items it knows about.
// 'hasConfigInfo' to map from a configuration to the runnable items it describes.
// 'hasInvenInfo' to map from an inventory to the runnable items it knows about (further defined below).
// 'hasMType' to map from a machine to the type it is, and
// 'haveRes' to map from machines to their physical resources.

hasConfigInfo : Configuration ! -> RunnableItem
hasInvenInfo : Inventory -> RunnableItem
hasMType : Machine -> MachineType !
haveRes : Machine -> Resources

// Relations among other classes on the UML diagram: 'runsOn'. This relation maps compilers to
// the type of machine they run on. Note that any particular compiler might run on many types of
// machines.
// To finish the definition of a platform, there must be a mapping from a platform to a compiler and
// platform to machine type. These last relations are not on the UML diagrams, since UML has the notion
// of aggregation whereas Alloy does not support the notion of tuples of greater arity than 2.

runsOn : Compiler -> MachineType
platformC : Platform -> Compiler !
platformMT : Platform -> MachineType !

// Relations to set up a configuration. The relation to associate configuration items with a configuration is
// shown on the UML diagram. However, there are no relations shown to associate a history or
// configuration
// identifier with a configuration (since they are part of a composition), so these must be added to the
// Alloy model.

configHist : Configuration -> History !
configId : Configuration ! -> ConfigId !

// Relations for operation feedback: Feedback

Feedback : OpAttempted -> ReturnValue

// Relations for keeping track of runnable items and their references : hasRegInfo, refersToMe,
// csReg. The 'refersToMe' relation keeps track of all the other runnable items that refer to any particular
// runnable item. This is used to figure out when it is safe to deactivate a runnable item. The
// 'csReg' relation is used to get to registries used by a configuration service.

hasRegInfo (~registeredIn) : Registry -> RunnableItem
refersToMe : RunnableItem -> RunnableItem
csReg : ConfigService ! -> Registry

// Relations for resource usage. These do not appear directly in the type model. The 'available' relation
// is defined below as the machine resources 'minus' the resources being used by all runnable items
// on a particular machine. 'Minus' is used in the set sense, not in the mathematical sense. The type
// model shows an available attribute associated with machine resources.

available [ Resources ] : Machine -> Integers !
boundTo : RunnableItem -> Machine !

```

```

}

// Next add definitions for subsets and relations.

// Define all config items except object code (needed since this is a subset that has similar attributes
// of version and resources, whereas object code doesn't have these relations directly), and the runnable
// config items (objects, services, and applications).

def VersionableCI { VersionableCI = SourceCode + Object + Service + Application }
def RunnableItem { RunnableItem = Object + Service + Application }

// A LocalSet consists of all the runnable items that are on machines that share a common LocalSetId.
// plus the configuration service that controls them (also running on a machine with the same LocalSetId).
// All these runnable items must be controlled by the one configuration service, and it cannot control
// any runnable items that are running on machines that do not share the common LocalSetId. These
// latter two constraints are stated as invariants below ('inv oneCScontrols', 'inv oneCScontrolsAll').

def LocalSetRI { all ri | ri.LocalSetRI = ri.on.LocalSetM }

// The definition of the 'controlledBy' relation. A runnable item and its controlling configuration service
// are in the same local set.

def controlledBy { all ri : RunnableItem | one cs : ConfigService | ri.controlledBy = cs -> ri.on.LocalSetM =
cs.on.LocalSetM }

// The relation 'hasInvenInfo' maps an inventory to the runnable items it knows about. These are
// defined as all those in the local set that shares the same LocalSetId as the inventory itself.

def hasInvenInfo { all inven | all ri : RunnableItem | ri in inven.hasInvenInfo -> ri.LocalSetRI =
inven.LocalSetI }

// The dependency relation definition.

def dependsOn { all ci | { no ci1 : ConfigItem | ci.dependsOn = ci1 }
| { ( ci in SourceCode -> some src : SourceCode | ci.dependsOn = src && ! { src.dependsOn = ci } )
&& ( ci in ObjectCode -> some oc : ObjectCode | ci.dependsOn = oc && ! { oc.dependsOn = ci } ) && (
ci in Object -> some obj : Object | ci.dependsOn = obj && ! { obj.dependsOn = ci } ) && ( ci in Service ->
some ser : Service | ci.dependsOn = ser && ! { ser.dependsOn = ci } ) } | { ci in Service -> some obj :
Object | ci.dependsOn = obj && ! { obj.dependsOn = ci } ) && ( ci in Application -> some ser : Service |
ci.dependsOn = ser && ! { ser.dependsOn = ci } ) } | { ci in Application -> some obj : Object |
ci.dependsOn = obj && ! { obj.dependsOn = ci } ) } }

// The 'createdFrom' relation.

def createdFrom { all ci | { ci in SourceCode -> ci.createdFrom = ci } && ( ci in ObjectCode -> one src :
SourceCode | ci.createdFrom = src ) && ( ci in Object -> some oc : ObjectCode | ci.createdFrom = oc )
&& ( ci in Service -> some obj : Object | ci.createdFrom = obj ) && ( ci in Application -> some obj : Object
| ci.createdFrom = obj ) }

// The 'available' relation. Available resources on a machine are those on the machine 'minus' (in the
// set sense, not in the mathematical sense) the resources being used by runnable items on the machine.
// This definition gives available as the maximum resources on the machine if there are no runnable
// items currently running on the machine.

def available { all m | all ri : RunnableItem, res : Resources | ri.on = m && m.available [ res ] = m.available [
res ] - ri.uses [ res ] }

// The 'okVersion' relation. An item in the set defined by this relation is part of the dependency
// relation of configuration items (except object code) and has a version that is compatible with
// the configuration item.

def okVersions { all notOC | all notOC1 : { notOC.dependsOn } | some ver : Version | ver in
notOC1.okVersions }

// The riType relation.

```

```

def riType { all ri | ri.riType = ri.createdFrom.srcType }

// Next add the invariants.

// The invariant that runnable items can have the same runnable item identifier if they have the same
// source types.

inv sameRlid { all ri1, ri2 : RunnableItem | ( ri1 != ri2 && ri1.riHasId = ri2.riHasId ) -> ri1.riType = ri2.riType }

// The invariant that all runnable items in a local set are controlled by its one configuration service.
// The invariant that there is only one configuration service in a local set and
// it controls all runnable items in the local set.

inv oneCSControls { all cs1, cs2 : ConfigService | no ri : RunnableItem | ri in cs1.controls & cs2.controls
&& ! ( cs1 = cs2 ) }
inv oneCSControlsAll { all lsid : LocalSetId | one cs : ConfigService | cs in lsid.lsRI && lsid.lsRI = cs.controls }

// An invariant that populates some of the configuration item sets so the createdFrom relation can be
// properly built.

// NEED TO SET SCOPE TO AT LEAST 5 FOR CONFIG ITEMS
inv sets { some ci1, ci2, ci3, ci4 : ConfigItem | ci1 in Service && ci2 in Object && ci3 in ObjectCode &&
ci4 in SourceCode }

// The invariant that error return values cannot be 'OK'.

inv errNotOK { all rv : ReturnValue | rv in Error -> ! ( rv in OK ) }

// The invariant that the required states (IDLE and STOP) are ones that every configuration item
// (except object code) can transition to, and that they are not the same state.

inv reqSt { all vCI : VersionableCI, stop : STOP-state, idle : IDLE-state | stop in vCI.transitionsTo && idle in
vCI.transitionsTo -> ! ( stop = idle ) }

// A configuration item that is not object code cannot use more resources than it requires. So the
// requires set must always be greater than or equal to the uses set.
// Note that Alloy does not support Integers, so this statement cannot really be validated.
// The new version of Alloy support set cardinality testing, so a more proper statement will be to simply test
// the
// relative cardinality of the sets.

inv resourceUsage { all ri : RunnableItem, res : Resources | some int : positiveIntegers | int = ri.requires
[res] - ri.uses [ res ] }

// Next add the operations defined in the UML diagrams.
//
// For RunnableItem, these are: check-RV, set-bind, release-res, register, instantiate, deploy, locate,
// activate,
// checkpoint, state-tran, de-activate, set-auto, and get-inven.
// Additional operations that had to be defined due to limitations in the Alloy model are:
// check-Res and check-Ver to replace check-RV, set-unbind-bind and deAct-act to make up for lack of
// sequencing in Alloy, and deActivate to make up for lack of recursion in Alloy.
//
// For ConfigService, these are: get-config, get-inventory, compare-change, and init-auto.
//
// For Inventory, the only operation is addOrdinalities.

// *** RunnableItem Operations ***

// An operation to check the resources needed by a runnable item.
// In this operation, we need to check that there are enough resources on the machine in question
// for the runnable item.

op check-Res ( ri : RunnableItem !, m : Machine ! ) {
  all res : Resources | some int : positiveIntegers | int in m.available [ res ] - ri.requires [res ]
}

```

```

    checkedRes.FeedBack' in OK
}

// An operation to check the version of a runnable item.
// In this operation, we need to check the version of the runnable item to make sure it is compatible with
// the
// other runnable items in its dependency set.

op check-Ver ( ri : RunnableItem ! ) {
    all depenRI : { ri.dependsOn } | some ver : { depenRI.hasVersion } | ver in depenRI.okVersions
    checkedVer.FeedBack' in OK
}

// An operation to set bindings for a runnable item.
// An assumption is that we've already checked to make sure there are enough resources on the
// machine via the check-Res operation.

op set-bind ( ri : RunnableItem !, m : Machine ! ) {
    all res : Resources | m.available' [ res ] = m.available [ res ] - ri.requires [ res ]
    ri.boundTo' = m
    bound.FeedBack' in OK
}

// An operation to release resources.
// This operation releases all the resources it is using on the machine it is running on.

op release-res ( ri : RunnableItem ! ) {
    all res : Resources | ri.on.available' [ res ] = ri.on.available [ res ] + ri.uses [ res ]
    ri.boundTo' = notBound
    released.FeedBack in OK
}

// An operation to release resources and re-bind them.
// This is needed since Alloy does not support sequencing; when the version of a runnable item
// needs to change the old version needs to get deactivated and the new one activated. This includes
// releasing the resources of the old version and binding the resources of the new version.

op set-unbind-bind ( ri : RunnableItem !, m : Machine ! ) {
    all res : Resources | ri.on.available' [ res ] = ri.on.available [ res ] + ri.uses [ res ] && m.available' [ res ] =
m.available [ res ] - ri.requires [ res ]
    ri.boundTo' = m
    setUB.FeedBack in OK
}

// An operation to register a runnable item in a registry.

op register ( ri : RunnableItem, reg : Registry ! ) {
    reg in ri.registeredIn'
    registered.FeedBack in OK
}

// An operation to instantiate a runnable item.

op instantiate ( ri : RunnableItem ! ) {
    ri.hasState' = IDLE-state
    instantiated.FeedBack' = OK
}

// An operation to deploy a runnable item.
// This means put the right version of a configuration item on the right machine. The latter means that
// we also have to make sure that the platform associated with the thing we're putting on matches the
// machine type of the machine we're putting it onto. We can do this by using the createdFrom relation
// one or more times to get the object code associated with the runnable item, then getting to its platform
// using the developedToRunOn relation, getting to the platform machine type with the platformMT
// relation. This final machine type needs to match that of the machine we're deploying to. At the
// end, set the feedback mechanism.

```

```

// This operation is one that needs a replicated repository for dependability purposes. This might be
// needed if there are problems finding the correct version of the runnable item. It is possible that
// the operation may have to go to a repository to find the runnable item, its objects, object code, or
// source code and then rebuild the runnable item.
// There isn't a way to put that search in this specification since it only declares the end result, that the
// correct type of runnable item is in the correct location.

op deploy ( ri : RunnableItem !, ver : Version !, m : Machine ! ) {
  m.hasMType in ri.+createdFrom.developedToRunOn.platformMT
  ri.hasVersion = ver
  ri.on' = m
  deployed.FeedBack' in OK
}

// An operation to locate a runnable item.
// For this operation, the item may be in the replicated registries passed in.
// The end result of this operation is that the runnable item is on the correct machine, its resources
// are bound, and it is instantiated and registered.
//
// This operation could be written so that it 'calls' deploy, check-Res, etc., but the Alloy tool appears to
// hang if this is done. Re-writing the post conditions of each operation allows the tool to operate
// properly.

op locate ( ri : RunnableItem !, ver : Version !, reg : Registry !, m : Machine !, wantsMe : RunnableItem ! ) {

  // From deploy...
  m.hasMType in ri.+createdFrom.developedToRunOn.platformMT
  ri.hasVersion = ver
  ri.on' = m

  // From check-Res
  all res : Resources | some int : positiveIntegers | int in m.available [ res ] - ri.requires [ res ]

  // From check-Ver
  all depenRI : ( ri.dependsOn ) | some vers : ( depenRI.okVersions ) | vers in depenRI.okVersions

  // From set-bind
  all res : Resources | m.available' [ res ] = m.available [ res ] - ri.requires [ res ]
  ri.boundTo' = m

  // From instantiate
  ri.hasState' = IDLE-state

  // From register
  reg in ri.registeredIn'

  ri.refersToMe' = ri.refersToMe + wantsMe
  located.FeedBack in OK
}

// An operation to activate a runnable item.
// In this operation, all dependencies first have to be located. Then the runnable item can be deployed,
// available resources checked, and the runnable item bound, instantiated, and registered. The set of
// items referring to the new one is augmented as well.
//
// This operation could be written so that it 'calls' deploy, check-Res, etc., but the Alloy tool appears to
// hang if this is done. Re-writing the post conditions of each operation allows the tool to operate
// properly.

op activate ( ri : RunnableItem !, m : Machine !, wantsMe : RunnableItem !, reg : Registry ! ) {
  all depenRI : ( ri.dependsOn ) | some vers : ( depenRI.okVersions ) | locate ( depenRI, vers, reg, m, ri )

  // From deploy...
  m.hasMType in ri.+createdFrom.developedToRunOn.platformMT
  ri.hasVersion = ri.desiredVersion

```

```

    ri.on' = m

// From check-Res
    all res : Resources | some int : positiveIntegers | int in m.available [ res ] - ri.requires [ res ]

// From check-Ver
    all depenRI : ( ri.dependsOn ) | some vers : ( depenRI.hasVersion ) | vers in depenRI.okVersions

// From set-bind
    all res : Resources | m.available' [ res ] = m.available [ res ] - ri.requires [ res ]
    ri.boundTo' = m

// From instantiate
    ri.hasState' = IDLE-state

// From register
    reg in ri.registeredIn'

    ri.refersToMe' = ri.refersToMe + wantsMe
    activated.FeedBack' in OK
}

// An operation to checkpoint the execution of a runnable item.
// This means put it into an idle state. Note that in the concrete world, putting something into an idle
// state probably also means stopping computations and freeing resources, etc. These design decisions
// are part of the definition of the idle state, and must be consistent across all the components.
// This specification on the other hand, just says that each configuration item needs an idle state
// and can be put into it in a consistent way.

op checkpoint ( ri : RunnableItem ! ) {
    ri.hasState = IDLE-state
    checkPt.FeedBack' in OK
}

// An operation to perform a state transition on a runnable item.
// The target state needs to be in the runnable item's set of states to which it can transition, then the
// runnable item can be put in the new state.

op state-tran ( ri : RunnableItem !, st : State ! ) {
    st in ri.transitionsTo
    ri.hasState' = st
    transitioned.FeedBack' in OK
}

// An operation to deactivate a dependency.
// This operation is needed since the operation to deactivate a runnable item cannot call itself
// using members of the dependency set of the runnable item as arguments.

op deActivate ( ri : RunnableItem !, wantsToDeI Me : RunnableItem ! ) {
    ri.refersToMe' = ri.refersToMe - wantsToDeI Me
    no referRI : ( ri.refersToMe ) | ri.hasState' = STOP-state && release-res ( ri )
    && ( all reg : ( ri.registeredIn ) | reg.hasRegInfo' = reg.hasRegInfo - ri )

    deActivated.FeedBack' in OK
}

// An operation to deactivate a runnable item.
// The reference set to this runnable item is decremented by the item that wants to deactivate it.
// (Note that this could either be a runnable item dependent on it or a configuration service.)
// If this set is empty, then the item can be moved to the stop state, its resources released,
// and its dependency set can be attempted to be deactivated. Note that if other items still reference
// items in the dependency set they will not be deactivated. (The idea is basically: "I'm quitting so I
// no longer depend on you. If nobody else depends on you either, then you can quit too.")
// If the original item is stopped, then it needs to be taken out of any registries that contain it.
// The operation 'deActivate' is needed to attempt to deactivate dependencies because Alloy will
// not allow an operation to be called from inside itself. The operation 'deActivate' is essentially

```



```

// the same as 'de-activate', except it does not traverse the dependency set.
//
// Note that Alloy does not support sequencing, so the order in which things happen is non-deterministic.

op de-activate ( ri : RunnableItem !, wantsToDelMe : RunnableItem ! ) {
  ri.refersToMe' = ri.refersToMe - wantsToDelMe
  no referRI : ( ri.refersToMe ) | ri.hasState' = STOP-state && release-res ( ri ) && ( all reg : ( ri.registeredIn )
| reg.hasRegInfo' = reg.hasRegInfo - ri ) && ( all depenRI : ( ri.dependsOn ) | deActivate ( depenRI, ri ) )
  de-activated.FeedBack' in OK
}

// An operation to "deactivate" one runnable item and then "activate" it as a new version.
// This is needed since Alloy does not support sequencing; we basically have to have a new operation
// that composes the post conditions of the operations that need to be sequenced; in this case
// deactivate and then activate.
//
// This operation could be written so that it 'calls' deploy, check-Res, etc., but the Alloy tool appears to
// hang if this is done. Re-writing the post conditions of each operation allows the tool to operate
// properly.

op deAct-act ( ri : RunnableItem !, m : Machine !, reg : Registry !, desiredVer : Version ! ) {
  ri.refersToMe' = ri.refersToMe
  ri.hasVersion = desiredVer
  all depenRI : ( ri.dependsOn ) | some vers : ( depenRI.okVersions ) | locate ( depenRI, vers, reg, m, ri )

// From deploy...
  m.hasMType in ri.+createdFrom.developedToRunOn.platformMT
  ri.hasVersion = ri.desiredVersion
  ri.on' = m

// From check-Res
  all res : Resources | some int : positiveIntegers | int in m.available [ res ] - ri.requires [ res ]

// From check-Ver
  all depenRI : ( ri.dependsOn ) | some vers : ( depenRI.hasVersion ) | vers in depenRI.okVersions

// From set-unbind-bind
  all res : Resources | ri.on.available' [ res ] = ri.on.available [ res ] + ri.uses [ res ] && m.available' [ res ] =
m.available [ res ] - ri.requires [ res ]
  ri.boundTo' = m

// From instantiate
  ri.hasState' = IDLE-state

// From register
  reg in ri.registeredIn'
  deAct-a.FeedBack' in OK
}

// An operation to set the automation level of a runnable item.
// For simplicity, this operation sets the same level of automation for all state transitions of the configuration
// item.

op set-auto ( vCI : VersionableCI !, auto : Automation ! ) {
  all st : ( vCI.transitionsTo ) | st.hasAutomation' = auto
  setAuto.FeedBack in OK
}

// An operation to get an inventory from a runnable item.
// The inventory to augment is passed in. The information of interest in an inventory is the runnable item's
// version (via the 'hasVersion' relation), whether it is active (via the 'isActive' relation), state
// (via the 'hasState' relation), and automation, which is on a per-state basis (via the 'hasAutomation'
// relation). However, all these items are available once the runnable item is part of the inventory,
// so this operation just needs to augment the inventory by the runnable item itself (via the 'hasInvenInfo'
// relation).

```

```

op get-inven ( ri : RunnableItem !, inven : Inventory ! ) {
    inven.hasInvenInfo' = inven.hasInvenInfo + ri
    inventoried.FeedBack' in OK
}

// *** ConfigService Operations ***

// A ConfigService operation to get a configuration (a desired configuration definition).
// A configuration identifier is passed in and the configuration that corresponds to it is found.
// This is another operation that needs a replicated repository which contains configurations.

op get-config ( confName : ConfigId ! ) {
    some conf : Configuration | conf.configId = confName
    gotDC.FeedBack' in OK
}

// A ConfigService operation to get an inventory for the local set.
// The operation figures out what inventory to use and passes this on to all the runnable
// items in the local set for them to augment with their own inventory information.
// Additional comments in the operation give RTL-style temporal constraints: the called
// operations need to finish within the tOP timeout. The 'theta' operation gives the time at which
// the i-th specified event occurred. So 'theta (begin (get-inven), i, t1)' sets t1 to the time at which
// the get-inven operation was started for the i-th time, and 'theta (end (get-inven), i, t2)' sets t2
// to the time at which that same i-th get-inven operation was finished. The specification says that
// the operation must finish within the timeout time.

op get-inventory ( configS : ConfigService !, inven : Inventory ! ) {
    inven.LocalSetI = configS.LocalSetRI
    all ri : ( configS.controls ) | get-inven ( ri, inven )

    // && all i : integer, t1 : Time | theta ( begin ( get-inven ), i, t1 )
    //      -> one t2 : Time | theta ( end ( get-inven ), i, t2 ) && ( t1 < t2 ) && ( t2 < t1 + tOP )
    //      inventoried.FeedBack' in OK
}

// A ConfigService operation to compare a desired configuration and an inventory of a local set and
// make changes to bring the inventory into alignment with the desired configuration.
// The cases are as follows:
// 1. An item is in the inventory and in the desired configuration.
//    Therefore check the state, version, ordinality and automation of the existing item and the
//    desired one. Make changes as follows:
//    a. state different: if the automation is OK for the state, then change it
//    b. ordinality different: if the desired ordinality is bigger than what exists, start a new one,
//    if the desired ordinality is less, stop one
//    c. version different: deactivate the current item and activate a new one with the new version
//    d. automation different: set the automation for the states to the desired automation
// 2. An item is not in the desired configuration: deactivate the item.
// 3. An item is in the desired configuration but not in the inventory: activate the item.
//
// Note that there is a problem checking ordinality. A runnable item can only know if it is active or
// not, not how many runnable items of the same executable exist. So the inventory has each runnable
// item and whether it is active. Some inventory operation must gather all the same kind of runnable items
// into
// an ordinality count before step 1b can be accomplished. It is not clear how to accomplish this
// specification in Alloy since there is no way in this version of Alloy to check set cardinality.

op compare-change ( inven : Inventory !, dConf : desiredConfig, configS : ConfigService ! ) {

    // all i : integer, t1 : Time | theta ( begin ( compare-change ), i, t1 )
    //      -> one t2 : Time | ( theta ( begin ( compare-change ), i+1, t2 ) ) && ( t1 + tCS <= t2 )

    all riID : ( dConf.hasConfigInfo.riHasId - inven.hasInvenInfo.riHasId ) | activate ( riID.idOfRI, riID.idOfRI.on,
    configS, configS.csReg )

    // && all i : integer, t1 : Time | theta ( begin ( activate ), i, t1 )
    //      -> one t2 : Time | theta ( end ( activate ), i, t2 ) ) && ( t1 < t2 ) && ( t2 < t1 + tOP )

```

```

    all riID : (inven.hasInvenInfo.riHasId & dConf.hasConfigInfo.riHasId ) | riID.idOfRI.hasState.hasAutomation
= autoOK -> de-activate ( riID.idOfRI, configS )

// && all i : integer, t1 : Time | theta ( begin ( de-activate ), i, t1 )
//      -> one t2 : Time | theta ( end ( de-activate ), i, t2 ) ) && ( t1 < t2 ) && ( t2 < t1 + TOP )

    all riID : (inven.hasInvenInfo.riHasId & dConf.hasConfigInfo.riHasId ) | one ri1, ri2 : riID.idOfRI, st1, st2 :
State | ( ri1.hasState = st1 && ri2.desiredState = st2 && ri1 != ri2 && st1 != st2 && st1.hasAutomation =
autoOK ) ->
    state-tran ( ri1, st2 )

// && all i : integer, t1 : Time | theta ( begin ( state-tran ), i, t1 )
//      -> one t2 : Time | theta ( end ( state-tran ), i, t2 ) ) && ( t1 < t2 ) && ( t2 < t1 + TOP )

    all riID : (inven.hasInvenInfo.riHasId & dConf.hasConfigInfo.riHasId ) | one ri1, ri2 : riID.idOfRI, v1, v2 :
Version | ( ri1.hasVersion = v1 && ri2.desiredVersion = v2 && ri1 != ri2 && v1 != v2 &&
ri1.hasState.hasAutomation = autoOK ) ->
    deAct-act ( ri1, ri2.on, configS.csReg, v2 )

// && all i : integer, t1 : Time | theta ( begin ( deAct-act ), i, t1 )
//      -> one t2 : Time | theta ( end ( deAct-act ), i, t2 ) ) && ( t1 < t2 ) && ( t2 < t1 + TOP )

    all riID : (inven.hasInvenInfo.riHasId & dConf.hasConfigInfo.riHasId ) | one ri1, ri2 : riID.idOfRI, o1, o2 :
Integers | ( ri1.hasOrdinality = o1 && ri2.desiredOrdinality = o2 && ri1 != ri2 && o1 != o2 ) ->
    ( ( some intP : positiveNotZeroIntegers |
        ( intP = ri2.desiredOrdinality - ri1.hasOrdinality && ri1.hasState.hasAutomation = autoOK ) ->
        activate ( ri2, ri2.on, configS, configS.csReg ) )

// && all i : integer, t1 : Time | theta ( begin ( activate ), i, t1 )
//      -> one t2 : Time | theta ( end ( activate ), i, t2 ) ) && ( t1 < t2 ) && ( t2 < t1 + TOP )

    | ( ( some intP : positiveNotZeroIntegers |
        ( intP = ri1.hasOrdinality - ri2.desiredOrdinality && ri1.hasState.hasAutomation = autoOK ) ->
        de-activate ( ri1, configS ) ) )

// && all i : integer, t1 : Time | theta ( begin ( de-activate ), i, t1 )
//      -> one t2 : Time | theta ( end ( de-activate ), i, t2 ) ) && ( t1 < t2 ) && ( t2 < t1 + TOP )

    all riID : (inven.hasInvenInfo.riHasId & dConf.hasConfigInfo.riHasId ) | one ri1, ri2 : riID.idOfRI, a1, a2 :
Automation | ( ri1.hasState.hasAutomation = a1 && ri2.hasState.desiredAutomation = a2 && ri1 != ri2 &&
a1 != a2 && ri1.hasState.hasAutomation = autoOK ) ->
    set-auto ( ri1, a2 )

// && all i : integer, t1 : Time | theta ( begin ( set-auto ), i, t1 )
//      -> one t2 : Time | theta ( end ( set-auto ), i, t2 ) ) && ( t1 < t2 ) && ( t2 < t1 + TOP )

    compared.FeedBack' in OK
}

// Operation to initialize all the automation to be OK to make configuration changes on runnable items
// no matter what state they are in. A default initialization.

op init ( configS : ConfigService ! ) {
    all ri : ( configS.controls ) | all st : ( ri.transitionsTo ) | st.hasAutomation = autoOK
    initialized.FeedBack' in OK
}

// *** Inventory Operations ***

// Operation to add the ordinality of the same types of active runnable items.

op addOrdinalities ( inven : Inventory ) {
    all ri1, ri2 : ( inven.hasInvenInfo ) | ( ( ri1 != ri2 )
        && ( ri1.isActive = true )
        && ( ri2.isActive = true )

```

```

        && ( ri1.+createdFrom.srcType = ri2.+createdFrom.srcType ) )
        -> ( ( ri1.hasOrdinality = ri1.hasOrdinality + ri2.hasOrdinality )
            && ( ri2.hasOrdinality = ri2.hasOrdinality + ri1.hasOrdinality ) )
    addedOrd.FeedBack' in OK
}
}

```

1. INTRODUCTION

This chapter reports on the validation and verification of the run-time configuration management specification of the ATM network monitoring system (see Chapter 6). This step ensures that each of the system key requirements is covered in the specification, and that the specification is internally consistent. The ability of validation and verification methods to achieve this purpose is discussed in the sections below. The methods discussed in this chapter are multiple view analysis (discussed in Section 2), problem abstraction (discussed in Section 3), and composability reachability analysis (discussed in Section 4). Automated techniques, based on the Alloy specification language are briefly discussed in Section 5. The results of each technique are presented, along with a method evaluation.²⁹

No single validation or verification method is sufficient to validate and verify the monitoring system software run-time configuration management specification. Some of the methods are more applicable to design or implementation (problem abstraction and composability reachability analysis), and some seem to work well during specification (multiple view analysis and the Alloy analysis tool). However, without modification, even these latter methods do not fully validate and verify the key requirements of incremental and automatic operation. Applicable portions of the methods must be

²⁹ Evaluation criteria are developed in Chapter 3.

combined to create a new validation and verification method. This method is discussed in Section 6.

2. THE MULTIPLE VIEW ANALYSIS TECHNIQUE

Issarny, Saridakis, and Zarras [33] propose using multiple architectural views to validate system consistency. They use a functional view of the system, an interaction view, a dependability view, and a performance view in their validation technique. Aspects of each of these views are compared as part of the consistency checks. Views are compared two at a time, and each is compared against all the others.

The functional view of a system is really the synergistic effect of simultaneously specifying all system requirements, and architects often document the functional view as class or object models. The *hpa* and Alloy models comprise the functional view of the ATM/IP network monitoring system software run-time configuration management. This view consists of the software-related items, their relationships, and their operations.

The interaction view consists of the interaction protocol between clients and servers. The interaction is between a ConfigService and RunnableItems in the ATM network monitoring system specification.

The dependability view contains system structures used in fault tolerance mechanisms. These mechanisms are data replication and ConfigService redundancy/mirroring in the ATM system.

The multiple view technique described by Issarny, et. al. validates consistency across these three views, but does nothing to validate specification of key system requirements (e.g. the ability to handle large scale, incremental and automatic operation). The technique needs to be modified to validate key requirements. The modification is to

add a view for each key system requirement. Thus, three new views are added: scalability, incremental, and automatic. This modified approach is shown in Figure 7-1.

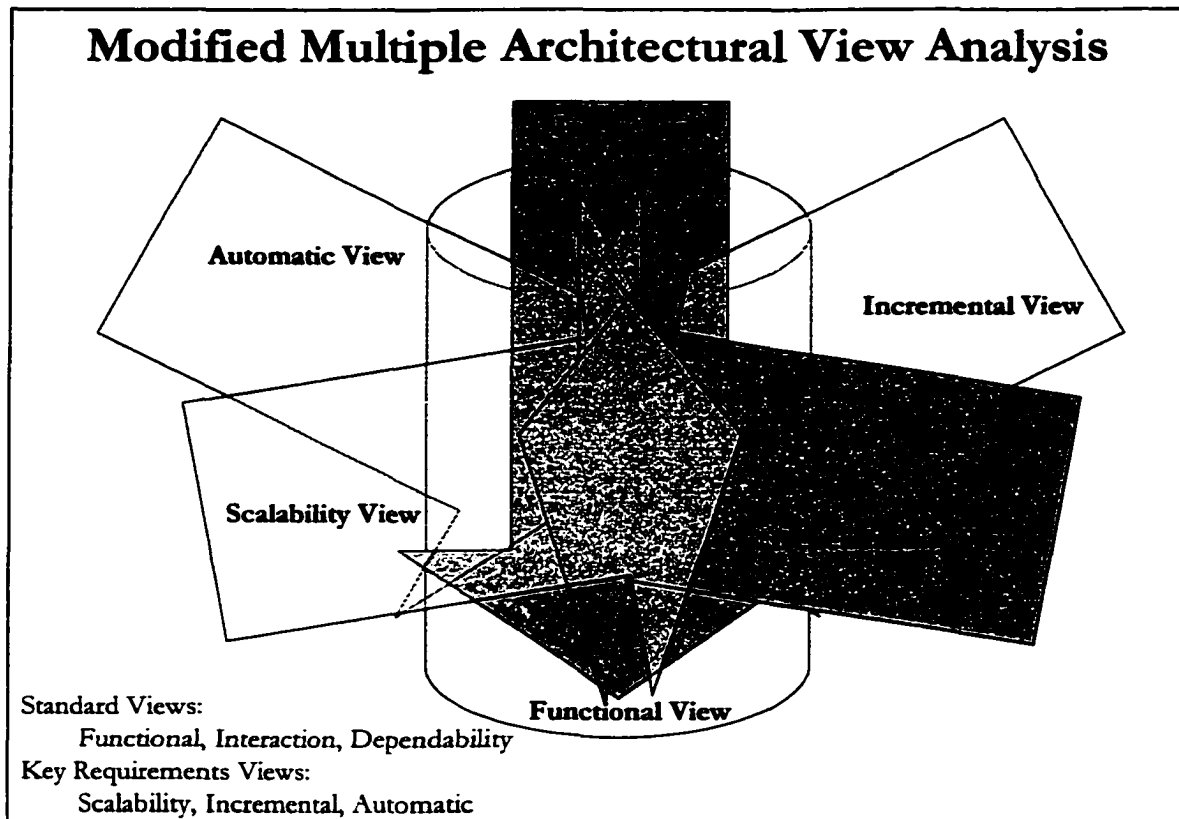


Figure 7-1. An abstract view of the modified multiple view analysis of the configuration management needs of the ATM network monitoring system. Three views are needed for any distributed system. These are functional, interaction, and dependability. The functional view of the system is the culmination of all system requirements and is what is traditionally known as the "specification". This view is represented by the cylinder in the diagram. The interaction and dependability views are shown as intersecting arrows. These three views are taken two at a time and compared for consistency. Views that represent each of the key system requirements must also be added to verify that these requirements are adequately specified. The additional views are scalability, incremental, and automatic views. These alternate views are also shown as intersecting arrows. Each alternate view must be checked against each of the three primary views (functional, interaction, and dependency) for consistency. The alternate views must also be checked against each other.

The multiple view analysis technique is changed to meet the needs of the ATM monitoring system:

- Performance is not a key requirement in the ATM monitoring system, so the performance view is deleted.

- The interaction view needs to be defined for the ATM monitoring system. In the work by Issarny, et. al, the interaction view consists of interaction protocols across all operations in all components. This definition is narrowed to configuration management interactions, specifically actions performed by a ConfigService in order to effect configuration management changes in the monitoring system.
- Additional views corresponding to scalability, incremental operation, and automatic operation are added.

These changes are reflected in Figures 7-2 through 7-7, which show the resulting views of the system: functional, interaction, dependability, scalability, incremental, and automatic.

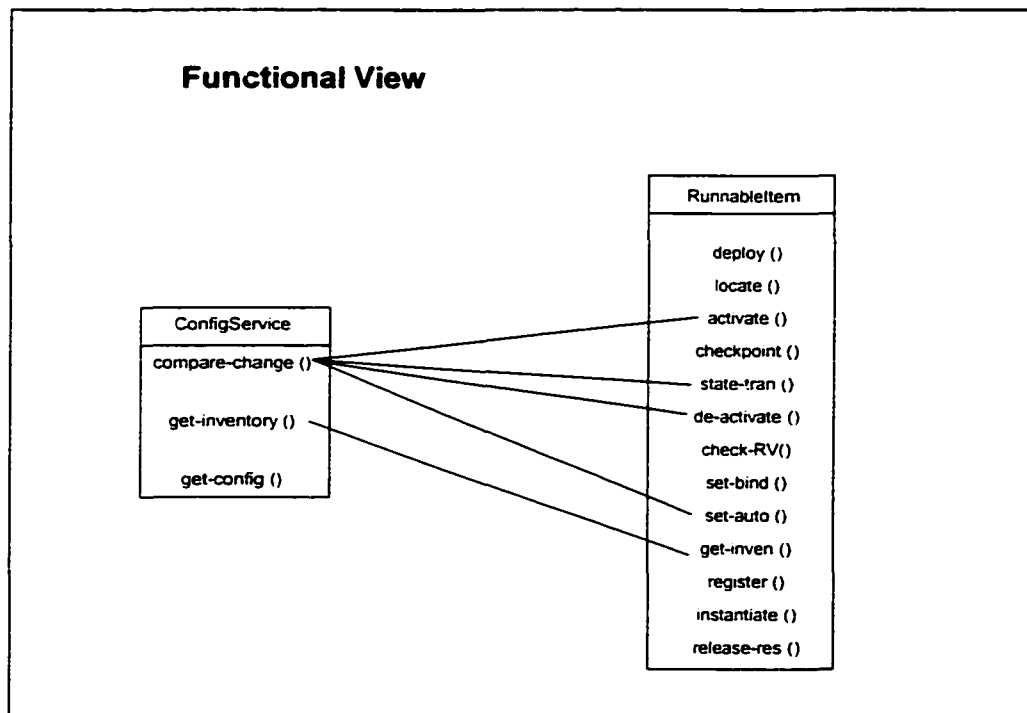


Figure 7-2. The *functional view* of the software configuration management of the ATM network monitoring system. A ConfigService acts as a client to the operations in RunnableItem, which acts as a server. The *functional view* shows operations callable by the client as well as server operations.

The *functional view* in Figure 7-2 consists of software-related items, their relationships, and the operations they perform.

The functional view shown in Figure 7-2 consists of RunnableItem operations callable by a ConfigService as well as other operations in RunnableItem. Lines between operations in ConfigService and operations in RunnableItem show the path of requests between ConfigService and RunnableItem. RunnableItem acts as a server to a ConfigService client.

The *interaction view* is the interaction protocol between ConfigService and RunnableItem, again viewed as client/server pairs. Each client request is encompassed in a timeout so the client can recover if there is no response from the server. Servers must respond to each client request. This protocol is shown in Figure 7-3.

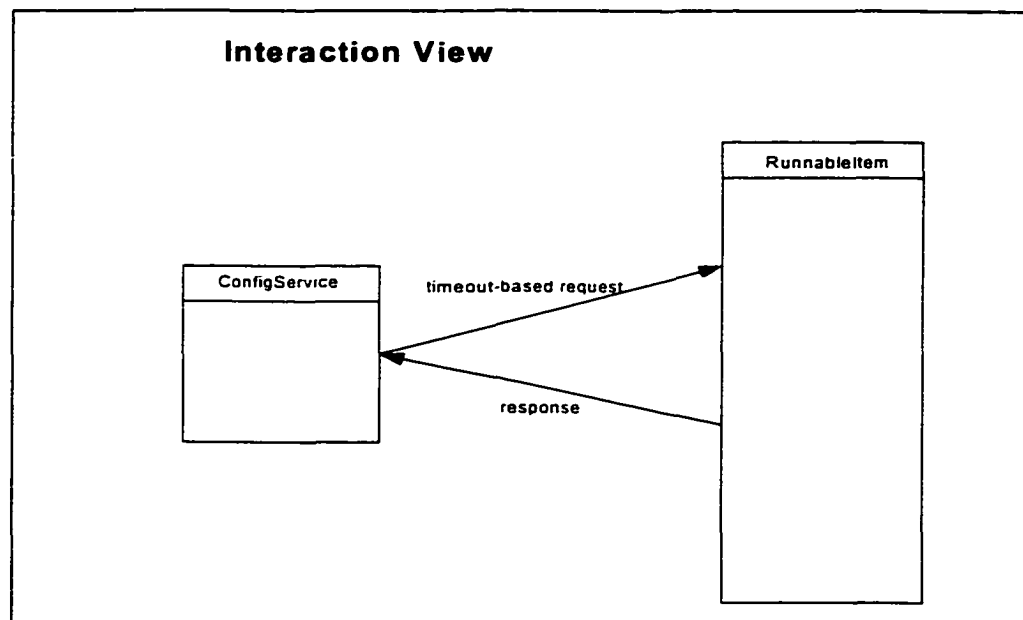


Figure 7-3. Interaction view; client (ConfigService) requests include timeouts for recovery purposes. Servers (RunnableItem) must respond to requests.

The *dependability view* shows fault tolerance mechanism structures. This view is shown in Figure 7-4.

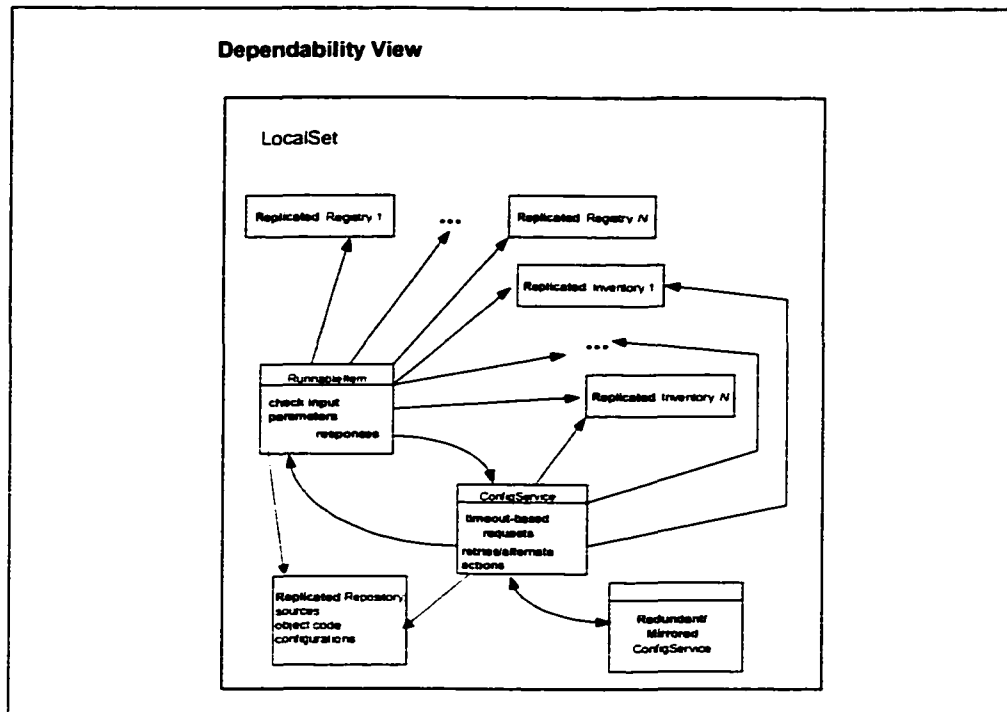


Figure 7-4. The dependability view. Redundancy in time and space (e.g. retries and redundant registries and inventories), timeouts, and replicated/mirrored ConfigServices are used for fault tolerance.

Figure 7-4 shows some mechanisms that support fault tolerance. These mechanisms are chosen based on errors likely to occur in RunnableItem and ConfigService operations. The mechanisms allow consistent error handling in the network monitoring system. It may or may not be reasonable to include some fault-handling techniques in a specification, although it certainly makes sense to put desirable techniques into architectural guidelines. However, it is easy to move into detailed design when choosing error handling techniques. Fault tolerance design needs to be avoided since it can lead to system functional design during requirements specification.

The *scalability view* identifies ideas and techniques in the system that help support large scale deployments, and ensures their use is consistent across the system. The scalability guiding principles used in the system are:

- aggregate actions and information where possible

- perform actions in parallel
- be able to perform all configuration actions without a central service or user help

Two ideas that help achieve these principles are:

- the idea of a local set
- a configuration service that operates autonomously on a local set

The scalability view is shown in Figure 7-5. The scalability view shows system structures that allow large scale, namely local set and its controlling ConfigService.

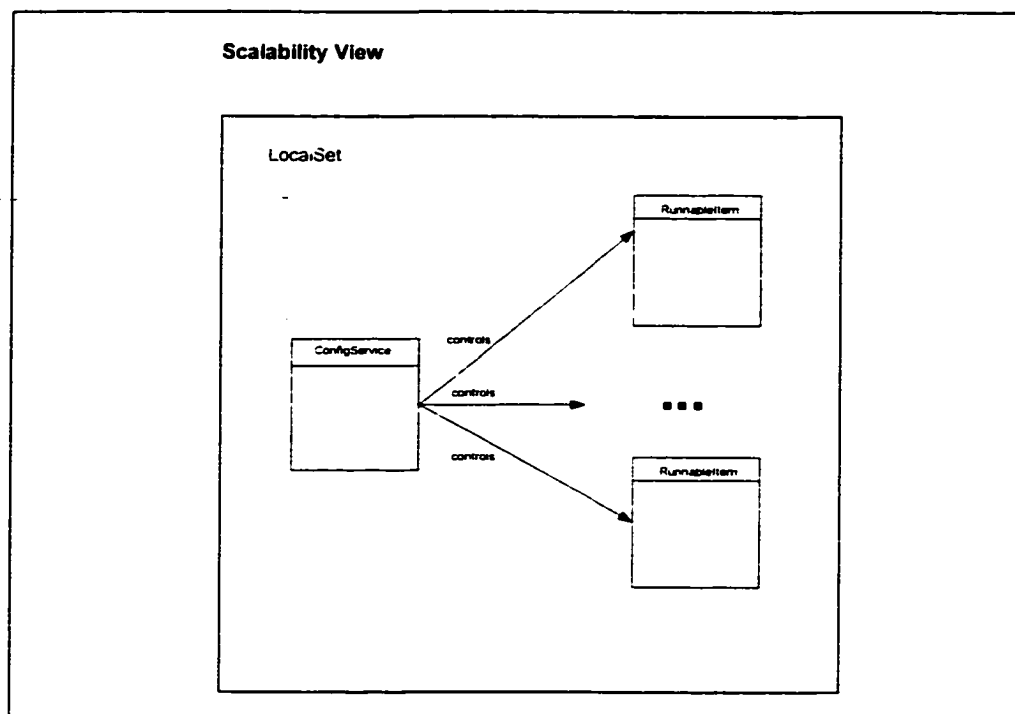


Figure 7-5. The scalability view. System structures that allow large scale are LocalSets, with their accompanying ConfigServices. ConfigServices operate autonomously on RunnableItems in the LocalSet.

The ideas and techniques supporting *incremental* and *automatic* operation are:

- move toward a desired configuration over time, rather than insisting movement occurs at the same time across the entire system
- allow local sets to operate independently from other parts of the system

- ensure all operations and information needed to perform configuration changes are available to ConfigServices

The incremental view in Figure 7-6 shows system structures that allow incremental operation. Structures are based on the idea of a LocalSet. A number of LocalSets constitute a running system, and they operate autonomously, so they might not all be in the same configuration at the same time. The notion of a minimum time between attempts to move to a new configuration allows the system to stabilize around any given configuration.

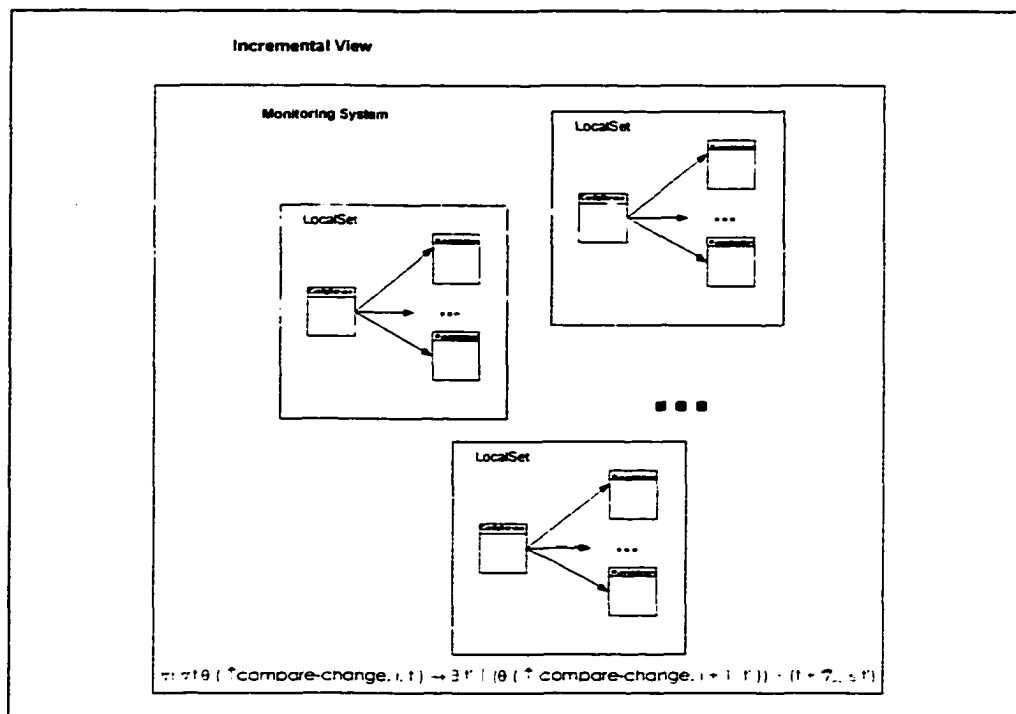


Figure 7-6. The incremental view. Local sets operate independently and therefore are not required to be in the same state at the same time. However, there is a minimum time between attempts to move to a new configuration so the system can stabilize around a desired configuration. This time constraint is shown in the RTL occurrence relation. The relation states that for all values of i and t , the start of the $i' + 1$ occurrence of the *compare-change* operation cannot occur until at least T_{cs} time after the start of the i' occurrence of the same operation. $\uparrow \text{compare-change}$ is the start event of the *compare-change* operation of a configuration service, i is the i' occurrence of the start of the *compare-change* operation, and $i + 1$ is the $i' + 1$ occurrence of the start of the *compare-change* operation.

The automatic view in Figure 7-7 shows system structures that allow automatic operation. These are the Automation concept and configuration operations in RunnableItem. The configuration operations must enable all configuration changes that can be performed by a ConfigService.

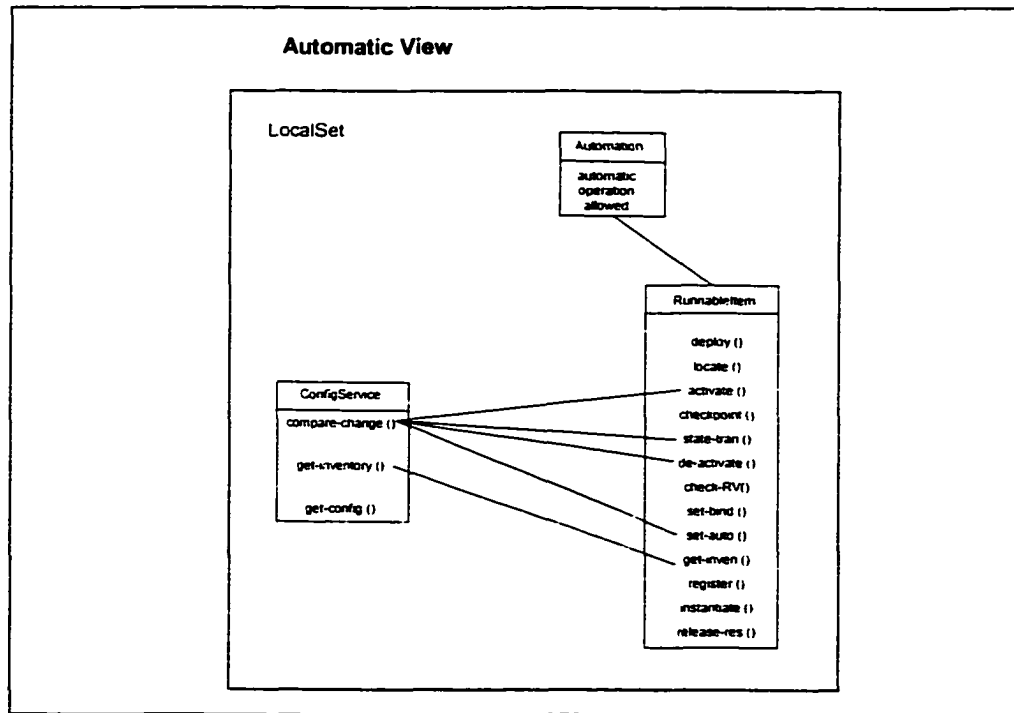


Figure 7-7. The automatic view. The ConfigService of a LocalSet has access to all the information and operations needed RunnableItems to perform configuration changes. The ConfigService can make configuration changes based on the Automatic concept.

Consistency checks can be applied after all six views are defined. Table 7-1 shows checks that must be performed between views, and their results. Views are checked in pairs to simplify comparisons. All views must all be checked against all other views. A discussion of the consistency checks for the ATM monitoring system is presented in Appendix 7.A. The specification is validated from the perspective of identifying whether key requirements are adequately reflected once consistency checks have been performed. Checks also verify the specification for internal consistency since the different architectural views are checked against one another.

Table 7-1. Views and Consistency Checks for the ATM Monitoring System. Consistency check results are italicized.

	Functional View	Interaction View	Dependability View	Scalability View	Incremental View
Interaction View	Do functional view components include interaction protocol? <i>missing timeouts</i>				
Dependability View	Do functional view components include dependability structures? <i>no</i>	Do dependability components include interaction protocol? <i>no</i>			
Scalability View	Do functional view components include scalability structures? <i>yes</i>	Do scalability components include interaction protocol? <i>yes</i>	Do dependability components include scalability structures? <i>yes</i> Do scalability components include dependability structures? <i>no</i>		
Incremental View	Do functional view components include incremental operation structures? <i>yes</i>	Do incremental components include interaction protocol? <i>yes</i>	Do dependability components include incremental structures? <i>yes</i> Do incremental components include dependability structures? <i>no</i>	Do scalability components include incremental structures? <i>yes</i> Do incremental components include scalability structures? <i>yes</i>	
Automatic View	Do functional view components include automatic operation structures? <i>yes</i>	Do automatic components include interaction protocol? <i>yes</i>	Do dependability components include automatic structures? <i>yes</i> Do automatic components include dependability structures? <i>no</i>	Do scalability components include automatic structures? <i>yes</i> Do automatic components include scalability structures? <i>yes</i>	Do incremental components include automatic structures? <i>yes</i> Do automatic components include incremental structures? <i>yes</i>

2.1 Consistency Check Results:

Results presented in Table 7-1 are from the initial specification presented in Chapter 6. The specification was changed to fix problems found by the multiple view analysis. (See Chapter 8 for the revised specification.)

The results of multiple view analysis are that the views are consistent with each other, except for the dependability view. The first version of the specification did not

contain any description of the dependability components, and so no view was consistent with the dependability structures. This deficiency leads to interesting observations about the dependability view. For example, some dependability techniques, such as replicated registries and inventories, can be included in both the *type* model and the Alloy operation specifications. However, when the system should use these dependability concepts cannot be expressed in a specification that does not include dynamic aspects of exception handling. Another instance of this problem is present in the idea of redundant ConfigServices in a LocalSet. ConfigService redundancy can be included in the *type* model as an attribute, but how that redundant service is used cannot be specified without considering dynamic aspects of exception handling.

How and when dependability concepts are used (and perhaps even which concepts are used) may really be a matter of design and not specification. However, in a complex distributed system, dependability must be an integral part of the architecture and consistently represented throughout the system. This is best accomplished by including dependability concepts and their static existence in the system specification. When necessary, dynamic aspects of the behavior need to be included in the specification as well. Thus, the specification needs to include dynamic models such as interaction diagrams. In the case of the ATM network monitoring system, the dependability requirements are straightforward. The system needs to be able to continue operation in the presence of unexpected network failures. Thus, simply specifying the existence of replicated registries, inventories, and repositories and redundant configuration services should give enough context for design verification.

2.2 Observations

Multiple view analysis depends on consistency checks between pairs of views, some present in all systems (i.e. functional, interaction, and dependability) and some unique to each system. The soft goals specification method helps define the functional view by identifying attributes and actions of items being configured as well as configuration operations needed by a ConfigService.

Either fuzzy sets or key domain requirements identify unique architectural views. The fuzzy set technique identifies the most important system requirements (those that need to be satisfied as HIGH or LOW). For the ATM system, scale is the most important requirement (and needs to be HIGH), and automatic operation and incremental operation are next most important (and also need to be HIGH). The key problem domain requirements also identify the most important requirements. These top requirements define additional views needed by the multiple view validation technique.

The multiple view technique is generally applicable to specification validation and verification as long as the verification is limited to checking minimum consistency across views. In particular, the dependency view is susceptible to over-specification.

The multiple view technique can be used to reason about how well key requirements are reflected in the specification. It can therefore validate the specification with respect to the presence of key requirements. In addition, the consistency checks across views allow reasoning about conditions under which key requirements can be met. For example, checking the dependability and incremental views may lead to the observation that if the network goes down between one local set and the rest of the system for an extended period of time, the isolated local set will miss intervening moves

to new configurations. Thus, the entire system does not have the property of moving toward a new desired configuration. Only those local sets that are accessible will reconfigure.

The multiple view technique found that the initial ATM specification lacks replication and redundancy-related structures. Dependability-related structures are present in the interaction protocol (i.e. timeout-based client/server protocol). The corrected specification is presented in Chapter 8.

2.3 Multiple View Analysis Evaluation

The multiple view analysis technique is evaluated using the technique evaluation criteria discussed in Chapter 3:

- 1) Validation of results using system architect or user feedback: There is no feedback available to validate the results of multiple view analysis.
- 2) Ability to validate and verify all necessary requirements: Multiple view analysis can be used to validate the existence of key run-time configuration management requirements in the specification. The method found the lack of dependency view structures in the functional view, as well as other errors of omission. Since the views are checked against each other, this method also provides an informal verification of internal consistency in the specification.
- 3) Applicability to specification: Multiple view analysis is suited to specification.
- 4) Ability to fit into industry:

- 1) Automation potential: The multiple view analysis technique does not seem to be suited to automation. However, if used for design-phase verification, automatic tools might check attributes, methods, and interface signatures as part of consistency checks.
- 2) Existing tools: none
- 3) Skill set needed by practitioners: No particular skills are needed for this technique.
- 4) Flexibility to fit into existing development processes: This technique can fit into specification activities. The technique also provides a clear partitioning of structures in terms of the views they support, and thus provides a rationale behind their use.
- 5) Ability to use partially or incrementally: The technique works for partial or incremental use. In this example, it only validates the run-time configuration management portion of a specification.

3. THE PROBLEM ABSTRACTION TECHNIQUE

The problem abstraction validation technique defines observable events from abstract sequence constraints. These events become test sequences. An example of a sequence constraint is that a move to the desired configuration *always* occurs when the following are true: 1) all the items in a local set are in a state where a configuration change is acceptable, 2) all the version, resources, and ordinality requests in the desired configuration can be met, and 3) a configuration change has not previously occurred within a specified minimum time. Similarly, if any of these requirements are not met, the configuration change *never* occurs.

A key aspect of problem abstraction is that all requirements embodied in a constraint statement must be observable. There must be some way to get feedback from the system demonstrating constraint satisfaction.

Since problem abstraction can generate test cases, the technique must be considered in light of testing an implemented system, not a specification. Problem abstraction might be used to create constraint statements and event sequences from a portion of a specification. For example, use cases can direct specific event statements that will be tested. Event statements in turn identify points in the system where events must be observable, along with the particular information needed in each event. This information can then be included in a specification.

Problem abstraction is not suitable for specification verification and validation. It can help identify test sequences during design. A complete problem abstraction example is presented in Appendix 7B.

3.1 Observations

Problem abstraction is related to design. If use cases exist as part of a specification, they can be used to create sequence constraints. However, ensuring that necessary events are externally visible is a design consideration. Problem abstraction does not help validate or verify a specification, rather it is useful in validating an implementation.

3.2 Problem Abstraction Evaluation

Problem abstraction is evaluated using the technique evaluation criteria discussed in Chapter 3:

- 1) Validation of results using system architect or user feedback: There is no feedback available to validate the results of problem abstraction.
- 2) Ability to validate and verify all necessary requirements: This method can only validate observable events. It requires that events related to all internal operations associated with run-time configuration management are visible.
- 3) Applicability to specification: Problem abstraction is more suited to design, and works on a system implementation. It is not suited to specification validation.
- 4) Ability to fit into industry:
 - 1) Automation potential: Once a system is implemented and sequence constraints are defined, automated tools can test the system against constraints. Clearly if the necessary events are not externally visible errors exist in the design or implementation.
 - 2) Existing tools: none
 - 3) Skill set needed by practitioners: No particular skills are needed for this technique.
 - 4) Flexibility to fit into existing development processes: This technique can possibly fit into design activities to identify test sequences. Its results can also be added to automated testing on the system implementation.

- 5) Ability to use partially or incrementally: The technique works for partial use since it is necessary to identify different sequences for each observable event that is to be tested.

4. THE COMPOSABILITY REACHABILITY ANALYSIS TECHNIQUE

In order to use composability reachability analysis, system processes must be described as a set of state transitions and actions that effect those transitions. The lowest level of the system hierarchy processes are described as labeled transition systems. Labeled transition systems can be composed into subsystems whose outward events are visible to higher levels of the hierarchy. Only the actions of interest in the validation of the system are included as part of the composition. An undefined state is included at the lowest level for transitions that are errors, and transitions to this state are included in the composition of states into subsystems. Thus, at the end of a system composition, if there exists a set of state transitions at lower levels of the system that can lead to the error state, the error state is also seen at the highest level of the system. Similarly, if no such error state exists in the highest composition, then the lower levels of the system did not have error transitions.

The de-activation of an application is presented as an example of composability reachability analysis. In this example, there are many processes that may be involved, but they are of only two types: 1) a ConfigService process, which initiates the de-activation of the application, and 2) a target application, which in turn will try to deactivate all of its dependent applications, services, and/or objects. As soon as a situation arises where there is some other application, service, or object still dependent on an item trying to de-activate itself, the de-activation stops. Once the item knows that all of its

dependencies are de-activated, the item can terminate itself and let the *ConfigService* know the status of termination; i.e. whether there were any problems de-activating dependencies.

Composability reachability analysis creates labeled transition diagrams for both a *ConfigService* and an application the configuration service de-activates. These labeled transition systems compose into a single system diagram. Another diagram describes a safety property of the system, specifically that a *ConfigService* does not attempt to move to a new configuration until a specific time has passed since the last attempt to move to a new configuration. The safety property composes with the system diagram. The composition demonstrates that the original *ConfigService* transition diagram does not include the timing constraint. The system is recomposed with a *ConfigService* labeled transition system that includes the timing constraint. The resulting labeled transition system composes with the safety property, which this time is fulfilled. Details of the labeled transition diagrams and compositions are shown in Appendix 7-C.

4.1 Observations

Composability reachability analysis may be more appropriate for design verification than for specification verification. However, it is a technique that verifies that the invariant dealing with minimum time between moves to a new configuration is present in the specification. The labeled transition technique found an error in the original *ConfigService* transition diagram: it did not include this minimum time between configuration changes. This temporal constraint is included in the Alloy model in the *change-config* operation. However, since Alloy cannot specify temporal constraints the

constraint is a comment. Thus, the Alloy tool cannot test the temporal constraint, and another technique must be used to verify existence of the temporal constraint.

Composability reachability analysis verifies existence of the timing constraint in the labeled transition systems. However, state transitions and events that cause them must be defined in composability reachability analysis. These transitions and events can be over-specified and design can occur, so care must be taken to keep the system description abstract.

4.2 Composability Reachability Analysis Evaluation

Composability reachability analysis is evaluated using the technique evaluation criteria discussed in Chapter 3:

- 1) Validation of results using system architect or user feedback: There is no feedback available to validate the results of composability reachability analysis.
- 2) Ability to validate and verify all necessary requirements: This method can only verify properties described as state transition systems.
- 3) Applicability to specification: Composability reachability analysis is more suited to design than specification. Composability reachability analysis also requires a different state transition system for each validated property and associated operations.
- 4) Ability to fit into industry:
 - 1) Automation potential: A tool can compose labeled transition systems once a system is described in terms of states, transitions, and

events. Similarly, if an invariant is described as a labeled transition system, a tool can compose it with the system labeled transition diagram in an attempt to find errors. Changing low level labeled transition systems to correct errors probably cannot be handled by an automatic tool.

- 2) Existing tools: none
- 3) Skill set needed by practitioners: The only skills needed are the ability to create and compose labeled transition systems. Invariants must also be described as state transitions systems. A problem with this technique is that each invariant and its related processes require a different state transition description. This is a verification method that works from the most detailed system operation to more abstract levels, and therefore involves a great deal of work.
- 4) Flexibility to fit into existing development processes: This technique can possibly fit into design activities, but the result is probably not worth the work.
- 5) Ability to use partially or incrementally: The technique works for partial use since it is necessary to create different labeled transition systems for each process involved in an invariant as well as for each invariant.

5. AUTOMATED TOOLS BASED ON SPECIFICATION NOTATION

A *type* model complemented by an Alloy model makes it possible that the tool associated with Alloy can be used to help validate and verify the specification. Since the Alloy model is complete, it duplicates and extends the type model. Any verification that can be done on the Alloy model provides verification of the *type* model as well, (provided the information in each model accurately represents the other).

The Alloy tool checks a model's state, invariants, definitions, assertions, and operations for consistency.

The Alloy tool creates a Boolean satisfaction formula from an Alloy model and assigns a scope to it. (Scope is the number of elements in each type of domain set in the model; scope is limited in order to create a decidable analysis.) The analysis determines if an instance exists for the formula within the scope bound. If one does exist, then the Alloy model is consistent. The tool can also be used to look for theorem (assertion) counterexamples that indicate model inconsistency. Jackson [34] states that scope is kept low in the tool (3 by default), so some consistent formulas may not have models at this scope. Experience has found that many formulas do have models in a small scope. Detailed operation of the tool is described in [34]. Note that finding an instance of a formula indicates model consistency, but not model correctness since the Alloy first order logic is undecidable. Failing to find an instance of a formula does not necessarily indicate that the model is inconsistent; it may simply have an instance at a higher scope. Similarly, failing to find a counterexample to an assertion does not mean that the assertion is consistent; a counter example may exist at a higher scope.

The Alloy model presented in Chapter 6 was analyzed using the Alloy tool. The model was analyzed on a Pentium III system with 128 Mbytes of RAM. It took 7 seconds to compile. Each state, invariant, and operation³⁰ was analyzed individually, and found consistent. Selected results are shown in Table 7-2.

Table 7-2. Analysis Time for Alloy Model Components.

Component Name	Component Type	Analysis Time (seconds)	Comments
state	model state	25	entire state schema; sets based on domain sets and relations
oneCSControls	invariant	22	there is only one controlling ConfigService in a LocalSet
errNotOk	invariant	2	error return values are different from OK return values
locate	operation	100	finds a runnable item or deploys, binds, instantiates and registers it
checkpoint	operation	27	moves a runnable item to its IDLE state

Table 7-2 shows a wide variation in analysis time, depending on the complexity of the component being analyzed. Jackson [34], discusses tool performance based on the number of items in the search space and the number of configurations present in the model. Each domain state and relation increases the potential search space, related to the scope used during analysis (the number of items of each domain type used in the analysis). Since the scope used for these analyses was the default, 3, this means that each

³⁰ The *compare-change* operation caused an Alloy tool failure, so this operation was not tested successfully.

of the 21 domain sets in the model has 2^3 possible values. Jackson states that each relation increases the potential search space by a factor of 2 to the k^2 , where k is the scope used during analysis. This means that each of the 43 relations in the model has an impact of 2^9 on the search space. Thus, the number of possible states is $2^{(21 \cdot 3 + 43 \cdot 9)}$, or $2.91e^{135}$. The number of transitions is $2^{(2 \cdot (21 \cdot 3 + 43 \cdot 9))}$, or $8.45e^{270}$.

The Alloy tool also analyzes operations against invariants. However, this analysis usually put the tool in an unspecified state or took too long to analyze. It was possible to analyze some of the resource-related operations and the resource usage invariant (a runnable item cannot use more resources than it requires). The tool found a counter-example, indicating that the operation is inconsistent with the invariant. This inconsistency is probably because the invariant and operation need access to resource set cardinality. Cardinality is not part of the first version of Alloy.

Each component of the Alloy specification was successfully validated for consistency by the Alloy tool with two exceptions. The first is the compare-config operation, which produced an assertion failure in the Alloy tool. (The cause of this failure is still being investigated.) The second is the combined analysis of the resource-usage invariant and operations that effect resource allocation. These operations are better specified using set cardinality, a feature included in the new version of the Alloy tool.

5.1 Automated Alloy Tool Evaluation

The Alloy tool is evaluated using the technique evaluation criteria discussed in Chapter 3:

- 1) Validation of results using system architect or user feedback: There is no feedback available to validate the results of the Alloy tool.
- 2) Ability to validate and verify all necessary requirements: The Alloy tool can be used to verify internal consistency of an Alloy model, so invariants, state, and operations modeled can be validated. Temporal constraints cannot be validated using this tool.
- 3) Applicability to specification: Alloy is applicable to specification.
- 4) Ability to fit into industry:
 - 1) Automation potential: The automated tool already exists, but temporal logic validation needs to be added to it.
 - 2) Existing tools: yes
 - 3) Skill set needed by practitioners: Practitioners need some skills in first order predicate logic and sets to use Alloy. No additional skills are needed to use the tool.
 - 4) Flexibility to fit into existing development processes: This technique can fit into specification. Current industrial validation practices involve peer review; verification using the Alloy tool can fit into this area.
 - 5) Ability to use partially or incrementally: The technique works for partial and incremental use since each state, invariant, or operation is a separate analysis (although the model only has to be compiled once if all these elements are present).

6. RECOMMENDED SPECIFICATION VERIFICATION METHOD

The recommended method to validate and verify a specification is to use two approaches.³¹ The first is the modified multiple view analysis described in Section 2 of this chapter. This technique validates how well key requirements are reflected in the specification. The second approach is to use the Alloy tool to test important invariants, operations, and overall system state. This technique checks for internal consistency across the specification.

This proposed validation and verification method allows reasoning about the existence of key requirements in the specification. The method also allows reasoning regarding internal consistency of the specification, but it does not allow statements to be made about provable correctness of the specification, nor of any resulting implementation.

³¹ See Figure 1.6 in Chapter 1 for an overall view of the validation and verification portion of the CMSpecV&V method.

APPENDIX 7A - MULTIPLE ARCHITECTURAL VIEW CONSISTENCY CHECKS

Figure 7A-1 shows consistency checks between architectural views that need to be performed for the software run-time configuration management of the ATM network monitoring system.

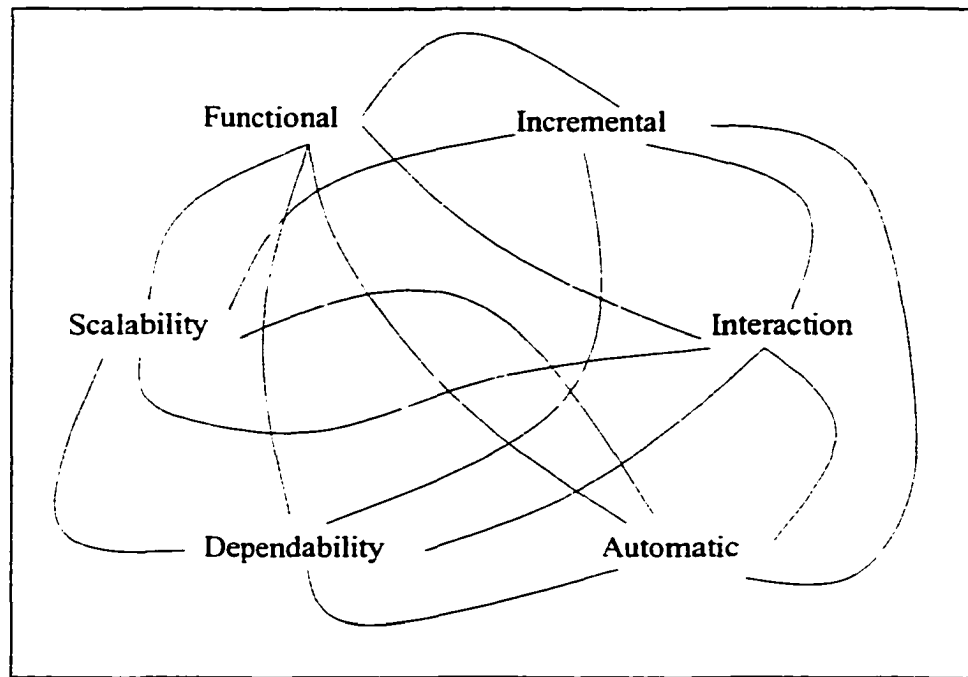


Figure 7A-1. Architectural view consistency checks that must be performed to validate the architectural description of the system.

Each consistency check is discussed in the following sections.

1. FUNCTIONAL/INTERACTION VIEWS

Synchronization of the interaction protocol and synchronization of the functional components must be consistent. The underlying interaction protocol consists of client/server requests (see Figure 7-3 in Chapter 7). Timeouts achieve

synchronization in this protocol. Each client request must be enclosed in a timeout, and servers are expected to respond to all requests they receive. Therefore, each client request will be answered by a server response, or it will timeout if there is no response (as in the case where a fault of some sort occurs). In either case, there is a fixed time beyond which the request is no longer valid, and the client can move onto other computations.

All functional view operations must support this same client/server protocol in order for interaction synchronization to be consistent with synchronization mechanisms in the functional view (see Figure 7-2 in Chapter 7). Operations include client operations used by ConfigService and server operations called in RunnableItem. Internal operations used by RunnableItem also must support this same synchronization mechanism.

The Alloy model in Chapter 6 must be used to verify that this timeout-based client/server protocol is supported by each operation. Specifically, each operation that acts as a client must have a timeout clause around each request that it makes. The ConfigService operations are: *compare-change* and *get-inventory*. Requests made by these operations are bound by RTL-based time constraint comments in the Alloy specification. Requests are made to RunnableItem operations *get-inven*, *activate*, *de-activate*, *state-tran*, *de-Act-act*, and *set-auto*. (The operation *de-Act-act* is a composition of the *de-activate* operation and the *activate* operation. A separate operation that composes the outcomes of the two operations is necessary since Alloy does not allow sequencing.)

Operations in RunnableItem that act as clients also have to utilize the timeout-based client/server protocol. These operations are: *activate*, *de-Activate*, *de-activate*, and

deAct-act. (Note that the operation *deActivate* is added to deal with the fact that Alloy does not allow recursion in operations; it is needed to deactivate dependencies from within the de-activate operation.) When the original Alloy specification was checked, there were no RTL time constraints in these operations. Thus, this check did find errors in the specification; these errors were corrected by adding the appropriate RTL-based time constraint comments.

The other portion of the interaction protocol is that sever operations always respond to requests they receive. ConfigService operations in this group are: *compare-change*, *get-inventory*, and *get-config*. These operations do have responses, which are in terms of the *Feedback* relation between an operation and return values (e.g. *compared.Feedback*). RunnableItem operations that need to be checked are: *get-inven*, *activate*, *de-activate*, *state-tran*, *de-Act-act*, *set-auto*, *deploy*, *locate*, *check-Res*, *check-Ver*, *set-unbind-bind*, *instantiate*, *set-bind*, *de-Activate*, *release-res*, *set-auto*, *checkpoint*, and *register*. The Alloy specification must be checked to see that each of these operations does set the appropriate *Feedback* relation, which they do. Thus, this consistency check found some errors, and also validated that most operations comply with the underlying interaction protocol.

2. INTERACTION/DEPENDABILITY VIEWS

Synchronization of the interaction protocol and synchronization of the dependability components must be consistent. The interaction protocol is timeout-based client/server requests (see Figure 7-3 in Chapter 7). The dependability components are shown in the dependability view (see Figure 7-4 in Chapter 7). The dependability components consist of replicated registries, inventories, repositories of

sources, object code, and configurations, and also of redundant or mirrored ConfigServices.

Since a redundant or mirrored ConfigService is just another ConfigService, it has already been checked against the interaction protocol in the discussion above. The idea of replicated repositories is not addressed in the Alloy specification. Alloy operation post conditions do not specify how or where items are found in repositories. However, operations that would access these repositories already exist and have been checked against the interaction protocol. The operations' specifications will need to be augmented to access repositories using timeout-based requests. The result of this consistency check is that some dependency components are consistent with the interaction protocol, but the check also found a lack of integration of most of the dependency components with the functional operations, and thus a potential inconsistency between the dependency components and the interaction protocol.

3. INTERACTION/SCALABILITY VIEWS

Synchronization of the interaction protocol and synchronization of the scalability components must be consistent. The interaction protocol is timeout-based client/server requests (see Figure 7-3 in Chapter 7). Scalability components are shown in the scalability view (see Figure 7-5 in Chapter 7). These components are LocalSets that contain one ConfigService and a number of RunnableItems that the ConfigService controls.

A LocalSet is defined as a set of relations between RunnableItems, Machines, Inventories, Registries, and a LocalSetID. Since it is an aggregation, there are no operations specific to a LocalSet. Thus, if the operations contained in RunnableItem

and ConfigService are consistent in their synchronization mechanisms with that of the interaction protocol, then the scalability components are also consistent. This has already been shown above, so this check is complete.

4. INTERACTION/INCREMENTAL VIEWS

Synchronization of the interaction protocol and synchronization of the incremental components must be consistent. The interaction protocol is timeout-based client/server requests (see Figure 7-3 in Chapter 7). Incremental components are shown in the incremental view (see Figure 7-6 in Chapter 7). They consist of a number of LocalSets with a timing constraint that prevents moves to new desired configurations unless a minimum amount of time has passed.

As in the argument above, a LocalSet is an aggregation and thus does not have any operations of its own that need to be checked against the interaction protocol synchronizations. The timing constraint is embedded in the ConfigService *compare-change* operation, which has already been successfully checked against the interaction protocol.

5. INTERACTION/AUTOMATIC VIEWS

Synchronization of the interaction protocol and synchronization of the automatic components must be consistent. The interaction protocol is timeout-based client/server requests (see Figure 7-3 in Chapter 7). Automatic components are shown in the automatic view (see Figure 7-7 in Chapter 7). They consist of LocalSets that in turn consist of a ConfigService and RunnableItems and the Automation concept.

There are no new components or operations in this view, so this check is complete.

6. FUNCTIONAL/SCALABILITY VIEWS

Functional components must have a scaling behavior consistent with the scalability components. Scalability components are shown in the scalability view (see Figure 7-5 in Chapter 7). They consist of LocalSets that contain one ConfigService and a number of RunnableItems that it controls. LocalSets operate independently; a ConfigService only controls the RunnableItems in its own LocalSet. The idea of independently operating LocalSets allows the system to scale to large size. (It should be noted, of course, that choosing the scope of a LocalSet can greatly affect the ability of the system to scale. For example, if a LocalSet is too large, scale issues due to the centralized control of the ConfigService can arise. The ideal size of a LocalSet is not specified in the *type* model or Alloy model shown in Chapter 6.) Independent operation of a LocalSet is supplied by invariants in the Alloy model that specify that there is only one ConfigService controlling any given RunnableItem, and that ConfigServices control only those RunnableItems in their own LocalSet. Thus, the items needed to fulfill the scalability view are present in the functional components via the LocalSet aggregation and the invariants on the items controlled by the ConfigServices.

7. FUNCTIONAL/INCREMENTAL VIEWS

Functional components must have an incremental behavior consistent with the incremental components. Incremental components are shown in the incremental view (see Figure 7-6 in Chapter 7). They consist of a number of LocalSets with a timing constraint that prevents moves to new desired configurations unless a minimum amount of time has passed. The timing constraint allows the system to settle into a particular desired configuration prior to attempting to move to another. The independently

operating LocalSets allow portions of the system to move toward a desired configuration at different times, thus creating partial movement toward the desired configuration. The functional components are consistent with this incremental behavior due to the independent operation of the LocalSets (via ConfigServices that only control the RunnableItems in their own LocalSets) and the timing constraints on the attempt to move to a new desired configuration. This constraint is shown in the Alloy model, in the *compare-change* operation. It is an RTL-style comment that states that the *compare-change* operation cannot begin until a specific amount of time has passed since it last began.

8. FUNCTIONAL/AUTOMATIC VIEWS

Functional components must have an automatic behavior consistent with the automatic components. Automatic components are shown in the automatic view (see Figure 7-7 in Chapter 7). They consist of LocalSets that in turn consist of a ConfigService and RunnableItems and the Automation concept. The idea of a LocalSet has already been shown to be consistent with functional components. In the automatic view, if automation is allowed, a ConfigService can effect configuration changes on RunnableItems, provided it can access the necessary operations in RunnableItem. This view is consistent with the functional view since all configuration operations needed by a ConfigService are available in each RunnableItem, and each RunnableItem has an Automation relation associated with its stop and idle states. There is also an operation to set the Automation relation.

9. FUNCTIONAL/DEPENDABILITY VIEWS

Functional components must have a dependability behavior consistent with the dependability components. Dependability components are shown in the dependability view (see Figure 7-4 in Chapter 7). They consist of replicated registries, inventories, repositories of sources, object code, and configurations, and also of redundant or mirrored ConfigServices. Interestingly, none of these components is present in the initial Alloy specification. Replicated registries and inventories are not precluded in functional operations, but neither are they, nor their use, specified. Replicated repositories and redundant/mirrored ConfigServices are not present at all. Thus, this consistency check found a large problem in the specification. Part of this problem is due to the fact that the *type* model and Alloy model are specifications of the static system and do not specify its dynamic behavior. The operations specified in the Alloy model are specified in terms of post conditions. These are declarative in nature and do not allow specification of when to use replicated/redundant components. However, it is still possible to include the creation of replicated/redundant components in operation post conditions. This step was simply left out of the original specification.

10. DEPENDABILITY /SCALABILITY VIEWS

Scalability components must have a dependability behavior consistent with dependability components, and dependability components must have a scaling behavior consistent with scalability components. Scalability components are shown in the scalability view (see Figure 7-5 in Chapter 7). They consist of LocalSets that contain one ConfigService and a number of RunnableItems it controls. LocalSets operate independently; a ConfigService only controls the RunnableItems in its own LocalSet.

Dependability components consist of replicated registries, inventories, repositories of sources, object code, and configurations, and also of redundant or mirrored ConfigServices (see Figure 7-3 in Chapter 7). Scalability components are no different than the functional components, so the fact that the functional components did not have any specification of dependency behavior means that the scalability components do not either. On the other hand, the dependency components can be shown to support scalability behavior provided that they exist completely within LocalSets. That is, replicated registries and inventories must contain only information about their own LocalSet. Similarly, redundant/mirrored ConfigServices must only mirror action in one LocalSet. A replicated repository can contain information beyond the LocalSet, but it must be constantly available to the LocalSet.

11. DEPENDABILITY /INCREMENTAL VIEWS

Incremental components must have a dependability behavior consistent with dependability components, and dependability components must have an incremental behavior consistent with incremental components. Incremental components are shown in the incremental view (see Figure 7-6 in Chapter 7). They consist of a number of LocalSets with a timing constraint that prevents moves to new desired configurations unless a minimum amount of time has passed. Dependability components consist of replicated registries, inventories, repositories of sources, object code, and configurations, and also of redundant or mirrored ConfigServices (see Figure 7-3 in Chapter 7). Since incremental components are not different from functional components, incremental components do not include any dependability behavior. Dependability components do

not have any affect on the timing constraints of LocalSets, and if they are completely contained within each LocalSet then they are consistent with incremental operation.

12. DEPENDABILITY /AUTOMATIC VIEWS

Automatic components must have a dependability behavior consistent with dependability components, and dependability components must have an automatic behavior consistent with automatic components. Automatic components are shown in the automatic view (see Figure 7-7 in Chapter 7). They consist of LocalSets that in turn consist of a ConfigService and RunnableItems and the Automation concept. Dependability components consist of replicated registries, inventories, repositories of sources, object code, and configurations, and also of redundant or mirrored ConfigServices (see Figure 7-3 in Chapter 7). Per previous arguments, since the automatic components do not introduce any new operations over the functional components, the automatic view is not consistent with the dependability view. However, if the dependability components are fully contained within the LocalSet, then the dependability components are consistent with the automatic view.

13. SCALABILITY /INCREMENTAL VIEWS

Incremental components must have a scalability behavior consistent with scalability components, and scalability components must have an incremental behavior consistent with incremental components. Incremental components are shown in the incremental view (see Figure 7-6 in Chapter 7). They consist of a number of LocalSets with a timing constraint that prevents moves to new desired configurations unless a minimum amount of time has passed. Scalability components are shown in the scalability view (see Figure 7-5 in Chapter 7). They consist of LocalSets that contain one

ConfigService and a number of RunnableItems that it controls. LocalSets operate independently; a ConfigService only controls the RunnableItems in its own LocalSet. Since LocalSets are the basis for both views and they have the same components, the views are consistent. The timing constraint of the incremental view further supports scalability by ensuring that the system will settle into one desired configuration prior to attempting to move to another. (Note that the length of time between moves may need to be increased with increased system scale. However, this time is normally on the order of several hours to a day, so an increase in the minimum time between configuration changes may not be necessary.)

14. SCALABILITY /AUTOMATIC VIEWS

Automatic components must have a scalability behavior consistent with scalability components, and scalability components must have an automatic behavior consistent with automatic components. Automatic components are shown in the automatic view (see Figure 7-7 in Chapter 7). They consist of LocalSets that in turn consist of a ConfigService and RunnableItems and the Automation concept. Scalability components are shown in the scalability view (see Figure 7-5 in Chapter 7). They consist of LocalSets that contain one ConfigService and a number of RunnableItems that it controls. LocalSets operate independently; a ConfigService only controls the RunnableItems in its own LocalSet. Since the components of both of these views are LocalSets, the views are consistent. Indeed, the idea of independently operating loci in the system directly supports its scalability since there are no central services or operations that can become bottlenecks.

15. INCREMENTAL /AUTOMATIC VIEWS

Automatic components must have an incremental behavior consistent with incremental components, and incremental components must have an automatic behavior consistent with automatic components. Automatic components are shown in the automatic view (see Figure 7-7 in Chapter 7). They consist of LocalSets that in turn consist of a ConfigService and RunnableItems and the Automation concept. Incremental components are shown in the incremental view (see Figure 7-6 in Chapter 7). They consist of a number of LocalSets with a timing constraint that prevents moves to new desired configurations unless a minimum amount of time has passed. Since both views contain the LocalSet component, they are consistent.

APPENDIX 7B – USING PROBLEM ABSTRACTION TO GENERATE TEST CASES FOR THE ATM MONITORING SYSTEM

Consider the use case of starting an application as an example of problem abstraction. The application can already exist in the system or it can be a new application. The main sub-tasks are to choose an application to start, create a new desired configuration with the application if one does not already exist, and start a configuration change to the new desired configuration. The end result of the task is getting feedback regarding its execution. The general use case for this task is shown in Figure 7B-1 below.

The actual actions that need to take place inside the system boundary are shown in Figure 7B-2.

A possible set of abstract sequence constraints for the feedback mechanism are:

- A request to move to a new configuration ALWAYS results in feedback: OK or errors on a per-local set basis.
- If ordinality is “full” a new object, service, or application is NEVER started.
- If an error occurs while starting dependencies, new dependencies are NEVER started and ones already started are ALWAYS cleaned up if no other object/service/or application is using them.
- Something is NEVER activated if all of its dependencies cannot be activated or if ordinality, version, or resources are not OK.

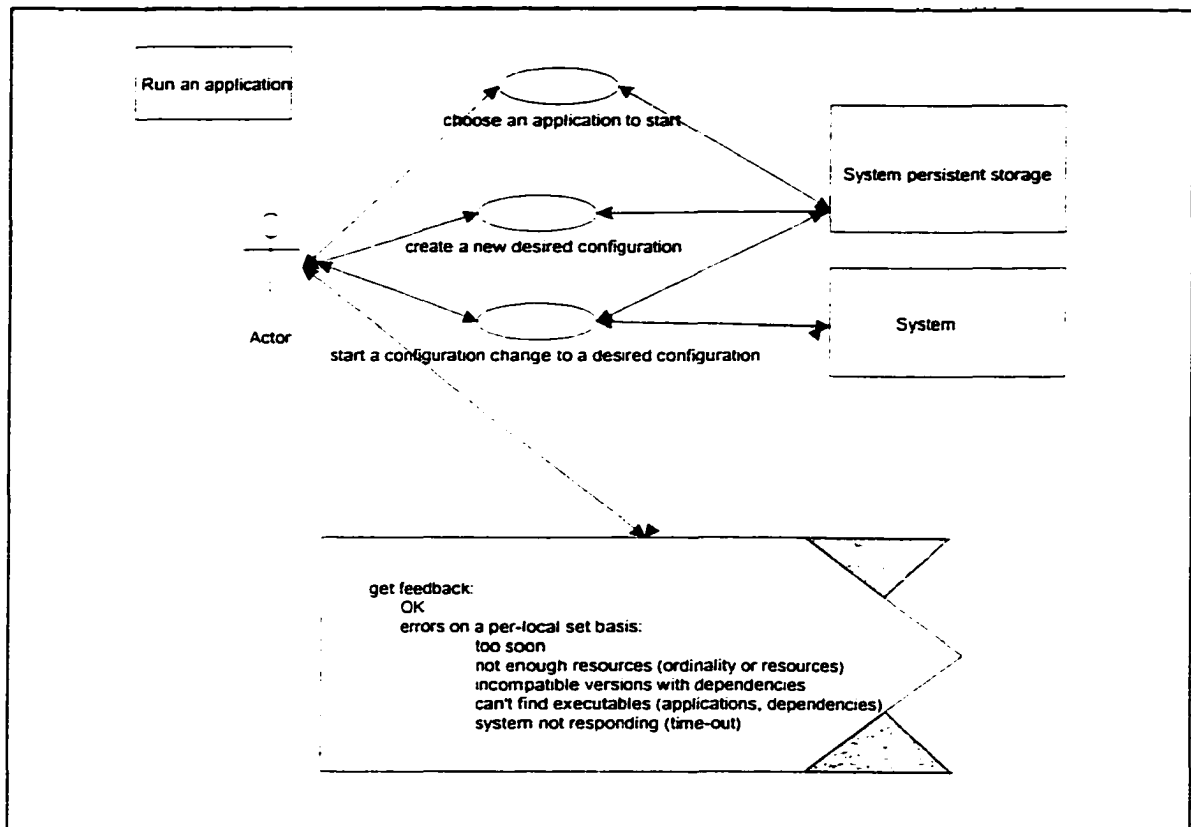


Figure 7B-1. Use case for starting a new application. The actor interacts with persistent storage to identify the application to be started along with creating a new desired configuration that contains the application, then beginning a transition to a new configuration that contains the application.

- Moving to a new configuration may POSSIBLY be delayed if some of the items in the local set are not in a state where they can change configuration. In this case a request to move to an OK state is ALWAYS sent to these items and a move is ALWAYS attempted again.
- A failure to receive information from an item is ALWAYS handled via a time-out ALWAYS followed by one re-try ALWAYS followed by a decision that a failure has occurred or POSSIBLY by attempting to get the information from an alternate source.

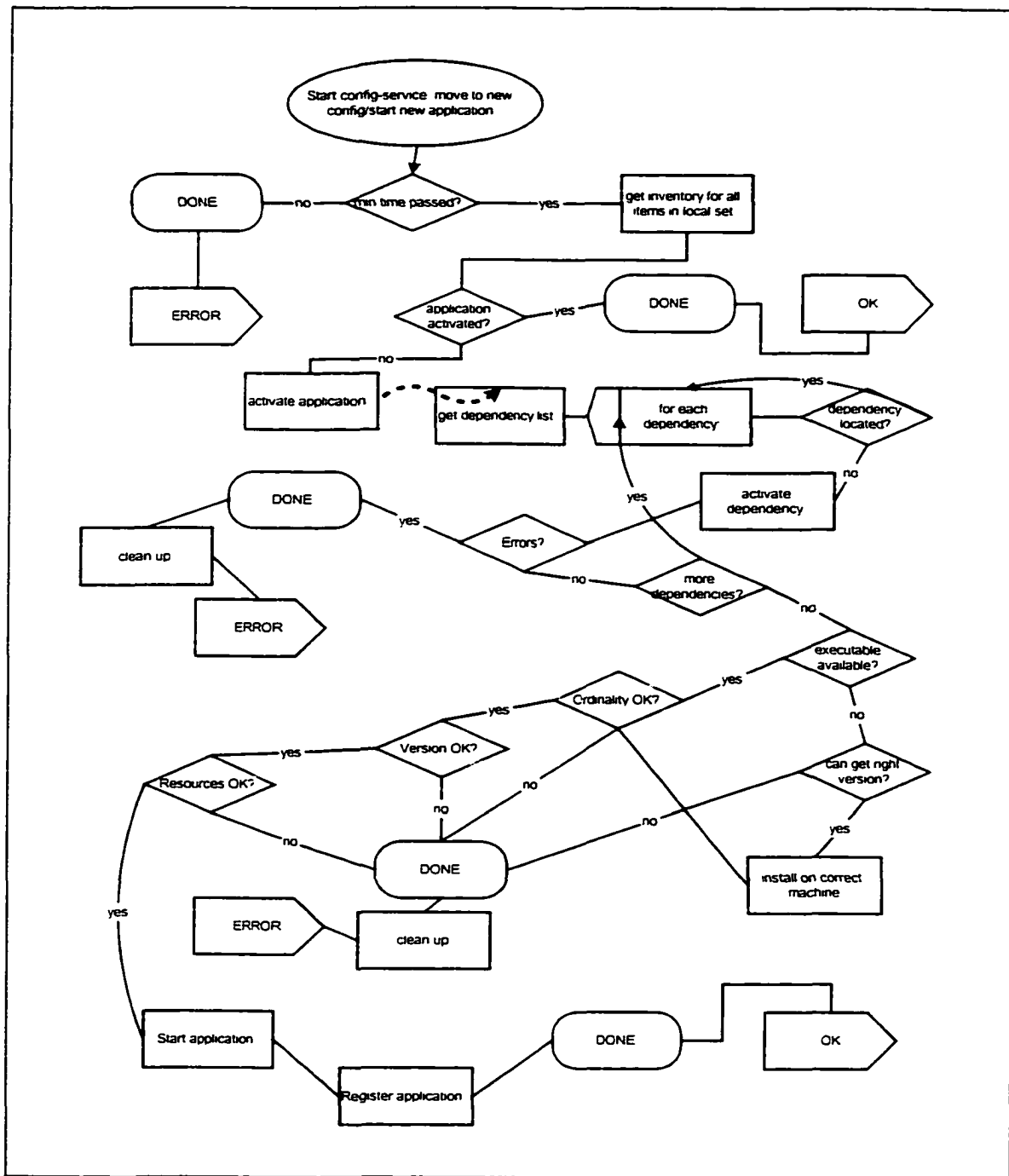


Figure TB-2. Actions needed to actually start an application as part of a move to a new desired configuration.

Note that these sequence constraints define what events must be observable in the final system in order to validate the sequence constraints. Some of these events are straightforward, for example the first, which requires that feedback always follows a

request to move to a new configuration and furthermore that the feedback must be OK or errors on a per-local set basis. Similarly, the second point is observable as an event provided that the error coming back from the attempt to start the object/service/or application is propagated to the user. The third point is much harder to show as observable events because during the process of clean up, there may be another item that has a dependency on one just started. In this case, the clean up needs to skip this item, but continue with other partially started items. Internal operation such as described by this third point are very difficult to validate with externally visible events; other mechanisms need to validate them to the point where their internal behavior can be abstracted up to some outwardly visible event. Once this has occurred, problem abstraction using constraint analysis may be a feasible validation mechanism.

APPENDIX 7C – USING COMPOSABILITY REACHABILITY TO VALIDATE THE ATM MONITORING SYSTEM

De-activating an application is an example of composability reachability analysis. There are many processes that may be involved, but they are of only two types. The first is the ConfigService process, which initiates the de-activation of an application. The second type is the target application, which in turn will try to de-activate all of its dependent applications, services, and/or objects. Note that if nothing else is depending on these dependencies, they will try to de-activate themselves, and try to de-activate their dependencies.

Figure 7C-1 shows the state transitions for the ConfigService process during de-activation. Note that there are many other state transitions in this process; only the ones having to do with de-activating an application are included. Thus, the labeled transition system only deals with the execution trace of interest.

Figure 7C-2 shows the state transitions for the application/service/or object that is being de-activated. Note that this is also the same labeled transition system as that of one of the dependencies of the item being de-activated. Once again, only the states and transitions associated with de-activation are shown.

The next step in composability reachability analysis is to compose the *Dependency* process at the lower states, collapsing states with transitions on the τ action. The resulting labeled transition system for this is shown in Figure 7C-3.

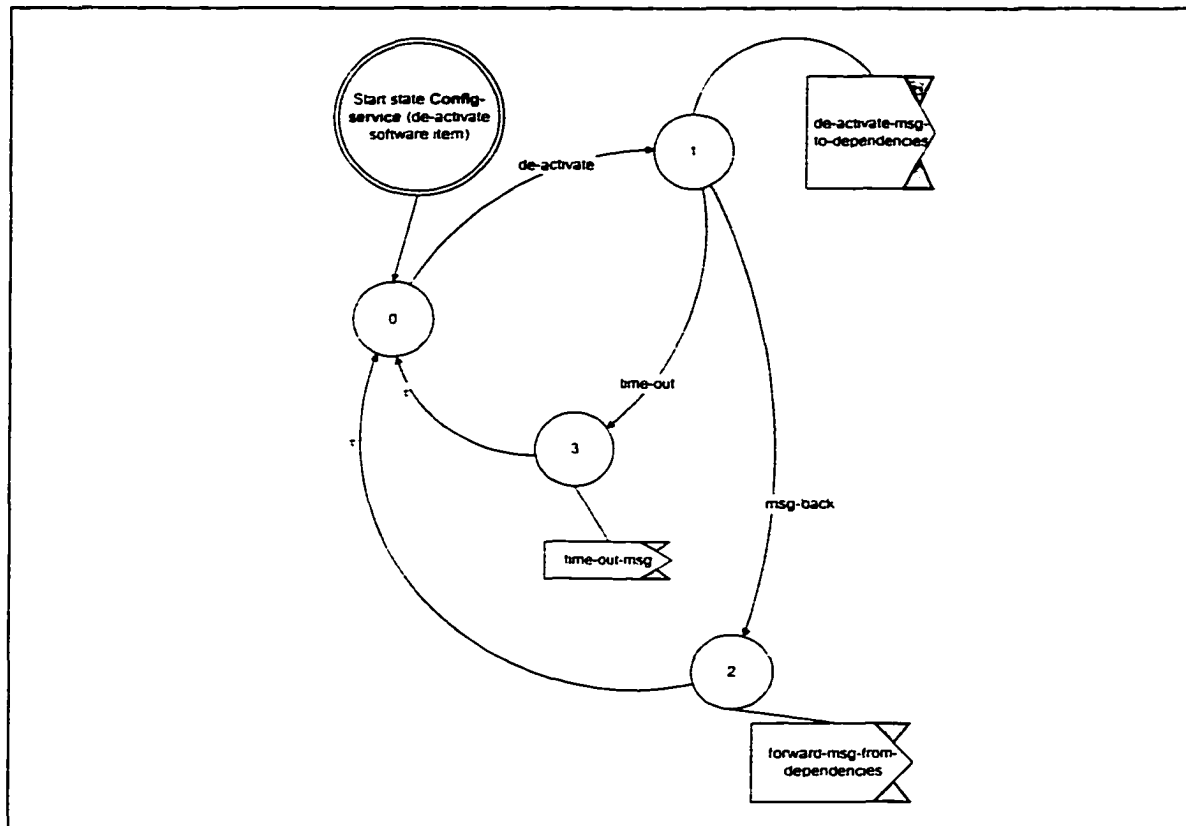


Figure 7C-1. A labeled transition system for the config-service process when de-activating another application/service/or object. The action τ stands for an internal transition within the process that is not effected by an outside action. Messages back to the process initiating the de-activation are shown as messages out of various states.

The composition of dependencies leads to the definition of the labeled transition system for an application, and this, composed with the labeled transition system of the config-service leads to the definition of the system for the purpose of de-activation. The composed system is shown in Figure 7C-4.

Additional state transitions help investigate errors in labeled transition systems. For example, it is a known aspect of the ATM monitoring system that system components and networks will be faulty, and at any given time some part of the system will not be functioning properly. It is possible that a request for a configuration change is received before the previous request has been completely implemented. One way to alleviate this problem is to impose a maximum frequency to the configuration change

requests that will be allowed. This “safety” property can be added to the lower levels of the labeled transition system as another automaton and used to determine if the system can handle this situation.

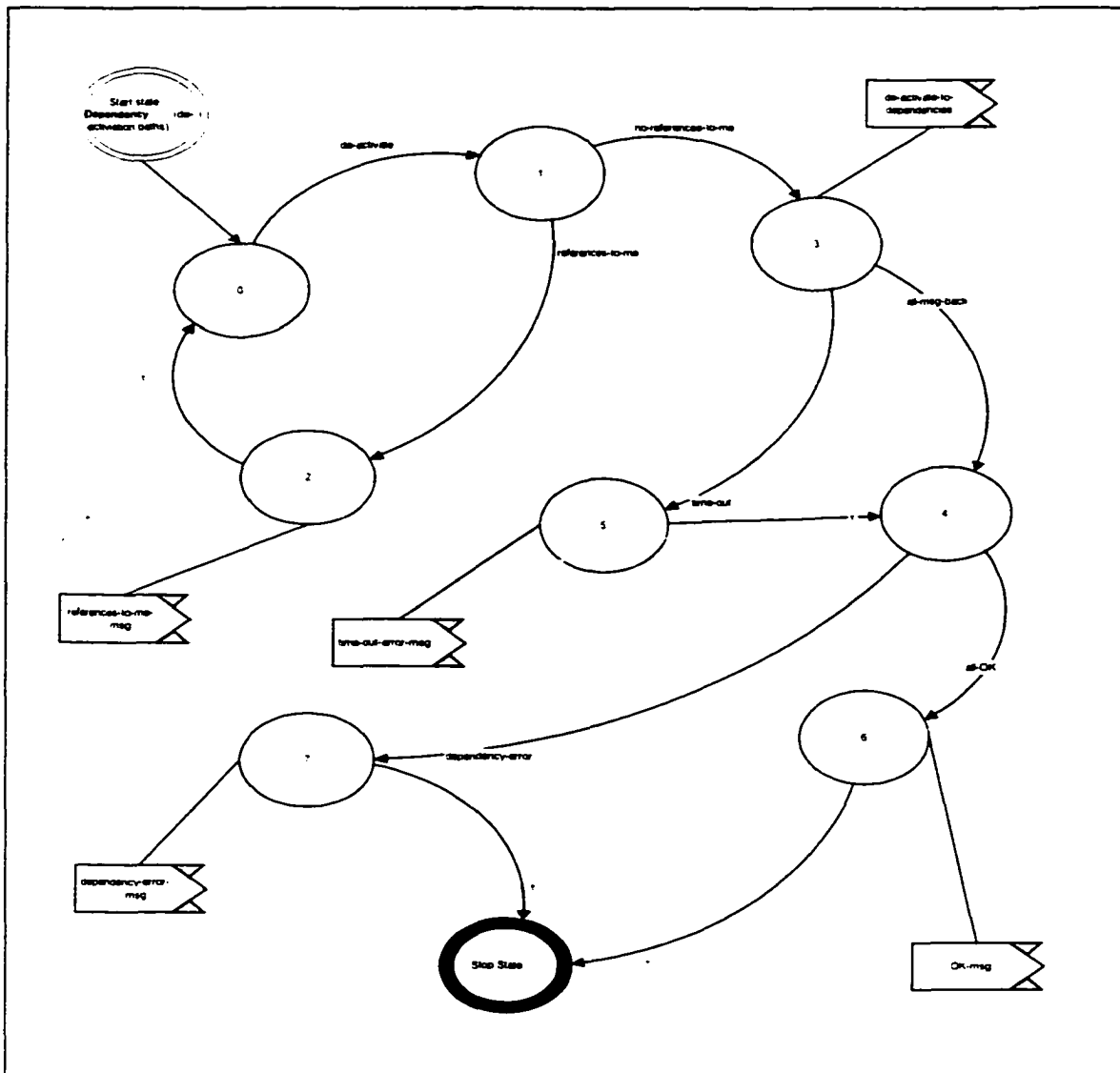


Figure 7C-2. A labeled transition system for software being de-activated. The action τ stands for an internal transition within the process. Messages travel back to items requesting de-activation; the items may be other software items or the ConfigService.

Figure 7C-5 shows a labeled transition system automaton for a time-between-changes property.

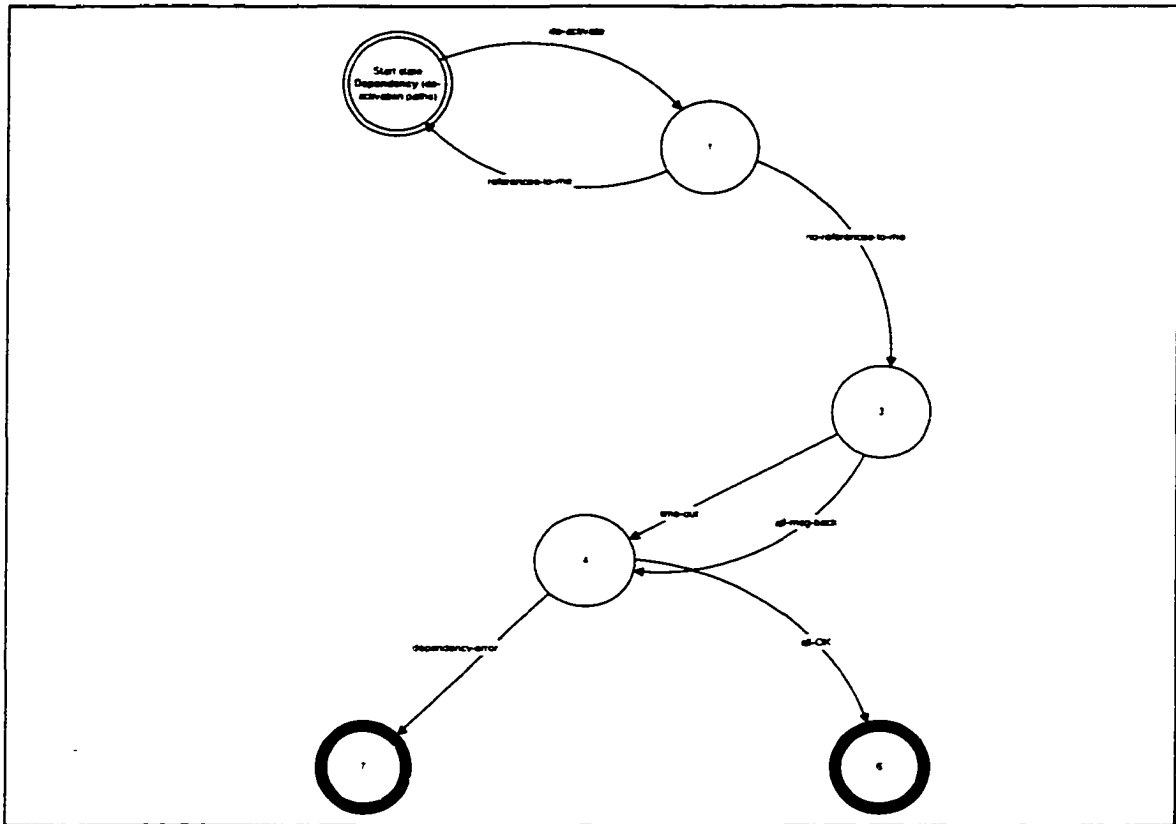


Figure 7C-3. A labeled transition system for the *Dependency* process. Note this is also the labeled transition system for a composition of dependency processes and the de-activation action.

The labeled transition system for the safety property composes with the composite labeled transition system. If there are any errors in the composed system, they will be expressed as a transition to the error state, π . The error state is entered when the minimum time has not passed between the previous configuration change request and the current request. The error occurs because there is no transition that covers it in the composed labeled transition system. The labeled transition system is shown in Figure 7C-6.

The errant portion of the system is changed to handle the condition that led to the error state. The revised ConfigService is shown in Figure 7C-7.

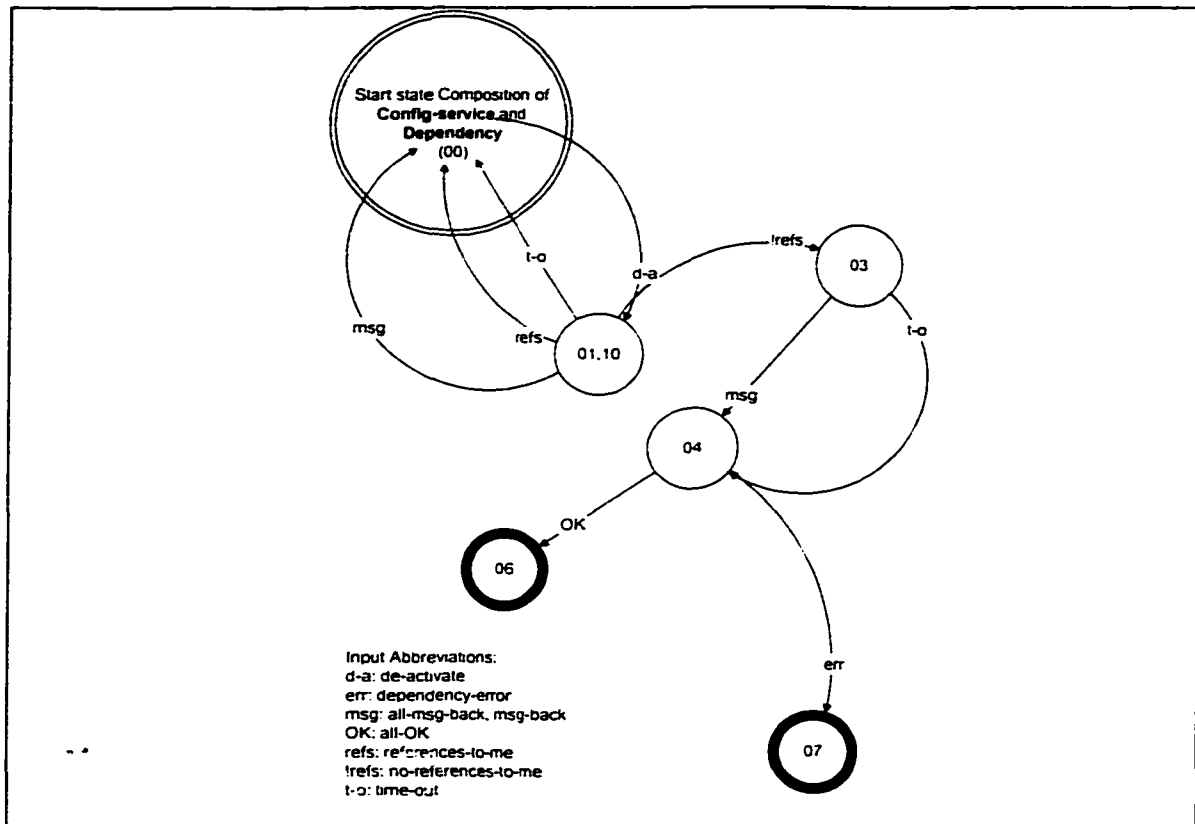


Figure 7C-4 The composition of dependencies, applications, and config-service processes into a system exhibiting the de-activation action.

The revised portion of the system is then recomposed first with the other parts of the system and then with the property of interest to see if the error condition has been fixed. The system no longer enters the error state π , so the labeled transition system is correct from the point of view of the time-between-changes property. This system is shown in Figure 7C-8.

A labeled transition system must be developed for each property of interest, and the transition system must be composed with the rest of the system that it impacts or that impacts it. Performing this task on a complete system is clearly not feasible, although the technique seems to work on the simplified version of the config-service shown in this discussion.

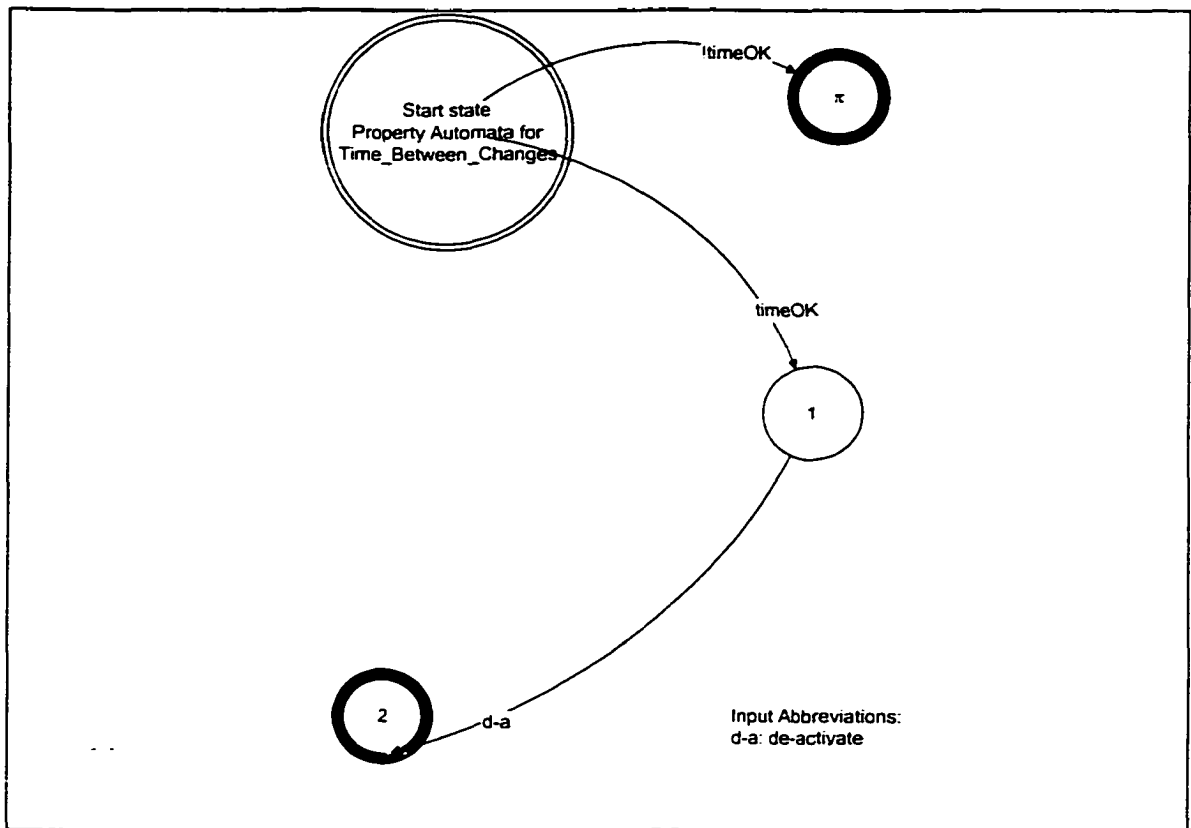


Figure 7C-5. The state transitions for the time-between changes property.

Observations

This technique may be more appropriate for design verification than for specification verification. However, it is shown here as a technique to verify that the invariant dealing with the minimum time between moves to a new configuration is validated. The labeled transition technique found an error in the original ConfigService shown in Figure 7C-1. The original ConfigService did not include the minimum time between configuration changes. This constraint was included in the Alloy model as a comment in the *change-config* operation. However, since Alloy does not include time capabilities, there is no way the Alloy tool can be used to test this timing constraint.

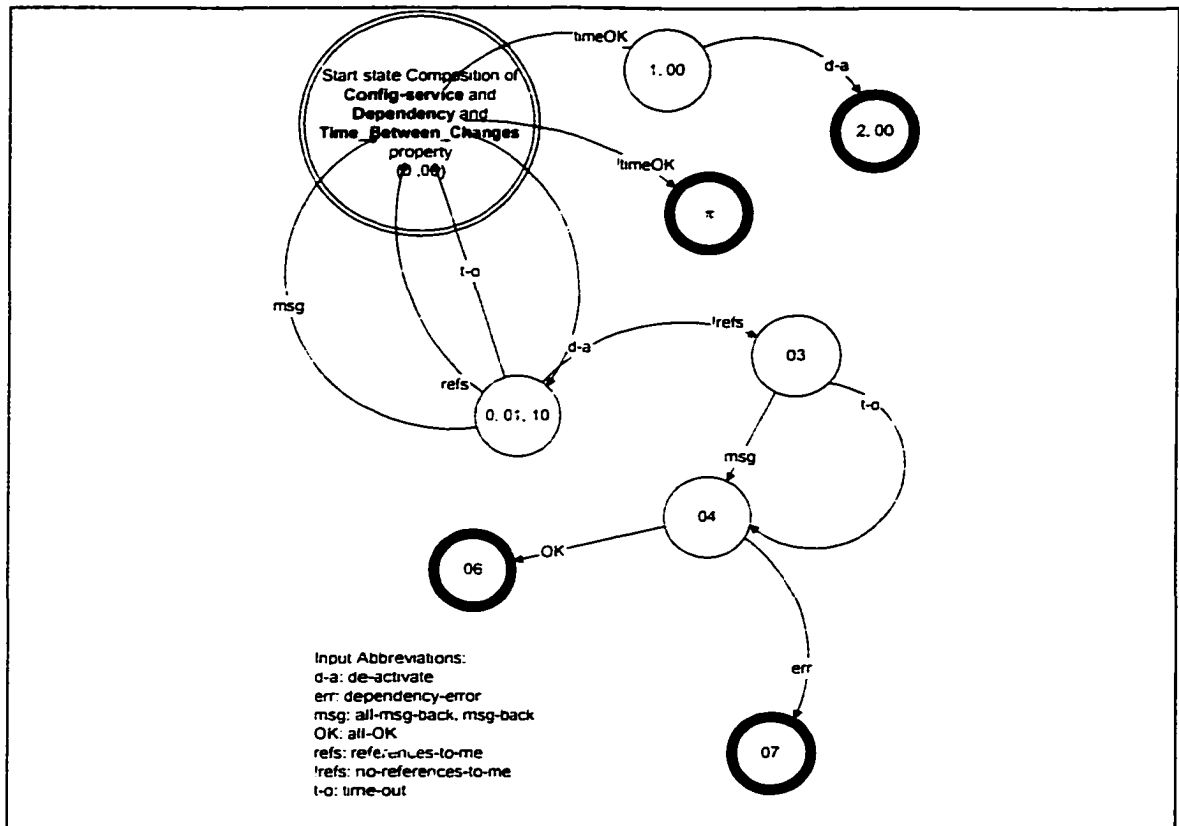


Figure 7C-6. Labeled transition system for the composition of the overall system with the time-between-changes property. Note that when not enough time has passed since the last configuration change request, the composition enters the error state, π .

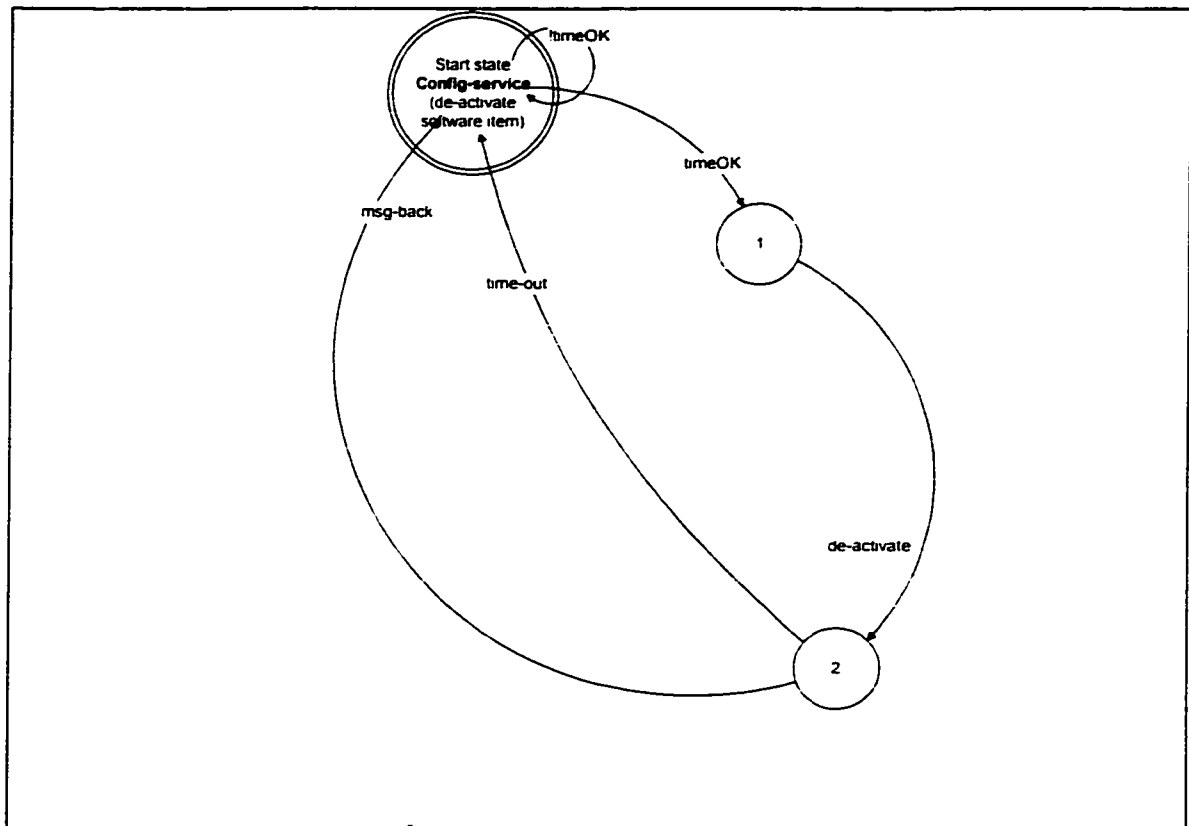


Figure 7C-7. Revised labeled transition system for the config-service de-activating another application/service/object.

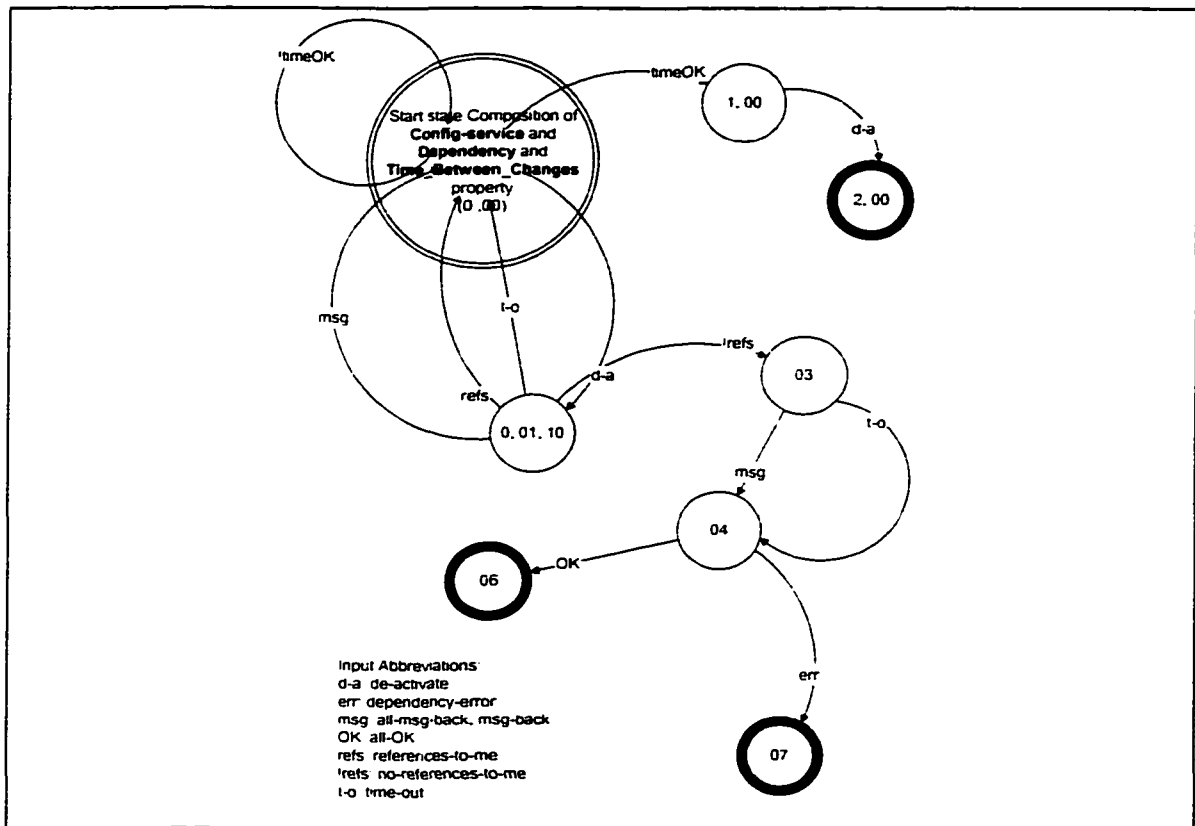


Figure 7C-8. Labeled transition system for the entire system composition with the time-between-changes property. Since the error state, π , does not exist in the diagram, the system is correct with respect to the time-between-changes property.

CHAPTER 8 - ATM MONITORING SYSTEM SOFTWARE RUN-TIME CONFIGURATION MANAGEMENT SPECIFICATION – VERSION 2

This chapter presents the revised software run-time configuration management specification, based on problems found during specification. (These problems are detailed in Chapter 7.) The differences between the revised specification and the original lie in the creation of a fault tolerant runnable item, the use of replicated repositories, inventories, registries, and the use of redundant ConfigServices.

A fault tolerant runnable item is used to completely separate the portions of the specification that are needed for fault tolerance from those needed for main-line execution of the system.³² In the ATM network monitoring system, all runnable items need to be fault tolerant; this is shown by the *{complete}* designation on the FaultTolerantRunnableItem type in Figure 8-1.

The revised *type* model based on UML/OCL is presented in Appendix 8A.

1. THE *TYPE* MODEL FIGURES

There are four *type* model diagrams associated with the ATM network monitoring system run-time configuration management specification of software items. The first shows the software items and the relations between them. This is shown in Figure 8-1. The second shows the relations between software items and other configuration-related concepts, such as states, resources, and versions. This is shown in Figure 8-2. Figure 8-3 shows the definition of a local set. Finally, the relation between

³² See [12] for a discussion of separating concerns in a specification, in the context of the TRIO language.

the software items and the configuration, inventory, and registry concepts is shown, along with the relation to machines, in Figure 8-4.

2. THE ALLOY MODEL

The Alloy model complements and extends the *type* model shown in Figures 8-1, 8-2, 8-3, and 8-4. Alloy complements the *type* models in terms of adding constraints and complete definitions to relations such as *dependsOn*. It extends the model by defining the post conditions of the operations and defining additional sets and relations needed to define those operations. The Alloy model also contains RTL-style temporal constraints in the form of comments (since Alloy does not include the notion of time). The complete Alloy model for the ATM network monitoring system's software run-time configuration management is shown below.

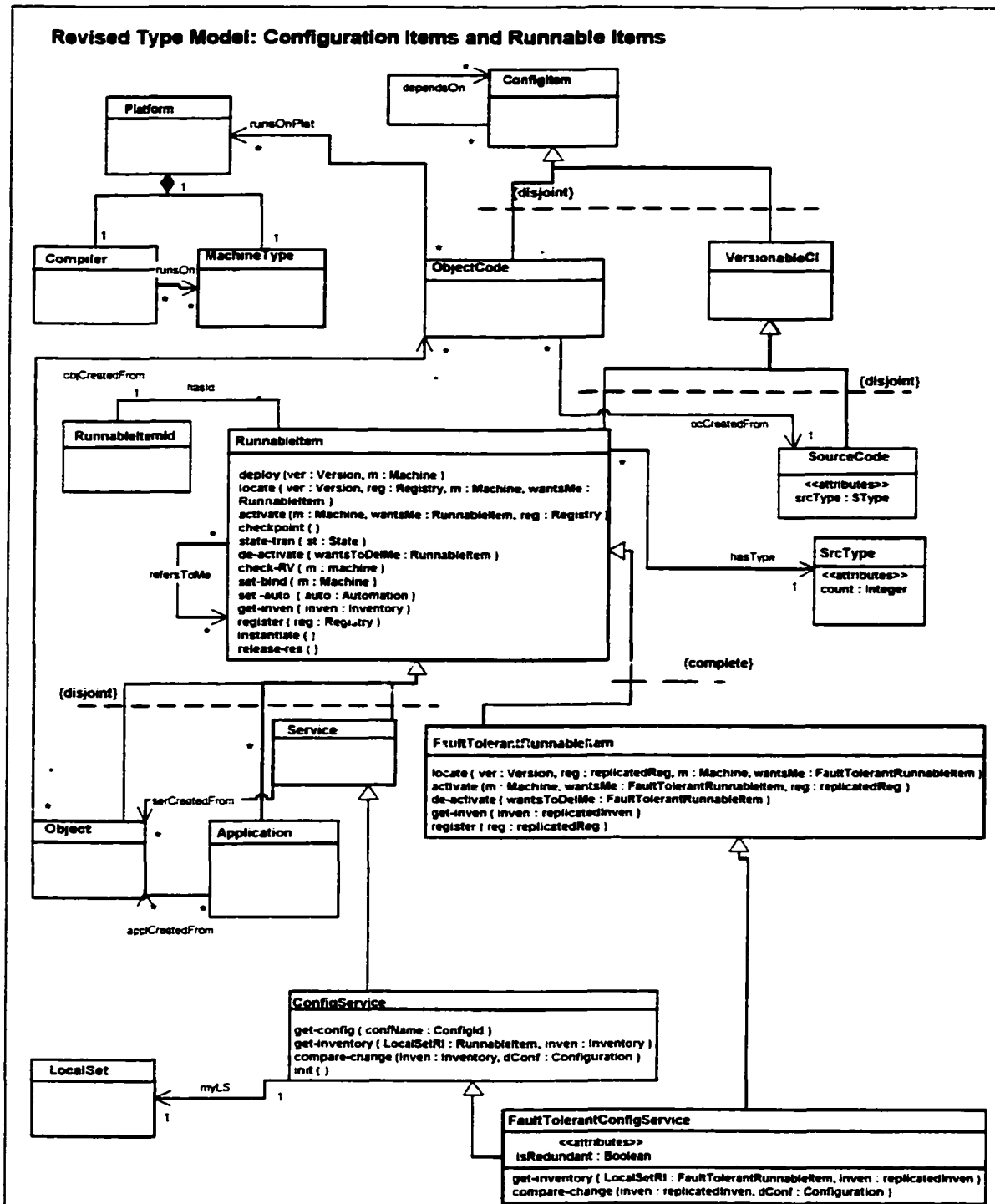


Figure 8-1. The software items and the relationships between them needed for run-time configuration management of the ATM/IP hybrid network monitoring system.

Revised Type Model: Runnable Items, States, Versions, and Resources

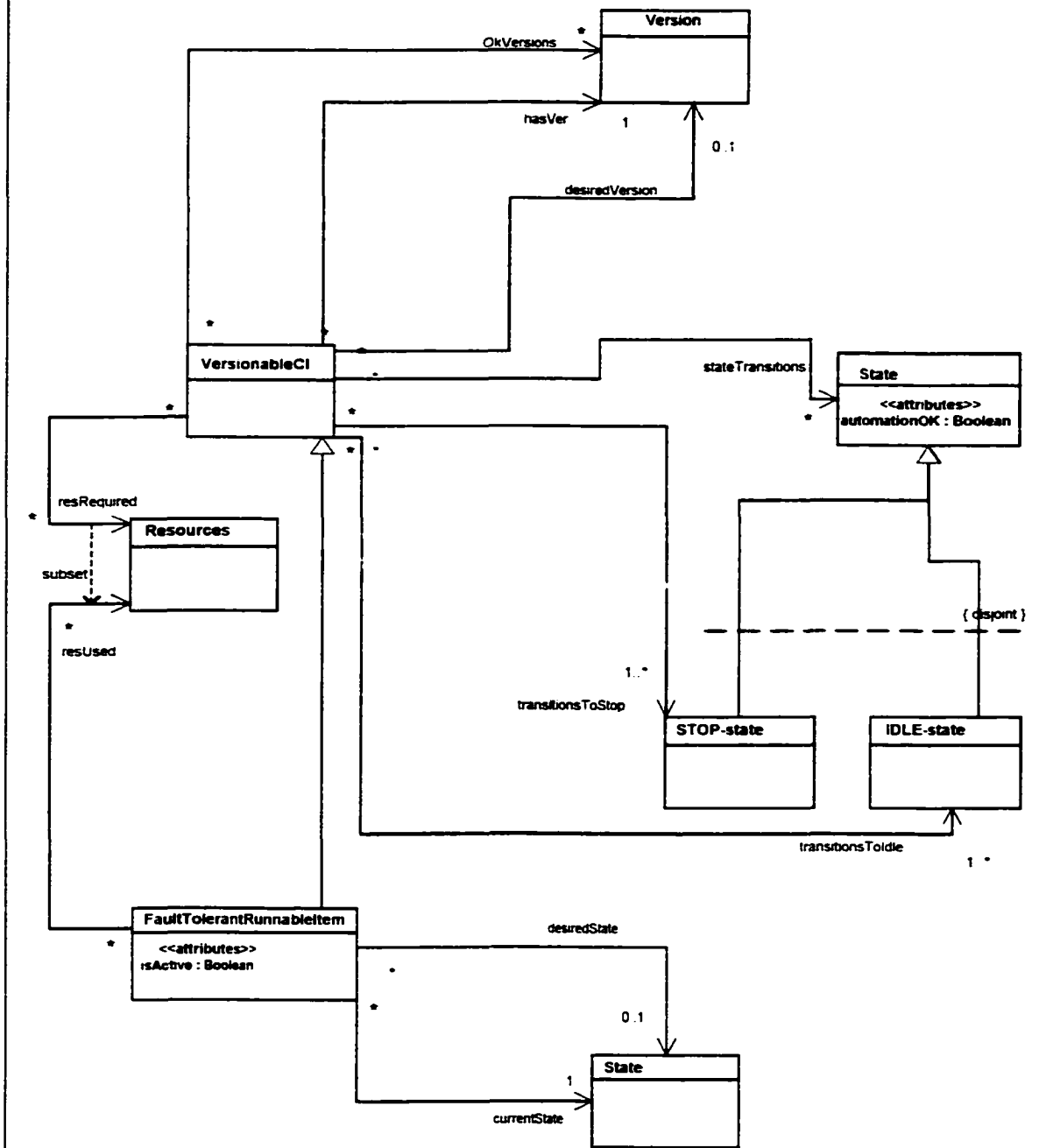


Figure 8-2. The software items and the relationships between them and versions, resources, and states.

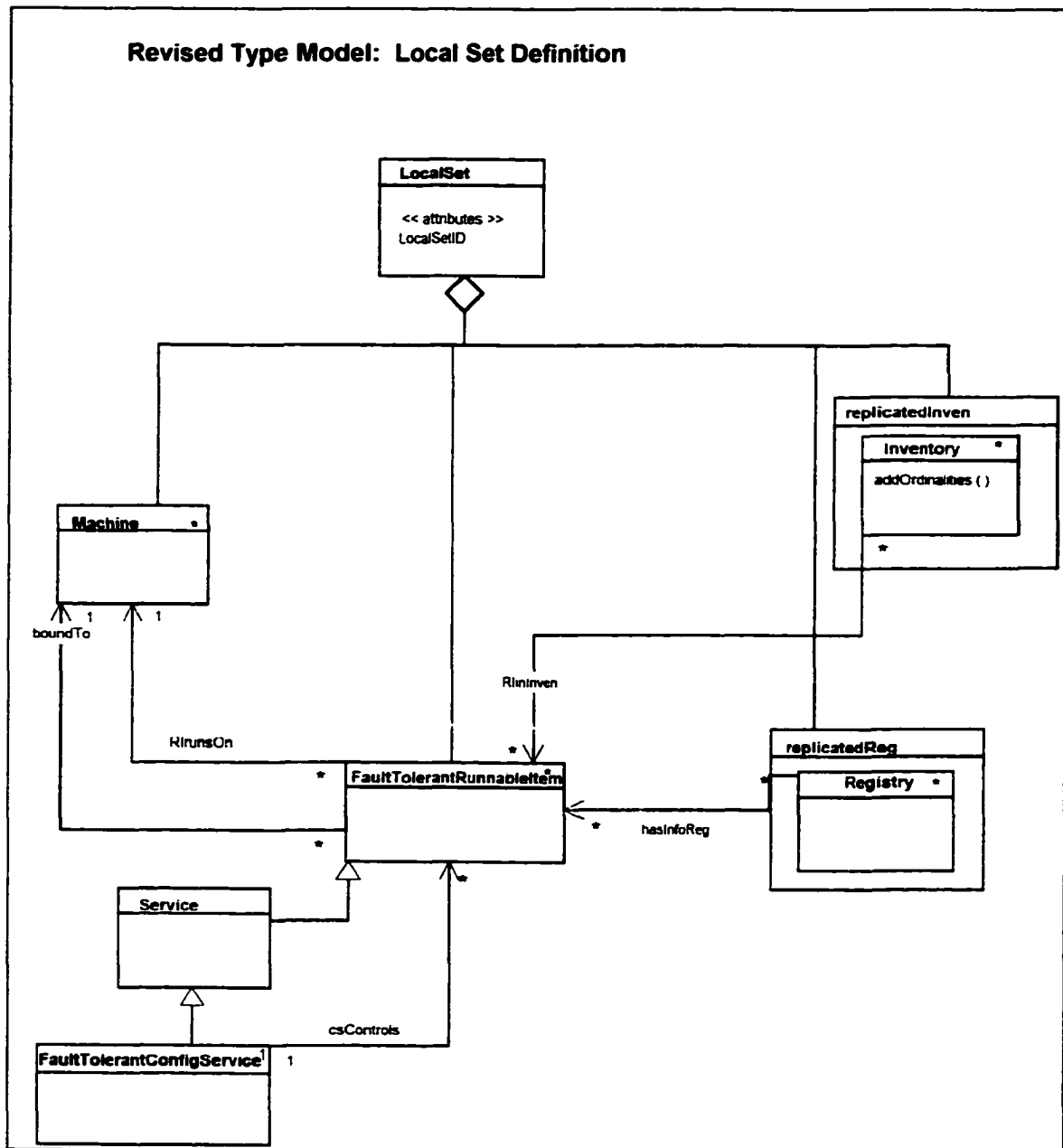


Figure 8-3. The definition of a local set.

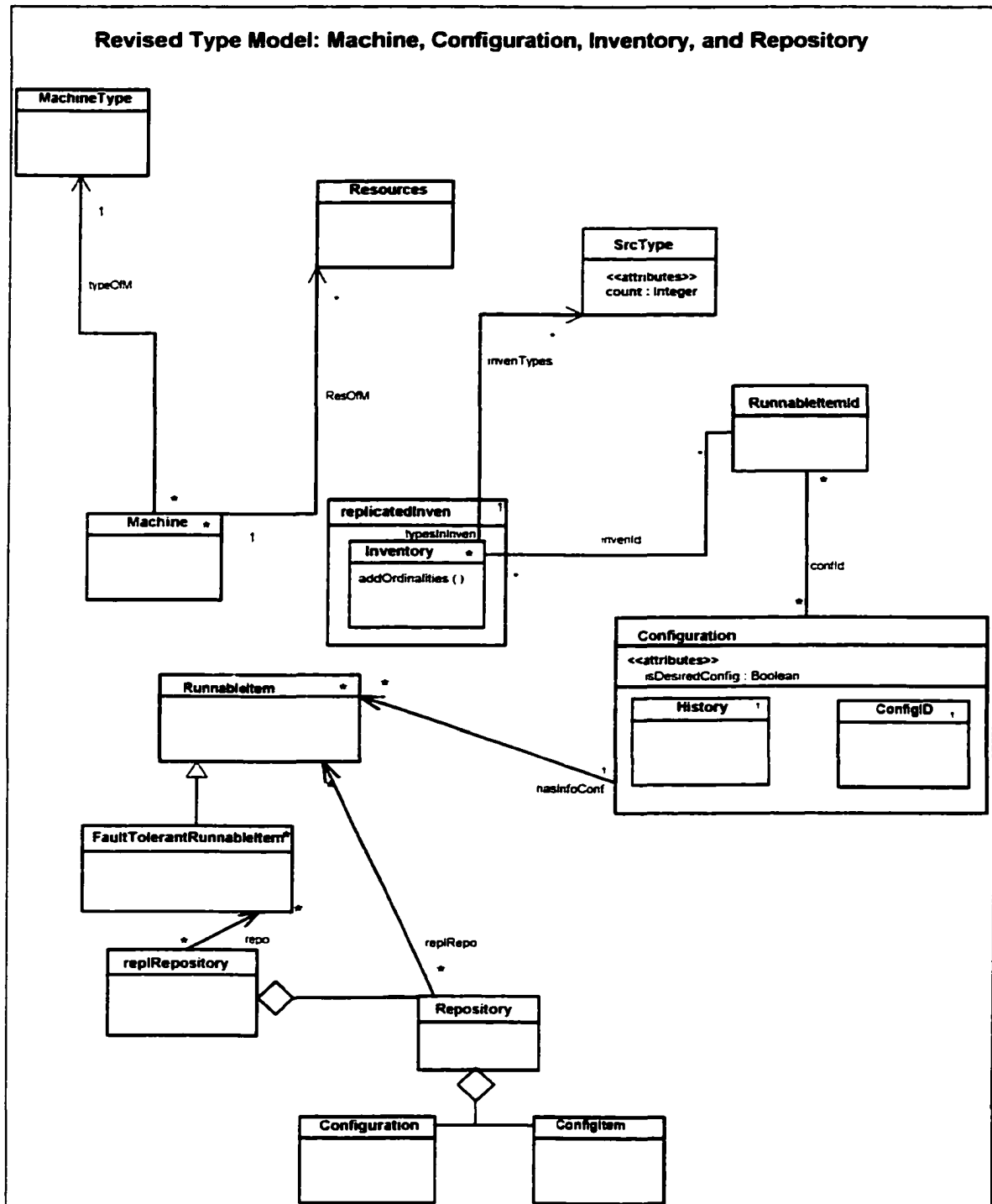


Figure 8-4. The executable software items and their relationships with the inventory, configuration, registry, and machines.

```

model ATM-network-monitoring-CM-fault-tolerant-version
{
// The format of this Alloy model is:
//
//   model ATM-monitor
//   {
//       domain { ... } – the domain-level sets used in the model
//       state { ... } – subsets of the domain-level sets and relations among sets
//       def { ... } – definitions of subsets and of relations
//       inv { ... } – invariants that constrain the state of the model
//       op { ... } – definitions of the operations that act on the model
//   }

// This model was created as an attempt to link an UML meta-model showing the abstract model of
// an ATM/IP network monitoring system's dynamic configuration needs to a specification that held
// more detail than just the UML diagrams. The hope was also that this Alloy model could be used to
// test some of the specification consistency, and thus be used to help verify/validate the specification.

// Start by looking at the UML diagrams, and figuring out which classes represent primary sets for Alloy.
// In this model, all the classes that have generalizations (e.g. most of the configuration items
// of 'ConfigItem', so 'ConfigItem' is a domain set for Alloy) fall into this category. So do the class
// components of things like 'Platform' (i.e. 'Compiler' and 'MachineType') since the sub-components
// do not have any relation among themselves that can be used for definition purposes.

// The resulting domain-level sets are:
// Machine, Resources, LocalSetId, Inventory, Configuration, History, ConfigId, ReplicatedRepository,
// Platform, Compiler, MachineType, ConfigItem, State, Registry, Automation, and Version.

// We add Integers as a domain set; things like Resources can be defined from the set of integers.
// Using them also allows the use of equalities where one item's Resources is equal to that of another.
// Boolean is also added; this is useful for relations such as when a runnable item is either active or not.

// We also have to add OpAttempted and ReturnValue as domain sets in order to model the feedback
// mechanisms necessary to the operations of the system. These are not shown in the UML diagrams,
// they need to be specified as part of the actual operations to provide consistency in the feedback
// mechanisms of each of these operations.

// The type Boolean is used for some of the UML attributes, and STypes is used to carry the types of
// sources, for use in figuring out the actual number of any given type of runnable item that is active.
// SType is also used along with RunnableItemId to identify runnable items that are of the same type.
// This is used for comparison purposes between information in an inventory about a runnable
// item and information in a desired configuration.

    domain { Machine, Resources, LocalSetId, Inventory, Configuration, History, ConfigId,
ReplicatedRepository, Platform, Compiler, MachineType, ConfigItem, State, Registry, Automation, Version,
Integers, OpAttempted, ReturnValue, Boolean, SType, RunnableItemId }

// Next define all the other sets and relations needed in the model. For the domain sets that correspond
// to classes with generalizations in the UML diagram, this simply means declaring subsets.

state
{
// The sub-classes of 'ConfigItem' partition this domain set.
// In addition, they are grouped in various ways to get everything but object code and runnable
// items. These groupings are defined in the definition section. For example, configuration items are
// source code, object code, objects, services, or applications. Runnable items are objects, services,
// or applications.

    partition SourceCode, ObjectCode, Object, Service, Application : ConfigItem
    VersionableCI : ConfigItem
    FaultTolerantRunnableItem : ConfigItem

// FaultTolerantConfigService is a subset of the Service set.

    FaultTolerantConfigService : Service

```

```
// States. There are two required states that must be a subset of all the states that exist in the system for
// any configuration item (except object code, which takes its states from its corresponding source code).
// These two required states can be defined as subsets of all the states. They can't be the same however,
// so this is added as an invariant later in the model ('inv reqSt').
```

```
STOP-state : State
IDLE-state : State
```

```
// Operations and return values. These are needed for feedback purposes in the operation definitions.
// The type of operations is a group of disjoint sets that partition all possible operations so that a type only
// corresponds to one operation. For return values, there is a set of OK return values and a set of possible
// error
// values. The error values are constrained by an invariant below so that they are not OK ('inv errNotOK').
```

```
partition deployed, located, instantiated, registered, bound, released, checkPt, transitioned,
inventoried, activated, de-activated, deActivated, setAuto, gotDC, checkedRes, checkedVer,
compared, deAct-a, setUB, addedOrd, initialized : OpAttempted
OK : ReturnValue
Error : ReturnValue
```

```
// Miscellaneous subsets useful in the Alloy model. The integer and boolean subsets are self-explanatory.
// The
// notBound subset of AllMachines is used in the 'boundTo' relation to indicate when a runnable item
// is not bound to any machine's resources. The automation values are used to determine if configuration
// changes can be made.
```

```
positiveIntegers : Integers
positiveNotZeroIntegers : positiveIntegers
true : Boolean !
false : Boolean !
notBound : Machine !
autoOK : Automation
```

```
// A desired configuration is a singleton subset of the configurations.
```

```
desiredConfig : Configuration :
```

```
// Replication and redundancy/mirroring are mechanisms used for dependability
// purposes. These are used for registries, inventories, repositories of
// the configuration items and for config services. The configuration item
// replicated repository is not mentioned further in this Alloy model since
// operation specifications do not include retrievals. The redundant/
// mirrored configuration service is also not used further in this model.
```

```
replicatedReg : Registry
replicatedInven : Inventory
redundantConfigService : FaultTolerantConfigService
```

```
// Next define the relations.
```

```
// Relations needed to define local sets: 'on' and 'controlledBy' (inversely, 'controls'). The 'on' relation maps
// runnable items to the machines they are on, and 'controlledBy' maps them to the configuration
// service that controls them. The 'controls' relation is the mapping the other direction, from a
// configuration
// service to all the runnable items it controls. The 'controlledBy' relation is further
// defined below, in terms of the 'on' relation as well as the LocalSetId common to all the machines in the
// local set.
```

```
controlledBy (~controls) : FaultTolerantRunnableItem -> FaultTolerantConfigService !
on (~riOn) : FaultTolerantRunnableItem -> Machine !
```

```
// A LocalSet is a group of runnable items. They all are controlled by a single configuration service,
// and are all on machines that share a common LocalSetId
// This definition is shown below in the definition section of the model.
```

```
LocalSetFTRI! (~IsFTRI) : FaultTolerantRunnableItem -> LocalSetId !
```

```

LocalSetM (~IsM) : Machine -> LocalSetId !
LocalSetI (~IsI) : Inventory -> LocalSetId !
LocalSetR (~IsR) : Registry -> LocalSetId !

// Relations among the configuration items. These are from the UML diagram, the 'dependsOn' and
// 'createdFrom' relations. They have additional definitions shown in the definition section.

dependsOn : ConfigItem -> ConfigItem
createdFrom : ConfigItem -> ConfigItem

// Relations among the configuration items and their attributes. These are from the UML diagram:
// 'requires', 'uses', 'transitionsTo', 'developedToRunOn', 'hasState', 'hasAutomation',
// 'hasOrdinality', and 'hasVersion'.
// Note that the 'hasAutomation' relation is between all the states a runnable item can move
// to and the level of acceptable automation for moving among them. The 'transitionsTo' relation
// needs an invariant to ensure that the required states are part of this set of states the configuration
// item can be in (part of 'inv reqSt'). The uses and requires relations need an invariant to ensure
// a runnable item does not use more resources than it requires ('inv resourceUsage' below).
// The srcTypes relation is added to aid in the identification of current ordinality of runnable items,
// needed by the compare-change operation of the ConfigService. The idea is that the createdFrom
// relation can be used to get to the source code actually used to create a runnable item, then the
// srcTypes relation can be used to compare the types of the sources. If the set of types is the same,
// then the runnable items are the same type, and their ordinality can be 'added'. Since Alloy only
// supports sets, the sets will actually be unioned.
// The relations riType and riHasId are used to identify runnable items that are the same in that their
// sources create the same kind of runnable item, but they may differ in terms of their current state,
// ordinality, or version. This situation occurs when a runnable item is represented in an inventory
// and the configuration service has to determine if it is in the proper state, has the proper version, or
// has the proper ordinality as required by a desired configuration. Just comparing the complete
// runnable item to see if it is the same won't work since the relations may not be the same between
// the two descriptions of the runnable item. Using an identifier that is the same, based on the source
// type (SType) that is also the same allows such comparisons to be made.

requires [Resources] : VersionableCI -> Integers
uses [Resources] : FaultTolerantRunnableItem -> Integers
transitionsTo : VersionableCI -> State
developedToRunOn : ObjectCode -> Platform
hasState : FaultTolerantRunnableItem -> State !
hasAutomation : State -> Automation !
hasVersion : VersionableCI -> Version !
hasOrdinality : FaultTolerantRunnableItem -> Integers !
isActive : FaultTolerantRunnableItem -> Boolean !
srcType : SourceCode -> SType
riType : FaultTolerantRunnableItem -> SType
riHasId (~ idOfRI) : FaultTolerantRunnableItem -> RunnableItemId !

// More relations between the configuration items and attributes. These are not part of the UML
// diagrams but are needed to specify the operations. In particular, for changing configurations,
// things like the desired ordinality, the maximum allowed ordinality, desired version, desired state,
// desired automation, and a set of compatible versions are needed.
// The set of compatible versions for a single
// configuration item is defined by the relation 'okVersions'. This relation is used to define a set
// of compatible configuration items. (That definition
// is given below.)

desiredOrdinality : FaultTolerantRunnableItem -> Integers !
allowedOrdinality : FaultTolerantRunnableItem -> Integers !
desiredVersion : VersionableCI -> Version !
desiredState : FaultTolerantRunnableItem -> State !
desiredAutomation : State -> Automation !
okVersions : VersionableCI -> Version

// Relations among sets of configuration items and other classes. These are from the UML diagram:
// 'hasRegInfo' to map from a registry to the runnable items it knows
// about.

```

```

// 'hasConfigInfo' to map from a configuration to the runnable items it describes.
// 'hasInvenInfo' to map from an inventory to the runnable items it knows about (further defined below).
// 'hasMType' to map from a machine
// to the type it is.
// 'hasInventId' to map an inventory to the local set it represents. and
// 'haveRes' to map from machines to their physical resources.

hasConfigInfo : Configuration ! -> FaultTolerantRunnableItem
hasInvenInfo : Inventory -> FaultTolerantRunnableItem
hasMType : Machine -> MachineType !
haveRes : Machine -> Resources

// Relations among other classes on the UML diagram: 'runsOn'. This relation maps compilers to
// the type of machine they run on. Note that any particular compiler might run on many types of
// machines.
// To finish the definition of a platform, there must be a mapping from a platform to a compiler and
// platform to machine type. These last relations are not on the UML diagrams, since UML has the notion
// of aggregation whereas Alloy does not support the notion of tuples of greater arity than 2.

runsOn : Compiler -> MachineType
platformC : Platform -> Compiler !
platformMT : Platform -> MachineType !

// Relations to set up a configuration. The relation to associate configuration items with a configuration is
// shown on the UML diagram. However, there are no relations shown to associate a history or
// configuration
// identifier with a configuration (since they are part of a composition), so these must be added to the
// Alloy model.

configHist : Configuration -> History !
configId : Configuration ! -> ConfigId !

// Relations for operation feedback: FeedBack

FeedBack : OpAttempted -> ReturnVal

// Relations for keeping track of runnable items and their references : hasRegInfo, refersToMe,
// csReg. The 'refersToMe' relation keeps track of all the other runnable items that refer to any particular
// runnable item. This is used to figure out when it is safe to deactivate a runnable item. The
// 'csReg' relation is used to get to registries used by a configuration service.

hasRegInfo (~registeredIn) : Registry -> FaultTolerantRunnableItem
refersToMe : FaultTolerantRunnableItem -> FaultTolerantRunnableItem
csReg : ConfigService ! -> Registry

// Relations for resource usage. These do not appear directly on the type model. The 'available' relation
// is defined below as the machine resources 'minus' the resources being used by all runnable items
// on a particular machine. 'Minus' is used in the set sense, not in the mathematical sense. The
// type model shows an available attribute associated with machine resources.

available [ Resources ] : Machine -> Integers !
boundTo : FaultTolerantRunnableItem -> Machine !

}

// Next add definitions for subsets and relations.

// Define all config items except object code (needed since this is a subset that has similar attributes
// of version and resources, whereas object code doesn't have these relations directly), and the runnable
// config items (objects, services, and applications).

def VersionableCI { VersionableCI = SourceCode + Object + Service + Application }
def FaultTolerantRunnableItem { FaultTolerantRunnableItem = Object + Service + Application }

// A LocalSet consists of all the runnable items that are on machines that share a common LocalSetId.

```



```

// plus the configuration service that controls them (also running on a machine with the same LocalSetId).
// All these runnable items must be controlled by the one configuration service, and it cannot control
// any runnable items that are running on machines that do not share the common LocalSetId. These
// latter two constraints are stated as invariants below ('inv oneCScontrols', 'inv oneCSControlsAll').

def LocalSetFTRI { all fTRI | fTRI.LocalSetFTRI = fTRI.on.LocalSetM }

// The definition of the 'controlledBy' relation. A runnable item and its controlling configuration service
// are in the same local set.

def controlledBy { all fTRI : FaultTolerantRunnableItem | one fTCS : FaultTolerantConfigService |
fTRI.controlledBy = fTCS -> fTRI.on.LocalSetM = fTCS.on.LocalSetM }

// The relation 'hasInvenInfo' maps an inventory to the runnable items it knows about. These are
// defined as all those in the local set that shares the same LocalSetId as the inventory itself.

def hasInvenInfo { all inven | all fTRI : FaultTolerantRunnableItem | fTRI in inven.hasInvenInfo ->
fTRI.LocalSetFTRI = inven.LocalSetI }

// The dependency relation definition.

def dependsOn { all ci | ( no ci1 : ConfigItem | ci.dependsOn = ci1 )
| | ( ci in SourceCode -> some src : SourceCode | ci.dependsOn = src && ! (src.dependsOn = ci) )
&& ( ci in ObjectCode -> some oc : ObjectCode | ci.dependsOn = oc && ! ( oc.dependsOn = ci ) ) && (
ci in Object -> some obj : Object | ci.dependsOn = obj && ! ( obj.dependsOn = ci ) ) && ( ci in Service ->
some ser : Service | ci.dependsOn = ser && ! ( ser.dependsOn = ci ) ) | | ( ci in Application -> some obj : Object |
ci.dependsOn = obj && ! ( obj.dependsOn = ci ) ) && ( ci in Application -> some ser : Service |
ci.dependsOn = ser && ! ( ser.dependsOn = ci ) ) | | ( ci in Application -> some obj : Object |
ci.dependsOn = obj && ! ( obj.dependsOn = ci ) ) )
}

// The 'createdFrom' relation.

def createdFrom { all ci | ( ci in SourceCode -> ci.createdFrom = ci ) && ( ci in ObjectCode -> one src :
SourceCode | ci.createdFrom = src ) && ( ci in Object -> some oc : ObjectCode | ci.createdFrom = oc )
&& ( ci in Service -> some obj : Object | ci.createdFrom = obj ) && ( ci in Application -> some obj : Object
| ci.createdFrom = obj ) }

// The 'available' relation. Available resources on a machine are those on the machine 'minus' (in the
// set sense, not in the mathematical sense) the resources being used by runnable items on the machine.
// This definition gives available as the maximum resources on the machine if there are no runnable
// items currently running on the machine.

def available { all m | all fTRI : FaultTolerantRunnableItem, res : Resources | fTRI.on = m && m.available [ res
] = m.available [ res ] - fTRI.uses [ res ] }

// The 'okVersion' relation. An item in the set defined by this relation is part of the dependency
// relation of configuration items (except object code) and has a version that is compatible with
// the configuration item.

def okVersions { all vCI | all notOCI : ( vCI.dependsOn ) | some ver : Version | ver in notOCI.okVersions }

// The riType relation.

def riType { all fTRI | fTRI.riType = fTRI.+createdFrom.srcType }

// Next add the invariants.

// The invariant that runnable items can have the same runnable item identifier if they have the same
// source types.

inv sameRiid { all fTRI1, fTRI2 : FaultTolerantRunnableItem | ( fTRI1 != fTRI2 && fTRI1.riHasId = fTRI2.riHasId ) ->
fTRI1.riType = fTRI2.riType }

// The invariant that all runnable items in a local set are controlled by its one configuration service.
// The invariant that there is only one configuration service in a local set and

```

```

// it controls all runnable items in the local set.

inv oneCSControls { all ftCS1, ftCS2 : FaultTolerantConfigService | no ftRI : FaultTolerantRunnableItem |
ftRI in ftCS1.controls & ftCS2.controls && ! ( ftCS1 = ftCS2 ) }
inv oneCSControlsAll { all lsid : LocalSetId | one ftCS : FaultTolerantConfigService | ftCS in lsid.IsFTRI &&
lsid.IsFTRI = ftCS.controls }

// An invariant that populates some of the configuration item sets so the createdFrom relation can be
// properly built.

// NEED TO SET SCOPE TO AT LEAST 5 FOR CONFIG ITEMS
inv sets { some ci1, ci2, ci3, ci4 : ConfigItem | ci1 in Service && ci2 in Object && ci3 in ObjectCode &&
ci4 in SourceCode }

// The invariant that error return values cannot be 'OK'.

inv errNotOK { all rv : ReturnValue | rv in Error -> ! ( rv in OK ) }

// The invariant that the required states (IDLE and STOP) are ones that every configuration item
// (except object code) can transition to, and that they are not the same state.

inv reqSt { all vCI : VersionableCI, stop : STOP-state, idle : IDLE-state | stop in vCI.transitionsTo && idle in
vCI.transitionsTo -> ! ( stop = idle ) }

// A configuration item that is not object code cannot use more resources than it requires. So the
// requires set must always be greater than or equal to the uses set.
// Note that Alloy does not support Integers, so this statement cannot really be validated.
// The new version of Alloy support set cardinality testing, so a more proper statement will be to simply test
// the relative cardinality of the sets.

inv resourceUsage { all ftRI : FaultTolerantRunnableItem, res : Resources | some int : positiveIntegers | int
= ftRI.requires [ res ] - ftRI.uses [ res ] }

// Next! add the operations defined in the UML diagrams.
//
// For RunnableItem, these are: check-RV, set-bind, release-res, register, instantiate, deploy, locate,
// activate,
// checkpoint, state-tran, de-activate, set-auto, and get-inven.
// Additional operations that had to be defined due to limitations in the Alloy model are:
// check-Res and check-Ver to replace check-RV, set-unbind-bind and deAct-act to make up for lack of
// sequencing in Alloy, and deActivate to make up for lack of recursion in Alloy.
//
// For ConfigService, these are: get-config, get-inventory, compare-change, and init-auto.
//
// For Inventory, the only operation is addOrdinalities.

// *** RunnableItem Operations ***

// An operation to check the resources needed by a runnable item.
// In this operation, we need to check that there are enough resources on the machine in question
// for the runnable item.

op check-Res ( ftRI : FaultTolerantRunnableItem !, m : Machine ! ) {
  all res : Resources | some int : positiveIntegers | int in m.available [ res ] - ftRI.requires [ res ]
  checkedRes.FeedBack' in OK
}

// An operation to check the version of a runnable item.
// In this operation, we need to check the version of the runnable item to make sure it is compatible with
// the
// other runnable items in its dependency set.

op check-Ver ( ftRI : FaultTolerantRunnableItem ! ) {
  all depenRI : ( ftRI.dependsOn ) | some ver : ( depenRI.hasVersion ) | ver in depenRI.okVersions
  checkedVer.FeedBack' in OK
}

```

```

// An operation to set bindings for a runnable item.
// An assumption is that we've already checked to make sure there are enough resources on the
// machine via the check-Res operation.

op set-bind ( ftrI : FaultTolerantRunnableItem !, m : Machine ! ) {
  all res : Resources | m.available' [ res ] = m.available [ res ] - ftrI.requires [ res ]
  ftrI.boundTo' = m
  bound.FeedBack' in OK
}

// An operation to release resources.
// This operation releases all the resources it is using on the machine it is running on.

op release-res ( ftrI : FaultTolerantRunnableItem ! ) {
  all res : Resources | ftrI.on.available' [ res ] = ftrI.on.available [ res ] + ftrI.uses [ res ]
  ftrI.boundTo' = notBound
  released.FeedBack in OK
}

// An operation to release resources and re-bind them.
// This is needed since Alloy does not support sequencing; when the version of a runnable item
// needs to change the old version needs to get deactivated and the new one activated. This includes
// releasing the resources of the old version and binding the resources of the new version.

op set-unbind-bind ( ftrI : FaultTolerantRunnableItem !, m : Machine ! ) {
  all res : Resources | ftrI.on.available' [ res ] = ftrI.on.available [ res ] + ftrI.uses [ res ] && m.available' [
res ] = m.available [ res ] - ftrI.requires [ res ]
  ftrI.boundTo' = m
  setUB.FeedBack in OK
}

// An operation to register a runnable item in a registry.

op register ( ftrI : FaultTolerantRunnableItem, reg : replicatedReg ) {
  reg in ftrI.registeredIn'
  registered.FeedBack in OK
}

// An operation to instantiate a runnable item.

op instantiate ( ftrI : FaultTolerantRunnableItem ! ) {
  ftrI.hasState' = IDLE-state
  instantiated.FeedBack' = OK
}

// An operation to deploy a runnable item.
// This means put the right version of a configuration item on the right machine. The latter means that
// we also have to make sure that the platform associated with the thing we're putting on matches the
// machine type of the machine we're putting it onto. We can do this by using the createdFrom relation
// one or more times to get the object code associated with the runnable item, then getting to its platform
// using the developedToRunOn relation, getting to the platform machine type with the platformMT
// relation. This final machine type needs to match that of the machine we're deploying to. At the
// end, set the feedback mechanism.

// This operation is one that needs a replicated repository for dependability purposes. This might be
// needed if there are problems finding the correct version of the runnable item. It is possible that
// the operation may have to go to a repository to find the runnable item, its objects, object code, or
// source code and then rebuild the runnable item.
// There isn't a way to put that search in this specification since it only declares the end result, that the
// correct type of runnable item is in the correct location.

op deploy ( ftrI : FaultTolerantRunnableItem !, ver : Version !, m : Machine ! ) {
  m.hasMType in ftrI.+createdFrom.developedToRunOn.platformMT
  ftrI.hasVersion = ver
  ftrI.on' = m

```

```

    deployed.FeedBack' in OK
  }

// An operation to locate a runnable item.
// For this operation, the item may be in the replicated registries passed in.
// The end result of this operation is that the runnable item is on the correct machine, its resources
// are bound, and it is instantiated and registered.
//
// This operation could be written so that it 'calls' deploy, check-Res, etc., but the Alloy tool appears to
// hang if this is done. Re-writing the post conditions of each operation allows the tool to operate
// properly.

op locate ( ftrl : FaultTolerantRunnableItem !, ver : Version !, reg : replicatedReg, m : Machine !, wantsMe
: FaultTolerantRunnableItem ! ) {

  // From deploy...
  m.hasMType in ftrl.+createdFrom.developedToRunOn.platformMT
  ftrl.hasVersion = ver
  ftrl.on' = m

  // From check-Res
  all res : Resources | some int : positiveIntegers | int in m.available [ res ] - ftrl.requires [ res ]

  // From check-Ver
  all depenRI : ( ftrl.dependsOn ) | some vers : ( depenRI.hasVersion ) | vers in depenRI.okVersions

  // From set-bind
  all res : Resources | m.available' [ res ] = m.available [ res ] - ftrl.requires [ res ]
  ftrl.boundTo' = m

  // From instantiate
  ftrl.hasState' = IDLE-state

  // From register
  reg in ftrl.registeredIn'

  ftrl.refersToMe' = ftrl.refersToMe + wantsMe
  located.FeedBack in OK
}

// An operation to activate a runnable item.
// In this operation, all dependencies first have to be located. Then the runnable item can be deployed,
// available resources checked, and the runnable item bound, instantiated, and registered. The set of
// items referring to the new one is augmented as well.
//
// This operation could be written so that it 'calls' deploy, check-Res, etc., but the Alloy tool appears to
// hang if this is done. Re-writing the post conditions of each operation allows the tool to operate
// properly.

op activate ( ftrl : FaultTolerantRunnableItem !, m : Machine !, wantsMe : FaultTolerantRunnableItem !,
reg : replicatedReg ) {
  all depenRI : ( ftrl.dependsOn ) | some vers : ( depenRI.okVersions ) | locate ( depenRI, vers, reg, m,
ftrl )

  // && all i : integer, t1 : Time | theta ( begin ( locate ), i, t1 )
  //      -> one t2 : Time | theta ( end ( locate ), i, t2 ) ) && ( t1 < t2 ) && ( t2 < t1 + TOP )

  // From deploy...
  m.hasMType in ftrl.+createdFrom.developedToRunOn.platformMT
  ftrl.hasVersion = ftrl.desiredVersion
  ftrl.on' = m

  // From check-Res
  all res : Resources | some int : positiveIntegers | int in m.available [ res ] - ftrl.requires [ res ]

  // From check-Ver

```

```

    all depenRI : ( fIRI.dependsOn ) | some vers : ( depenRI.hasVersion ) | vers in depenRI.okVersions

// From set-bind
    all res : Resources | m.available' [ res ] = m.available [ res ] - fIRI.requires [ res ]
    fIRI.boundTo' = m

// From instantiate
    fIRI.hasState' = IDLE-state

// From register
    reg in fIRI.registeredIn'

    fIRI.refersToMe' = fIRI.refersToMe + wantsMe
    activated.FeedBack' in OK
}

// An operation to checkpoint the execution of a runnable item.
// This means put it into an idle state. Note that in the concrete world, putting something into an idle
// state probably also means stopping computations and freeing resources, etc. These design decisions
// are part of the definition of the idle state, and must be consistent across all the components.
// This specification on the other hand, just says that each configuration item needs an idle state
// and can be put into it in a consistent way.

op checkpoint ( fIRI : FaultTolerantRunnableItem ! ) {
    fIRI.hasState = IDLE-state
    checkPt.FeedBack' in OK
}

// An operation to perform a state transition on a runnable item.
// The target state needs to be in the runnable item's set of states to which it can transition, then the
// runnable item can be put in the new state.

op state-tran ( fIRI : FaultTolerantRunnableItem !, st : State ! ) {
    st in fIRI.transitionsTo
    fIRI.hasState' = st
    transitioned.FeedBack' in OK
}

// An operation to deactivate a dependency.
// This operation is needed since the operation to deactivate a runnable item cannot call itself
// using members of the dependency set of the runnable item as arguments.

op deactivate ( fIRI : FaultTolerantRunnableItem !, wantsToDeMe : FaultTolerantRunnableItem ! ) {
    fIRI.refersToMe' = fIRI.refersToMe - wantsToDeMe
    no referRI : ( fIRI.refersToMe ) | fIRI.hasState' = STOP-state && release-res ( fIRI )

    && all i : integer, t1 : Time | theta ( begin ( release-res ), i, t1 )
    && one t2 : Time | theta ( end ( release-res ), i, t2 ) && ( t1 < t2 ) && ( t2 < t1 + IOP )

    && ( all reg : ( fIRI.registeredIn ) | reg.hasRegInfo' = reg.hasRegInfo - fIRI )

    deactivated.FeedBack' in OK
}

// An operation to deactivate a runnable item.
// The reference set to this runnable item is decremented by the item that wants to deactivate it.
// (Note that this could either be a runnable item dependent on it or a configuration service.)
// If this set is empty, then the item can be moved to the stop state, its resources released,
// and its dependency set can be attempted to be deactivated. Note that if other items still reference
// items in the dependency set they will not be deactivated. (The idea is basically: "I'm quitting so I
// no longer depend on you. If nobody else depends on you either, then you can quit too.")
// If the original item is stopped, then it needs to be taken out of any registries that contain it.
// The operation 'deActivate' is needed to attempt to deactivate dependencies because Alloy will
// not allow an operation to be called from inside itself. The operation 'deActivate' is essentially
// the same as 'de-activate', except it does not traverse the dependency set.
//

```

```

// Note that Alloy does not support sequencing, so the order in which things happen is non-deterministic.

op de-activate ( ftRI : FaultTolerantRunnableItem !, wantsToDelMe : FaultTolerantRunnableItem ! ) {
    ftRI.refersToMe' = ftRI.refersToMe - wantsToDelMe
    no referRI : ( ftRI.refersToMe ) | ftRI.hasState' = STOP-state && release-res ( ftRI ) && ( all reg : (
ftRI.registeredIn ) | reg.hasRegInfo' = reg.hasRegInfo - ftRI ) && ( all depenRI : ( ftRI.dependsOn ) |
deActivate ( depenRI, ftRI ) )

// && all i : integer, t1 : Time | theta ( begin ( deActivate ), i, t1 )
// -> one t2 : Time | theta ( end ( deActivate ), i, t2 ) ) && ( t1 < t2 ) && ( t2 < t1 + tOP )

    de-activated.FeedBack' in OK
}

// An operation to "deactivate" one runnable item and then "activate" it as a new version.
// This is needed since Alloy does not support sequencing; we basically have to have a new operation
// that composes the post conditions of the operations that need to be sequenced; in this case
// deactivate and then activate.
//
// This operation could be written so that it 'calls' deploy, check-Res, etc., but the Alloy tool appears to
// hang if this is done. Re-writing the post conditions of each operation allows the tool to operate
// properly.

op deAct-act ( ftRI : FaultTolerantRunnableItem !, m : Machine !, reg : replicatedReg, desiredVer : Version
! ) {
    ftRI.refersToMe' = ftRI.refersToMe
    ftRI.hasVersion = desiredVer
    all depenRI : ( ftRI.dependsOn ) | some vers : ( depenRI.okVersions ) | locate ( depenRI, vers, reg, m,
ftRI )

// && all i : integer, t1 : Time | theta ( begin ( locate ), i, t1 )
// -> one t2 : Time | theta ( end ( locate ), i, t2 ) ) && ( t1 < t2 ) && ( t2 < t1 + tOP )

// From deploy...
    m.hasMType in ftRI.+createdFrom.developedToRunOn.platformMT
    ftRI.hasVersion = ftRI.desiredVersion
    ftRI.on' = m

// From check-Res
    all res : Resources | some int : positiveIntegers | int in m.available [ res ] - ftRI.requires [ res ]

// From check-Ver
    all depenRI : ( ftRI.dependsOn ) | some vers : ( depenRI.hasVersion ) | vers in depenRI.okVersions

// From set-unbind-bind
    all res : Resources | ftRI.on.available' [ res ] = ftRI.on.available [ res ] + ftRI.uses [ res ] && m.available' [
res ] = m.available [ res ] - ftRI.requires [ res ]
    ftRI.boundTo' = m

// From instantiate
    ftRI.hasState' = IDLE-state

// From register
    reg in ftRI.registeredIn'
    deAct-a.FeedBack' in OK
}

// An operation to set the automation level of a runnable item.
// For simplicity, this operation sets the same level of automation for all state transitions of the configuration
// item.

op set-auto ( vCI : VersionableCI !, auto : Automation ! ) {
    all st : ( vCI.transitionsTo ) | st.hasAutomation' = auto
    setAuto.FeedBack in OK
}

```

```

// An operation to get an inventory from a runnable item.
// The inventory to augment is passed in. The information of interest in an inventory is the runnable item's
// version (via the 'hasVersion' relation), whether it is active (via the 'isActive' relation), state
// (via the 'hasState' relation), and automation, which is on a per-state basis (via the 'hasAutomation'
// relation). However, all these items are available once the runnable item is part of the inventory,
// so this operation just needs to augment the inventory by the runnable item itself (via the 'hasInvenInfo'
// relation).

op get-inven ( ftrl : FaultTolerantRunnableItem !, inven : replicatedInven ) {
  all invent : ( inven ) | invent.hasInvenInfo = invent.hasInvenInfo + ftrl
  inventoried.FeedBack' in OK
}

// *** ConfigService Operations ***

// A ConfigService operation to get a configuration (a desired configuration definition).
// A configuration identifier is passed in and the configuration that corresponds to it is found.
// This is another operation that needs a replicated repository which contains configurations.

op get-config ( confName : ConfigId ! ) {
  some conf : Configuration | conf.configId = confName
  gotDC.FeedBack' in OK
}

// A ConfigService operation to get an inventory for the local set.
// The operation figures out what replicated inventories to use and passes this on to all the runnable
// items in the local set for them to augment with their own inventory information.
// Additional comments in the operation give RTL-style temporal constraints; the called
// operations need to finish within the tOP timeout. The 'theta' operation gives the time at which
// the i-th specified event occurred. So 'theta (begin (get-inven), i, t1)' sets t1 to the time at which
// the get-inven operation was started for the i-th time, and 'theta (end (get-inven), i, t2)' sets t2
// to the time at which that same i-th get-inven operation was finished. The specification says that
// the operation must finish within the timeout time.

op get-inventory ( configS : ConfigService !, inven : replicatedInven ) {
  all invent : ( inven ) | invent.LocalSetI = configS.LocalSetFTRI
  all ftrl : ( configS.controls ) | get-inven ( ftrl, inven )

  // && all i : integer, t1 : Time | theta ( begin ( get-inven ), i, t1 )
  // -> one t2 : Time | theta ( end ( get-inven ), i, t2 ) && ( t1 < t2 ) && ( t2 < t1 + tOP )
  inventoried.FeedBack' in OK
}

// A ConfigService operation to compare a desired configuration and an inventory of a local set and
// make changes to bring the inventory into alignment with the desired configuration.
// The cases are as follows:
// 1. An item is in the inventory and in the desired configuration.
//    Therefore check the state, version, ordinality and automation of the existing item and the
//    desired one. Make changes as follows:
//    a. state different: if the automation is OK for the state, then change it
//    b. ordinality different: if the desired ordinality is bigger than what exists, start a new one,
//    if the desired ordinality is less, stop one
//    c. version different: deactivate the current item and activate a new one with the new version
//    d. automation different: set the automation for the states to the desired automation
// 2. An item is not in the desired configuration: deactivate the item.
// 3. An item is in the desired configuration but not in the inventory: activate the item.
//
// Note that there is a problem checking ordinality. A runnable item can only know if it is active or
// not, not how many runnable items of the same executable exist. So the inventory has each runnable
// item and whether it is active. Some inventory operation must gather all the same kind of runnable items
// into
// an ordinality count before step 1b can be accomplished. It is not clear how to accomplish this
// specification in Alloy since there is no way in this version of Alloy to check set cardinality.

op compare-change ( inven : replicatedInven !, dConf : desiredConfig, ftrlConfigS :
FaultTolerantConfigService ! ) {

```

```

// all i : integer, t1 : Time | theta ( begin ( compare-change ), i, t1 )
//      -> one t2 : Time | ( theta ( begin ( compare-change ), i+1, t2 ) ) && ( t1 + tCS <= t2 )

    some invent : ( inven ) | all ftRlid : ( dConf.hasConfigInfo.riHasId - invent.hasInvenInfo.riHasId ) |
activate ( ftRlid.idOfRI, ftRlid.idOfRI.on, ftConfigS, ftConfigS.csReg )

// && all i : integer, t1 : Time | theta ( begin ( activate ), i, t1 )
//      -> one t2 : Time | theta ( end ( activate ), i, t2 ) ) && ( t1 < t2 ) && ( t2 < t1 + tOP )

some invent : ( inven ) | all ftRlid : ( invent.hasInvenInfo.riHasId - dConf.hasConfigInfo.riHasId ) |
ftRlid.idOfRI.hasState.hasAutomation = autoOK -> de-activate ( ftRlid.idOfRI, ftConfigS )

// && all i : integer, t1 : Time | theta ( begin ( de-activate ), i, t1 )
//      -> one t2 : Time | theta ( end ( de-activate ), i, t2 ) ) && ( t1 < t2 ) && ( t2 < t1 + tOP )

some invent : ( inven ) | all ftRlid : ( invent.hasInvenInfo.riHasId & dConf.hasConfigInfo.riHasId ) | one ftR11,
ftR12 : ftRlid.idOfRI, st1, st2 : State | ( ftR11.hasState = st1 && ftR12.desiredState = st2 && ftR11 != ftR12 && st1 !=
st2 && st1.hasAutomation = autoOK ) ->
state-tran ( ftR11, st2 )

// && all i : integer, t1 : Time | theta ( begin ( state-tran ), i, t1 )
//      -> one t2 : Time | theta ( end ( state-tran ), i, t2 ) ) && ( t1 < t2 ) && ( t2 < t1 + tOP )

some invent : ( inven ) | all ftRlid : ( invent.hasInvenInfo.riHasId & dConf.hasConfigInfo.riHasId ) | one ftR11,
ftR12 : ftRlid.idOfRI, v1, v2 : Version | ( ftR11.hasVersion = v1 && ftR12.desiredVersion = v2 && ftR11 != ftR12 &&
v1 != v2 && ftR11.hasState.hasAutomation = autoOK ) ->
deAct-act ( ftR11, ftR12.on, ftConfigS.csReg, v2 )

// && all i : integer, t1 : Time | theta ( begin ( deAct-act ), i, t1 )
//      -> one t2 : Time | theta ( end ( deAct-act ), i, t2 ) ) && ( t1 < t2 ) && ( t2 < t1 + tOP )

some invent : ( inven ) | all ftRlid : ( invent.hasInvenInfo.riHasId & dConf.hasConfigInfo.riHasId ) | one ftR11,
ftR12 : ftRlid.idOfRI, o1, o2 : integers | ( ftR11.hasOrdinality = o1 && ftR12.desiredOrdinality = o2 && ftR11 !=
ftR12 && o1 != o2 ) ->
( ( some intP : positiveNotZeroIntegers |
( intP = ftR12.desiredOrdinality - ftR11.hasOrdinality && ftR11.hasState.hasAutomation = autoOK ) ->
activate ( ftR12, ftR12.on, ftConfigS, ftConfigS.csReg ) )

// && all i : integer, t1 : Time | theta ( begin ( activate ), i, t1 )
//      -> one t2 : Time | theta ( end ( activate ), i, t2 ) ) && ( t1 < t2 ) && ( t2 < t1 + tOP )

    | | ( some intP : positiveNotZeroIntegers |
( intP = ftR11.hasOrdinality - ftR12.desiredOrdinality && ftR11.hasState.hasAutomation = autoOK ) ->
de-activate ( ftR11, ftConfigS ) ) )

// && all i : integer, t1 : Time | theta ( begin ( de-activate ), i, t1 )
//      -> one t2 : Time | theta ( end ( de-activate ), i, t2 ) ) && ( t1 < t2 ) && ( t2 < t1 + tOP )

some invent : ( inven ) | all ftRlid : ( invent.hasInvenInfo.riHasId & dConf.hasConfigInfo.riHasId ) | one ftR11,
ftR12 : ftRlid.idOfRI, a1, a2 : Automation | ( ftR11.hasState.hasAutomation = a1 &&
ftR12.hasState.desiredAutomation = a2 && ftR11 != ftR12 && a1 != a2 && ftR11.hasState.hasAutomation =
autoOK ) ->
set-auto ( ftR11, a2 )

// && all i : integer, t1 : Time | theta ( begin ( set-auto ), i, t1 )
//      -> one t2 : Time | theta ( end ( set-auto ), i, t2 ) ) && ( t1 < t2 ) && ( t2 < t1 + tOP )

    compared.Feedback' in OK
}

// Operation to initialize all the automation to be OK to make configuration changes on runnable items
// no matter what state they are in. A default initialization.

op init ( ftConfigS : FaultTolerantConfigService ! ) {
    all ftRI : ( ftConfigS.controls ) | all st : ( ftRI.transitionsTo ) | st.hasAutomation = autoOK

```



```

    initialized.FeedBack' in OK
}

// *** Inventory Operations ***

// Operation to add the ordinality of the same types of active runnable items.

op addOrdinalities ( inven : Inventory ) {
    all fIR11, fIR12 : ( inven.hasInvenInfo ) | { ( fIR11 != fIR12 )
        && ( fIR11.isActive = true )
        && ( fIR12.isActive = true )
        && ( fIR11.createdFrom.srcType = fIR12.createdFrom.srcType ) }
    -> ( ( fIR11.hasOrdinality = fIR11.hasOrdinality + fIR12.hasOrdinality )
        && ( fIR12.hasOrdinality = fIR12.hasOrdinality + fIR11.hasOrdinality ) )
    addedOrd.FeedBack' in OK
}
}

```

APPENDIX 8A – REVISED COMPLETE UML/OCL SPECIFICATION

This appendix presents the revised UML/OCL specification of run-time software configuration management in the ATM/IP network monitoring system. The specification is revised based on problems detailed in Chapter 7. The differences between this and the original specification (presented in Appendix 5A) lie in the creation of a fault tolerant runnable item, the use of replicated repositories, inventories, registries, and the use of redundant configuration services.

The *type* model shows *FaultTolerantRunnableItem* as a specialization of *RunnableItem* and *FaultTolerantConfigService* as a specialization of *ConfigService*. These *types* are especially required for fault tolerant run-time software configuration management in the ATM network monitoring system. They are shown as specializations of the non-fault tolerant *types* to specifically shown what aspects of the specification are needed only for fault tolerance. This separation improves the understandability of the specification since fault tolerant aspects are not interwoven with non-fault tolerant aspects. In addition, it may improve the reusability of the specification for other distributed systems that do not have the same fault tolerant needs as the ATM network monitoring system. The idea of separating different concerns in a specification is discussed in {Ciapessoni, et al. 1999}.

1. TYPE MODEL DIAGRAMS

Figures 8A-1 and 8A-2 show the fundamental software *types* (Figure 8A-1) and their operations (Figure 8A-2). Figure 8A-3 shows the interactions of the software *types*

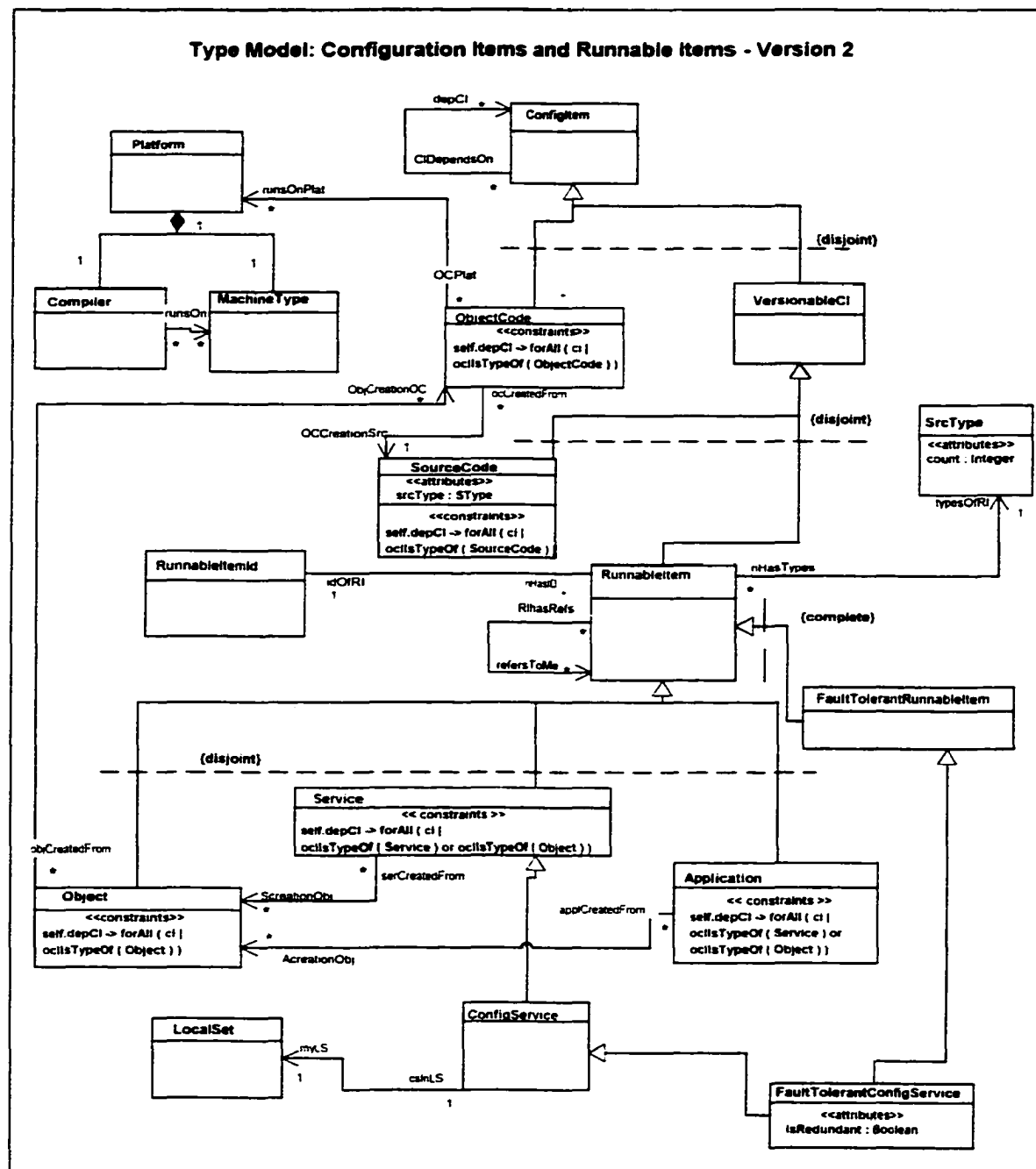
with other configuration management-related *concepts* and *types* (i.e. versions, resource, and software states).

Another fundamental concept in the ATM network monitoring system is the *LocalSet*. This concept allows incremental dynamic configuration across the system. It is shown in the *type* model. Figure 8A-4, shows the definition of the loose aggregation of a *LocalSet*. It consists of *RunnableItems*, the *Machines* they run on, and the associated *Registries* and *Inventories*. Figure 8A-5, shows the relations between portions of a *LocalSet* and other configuration management-related concepts (i.e. *MachineType*, *Resources*, *SrcType*, *Configurations*, and *Repositories*).

2. OCL DEFINITIONS

The OCL portion of the UML/OCL type models is shown below. Note that even though this is a requirements specification, and we used *type* models so that we could remain abstract in the operation definition, we seem to be forced to specify operation return value data types.

Pre- and post- conditions dealing with the state machine (as noted in the *type* model) use the notation *self.ocInState* to specify the state the item is in once the operation is complete. For example, *self.ocInState (deployed) = true* in the *deploy* operation definition of a *RunnableItem* indicates that a post-condition of the *deploy* operation is that the runnable item is left in its *deployed* state. These post-conditions are used in more complex operations that would ordinarily compose simpler functions, by using the previously defined post-conditions states to be stated as pre-conditions.



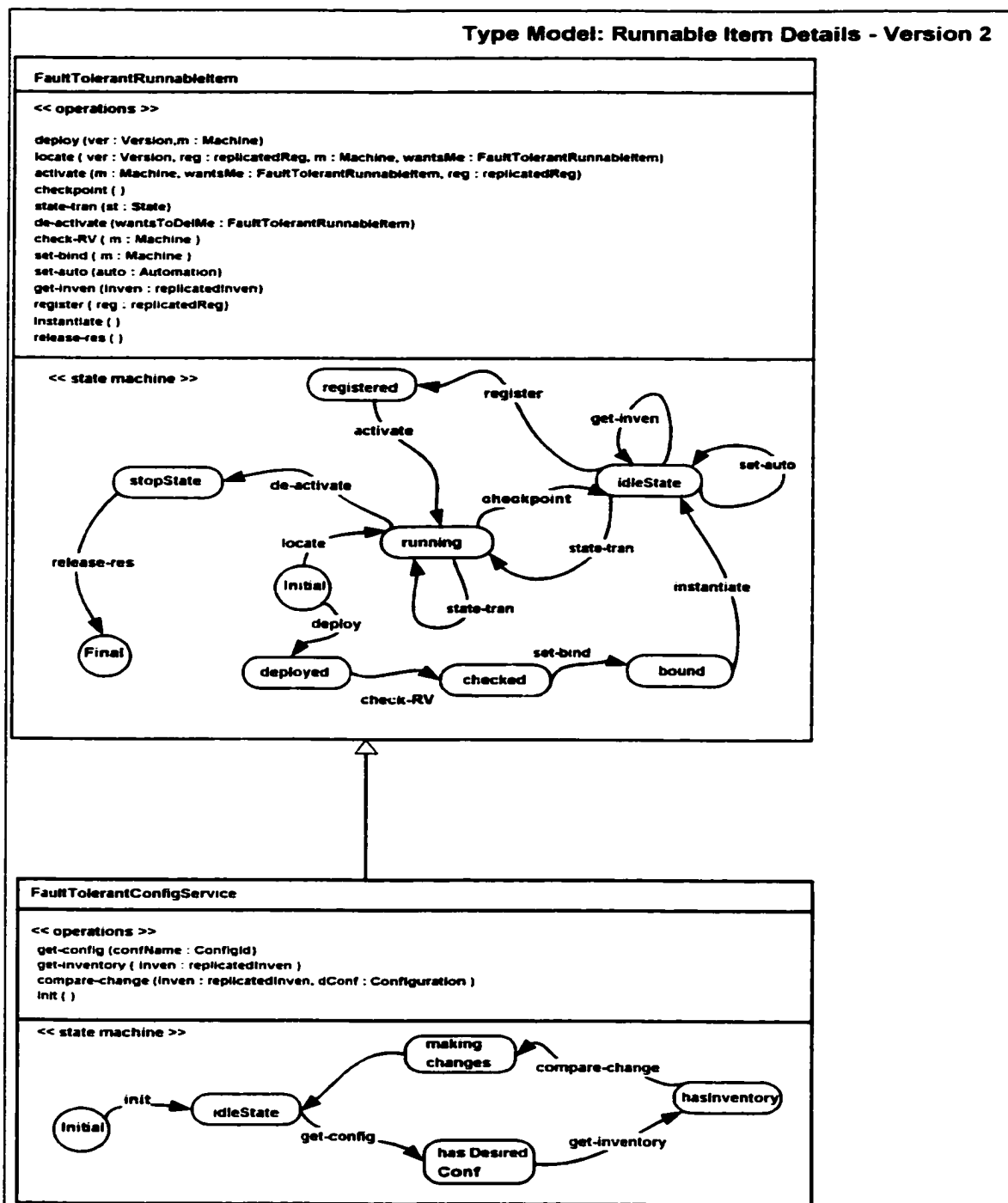


Figure 8.A-2. Operations and state machine details for *RunnableItem*. The <<operations>> section shows the calling signatures of the operations and the <<state machine>> section shows how each operation effects the state of the type. *ConfigService* is a specialization of *Service* and has additional operations that are needed to manage the dynamic configuration of the system.

Type Model: Runnable Items, States, Versions, and Resources - Version 2

```

classDiagram
    class Version {
        +desiredVersion
    }
    class VersionableCI {
        +depVers
        +VerCVersion
        +VerCIdlesVer
        +VerCISStates
        +VerCISStop
        +VerCIdle
    }
    class Resources {
        +resRequired
        +subset
        +resUsed
    }
    class FaultTolerantRunnableItem {
        +isActive : Boolean
    }
    class State {
        +automationOK : Boolean
    }
    class STOP_state {
    }
    class IDLE_state {
    }

    VersionableCI --> Version : OkVersions *
    VersionableCI --> Version : hasVer 1
    VersionableCI --> Version : desiredVersion 0 1
    VersionableCI --> Version : VerCVersion *
    VersionableCI --> Version : VerCIdlesVer *
    VersionableCI --> Version : VerCISStates *
    VersionableCI --> Version : VerCISStop *
    VersionableCI --> Version : VerCIdle *
    VersionableCI --> State : stateTransitions *
    VersionableCI --> STOP_state : transitionsToStop 1 *
    VersionableCI --> IDLE_state : transitionsToIdle 1 *
    VersionableCI --|> FaultTolerantRunnableItem
    FaultTolerantRunnableItem --> Resources : resRequired *
    FaultTolerantRunnableItem --> Resources : subset
    FaultTolerantRunnableItem --> Resources : resUsed *
    FaultTolerantRunnableItem --> State : RidesState *
    FaultTolerantRunnableItem --> State : RidesState *
    FaultTolerantRunnableItem --> State : currentState :
    FaultTolerantRunnableItem --> State : desiredState 0 1
    State --|> STOP_state
    State --|> IDLE_state
    
```

The diagram illustrates the relationships between several classes in a system:

- Version**: Contains a `desiredVersion` attribute.
- VersionableCI**: Contains attributes `depVers`, `VerCVersion`, `VerCIdlesVer`, `VerCISStates`, `VerCISStop`, and `VerCIdle`. It has associations with **Version** (labeled `OkVersions`, `hasVer`, and `desiredVersion`), **Resources** (labeled `resRequired`, `subset`, and `resUsed`), **State** (labeled `stateTransitions`), **STOP-state** (labeled `transitionsToStop`), and **IDLE-state** (labeled `transitionsToIdle`).
- Resources**: Contains attributes `resRequired`, `subset`, and `resUsed`.
- FaultTolerantRunnableItem**: Inherits from **VersionableCI**. It contains an attribute `isActive : Boolean` and has associations with **Resources** (labeled `resRequired`, `subset`, and `resUsed`), **State** (labeled `RidesState`, `RidesState`, `currentState`, and `desiredState`), and **State** (labeled `desiredState`).
- State**: Contains an attribute `automationOK : Boolean`. It has a generalization relationship with **STOP-state** and **IDLE-state**.
- STOP-state** and **IDLE-state**: Specializations of the **State** class.

Figure 8.A-3. *VersionableCI* interactions with other configuration management-related concepts. *VersionableCI* types have a version, require resources, and have transition states in this system. Two states are required to be in the set of transition states, a stop and an idle state. Other states in the transition set are a result of the actual functionality of the software (e.g. a fraud detection application). These other states are orthogonal to the issue of software run-time configuration management and vary according to the functionality of their associated software. They are not included in the software run-time configuration management type model. Configuration changes are made to software types when they are in the idle or stop state, provided *Automation* for these states is properly set. If *Automation* is not properly set, automatic configuration changes are not allowed. The *Automation* concept is defined as an attribute in this type model, so it must be given a data type. This model shows it as a Boolean value whose name is *automationOK*. In addition, the *RunnableItem* specialization of *VersionableCI* uses resources, always the same or a subset of the required resources.

Type Model: Local Set Definition - Version 2

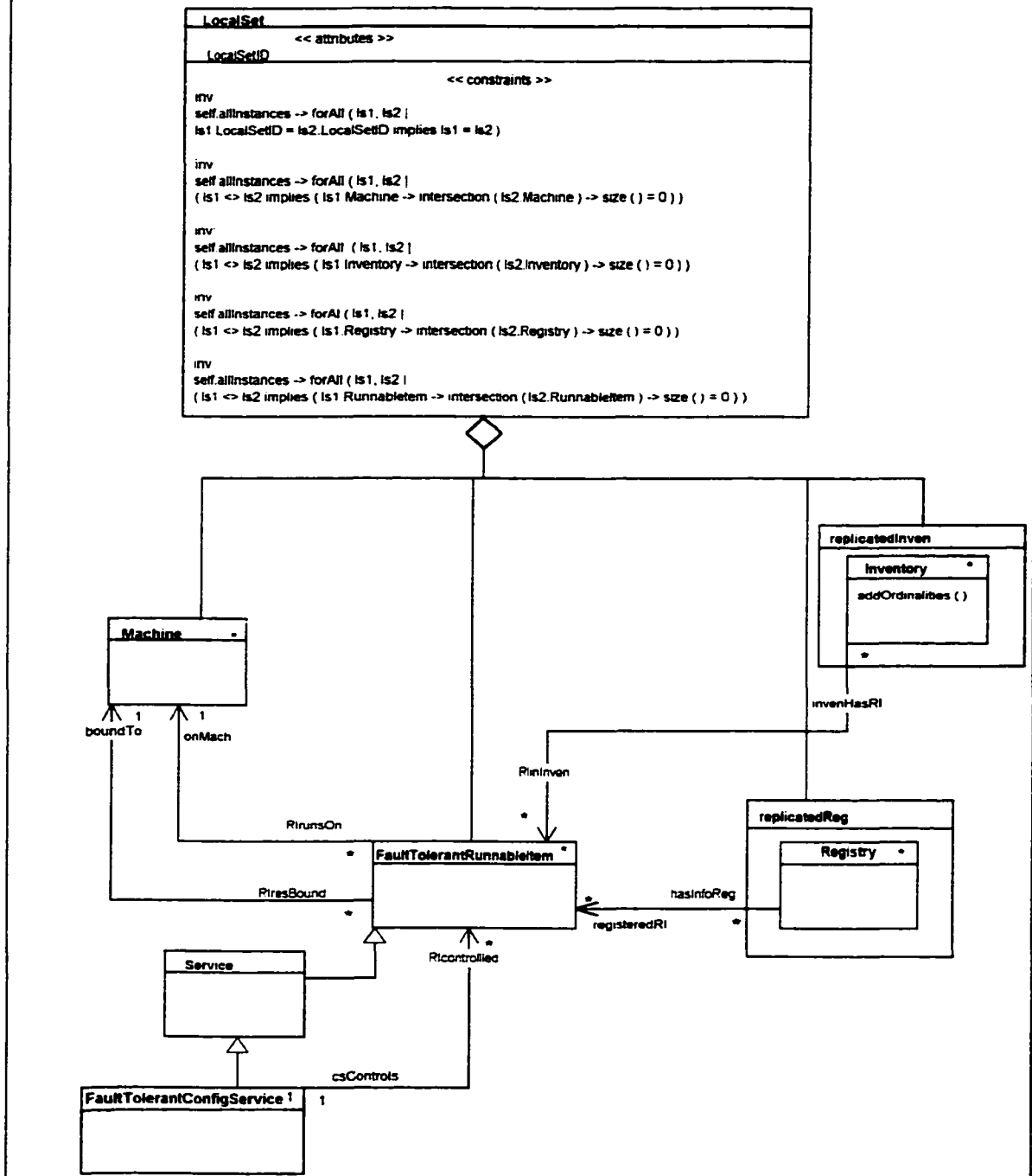


Figure 8.A-4. The type model of a *LocalSet*. A *LocalSet* partitions the overall system to allow independent run-time configuration management of the partitions. At any given time, machines and the runnable items executing on them can only be in one *LocalSet*. Inventories and registries are only associated with one *LocalSet*. A *LocalSet* is defined as a loose aggregation of runnable items, the physical machines on which they run and associated registries and inventories. The notion of a local set identifier is included to ensure unique local sets.

RunnableItem

The **deploy** operation must be overridden by each specialization: object, service, and application. Its signature is given in the generalization.

```
context RunnableItem :: deploy (ver : Version, m : Machine) : Boolean
pre: self.oclnState ( INITIAL ) = true
post: self.oclnState ( deployed ) = true and result = true
```

The **locate** operation. The post condition is that EITHER there is a registry at the beginning of the operation that already contains the runnable item in question, OR the runnable item is deployed, its resources and version checked, its resources bound, it is instantiated, and it is placed in the registry. In addition, its reference list is augmented to include the runnable item that wanted it. The result of this post condition is TRUE.

```
context RunnableItem :: locate ( ver : Version, reg : Registry, m : Machine,
                                wantsMe : RunnableItem ) : Boolean
pre: self.oclnState ( INITIAL ) = true
post : ( ( self.hasInfoReg @pre -> includes ( reg ) )
xor ( self.deploy ( ver, m ) and self.checkRV ( m ) and self.set-bind ( m ) and self.instantiate ( )
and ( reg -> including ( self ) )
and ( self.refersToMe -> including ( wantsMe )
and self.oclnState ( running ) = true
and result = true
```

The **activate** operation. The post condition states that all runnable items in the dependency list are located. In addition, the resources and version for this runnable item are checked, the resources are bound, the runnable item is instantiated and added to the registry. The result of the post condition is TRUE.

```
context RunnableItem :: activate ( m : Machine, wantsMe : RunnableItem, reg : Registry ) : Boolean
pre: self.oclnState ( registered ) = true
post : ( self.depRI -> iterate (
n : RunnableItem;
res : Boolean = true; |
n.locate ( self.hasVer, self.hasInfoReg, m, self ) ) )
and self.check-RV ( m ) and self.set-bind ( m ) and self.instantiate ( )
and ( reg -> including ( self ) )
and self.oclnState ( running ) = true
and result = true
```

The **checkpoint** operation simply places the runnable item into its IDLE state. The post condition result is TRUE.

```
context RunnableItem :: checkpoint ( ) : Boolean
pre: self.oclnState ( running ) = true
post : self.currentState = IDLE-state and self.oclnState ( idleState ) = true and result = true
```

The **state-tran** operation places the runnable item into the state passed in as an argument. The post condition result is TRUE.

```
context RunnableItem :: state-tran ( st : State ) : Boolean
pre: self.oclnState ( idleState ) = true or self.oclnState ( running ) = true
post : self.currentState = st and self.oclnState ( running ) = true and result = true
```

The **deactivate** operation. The post condition states that the reference list is "decremented" by the item that wants to deactivate this runnable item. Furthermore, if this list is empty that implies that the runnable item can move to its stop state, release all its resources, take itself out of the registries that have information about it, and attempt to deactivate all of the runnable items in its dependency set. The post condition result is TRUE.

```
context RunnableItem :: deactivate ( wantsToDelMe : RunnableItem ) : Boolean
pre: self.oclnState ( running ) = true
post : self.refersToMe -> excluding ( wantsToDelMe )
and self.refersToMe -> isEmpty implies self.state-tran ( STOP-state ) and self.release-res ( )
and self.hasInfoReg.allInstances -> forAll ( regis | regis.registeredRI -> excluding ( self ) )
and self.depRI.allInstances -> forAll ( ri | ri.deactivate ( self ) )
and self.oclnState ( stopState ) = true
and result = true
```

The **check-RV** operation. The resource check states that the resources available on the machine at the beginning of the operation must be greater or equal to those required by the runnable item. The version check is that the set of versions that are OK for each item in the dependency set must include the version of this runnable item. The post condition is TRUE.

```
context RunnableItem :: check-RV ( m : Machine ) : Boolean
pre: self.oclnState ( deployed ) = true
post : m.ResOfM.resAvail @pre >= self.resRequired and self.depRI -> forAll ( ri |
ri.OkVersions -> includes ( self.hasVersion ) ) and self.oclnState ( checked ) = true and result = true
```

The **set-bind** operation decrements the available resources of the machine passed in as an argument by the resources required by the runnable item and sets the runnable item `boundTo` relation to the machine. The resulting resources on the machine must be greater than or equal to 0. The result of the post condition is TRUE.

```
context RunnableItem :: set-bind ( m : Machine ) : Boolean
  pre: self.oclnState ( checked ) = true
  post : m.ResOfM.resAvail = m.ResOfM.resAvail - self.resRequired and self.boundTo = m
        and m.ResOfM.resAvail >= 0 and self.oclnState ( bound ) = true and result = true
```

The **set-auto** operation. This iterates through all the states of the runnable item and sets each state's `automationOK` attribute to the Boolean passed into the operation. The post condition result is TRUE.

```
context RunnableItem :: set-auto ( auto : Boolean ) : Boolean
  pre: self.oclnState ( idleState ) = true
  post : self.stateTransitions.allInstances -> forAll ( st | st.automationOK = auto )
        and self.oclnState ( idleState ) = true and return = true
```

The **get-inven** operation. The runnable item is added to the information in the inventory. The result of the post condition is TRUE.

```
context RunnableItem :: get-inven ( inven : Inventory ) : Boolean
  pre: self.oclnState ( idleState ) = true
  post : inven -> including ( self ) and self.oclnState ( idleState ) = true and return = true
```

The **register** operation. The set of registries that have information about the runnable item is augmented to include the registry passed into the operation.

```
context RunnableItem :: register ( reg : Registry ) : Boolean
  pre: self.oclnState ( idleState ) = true
  post : self.hasInfoReg -> includes ( reg ) and self.oclnState ( registered ) = true and result = true
```

The **instantiate** operation puts the runnable item in its IDLE state. The result of the post condition is TRUE.

```
context RunnableItem :: instantiate ( ) : Boolean
  pre: self.oclnState ( bound ) = true
  post : self.currentState = IDLE-state and self.oclnState ( idleState ) = true and result = true
```

The **release-res** operation. The available resources on the machine on which the runnable item is executing is augmented by the resources in use by the runnable item. The result of the post condition is TRUE.

```
context RunnableItem :: release-res ( ) : Boolean
  pre: self.oclnState ( stopState ) = true
  post : self.onMach.ResOfM.resAvail = self.onMach.ResOfM.resAvail + self.resUsed
        and self.oclnState ( FINAL ) = true and result = true
```

The **sameRlid** invariant. Runnable items with the same runnable item identifier share the same type of sources.

```
context RunnableItem :: inv sameRlid :
  self.idOfRI.nHasID -> forAll ( ri | ri.typesOfRI = self.typesOfRI )
```

FaultTolerantRunnableItem

The **locate** operation. The post condition is that EITHER there is a registry at the beginning of the operation that already contains the runnable item in question. OR the runnable item is deployed, its resources and version checked, its resources bound, it is instantiated, and it is placed in the replicated registries. In addition, its reference list is augmented to include the runnable item that wanted it. The result of this post condition is TRUE.

```
context FaultTolerantRunnableItem :: locate ( ver : Version, reg : replicatedReg, m : Machine,
                                             wantsMe : FaultTolerantRunnableItem ) : Boolean
  pre: self.oclnState ( INITIAL ) = true
  post : ( { self.hasInfoReg @pre -> includes ( reg ) }
        xor ( self.deploy ( ver, m ) and self.checkRV ( m ) and self.set-bind ( m ) and self.instantiate ( )
              and ( reg.Registry -> forAll regis | regis.registeredRI -> including ( self ) )
              and ( self.refersToMe -> including ( wantsMe )
              and self.oclnState ( running ) = true
              and result = true
```

The **activate** operation. The post condition states that all runnable items in the dependency list are located. In addition, the resources and version for this runnable item are checked, the resources are bound, the runnable item is instantiated and added to the replicated registries. The result of the post condition is TRUE.

```
context FaultTolerantRunnableItem :: activate ( m : Machine, wantsMe : FaultTolerantRunnableItem,
                                              reg : ReplicatedReg ) : Boolean
  pre: self.oclnState ( registered ) = true
```

```

post : ( self.depRI -> iterate (
  n : FaultTolerantRunnableItem;
  res : Boolean = true; |
  n.locate ( self.hasVer, self.hasInfoReg, m, self ) ) )
and self.check-RV ( m ) and self.set-bind ( m ) and self.instantiate ( )
and ( reg.Registry -> forAll regis | regis.registeredRI -> including ( self ) )
and self.oclnState ( running ) = true
and result = true

```

The **de-activate** operation. The post condition states that the reference list is "decremented" by the item that wants to deactivate this runnable item. Furthermore, if this list is empty that implies that the runnable item can move to its stop state and release all its resources and take itself out of the replicated registries and attempt to deactivate all of the runnable items in its dependency set. The post condition result is TRUE.

```

context FaultTolerantRunnableItem :: de-activate ( wantsToDelMe : FaultTolerantRunnableItem ) : Boolean
pre: self.oclnState ( running ) = true
post : self.refersToMe -> excluding ( wantsToDelMe )
and self.refersToMe -> isEmpty implies self.state-tran ( STOP-state ) and self.release-res ( )
and self.hasInfoReg.allInstances -> forAll ( regis | regis.registeredRI -> excluding ( self ) )
and self.depRI.allInstances -> forAll ( n | n.deactivate ( self ) )
and self.oclnState ( stopState ) = true
and result = true

```

The **get-inven** operation. For each inventory in the replicated inventory, the runnable item is added to the information in the inventory. The result of the post condition is TRUE.

```

context FaultTolerantRunnableItem :: get-inven ( inven : ReplicatedInven ) : Boolean
pre: self.oclnState ( idleState ) = true
post : inven.Inventory -> forAll ( invent | invent.LsinInven -> including ( self ) )
and self.oclnState ( idleState ) = true and return = true

```

The **register** operation. The set of registries that have information about the runnable item is augmented to include the replicated registry passed into the operation.

```

context FaultTolerantRunnableItem :: register ( reg : ReplicatedReg ) : Boolean
pre: self.oclnState ( idleState ) = true
post : self.hasInfoReg -> includes ( reg ) and self.oclnState ( registered ) = true and result = true

```

ConfigService

The **get-config** operation. The post condition states that the configuration that corresponds to the configuration name is the desired configuration. The result of the post condition is TRUE.

```

context ConfigService :: get-config ( confName : ConfigId ) : Boolean
pre: self.oclnState ( idleState ) = true
post : confName.configuration.isDesiredConfig = true and self.oclnState ( hasDesiredConf ) = true
and result = true

```

The **get-inventory** operation. The post condition states that the inventory has the same local set identifier as the configuration service and that the get-inven operation is called for each runnable item in the local set. The result of the post condition is TRUE.

```

context ConfigService :: get-inventory ( inven : Inventory ) : Boolean
pre: self.oclnState ( hasDesiredConf ) = true
post : inven.LSIDofInven = self.myLSID
and self.RlinLS.allInstances -> forAll ( n | n.get-inven ( inven ) )
and self.oclnState ( hasInventory ) = true and result = true

```

The **compare-change** operation. Depending on the contents of the desired configuration and an inventory, runnable items are activated, deactivated, their states changed, or their automationOK attributes changed. See the Alloy model for the logic of this operation.

```

context ConfigService :: compare-change ( inven : Inventory, dConf : Configuration ) : Boolean
pre: self.oclnState ( hasInventory ) = true
post: self.oclnState ( makingChanges ) = true and result = true

```

The **init** operation. This operation sets all the automation attributes so that automatic operation can occur. It also moves the item into its idle state. The result of the post condition is TRUE.

```

context ConfigService :: init ( ) : Boolean
pre: self.oclnState ( INITIAL ) = true
post: ( dConf.confId - inven.invenId ) -> forAll ( nId |
  nId.nHasID.csControls = self implies
  nId.nHasID.activate ( nId.nHasID.onMach, self, nId.nHasID.hasInfoReg ) )
and ( inven.invenId - dConf.confId ) -> forAll ( nId |
  nId.nHasID.csControls = self and nId.nHasID.currentState.automationOK = true implies

```

```

        nld.nHasID.de-activate ( self )
    and ( dConf.confId -> intersection ( inven.invenId ) -> forAll ( nld |
        ( ( dConf.nld.nHasID.desiredState <> inven.nld.nHasID.currentState and
            inven.nld.nHasID.currentState.automationOK = true ) implies
                inven.nld.nHasID.state-tran ( dConf.nld.nHasID.desiredState ) )
        and ( ( dConf.nld.nHasID.desiredVersion <> inven.nld.nHasID.hasVer and
            inven.nld.nHasID.currentState.automationOK = true ) implies
                inven.nld.nHasID.de-activate ( self ) and
                inven.nld.nHasID.activate ( dConf.nld.nHasID.onMach, self, inven.nld.nHasID.hasInfoReg
        ) )
    and ( ( dConf.nld.nHasID.stateTransitions.automationOK = false and
        inven.nld.nHasID.currentState.automationOK = true ) implies
        inven.nld.nHasID.set-auto ( false ) )
    and ( ( dConf.nld.nHasID.typesOfRI.count > inven.nld.nHasID.typesOfRI.count ) implies
        inven.nld.nHasID.activate ( dConf.nld.nHasID.onMach, self, inven.nld.nHasID.hasInfoReg ) )
    and ( ( dConf.nld.nHasID.typesOfRI.count < inven.nld.nHasID.typesOfRI.count and
        inven.nld.nHasID.currentState.automationOK = true )
        implies inven.nld.nHasID.de-activate ( self ) ) )
    and self.ocInState ( makingChanges ) = true and result = true

```

FaultTolerantConfigService

The **get-inventory** operation. The post condition states that all inventories in the replicated inventory have the same local set identifier as the configuration service and that the get-inven operation is called for each runnable item in the local set. The result of the post condition is TRUE.

```

context FaultTolerantConfigService :: get-inventory ( inven : replicatedInven ) : Boolean
pre: self.ocInState ( hasDesiredConf ) = true
post : inven.inventory -> forAll ( invent | invent.LSIDofInven = self.myLSID )
      and self.RinLS.allInstances -> forAll ( ri | ri.get-inven ( inven ) )
      and self.ocInState ( hasInventory ) = true and result = true

```

The **compare-change** operation. Depending on the contents of the desired configuration and an inventory, runnable items are activated, deactivated, their states changed, or their automationOK attributes changed. See the Alloy model for the logic of this operation.

```

context FaultTolerantConfigService :: compare-change ( inven : replicatedInven, dConf : Configuration ) :
    Boolean
pre: self.ocInState ( hasInventory ) = true
post: ( dConf.confId - inven.invenId ) -> forAll ( nld |
    nld.nHasID.csControls = self implies
        nld.nHasID.activate ( nld.nHasID.onMach, self, nld.nHasID.hasInfoReg ) )
    and ( inven.invenId - dConf.confId ) -> forAll ( nld |
        nld.nHasID.csControls = self and nld.nHasID.currentState.automationOK = true implies
            nld.nHasID.de-activate ( self ) )
    and ( dConf.confId -> intersection ( inven.invenId ) -> forAll ( nld |
        ( ( dConf.nld.nHasID.desiredState <> inven.nld.nHasID.currentState and
            inven.nld.nHasID.currentState.automationOK = true ) implies
                inven.nld.nHasID.state-tran ( dConf.nld.nHasID.desiredState ) )
        and ( ( dConf.nld.nHasID.desiredVersion <> inven.nld.nHasID.hasVer and
            inven.nld.nHasID.currentState.automationOK = true ) implies
                inven.nld.nHasID.de-activate ( self ) and
                inven.nld.nHasID.activate ( dConf.nld.nHasID.onMach, self,
                    inven.nld.nHasID.hasInfoReg ) )
        and ( ( dConf.nld.nHasID.stateTransitions.automationOK = false and
            inven.nld.nHasID.currentState.automationOK = true ) implies
                inven.nld.nHasID.set-auto ( false ) )
        and ( ( dConf.nld.nHasID.typesOfRI.count > inven.nld.nHasID.typesOfRI.count ) implies
            inven.nld.nHasID.activate ( dConf.nld.nHasID.onMach, self, inven.nld.nHasID.hasInfoReg ) )
        and ( ( dConf.nld.nHasID.typesOfRI.count < inven.nld.nHasID.typesOfRI.count and
            inven.nld.nHasID.currentState.automationOK = true )
            implies inven.nld.nHasID.de-activate ( self ) ) )
    and self.ocInState ( makingChanges ) = true and result = true

```

Object

The **deploy** operation for an Object. The post condition result starts out as TRUE. All of the object code used to create this object is checked to make sure it was created to run on the platform of the same type as the machine passed in. If any in the set was not created for the same platform, the post condition result becomes FALSE. The object's version is the one passed in and it is on the machine passed in.

```
context Object :: deploy ( ver : Version, m : Machine ) : Boolean
pre: self.oclnState ( INITIAL ) = true
post : result = true
and self.ObjCreationOC.allInstances -> forAll ( oc |
  if oc.runsOnPlat.machineType = m.typeOfM
  then ( if result = false then result = false else result = true endif )
  else result = false endif )
and self.hasVer = ver and self.onMach = m and self.oclnState ( deployed ) = true
```

The **notDepOnSerAppl** constraint. The configuration items in the object's dependency set can only be of the type Object.

```
context Object :: inv notDepOnSerAppl :
self.depCI -> forAll ( ci | oclIsTypeOf ( Object ) )
```

Service

The **deploy** operation for a Service. The post condition result starts out as TRUE. All of the object code used to create this service is checked to make sure it was created to run on the platform of the same type as the machine passed in. If any in the set was not created for the same platform, the post condition result becomes FALSE. The object's version is the one passed in and it is on the machine passed in.

```
context Service :: deploy ( ver : Version, m : Machine ) : Boolean
pre: self.oclnState ( INITIAL ) = true
post : result = true
and self.SCreationOC.allInstances -> forAll ( obj |
  obj.ObjCreationOC.allInstances -> forAll ( oc |
    if oc.runsOnPlat.machineType = m.typeOfM
    then ( if result = false then result = false else result = true endif )
    else result = false endif ) )
and self.hasVer = ver and self.onMach = m and self.oclnState ( deployed ) = true
```

The **notDepOnAppl** constraint. The configuration items in the service's dependency set can only be of the type Service or Object.

```
context Service :: inv notDepOnAppl :
self.depCI -> forAll ( ci | oclIsTypeOf ( Service ) or oclIsTypeOf ( Object ) )
```

Application

The **deploy** operation for an Application. The post condition result starts out as TRUE. All of the object code used to create this application is checked to make sure it was created to run on the platform of the same type as the machine passed in. If any in the set was not created for the same platform, the post condition result becomes FALSE. The object's version is the one passed in and it is on the machine passed in.

```
context Application :: deploy ( ver : Version, m : Machine ) : Boolean
pre: self.oclnState ( INITIAL ) = true
post : result = true
and self.ACreationOC.allInstances -> forAll ( obj |
  obj.ObjCreationOC.allInstances -> forAll ( oc |
    if oc.runsOnPlat.machineType = m.typeOfM
    then ( if result = false then result = false else result = true endif )
    else result = false endif ) )
and self.hasVer = ver and self.onMach = m and self.oclnState ( deployed ) = true
```

The **notDepOnAppl** constraint. The configuration items in the application's dependency set can only be of the type Application or Object.

```
Context Application :: inv notDepOnSer:
self.depCI -> forAll ( ci | oclIsTypeOf ( Application ) or oclIsTypeOf ( Object ) )
```

LocalSet

The **uniqueLSID** constraint. Every local set has a unique local set identifier.

```
context LocalSet :: inv uniqueLSID:
self.allInstances -> forAll ( ls1, ls2 | ls1.LocalSetID implies ls2 = ls1 )
```

The **machInOneLS** constraint. Every machine can only be in one local set at any given point in time.

```
context LocalSet :: inv machInOneLS:
```

```

self.allInstances -> forAll ( ls1, ls2 | ls1 <> ls2 implies
    ( ls1.Machine -> intersection ( ls2.Machine ) ) -> size ( ) = 0 )

```

The **invenInOneLS** constraint. Inventories are associated with only one local set.

```

context LocalSet :: inv invenInOneLS:
    self.allInstances -> forAll ( ls1, ls2 | ls1 <> ls2 implies
        ( ls1.Inventory -> intersection ( ls2.Inventory ) ) -> size ( ) = 0 )

```

The **registryInOneLS** constraint. Registries are associated with only one local set.

```

context LocalSet :: inv registryInOneLS:
    self.allInstances -> forAll ( ls1, ls2 | ls1 <> ls2 implies
        ( ls1.Registry -> intersection ( ls2.Registry ) ) -> size ( ) = 0 )

```

The **RlinOneLS** constraint. Executing runnable items are associated with only one local set.

```

context LocalSet :: inv RlinOneLS:
    self.allInstances -> forAll ( ls1, ls2 | ls1 <> ls2
        implies ( ls1.RunnableItem -> intersection ( ls2.RunnableItem ) ) -> size ( ) = 0 )

```

Configuration

The **uniqueID** constraint. Every configuration has a unique identifier.

```

context Configuration :: inv uniqueID:
    self.allInstances -> forAll ( conf1, conf2 | conf1.ConfigID = conf2.ConfigID implies conf1 = conf2 )

```

Inventory

The **addOrdinalities** operation. The post condition checks all runnable items in the local set for which the inventory has information, and if any two unique runnable items are active and have the same source type this implies that the source type count for that type of item in the inventory is incremented. The post condition is TRUE. This operation is needed in the change-config operation of a configuration service so that it can count how many active runnable item types are in the local set and activate or deactivate some if the count is different than that requested in the desired configuration.

```

context Inventory :: addOrdinalities ( ) : Boolean
    post : self.LsinInven.FaultTolerantRunnableItem -> forAll ( ri1, ri2 |
        { ri1 != ri2 and ri1.isActive = true and ri2.isActive = true
          and ri1.srcType = ri2.srcType } ) implies self.typesInInven = ri1.srcType and
        self.typesInInven.count = self.typesInInven.count + 1
    and return = true

```

1. INTRODUCTION

Chapter 4 presents a method that identifies key configuration management needs of the bioinformatics system. The key configuration need of this system is to be able to define how the system deals with new or changed data. Related to this is being able to configure how incrementally the system operates, the amount of persistence in the system, and when to re-start analyses. Another key need is to be able to configure the security protection of data and unpublished results.

The focus of the bioinformatics system is on data and using data. The configuration management issues revolve around the actual functionality of the system. (This is unlike the ATM network monitoring system, where the configuration management issues revolve around managing dynamic software components in the system.) Specification of actual search and analysis functionality in the bioinformatics system is an orthogonal issue. This functionality is not considered in this research.

The specification method and notation proposed in Chapter 5 is:

1. Identify the system focus and related configuration management issues using system knowledge and key configuration management requirements. Use this step to identify potential items that need to be included in subsequent steps as well as in specification validation and verification (e.g. architectural views for multiple view analysis).

2. Use property refinement to obtain function-related statements that embody the system-wide configuration management behavioral constraints.
3. Use the soft goals technique to identify and further refine configuration management functionality and to identify links (e.g. satisficing and argumentation goals) between requirements.
4. Create a *type* model using the information gained in the previous steps. In conjunction with this step,
5. Create an Alloy model to complement the *type* model and include information from previous steps that extends the *type* model. The process of creating an Alloy model can suggest refinement changes to the *type* model and visa-versa.
6. If needed, add temporal constraints to the Alloy model using RTL-style comments.

Section 2 of this chapter presents an overview of the concepts and configuration issues in the bioinformatics system. Section 3 presents property refinement for a key run-time configuration management requirement and Section 4 presents the soft goal analysis for this system. Section 5 presents the *type* model of the run-time configuration management specification, and Section 6 presents the Alloy model of the specification. Section 7 presents temporal issues in the specification and Section 8 presents conclusions regarding the proposed specification method and notation.

2. STEP 1: BIOINFORMATICS SYSTEM FOCUS AND CONFIGURATION MANAGEMENT ISSUES

The focus of the bioinformatics system is *dynamic data*. Data is dynamic in two ways. It can either change frequently or new data can be added frequently. Two mechanisms that deal with dynamic data are incremental operation and persistence. Incremental operation occurs on searches and analyses of data, and persistence stores partial results and experimental or warehoused data. Security protection is needed for persisted items as well as for unpublished results and transient data. Over the course of a project, links need to be kept between data, analyses, searches, and results. A time-ordering of research efforts is required (productive and dead-end efforts). A concept called a research thread provides this ordering and linking function. A research thread drives work on the data and collects results as work progresses. The final results from a thread are usually publishable results that are the basis of future experiments or papers.

Table 1 shows ideas that are useful in subsequent specification steps. This table is developed as the system focus and run-time configuration management issues are identified.

3. STEP 2: PROPERTY REFINEMENT

The purpose of property refinement is to refine constraints on system-wide behavior into functionality that can be specified and implemented in software components. Property refinement refines system-wide behaviors of incremental operation and persistence into specifiable functionality. Areas of incremental operation and persistence that need to be configured while the system is running are identified,

along with configuration management operations that must be included in the system to support them.

3.1 Incremental Operation Property

Incremental operation of analyses and searches occurs over time and space. Examples of particular times are when new data is available, when data changes, or at some time interval. Examples of the space concept are over part of the action (i.e. analysis or search) or over part of the data. Data can be further refined into logical or geographical space. An important constraint in incremental operation is when to stop any particular action. Action can stop when some accuracy is achieved, when some elapsed time or resources have been used, or when some number of incremental steps has occurred. The definitions of time, space, and when to stop are all used to drive actions of analyses and searches. These definitions need to be configurable, or changeable during system operation.

Incremental operation concerns bioinformatics functionality (i.e. analyses and searches), whereas in the ATM network monitoring system it concerned software configuration. Care must be taken to specify run-time configuration management aspects of incremental operation, not the functionality of incremental analyses or searches. Therefore, the configurable aspects of incremental operation (i.e. over what time it takes place, over what space it takes place, and its duration) are the only things that need to be included in the run-time configuration management specification. In addition, only the run-time configuration management operations that relate to these configurable aspects need to be specified.

Table 1. Items useful in subsequent specification steps

Specification step	Ideas that may be useful in the step
Property refinement	Incremental operation, persistence
Soft goals	Softgoals: scale, dynamic environment, security Sub/satisficing goals: incremental operation, persistence
Type model	Types/Concepts: data source, data, research thread, work (analysis, search), results Configuration operations: define, calculate, bind, locate, instantiate, activate, register, deactivate, view
Validation/Verification step	Items that may be useful in the step
Multiple architectural views	Additional views: scale, dynamic environment, automatic operation, security, heterogeneous data

The items that need to be included in the run-time configuration management requirements specification are: 1) when to use incremental operation (e.g. when new data is available, when changed data is available, or after some elapsed time), 2) over what space incremental operation must occur (e.g. over analyses, over searches, or over a logical or geographical data source), and 3) when to stop any particular incremental work (e.g. based on accuracy, time taken, or number of iterations). The configuration management operations that need to be included in the specification are to *define* any of these items, and to *view* them (i.e. *use* them).

3.2 Persistence Property

Persistence occurs over time and space. Both of these aspects of persistence need to be configurable, and are included in the run-time configuration management

specification. Persisted data and results allow incremental analyses and searches to start from a previously calculated state. Analyses and searches must also store their results in the persisted storage. Data storage and retrieval operations are not detailed in subsequent specification steps since these actions are not related to run-time configuration management.³³

Items included in the run-time configuration management requirements specification are: 1) what to persist (e.g. work, results, and/or data), 2) what format to use (e.g. normal, compressed, and/or encrypted), and 3) whether to persist over time or over space. The configuration management operations that need to be included in the specification are to *define* the persistence configurable items and to *view* (or *use*) them.

3.3 Conclusions

Property refinement refines incremental operation and persistence into functionality constraints that must be present in the system software. Constraints track whether incremental operation and persistence obligations have been fulfilled at the component level. However, from the viewpoint of run-time configuration management, bioinformatics functionality statements related to incremental operation and persistence are an orthogonal issue to run-time configuration management. Only aspects that must be configured while the system is operating need to be included in the run-time configuration management specification (e.g. when to stop incremental analysis, or what data or results to persist across analyses). Configuration actions that will be used to configure these aspects also need to be included in the specification. Soft goals analysis identifies these configuration management actions.

³³ These operations and other persistence functionality (e.g. where it occurs) are orthogonal to the issue of run-time configuration management. They must be addressed in other bioinformatics functionality specifications.

4. STEP 3: SOFT GOALS

Soft goals refines requirements into sub-goals (sub-goals completely satisfy a higher level goal) and satisficing goals (goals that are necessary but not sufficient to complete a higher level goal; one satisficing goal may also help complete multiple higher level goals), and to establish argumentation goals (additional requirements that help prove a requirement has been met or that it cannot be met).

The bioinformatics system has three soft goals: 1) dealing with scale, 2) dealing with a dynamic environment, and 3) protecting data and results.³⁴ These soft goals can be refined into more sub-goals, satisficing goals, and argumentation goals. Argumentation goals exist between soft goals. Soft goal refinement is shown in Figure 9-1. (This soft goal analysis shows run-time configuration management, rather than functionality of the bioinformatics system. As a result, there may be bioinformatics-related refinements that should be included to support soft goals such as dealing with a dynamic environment. These bioinformatics-related sub- and satisficing goals are not shown in Figure 9-1. Therefore it appears that a single sub-goal, *incremental* operation, is sufficient to satisfy the *dynamic* environment soft goal.)

A refinement example is the *dynamic* environment soft goal. Dynamic environment refines into an *incremental* sub-goal, which in turn refines into a *persistence* sub-goal along with satisficing goals (i.e. *define*, *calculate partial result*, *return result*, and *restart analysis/search*). Each goal refines into satisficing and argumentation goals. A satisficing goal may contain many other satisficing goals. An example is *define*. The *define* satisficing goal of *persistence* includes defining what needs to be persisted, over what time and space,

³⁴ The KR method presented in Chapter 4 identified these three soft goals as key requirements of the bioinformatics system.

and constraints on when persistence definitions can be changed. (Due to space constraints, Figure 9-1 shows *define* satisficing goals as labeled bubbles. The associated satisficing and argumentation goals are shown inside the bubbles.)

Figure 9-1 shows the basic refinement of the main soft goals in the bioinformatics system from a run-time configuration management point of view. There are three soft goals shown in Figure 9-1. These are *scale*, *dynamic environment*, and *security*. Satisficing goals listed under *scale* are all goals that help the system deal with large amounts of data. This includes high volume, often changing, multiple source data. Argumentation goals shown under *scale* show how these satisficing goals help support *scale*. For example, *pre-calculating* results can support complex work in the context of doing *part of the work* at any given time. Dealing with *part of the data* helps support large volumes of quickly changing data from multiple sources.

The *dynamic* soft goal has one sub-goal, *incremental operation*. *Incremental operation* has one sub-goal, *persistence*. (Single sub-goals result from only analyzing soft goals from a run-time configuration management point of view. There may be additional bioinformatics-related functionality that also supports the original soft goals.) In addition, *incremental operation* has several satisficing goals, including *define*, *calculate partial results*, *return result*, and *restart analysis/search*. The *define* sub-goal contains several satisficing goals as well, including *OK to define*, which helps support *security*. Similarly, *persistence* has several satisficing goals, including *define*, *retrieve partial result*, and *persist partial result*. The *persistence define* sub-goal again has several satisficing goals, including *OK to define* which provides an argumentation goal that the *security* goal is met.

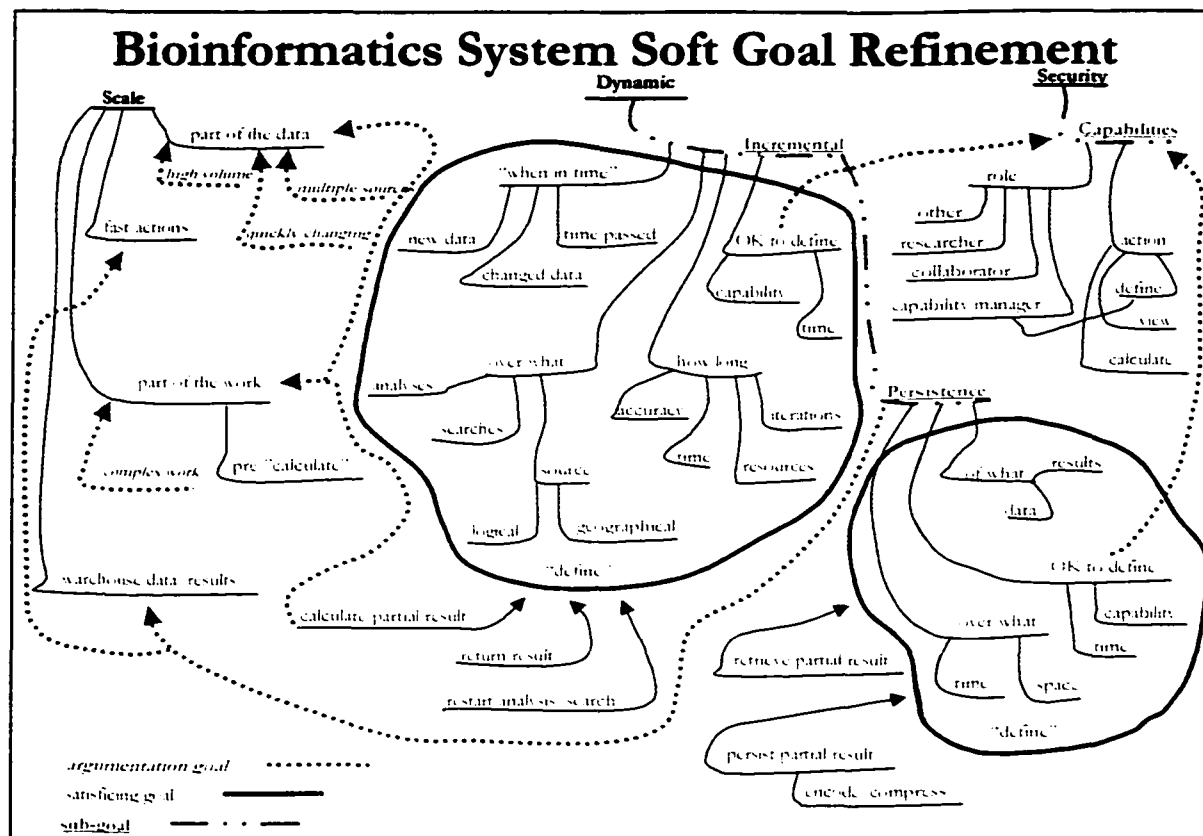


Figure 9-1. Refinement of major soft goals in the bioinformatics system, including sub-goals, satisficing goals, and argumentation goals within and between major soft goals. A dash-dot-dot line denotes sub-goals. Sub-goal names are bolded and underlined. Satisficing goals are shown in normal font and with solid lines and argumentation goals are shown in italicized font with dotted lines. The bubbles labeled "define" indicate satisficing goals.

The *security* soft goal has one sub-goal. This is *capabilities*. Role and action are satisficing goals for *capabilities*. One of the roles defined is that of a *capability manager*, which must define capabilities for all the other roles. (This is shown via a satisficing goal line between *capability manager* and *define* in Figure 9-1.)

4.1 Conclusions

The soft goal technique identifies run-time configuration management structuring operations. This technique identifies the *define* configuration management operation. The *define* operation is needed to configure aspects of *incremental operation*, *persistence*, and *capabilities* (e.g. when to use incremental operation, what to persist, or what

capabilities are assigned to each role). Operations included in the soft goals analysis are only related to the original soft goals of *scale*, *dynamic environment*, and *security*. Other configuration management operations are defined in the next step of the specification.

5. STEP 4: *TYPE* MODEL

Information from the previous three steps is used to create a *type* model. A *type* model consists of concepts, UML types, and the relations between them. In the UML, a “...Type is used to specify a domain of objects together with operations applicable to the objects without defining the physical implementation of those objects” ([63]). A concept is a domain of objects. (See [47] for a discussion of concepts.) A concept becomes a type when its operations are identified. Partial operation signatures are used to specify operations in the bioinformatics system. Specific operation return values are not needed at this level of abstraction, so operation signatures do not include return values.

Problem domain ideas are shown as concepts rather than attributes of existing concepts or types if they are shared by other concepts or types or if it is not clear what their data type should be (e.g. integer, Boolean, etc.).

5.1 General description of the *type* model

The bioinformatics *type* model shows the main problem domain ideas in the system and their relations. The *work* type is a generalization of analyses and searches. Work can be incremental and can also use persistence. The *WorkIncrement* type has relations to its configuration aspects identified by property refinement. Similarly, the *persistence* type has relations to its configurable aspects, identified by property refinement. *Persistence* also has relations to persistent stores, even though this relation is more closely

related to the actual functionality of the bioinformatics system rather than to its run-time configuration management. This is consistent with other relations between types and concepts that exist for bioinformatics-related functionality reasons rather than for run-time configuration management reasons. An example is the *workUsesData* relation between *Work* and *Data* in a *PersistentStore*. These relations are included in the *type* model to aid understanding of how run-time configuration management fits into overall bioinformatics functionality. The relations also overlap bioinformatics functional specifications that can aid in both understanding and in internal consistency checks.

The concept of a persistent store is shown in the *type* model, and it contains data (including results). Data can be partial. (This is the incremental concept again, but related to data.) *DataSources* are geographically dispersed databases, or other data sources that store enormous amounts of information. The *researchThread* type drives work and collects results. The *role* type and *capability* concept are included to satisfy the security soft goal. The *capabilityManager* role defines other roles as well as capabilities for all roles. These capabilities define what operations can be used in *ResearchThread*, *Work*, *DataSource*, *PersistentStore*, and *Persistence*.

Registries locate concepts and types (data sources, work, research threads, persistent stores, and roles). A client is also present in the *type* model. A client is not necessarily a software concept; it can be a researcher using the bioinformatics system. Some concepts and types have run-time configuration management operations in common; these are collected in interfaces for clarity. Soft goal analysis and the preliminary configuration matrix developed in Chapter 4 identify configuration management operations.

5.2 Type Model Figures

Figures 9-2 through 9-6 show the *type* model of the bioinformatics run-time configuration management requirements specification. Figure 9-2 shows concepts and types of persistent storage, data, results, data sources, research threads, work, and persistence. Figure 9-3 shows specializations of work: analysis, search, work increment, and pre-calculate. Work is completely partitioned into analyses or searches (these specializations of work are disjoint and complete). Work can be incremental, and/or part of a pre-calculation. (A pre-calculation is a guess by the system that an analysis or a search, not specifically requested by a user, will be needed at some point in the near future.) *WorkIncrement* has relations to configurable aspects of incremental operation. Figure 9-4 shows the registry type. Registries provide secure access to concepts and types through *register* and *locate* operations that use roles and capabilities. (*Register* and *locate* are equivalent to *define* and *view* operations). Figure 9-5 shows the role type and capability concept and their relations to the other system concepts and types. Finally, Figure 9-6 shows client interaction with other system types. A client corresponds to one of the system roles: researcher collaborator, capability manager, or some other person.

6. STEP 5: USING ALLOY TO COMPLEMENT THE TYPE MODEL

Alloy is a modeling notation, supported by a tool designed to test assertions, invariants, and operations of an Alloy model. An Alloy model consists of several parts: a list of sets that are fundamental to the model (*domain* sets), other sets defined from domain sets, relations between sets, relation definitions, invariants, assertions, and operations.

The *type* model described in Section 5 does not include any OCL expressions, so constraints and operation definitions are not included in the *type* model. The Alloy model presented in this section is initially created from the *type* model, using the *type* model to define domain sets, some subsets, and most of the relations. The Alloy model also includes constraints and invariants on system behavior as well as operation definitions. In this way, the Alloy model complements and extends the *type* model. The Alloy model presented in this chapter complies and can be analyzed using the associated Alloy tool. Analysis of the Alloy model is discussed in Chapter 10.

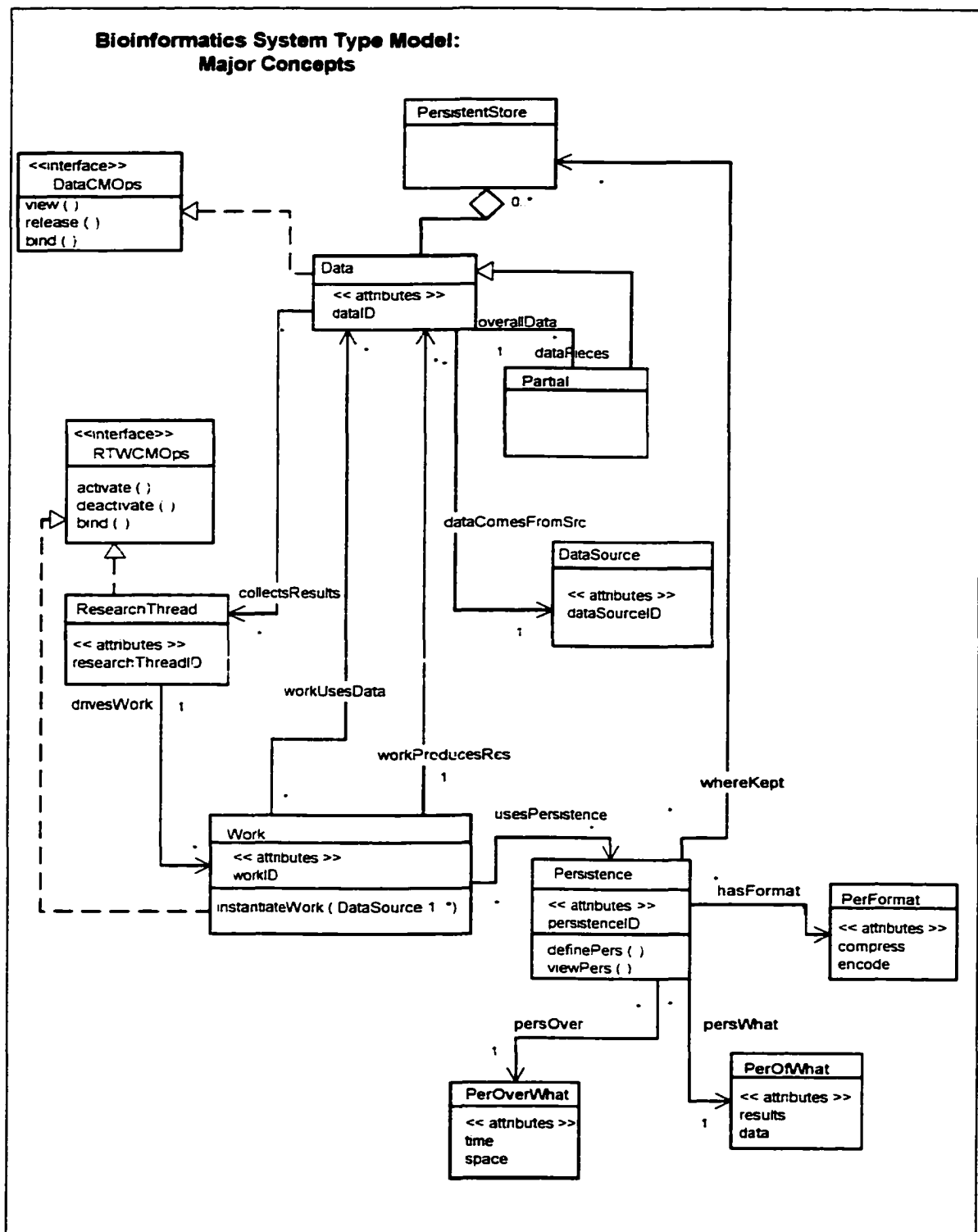


Figure 9-2. Conceptual model of bioinformatics system concepts. *ResearchThreads* drive *Work* in the system (either an *Analysis* or a *Search*), and collect results. *Work* uses *Persistence* and a *PersistentStore*. *DataSources* provide *Data* to a *PersistentStore*.

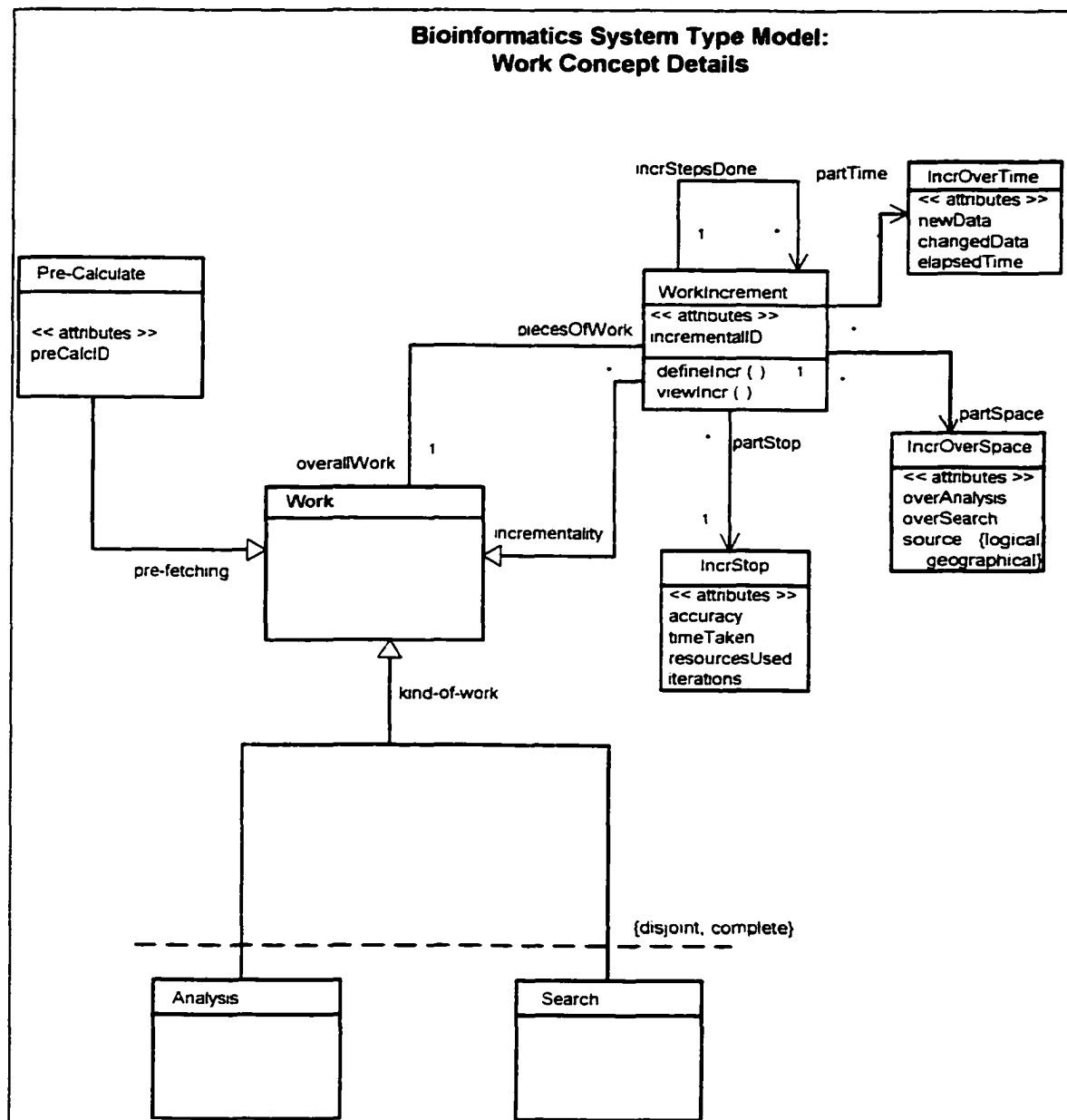


Figure 9-3. Specializations of the *Work* type. *Work* is either an *analysis* or a *search*. *Work* can also be *incremental* and/or *pre-calculated* (i.e. a guess by the system that the results of an analysis or search will be needed later).

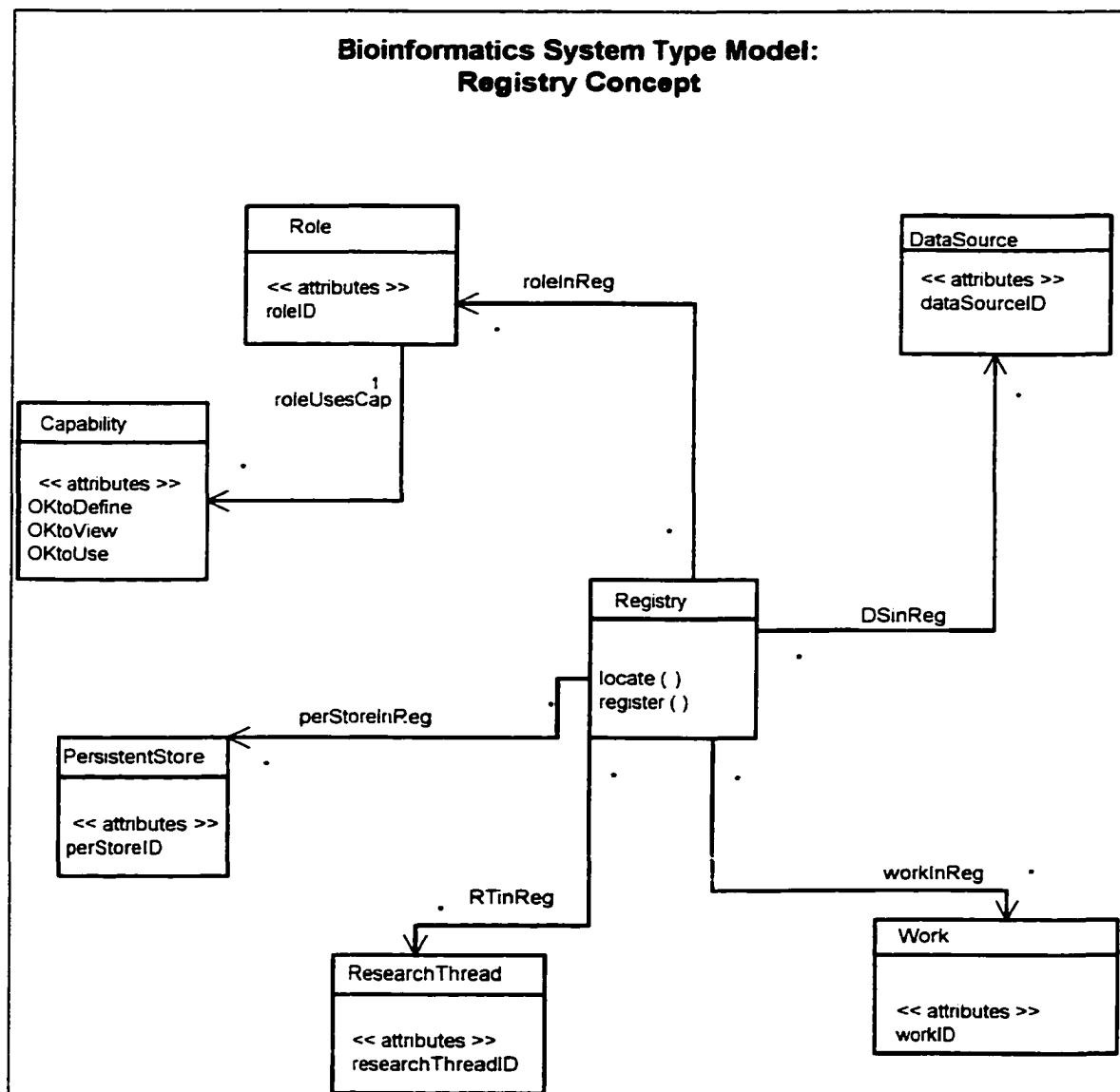


Figure 9-4. The *Registry* type and its relations with other types.

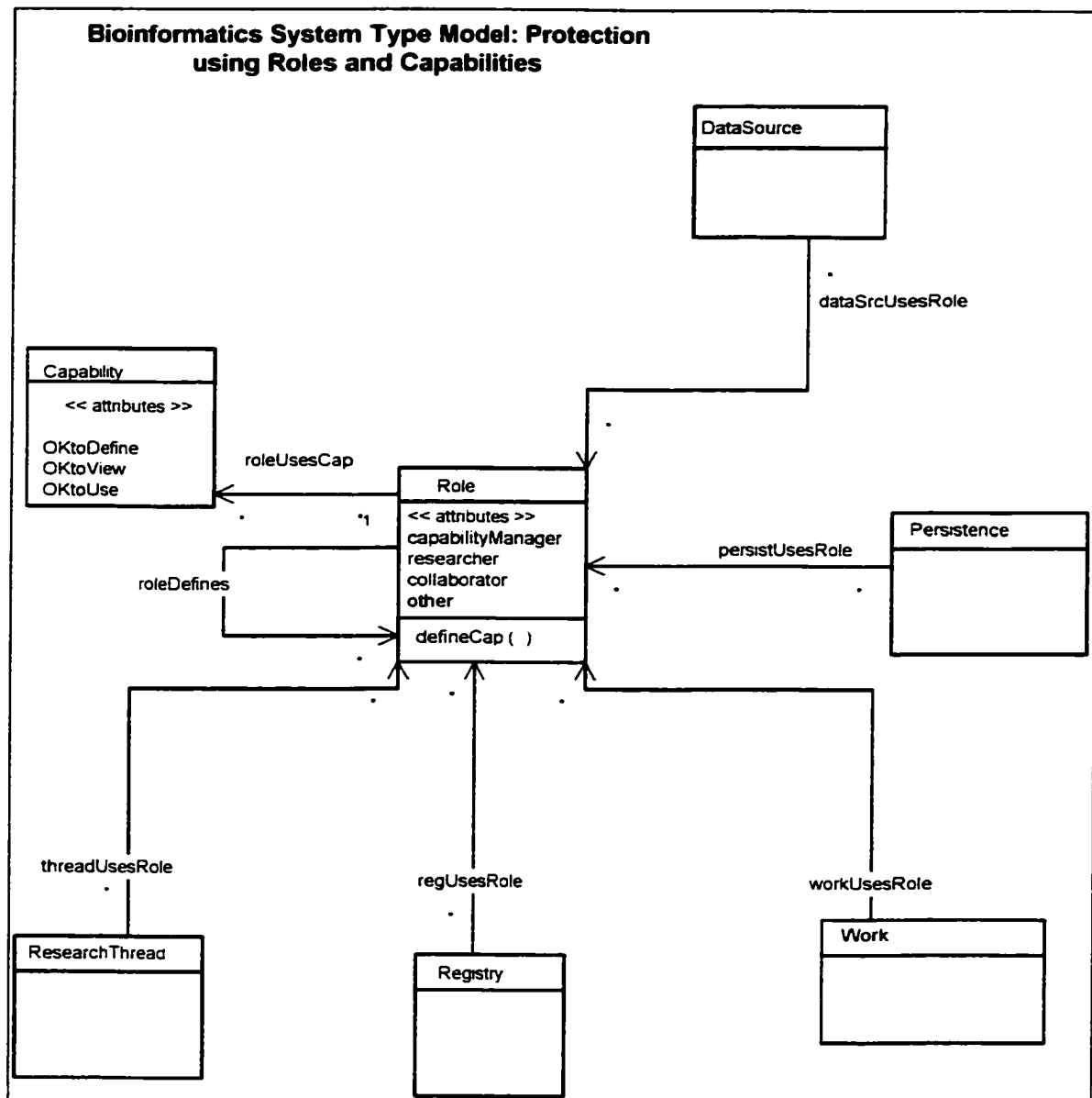


Figure 9-5. *Role* and *Capability* types in the bioinformatics system.

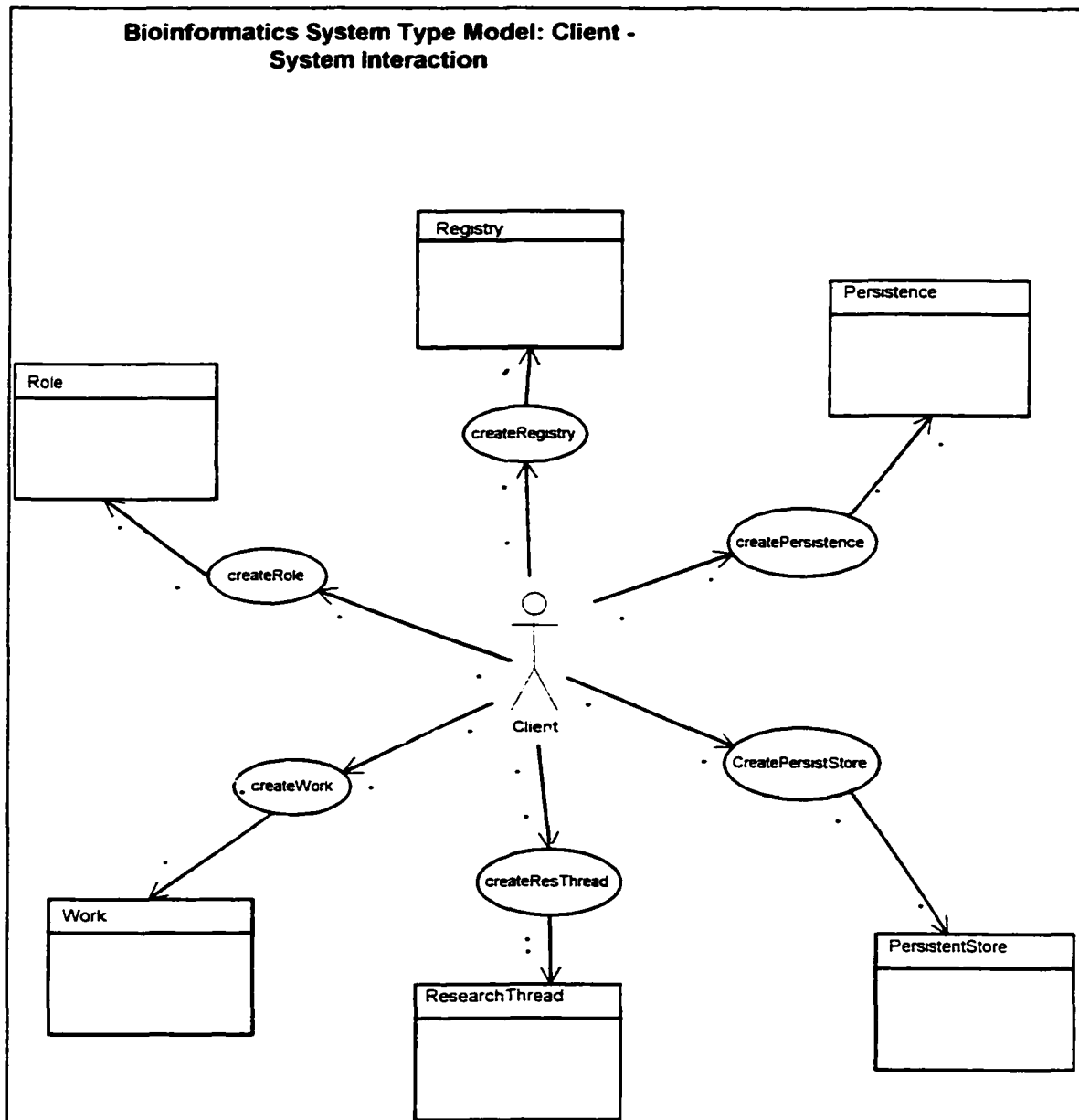


Figure 9-6. Interaction of a client with the bioinformatics system. The interaction creates one of the other types, or to accesses them via a registry.

6.1 Bioinformatics run-time configuration management specification Alloy model

```
model bioinformaticsCM
```

```
{
```

```
// First define the domain sets in the model.
```

```
domain { DataSource, Role, Capability, Data, OpAttempted, ReturnValue, Identifier, Work, IncrOverTime,
IncrOverSpace, IncrStop, PersistentStore, Result, Persistence, PersFormat, PersOfWhat, PersOverWhat,
ResearchThread, Registry }
```

```
// Next define the state: the subsets and relations.
```

```
state
```

```
{
```



```

// First define subsets of the domain sets.

// Work can be either an analysis or a search. In addition, it can be incremental or part of a
// pre-calculation.
    partition analysis, search : Work
    incremental : Work
    preCalculate : Work

// Data can also be incremental, or partial.
    partial : Data

// These are the kinds of capabilities in the system. Together they create the set Capability. They do not
// partition the set, so there is a 'def' statement below that creates the Capability set from them.
    OKtoDefine : Capability !
    OKtoUse : Capability !
    OKtoView : Capability !

// These are the definitions of incremental work. Incremental has to be defined over time, space, and
// duration (i.e. when to stop). When to do incremental work can be when new data is available, when
// data
// changes, or after some elapsed time. Incremental work can be done over an analysis, a search, or over
// data sources, which can be logical or geographical. Incremental work can be stopped after some
// accuracy is achieved, some time is taken, some resources are used, or some number of iterations occur.
    newData : IncrOverTime !
    changedData : IncrOverTime !
    elapsedTime : IncrOverTime !
    overAnalysis : IncrOverSpace !
    overSearch : IncrOverSpace !
    source : IncrOverSpace !
    logical : source !
    geographical : source !
    accuracy : IncrStop !
    timeTaken : IncrStop !
    resourcesUsed : IncrStop !
    iterations : IncrStop !

// These are the definitions of persistence. They include the format (compressed and/or encoded
// somehow),
// what should be persisted (data or results), and over what time (elapsed time or space taken).
    compress : PersFormat !
    encode : PersFormat !
    results : PersOfWhat !
    data : PersOfWhat !
    time : PersOverWhat !
    space : PersOverWhat !

// These are the operations that are defined.
    partition registered, located, instantiated, activated, deactivated, defined, viewed, bound, released :
    OpAttempted

// Operation return values.
    partition OK, Error : ReturnValue

// Next define the relations.

// Used to get results back from the operations.
    FeedBack : OpAttempted -> ReturnValue !

// Used to determine whether a role has the capability to perform some operation.
    roleUsesCap : Role ! -> Capability

// Relations having to do with research threads. These are from the UML concept models.
    drivesWork : ResearchThread ! -> Work
    collectsResults : ResearchThread -> Result

// Relations for work. From the UML concept models.

```

```

workProducesRes : Work ! -> Result
workUsesData : Work -> Data

// Relations for data. From the UML concept model.
dataComesFromSrc : Data -> DataSource !
dataPieces ( ~ overallData ) : Data ! -> partial

// Relations used to locate items. Some are used by the registry.
dataID : Data ! -> Identifier !
incrementalID : incremental ! -> Identifier !
persistenceID : Persistence ! -> Identifier !
roleID : Role ! -> Identifier !
workID : Work ! -> Identifier !
dataSourceID : DataSource ! -> Identifier !
researchThreadID : ResearchThread ! -> Identifier !
perStoreID : PersistentStore ! -> Identifier !

// Relations used by the registry for locating and registering items.
dsInRegistry ( ~ regHasDS ) : DataSource -> Registry
roleInRegistry ( ~ regHasRole ) : Role -> Registry
threadInRegistry ( ~ regHasRT ) : ResearchThread -> Registry
workInRegistry ( ~ regHasWork ) : Work -> Registry
perStoreInRegistry ( ~ regHasPerStore ) : PersistentStore -> Registry

// Relations that help define incremental work.
partStop : incremental -> IncrStop !
partSpace : incremental -> IncrOverSpace !
partTime : incremental -> IncrOverTime !

// Relations about work and incremental operation.
incrStepsDone : incremental ! -> incremental
overallWork ( ~ piecesOfWork ) : Work -> Work !

// Relations that help define persistence.
hasFormat : Persistence -> PersFormat !
persWhat : Persistence -> PersOfWhat !
persOver : Persistence -> PersOverWhat !

// Relations about work, data, results, and persistence.
usesPersistence ( ~ workUsingP ) : Work -> Persistence
dataInPerStorage : PersistentStore ! -> Data
resultsInPerStorage : PersistentStore ! -> Result
whereKept : Persistence -> PersistentStore

}

// Next add definitions and invariants.

// Define Capability in terms of its subsets, and add an invariant that the subsets are disjoint.
def Capability { Capability = OKtoDefine + OKtoUse + OKtoView }
inv disjointCap { ! ( OKtoDefine = OKtoUse ) && ! ( OKtoDefine = OKtoView ) && ! ( OKtoUse = OKtoView ) }
}

// Define the portions of incremental work in terms of their subsets. Unlike Capability, the subsets do not
// have to be disjoint.
def IncrOverTime { IncrOverTime = newData + changedData + elapsedTime }
def IncrOverSpace { IncrOverSpace = overAnalysis + overSearch + source }
def source { source = logical + geographical }
def IncrStop { IncrStop = accuracy + timeTaken + resourcesUsed + iterations }

// Define the portions of persistence in terms of their subsets. Unlike Capability, the subsets do not
// have to be disjoint.
def PersFormat { PersFormat = compress + encode }
def PersOfWhat { PersOfWhat = results + data }
def PersOverWhat { PersOverWhat = time + space }

```

```

// Next define operations.

// Operations for DATA. These include binding it to a data source, viewing a particular data, and releasing
// a
// data source.

// Bind data to a data source by creating the dataComesFromSrc relation. This operation can only take
// place if the role that is passed in has the USE capability. If so, the feedback is OK, otherwise there is
// an error.
op bindData ( whoWantsTo : Role !, dataSrc : DataSource !, dat : Data ! ) {

// Another bindData operation cannot start before the current one is complete.
// all i1 : integer, t1, t2 : Time | ( theta ( begin ( bindData, i1, t1 ) && theta ( end ( bindData, i1, t2 )
//      && ( t2 > t1 ) )
//      -> no i2 : integer, t3 : Time | ( theta ( begin ( bindData, i2, t3 ) && ( t2 > t3 > t1 ) && ( i2 > i1 ) ) )
//
// A releaseData operation cannot start before the current bindData operation completes.
// all i1 : integer, t1, t2 : Time | theta ( begin ( bindData, i1, t1 ) && theta ( end ( bindData, i1, t2 )
//      && ( t2 > t1 ) )
//      -> no i2 : integer, t3 : Time | ( theta ( begin ( releaseData, i2, t3 ) && ( t2 > t3 > t1 ) ) )
//
// A viewData operation cannot start before the current bindData operation is complete.
// all i1 : integer, t1, t2 : Time | theta ( begin ( bindData, i1, t1 ) && theta ( end ( bindData, i1, t2 )
//      && ( t2 > t1 ) )
//      -> no i2 : integer, t3 : Time | theta ( begin ( viewData, i2, t3 ) && ( t2 > t3 > t1 ) )

    ( OKtoDefine in whoWantsTo.roleUsesCap && dat.dataComesFromSrc' = dataSrc && bound.FeedBack'
in OK )
    || ( ! ( OKtoDefine in whoWantsTo.roleUsesCap ) && bound.FeedBack' in Error )
}

// View data. Data is located via the identifier passed in. This operation can only take place if the role
// passed in has the VIEW capability. If so, the feedback is OK, otherwise there is an error.
op viewData ( whoWantsTo : Role !, dID : Identifier ! ) {

// A releaseData operation cannot start before the current viewData operation completes.
// all i1 : integer, t1, t2 : Time | theta ( begin ( viewData, i1, t1 ) && theta ( end ( viewData, i1, t2 )
//      && ( t2 > t1 ) )
//      -> no i2 : integer, t3 : Time | ( theta ( begin ( releaseData, i2, t3 ) && ( t2 > t3 > t1 ) ) )
//
// A bindData operation cannot start before the current viewData operation is complete.
// all i1 : integer, t1, t2 : Time | theta ( begin ( viewData, i1, t1 ) && theta ( end ( viewData, i1, t2 )
//      && ( t2 > t1 ) )
//      -> no i2 : integer, t3 : Time | theta ( begin ( bindData, i2, t3 ) && ( t2 > t3 > t1 ) )

    ( some dat : Data | dat.dataID = dID && OKtoView in whoWantsTo.roleUsesCap
&& viewed.FeedBack' in OK )
    || ( ! ( OKtoView in whoWantsTo.roleUsesCap ) && viewed.FeedBack' in Error )
}

// Unbind data from a data source. This is accomplished by making sure there is no dataComesFromSrc
// relation from the data to any data source. This operation can only work if the role passed in has the
// DEFINE capability. If so, the feedback is OK, otherwise there is an error.
op releaseData ( whoWantsTo : Role !, dat : Data ! ) {

// Another releaseData operation cannot start before the current one is complete.
// all i1 : integer, t1, t2 : Time | ( theta ( begin ( releaseData, i1, t1 ) && theta ( end ( releaseData, i1, t2 )
//      && ( t2 > t1 ) )
//      -> no i2 : integer, t3 : Time | ( theta ( begin ( releaseData, i2, t3 ) && ( t2 > t3 > t1 ) && ( i2 > i1 ) ) )
//
// A bindData operation cannot start before the current releaseData operation completes.
// all i1 : integer, t1, t2 : Time | theta ( begin ( releaseData, i1, t1 ) && theta ( end ( releaseData, i1, t2 )
//      && ( t2 > t1 ) )
//      -> no i2 : integer, t3 : Time | ( theta ( begin ( bindData, i2, t3 ) && ( t2 > t3 > t1 ) ) )
//
// A viewData operation cannot start before the current releaseData operation is complete.

```

```

// all i1 : integer, t1, t2 : Time | theta { begin (releaseData, i1, t1) && theta { end (releaseData, i1, t2)
//      && ( t2 > t1 ) }
//      -> no i2 : integer, t3 : Time | theta { begin ( viewData, i2, t3 ) && ( t2 > t3 > t1 ) }

    { no ds : DataSource | OKtoDefine in whoWantsTo.roleUsesCap && dat.dataComesFromSrc' = ds
      && released.FeedBack' in OK }
    | { ! ( OKtoDefine in whoWantsTo.roleUsesCap ) && released.FeedBack' in Error }
  }

// Operations for INCREMENTAL. These include defining and viewing settings.

// Define the incremental settings: over what time, over what space, and when to stop. This can only take
// place if the role passed in has the DEFINE capability. If so the operation succeeds, otherwise an
// error occurs.
op defineIncr ( whoWantsTo : Role !, incr : incremental !, overT : IncrOverTime !, overS : IncrOverSpace !,
iStop : IncrStop ! ) {

// Another defineIncr operation cannot start before the current one is complete.
// all i1 : integer, t1, t2 : Time | { theta { begin ( defineIncr, i1, t1 ) && theta { end (defineIncr, i1, t2)
//      && ( t2 > t1 ) }
//      -> no i2 : integer, t3 : Time | { theta { begin (defineIncr, i2, t3) && ( t2 > t3 > t1 ) && ( i2 > i1 ) }
//
// A viewIncr operation cannot start before the current defineIncr operation completes.
// all i1 : integer, t1, t2 : Time | theta { begin (defineIncr, i1, t1) && theta { end (defineIncr, i1, t2)
//      && ( t2 > t1 ) }
//      -> no i2 : integer, t3 : Time | { theta { begin ( viewIncr, i2, t3 ) && ( t2 > t3 > t1 ) }

    { OKtoDefine in whoWantsTo.roleUsesCap && incr.partTime' = overT && incr.partSpace' = overS &&
incr.partStop' = iStop && defined.FeedBack' in OK }
    | { ! ( OKtoDefine in whoWantsTo.roleUsesCap ) && defined.FeedBack' in Error }
  }

// View incremental settings. This is done by matching an incremental item to the identifier passed in.
// This operation can only take place if the role passed in has the VIEW capability. Otherwise an error
// occurs.
op viewIncr ( whoWantsTo : Role !, iID : Identifier ! ) {

// A defineIncr operation cannot start before the current viewIncr operation completes.
// all i1 : integer, t1, t2 : Time | theta { begin (viewIncr, i1, t1) && theta { end (viewIncr, i1, t2)
//      && ( t2 > t1 ) }
//      -> no i2 : integer, t3 : Time | { theta { begin ( defineIncr, i2, t3 ) && ( t2 > t3 > t1 ) }

    { some incr : incremental | incr.incrementalID = iID && OKtoView in whoWantsTo.roleUsesCap
      && viewed.FeedBack' in OK }
    | { ! ( OKtoView in whoWantsTo.roleUsesCap ) && viewed.FeedBack' in Error }
  }

// Operations for PERSISTENCE. These include defining and viewing settings.

// Define persistence settings as well as the relation to persistent stores. This operation can only take
// place if the role passed in has the DEFINE capability. Otherwise an error occurs. The relation to the
// persistent store (whereKept) is set and then the data and result items associated with the work using
// persistence are associated with the persistent store via the dataInPerStorage and resultsInPerStorage
// relations.
op definePers ( whoWantsTo : Role !, pers : Persistence !, format : PersFormat !, ofWhat : PersOfWhat !,
overWhat : PersOverWhat ! ) {

// Another definePers operation cannot start before the current one is complete.
// all i1 : integer, t1, t2 : Time | { theta { begin ( definePers, i1, t1 ) && theta { end (definePers, i1, t2)
//      && ( t2 > t1 ) }
//      -> no i2 : integer, t3 : Time | { theta { begin (definePers, i2, t3) && ( t2 > t3 > t1 ) && ( i2 > i1 ) }
//
// A viewPers operation cannot start before the current definePers operation completes.
// all i1 : integer, t1, t2 : Time | theta { begin (definePers, i1, t1) && theta { end (definePers, i1, t2)
//      && ( t2 > t1 ) }
//      -> no i2 : integer, t3 : Time | { theta { begin ( viewPers, i2, t3 ) && ( t2 > t3 > t1 ) }

```

```

    ( some perSt : PersistentStore | all wrk : ( pers.workUsingP )
      | all dat : ( wrk.workUsesData ), res : ( wrk.workProducesRes )
      | perSt in pers.whereKept' && dat in perSt.dataInPerStorage' && res in
perSt.resultsInPerStorage'
      && OKtoDefine in whoWantsTo.roleUsesCap && pers.hasFormat' = format
      && pers.persWhat' = ofWhat && pers.persOver' = overWhat && defined.FeedBack' in OK )
    | | ( ! ( OKtoDefine in whoWantsTo.roleUsesCap ) && defined.FeedBack' in Error )
  }

// View persistence settings. This is done via matching the persInceID relation with the identifier
// passed in. This operation can only succeed if the role passed in has the VIEW capability. Otherwise
// an error occurs.
op viewPers ( whoWantsTo : Role !, pID : Identifier ! ) {

// A definePers operation cannot start before the current viewPers operation completes.
// all i1 : integer, t1, t2 : Time | theta ( begin (viewPers, i1, t1) && theta ( end (viewPers, i1, t2)
//   && ( t2 > t1 ) )
//   -> no i2 : integer, t3 : Time | ( theta ( begin (definePers, i2, t3) && ( t2 > t3 > t1 ) ) )

  ( some pers : Persistence | pers.persInceID = pID && OKtoView in whoWantsTo.roleUsesCap
    && viewed.FeedBack' in OK )
  | | ( ! ( OKtoView in whoWantsTo.roleUsesCap ) && viewed.FeedBack' in Error )
}

// Operations for WORK. These include binding work data to data sources, instantiating work
// (which includes binding it to results), activating a part of incremental work, and deactivating work
// (which includes unbinding data and results).

// Bind a work item's data to a single data source. This operation can only take place if the role passed
// in has the DEFINE capability. Otherwise an error occurs. The operation works by creating a
// dataComesFromSrc relation between the data item(s) and the data source.
op bindWork ( whoWantsTo : Role !, dataSrc : DataSource !, myWork : Work ! ) {

// Another bindWork operation cannot start before the current one is complete.
// all i1 : integer, t1, t2 : Time | ( theta ( begin ( bindWork, i1, t1 ) && theta ( end (bindWork, i1, t2)
//   && ( t2 > t1 ) ) )
//   -> no i2 : integer, t3 : Time | ( theta ( begin (bindWork, i2, t3) && ( t2 > t3 > t1 ) && ( i2 > i1 ) ) )
//
// A activateWork operation cannot start before the current bindWork operation completes.
// all i1 : integer, t1, t2 : Time | theta ( begin (bindWork, i1, t1) && theta ( end (bindWork, i1, t2)
//   && ( t2 > t1 ) )
//   -> no i2 : integer, t3 : Time | ( theta ( begin (activateWork, i2, t3) && ( t2 > t3 > t1 ) ) )
//
// A deactivateWork operation cannot start before the current bindWork operation is complete.
// all i1 : integer, t1, t2 : Time | theta ( begin (bindWork, i1, t1) && theta ( end (bindWork, i1, t2)
//   && ( t2 > t1 ) )
//   -> no i2 : integer, t3 : Time | theta ( begin (deactivateWork, i2, t3) && ( t2 > t3 > t1 ) )

  ( some dat : Data | OKtoDefine in whoWantsTo.roleUsesCap && dat in myWork.workUsesData'
    && dat.dataComesFromSrc' = dataSrc && bound.FeedBack' in OK )
  | | ( ! ( OKtoDefine in whoWantsTo.roleUsesCap ) && bound.FeedBack' in Error )
}

// Instantiate work. This operation can only work if the role passed in has the DEFINE capability.
// Otherwise an error occurs. The operation works by calling 'bindWork' to bind data items to each of
// data sources passed in, then binding the work to result item(s) using the workProducesRes relation.
op instantiateWork ( whoWantsTo : Role !, dataSrcSet : DataSource, myWork : Work ! ) {

// Another instantiateWork operation cannot start before the current one is complete.
// all i1 : integer, t1, t2 : Time | ( theta ( begin (instantiateWork, i1, t1) &&
//   theta ( end (instantiateWork, i1, t2) && ( t2 > t1 ) ) )
//   -> no i2 : integer, t3 : Time | ( theta ( begin (instantiateWork, i2, t3) && ( t2 > t3 > t1 ) && ( i2 > i1 ) ) )
//
// A bindWork operation cannot start before the current instantiateWork operation completes.

```

```

// all i1 : integer, t1, t2 : Time | theta ( begin (instantiateWork, i1, t1) && theta ( end (instantiateWork, i1, t2)
//      && ( t2 > t1 ) )
//      -> no i2 : integer, t3 : Time | ( theta ( begin (bindWork, i2, t3) && ( t2 > t3 > t1 ) ) )
//
// An activateWork operation cannot start before the current instantiateWork operation is complete.
// all i1 : integer, t1, t2 : Time | theta ( begin (instantiateWork, i1, t1) && theta ( end (instantiateWork, i1, t2)
//      && ( t2 > t1 ) ) )
//      -> no i2 : integer, t3 : Time | theta ( begin (activateWork, i2, t3) && ( t2 > t3 > t1 ) )
//
// A deactivateWork operation cannot start before the current instantiateWork operation is complete.
// all i1 : integer, t1, t2 : Time | theta ( begin (instantiateWork, i1, t1) && theta ( end (instantiateWork, i1, t2)
//      && ( t2 > t1 ) ) )
//      -> no i2 : integer, t3 : Time | theta ( begin (deactivateWork, i2, t3) && ( t2 > t3 > t1 ) )

    ( ( all ds : ( dataSrcSet ) | bindWork ( whoWantsTo, ds, myWork ) )
      && some res : Result | OKtoDefine in whoWantsTo.roleUsesCap && res in myWork.workProducesRes'
      && instantiated.FeedBack in OK )
    | | ( ! ( OKtoDefine in whoWantsTo.roleUsesCap ) && instantiated.FeedBack' in Error )
  }

// Activate work. This operation can only work if the role passed in has the USE capability. Otherwise an
// error occurs. This operation declares that work to activate is incremental, that the overall type of work is
// also incremental, and that the current work gets added to the set of incremental steps done for the
// overall work.
op activateWork ( whoWantsTo : Role !, currWork : Work ! ) {

// A bindWork operation cannot start before the current activateWork operation completes.
// all i1 : integer, t1, t2 : Time | theta ( begin (activateWork, i1, t1) && theta ( end (activateWork, i1, t2)
//      && ( t2 > t1 ) )
//      -> no i2 : integer, t3 : Time | ( theta ( begin (bindWork, i2, t3) && ( t2 > t3 > t1 ) ) )
//
// An activateWork operation cannot start before the current activateWork operation is complete.
// all i1 : integer, t1, t2 : Time | theta ( begin (activateWork, i1, t1) && theta ( end (activateWork, i1, t2)
//      && ( t2 > t1 ) ) )
//      -> no i2 : integer, t3 : Time | theta ( begin (activateWork, i2, t3) && ( t2 > t3 > t1 ) )
//
// A deactivateWork operation cannot start before the current activateWork operation is complete.
// all i1 : integer, t1, t2 : Time | theta ( begin (activateWork, i1, t1) && theta ( end (activateWork, i1, t2)
//      && ( t2 > t1 ) ) )
//      -> no i2 : integer, t3 : Time | theta ( begin (deactivateWork, i2, t3) && ( t2 > t3 > t1 ) )

    ( OKtoUse in whoWantsTo.roleUsesCap && currWork in incremental && currWork.overallWork in
    incremental
      && currWork in currWork.overallWork.incrStepsDone' && activated.FeedBack' in OK )
    | | ( ! ( OKtoUse in whoWantsTo.roleUsesCap ) && activated.FeedBack' in Error )
  }

// Deactivate work. This operation can only take place if the role passed in has both the USE and DEFINE
// capabilities. Otherwise an error occurs. The operation works by ensuring that there are no data items
// nor result items related to the work via the workUsesData and workProducesRes relations, respectively.
op deactivateWork ( whoWantsTo : Role !, myWork : Work ! ) {

// Another deactivateWork operation cannot start before the current one is complete.
// all i1 : integer, t1, t2 : Time | ( theta ( begin (deactivateWork, i1, t1) &&
//      theta ( end (deactivateWork, i1, t2) && ( t2 > t1 ) ) )
//      -> no i2 : integer, t3 : Time | ( theta ( begin (deactivateWork, i2, t3) && ( t2 > t3 > t1 ) && ( i2 > i1 ) ) )
//
// A bindWork operation cannot start before the current deactivateWork operation completes.
// all i1 : integer, t1, t2 : Time | theta ( begin (deactivateWork, i1, t1) &&
//      theta ( end (deactivateWork, i1, t2) && ( t2 > t1 ) ) )
//      -> no i2 : integer, t3 : Time | ( theta ( begin (bindWork, i2, t3) && ( t2 > t3 > t1 ) ) )
//
// An instantiateWork operation cannot start before the current deactivateWork operation is complete.
// all i1 : integer, t1, t2 : Time | theta ( begin (deactivateWork, i1, t1) &&
//      theta ( end (deactivateWork, i1, t2) && ( t2 > t1 ) ) )
//      -> no i2 : integer, t3 : Time | theta ( begin (instantiateWork, i2, t3) && ( t2 > t3 > t1 ) )

```

```

//
// An activateWork operation cannot start before the current deactivateWork operation is complete.
// all i1 : integer, t1, t2 : Time | theta ( begin (deactivateWork, i1, t1) &&
//      theta ( end (deactivateWork, i1, t2) && ( t2 > t1 ) )
//      -> no i2 : integer, t3 : Time | theta ( begin (activateWork, i2, t3) && ( t2 > t3 > t1 ) )
//
( no dat : Data, res : Result | OKtoUse in whoWantsTo.roleUsesCap
    && OKtoDefine in whoWantsTo.roleUsesCap && dat in myWork.workUsesData'
    && res in myWork.workProducesRes' && deactivated.FeedBack' in OK )
| | ( ! ( OKtoUse in whoWantsTo.roleUsesCap ) | ! ( OKtoDefine in whoWantsTo.roleUsesCap ) )
    && deactivated.FeedBack' in Error )
}

// Operations for RESEARCH THREADS. These include binding a research thread to some work.
// activating and deactivating a research thread.

// Bind a research thread to work. This operation can only work if the role passed in has the DEFINE
// capability. Otherwise an error occurs. The operation works by creating drivesWork relations between
// the research thread and the work item(s).
op bindRT ( whoWantsTo : Role !, currRT : ResearchThread !, work : Work ) {

// Another bindRT operation cannot start before the current one is complete.
// all i1 : integer, t1, t2 : Time | ( theta ( begin (bindRT, i1, t1) && theta ( end (bindRT, i1, t2)
//      && ( t2 > t1 ) )
//      -> no i2 : integer, t3 : Time | ( theta ( begin (bindRT, i2, t3) && ( t2 > t3 > t1 ) && ( t2 > i1 ) ) )
//
// An activateRT operation cannot start before the current bindRT operation completes.
// all i1 : integer, t1, t2 : Time | theta ( begin (bindRT, i1, t1) && theta ( end (bindRT, i1, t2)
//      && ( t2 > t1 ) )
//      -> no i2 : integer, t3 : Time | ( theta ( begin (activateRT, i2, t3) && ( t2 > t3 > t1 ) ) )
//
// A deactivateRT operation cannot start before the current bindRT operation is complete.
// all i1 : integer, t1, t2 : Time | theta ( begin (bindRT, i1, t1) && theta ( end (bindRT, i1, t2)
//      && ( t2 > t1 ) ) )
//      -> no i2 : integer, t3 : Time | theta ( begin (deactivateRT, i2, t3) && ( t2 > t3 > t1 ) )

( work in currRT.drivesWork' && work.workProducesRes in currRT.collectsResults'
    && OKtoDefine in whoWantsTo.roleUsesCap && bound.FeedBack' in OK )
| | ( ! ( OKtoDefine in whoWantsTo.roleUsesCap ) && bound.FeedBack' in Error )
}

// Activate a research thread. This operation only ensures that the role passed in has the USE capability.
// Otherwise an error occurs.
op activateRT ( whoWantsTo : Role !, currRT : ResearchThread ! ) {

// A bindRT operation cannot start before the current activateRT operation completes.
// all i1 : integer, t1, t2 : Time | theta ( begin (activateRT, i1, t1) && theta ( end (activateRT, i1, t2)
//      && ( t2 > t1 ) )
//      -> no i2 : integer, t3 : Time | ( theta ( begin (bindRT, i2, t3) && ( t2 > t3 > t1 ) ) )
//
// A deactivateRT operation cannot start before the current activateRT operation is complete.
// all i1 : integer, t1, t2 : Time | theta ( begin (activateRT, i1, t1) && theta ( end (activateRT, i1, t2)
//      && ( t2 > t1 ) ) )
//      -> no i2 : integer, t3 : Time | theta ( begin (deactivateRT, i2, t3) && ( t2 > t3 > t1 ) )

( OKtoUse in whoWantsTo.roleUsesCap && activated.FeedBack' in OK )
| | ( ! ( OKtoUse in whoWantsTo.roleUsesCap ) && activated.FeedBack' in Error )
}

// Deactivate a research thread. This operation can only work if the role passed in has the DEFINE and
// USE capabilities. Otherwise an error occurs. The operation works by deactivating all
// the work items associated with the research thread via the drivesWork relation.
op deactivateRT ( whoWantsTo : Role !, currRT : ResearchThread ! ) {

// Another deactivateRT operation cannot start before the current one is complete.
// all i1 : integer, t1, t2 : Time | ( theta ( begin (deactivateRT, i1, t1) && theta ( end (deactivateRT, i1, t2)

```

```

//      && ( t2 > t1 ) }
//      -> no i2 : integer, t3 : Time | ( theta ( begin (deactivateRT, i2, t3) && ( t2 > t3 > t1 ) && ( i2 > i1 ) )
//
// An bindRT operation cannot start before the current deactivateRT operation completes.
// all i1 : integer, t1, t2 : Time | theta ( begin (deactivateRT, i1, t1) && theta ( end (deactivateRT, i1, t2)
//      && ( t2 > t1 ) )
//      -> no i2 : integer, t3 : Time | ( theta ( begin (bindRT, i2, t3) && ( t2 > t3 > t1 ) )
//
// An activateRT operation cannot start before the current deactivateRT operation is complete.
// all i1 : integer, t1, t2 : Time | theta ( begin (deactivateRT, i1, t1) && theta ( end (deactivateRT, i1, t2)
//      && ( t2 > t1 ) )
//      -> no i2 : integer, t3 : Time | theta ( begin (activateRT, i2, t3) && ( t2 > t3 > t1 ) )

    ( all wrk : ( currRT.drivesWork ) | deactivateWork ( whoWantsTo, wrk)
      && OKtoDefine in whoWantsTo.roleUsesCap && OKtoUse in whoWantsTo.roleUsesCap
      && deactivated.FeedBack' in OK )
    | | ( ! ( OKtoDefine in whoWantsTo.roleUsesCap ) | ! ( OKtoUse in whoWantsTo.roleUsesCap ) )
      && deactivated.FeedBack' in Error )
  }

// Operations for a REGISTRY. These include operations to register and locate a data source.
// role, research thread, work, and persistent store.

// Register a data source. The operation only takes place if the role passed in has the DEFINE
// capability. Otherwise an error occurs. The operation creates a regHasDS relation between
// the data source and the registry.
op registerDS ( whoWantsTo : Role !, what : DataSource !, reg : Registry ! ) {

// Another registerDS operation cannot start before the current one is complete.
// all i1 : integer, t1, t2 : Time | ( theta ( begin (registerDS, i1, t1) && theta ( end (registerDS, i1, t2)
//      && ( t2 > t1 ) )
//      -> no i2 : integer, t3 : Time | ( theta ( begin (registerDS, i2, t3) && ( t2 > t3 > t1 ) && ( i2 > i1 ) )
//
// A locateDS operation cannot start before the current registerDS operation completes.
// all i1 : integer, t1, t2 : Time | theta ( begin (registerDS, i1, t1) && theta ( end (registerDS, i1, t2)
//      && ( t2 > t1 ) )
//      -> no i2 : integer, t3 : Time | ( theta ( begin (locateDS, i2, t3) && ( t2 > t3 > t1 ) )

    ( OKtoDefine in whoWantsTo.roleUsesCap && what in reg.regHasDS' && registered.FeedBack' in OK )
    | | ( ! ( OKtoDefine in whoWantsTo.roleUsesCap ) && registered.FeedBack' in Error )
  }

// Locate a data source. The operation only takes place if the role passed in has the USE
// capability. Otherwise an error occurs. The operation attempts to find a dataSourceID
// relation between data sources and the identifier passed in.
op locateDS ( whoWantsTo : Role !, whichOne : Identifier !, reg : Registry ! ) {

// A registerDS operation cannot start before the current locateDS operation completes.
// all i1 : integer, t1, t2 : Time | theta ( begin (locateDS, i1, t1) && theta ( end (locateDS, i1, t2)
//      && ( t2 > t1 ) )
//      -> no i2 : integer, t3 : Time | ( theta ( begin (registerDS, i2, t3) && ( t2 > t3 > t1 ) )

    ( some ds : ( reg.regHasDS ) | ds.dataSourceID = whichOne && OKtoUse in whoWantsTo.roleUsesCap
      && located.FeedBack' in OK )
    | | ( ! ( OKtoUse in whoWantsTo.roleUsesCap ) && registered.FeedBack' in Error )
  }

// Register a role. The operation only takes place if the role passed in has the DEFINE
// capability. Otherwise an error occurs. The operation creates a regHasRole relation
// between the role and the registry.
op registerRole ( whoWantsTo : Role !, what : Role !, reg : Registry ! ) {

// Another registerRole operation cannot start before the current one is complete.
// all i1 : integer, t1, t2 : Time | ( theta ( begin (registerRole, i1, t1) && theta ( end (registerRole, i1, t2)
//      && ( t2 > t1 ) )
//      -> no i2 : integer, t3 : Time | ( theta ( begin (registerRole, i2, t3) && ( t2 > t3 > t1 ) && ( i2 > i1 ) )

```



```

//
// A locateDS operation cannot start before the current registerRole operation completes.
// all i1 : integer, t1, t2 : Time | theta ( begin (registerRole, i1, t1) && theta ( end (registerRole, i1, t2)
//      && ( t2 > t1 )
//      -> no i2 : integer, t3 : Time | ( theta ( begin (locateRole, i2, t3) && ( t2 > t3 > t1 ) )
//
// { OKtoDefine in whoWantsTo.roleUsesCap && what in reg.regHasRole' && registered.FeedBack' in OK )
//   | | ( ! ( OKtoDefine in whoWantsTo.roleUsesCap ) && registered.FeedBack' in Error )
// }

// Locate a role. The operation only takes place if the role passed in has the USE
// capability. Otherwise an error occurs. The operation attempts to find a roleID
// relation between roles and the identifier passed in.
op locateRole ( whoWantsTo : Role !, whichOne : Identifier !, reg : Registry ! ) {

// A registerRole operation cannot start before the current locateRole operation completes.
// all i1 : integer, t1, t2 : Time | theta ( begin (locateRole, i1, t1) && theta ( end (locateRole, i1, t2)
//      && ( t2 > t1 )
//      -> no i2 : integer, t3 : Time | ( theta ( begin (registerRole, i2, t3) && ( t2 > t3 > t1 ) )

// ( some role : ( reg.regHasRole ) | role.roleID = whichOne && OKtoUse in whoWantsTo.roleUsesCap
//   && located.FeedBack' in OK )
//   | | ( ! ( OKtoUse in whoWantsTo.roleUsesCap ) && registered.FeedBack' in Error )
// }

// Register a research thread. The operation only takes place if the role passed in has the DEFINE
// capability. Otherwise an error occurs. The operation creates a regHasRT relation between the
// research thread and the registry.
op registerRT ( whoWantsTo : Role !, what : ResearchThread !, reg : Registry ! ) {

// Another registerRT operation cannot start before the current one is complete.
// all i1 : integer, t1, t2 : Time | ( theta ( begin (registerRT, i1, t1) && theta ( end (registerRT, i1, t2)
//      && ( t2 > t1 ) )
//      -> no i2 : integer, t3 : Time | ( theta ( begin (registerRT, i2, t3) && ( t2 > t3 > t1 ) && ( i2 > i1 ) )
//
// A locateDS operation cannot start before the current registerRT operation completes.
// all i1 : integer, t1, t2 : Time | theta ( begin (registerRT, i1, t1) && theta ( end (registerRT, i1, t2)
//      && ( t2 > t1 )
//      -> no i2 : integer, t3 : Time | ( theta ( begin (locateRT, i2, t3) && ( t2 > t3 > t1 ) )

// { OKtoDefine in whoWantsTo.roleUsesCap && what in reg.regHasRT' && registered.FeedBack' in OK )
//   | | ( ! ( OKtoDefine in whoWantsTo.roleUsesCap ) && registered.FeedBack' in Error )
// }

// Locate a research thread. The operation only takes place if the role passed in has the USE
// capability. Otherwise an error occurs. The operation attempts to find a researchThreadID
// relation between research threads and the identifier passed in.
op locateRT ( whoWantsTo : Role !, whichOne : Identifier !, reg : Registry ! ) {

// A registerRT operation cannot start before the current locateRT operation completes.
// all i1 : integer, t1, t2 : Time | theta ( begin (locateRT, i1, t1) && theta ( end (locateRT, i1, t2)
//      && ( t2 > t1 )
//      -> no i2 : integer, t3 : Time | ( theta ( begin (registerRT, i2, t3) && ( t2 > t3 > t1 ) )

// ( some rt : ( reg.regHasRT ) | rt.researchThreadID = whichOne && OKtoUse in whoWantsTo.roleUsesCap
//   && located.FeedBack' in OK )
//   | | ( ! ( OKtoUse in whoWantsTo.roleUsesCap ) && registered.FeedBack' in Error )
// }

// Register a work item. The operation only takes place if the role passed in has the DEFINE
// capability. Otherwise an error occurs. The operation creates a regHasWork relation between the
// work item and the registry.
op registerWork ( whoWantsTo : Role !, what : Work !, reg : Registry ! ) {

// Another registerWork operation cannot start before the current one is complete.
// all i1 : integer, t1, t2 : Time | ( theta ( begin (registerWork, i1, t1) && theta ( end (registerWork, i1, t2)

```

```

//    && ( t2 > t1 ) )
//    -> no i2 : integer, t3 : Time | ( theta ( begin (registerWork, i2, t3) && ( t2 > t3 > t1 ) && ( i2 > i1 ) )
//
// A locateWork operation cannot start before the current registerWork operation completes.
// all i1 : integer, t1, t2 : Time | theta ( begin (registerWork, i1, t1) && theta ( end (registerWork, i1, t2)
//    && ( t2 > t1 ) )
//    -> no i2 : integer, t3 : Time | ( theta ( begin (locateWork, i2, t3) && ( t2 > t3 > t1 ) )

    ( OKtoDefine in whoWantsTo.roleUsesCap && what in reg.regHasWork' && registered.FeedBack' in OK
  )
  || ( ! ( OKtoDefine in whoWantsTo.roleUsesCap ) && registered.FeedBack' in Error )
}

// Locate a work item. The operation only takes place if the role passed in has the USE
// capability. Otherwise an error occurs. The operation attempts to find a workID relation
// between work items and the identifier passed in.
op locateWork ( whoWantsTo : Role !, whichOne : Identifier !, reg : Registry ! ) {

// A registerWork operation cannot start before the current locateWork operation completes.
// all i1 : integer, t1, t2 : Time | theta ( begin (locateWork, i1, t1) && theta ( end (locateWork, i1, t2)
//    && ( t2 > t1 ) )
//    -> no i2 : integer, t3 : Time | ( theta ( begin (registerWork, i2, t3) && ( t2 > t3 > t1 ) )

    ( some work : ( reg.regHasWork ) | work.workID = whichOne && OKtoUse in whoWantsTo.roleUsesCap
      && located.FeedBack' in OK )
    || ( ! ( OKtoUse in whoWantsTo.roleUsesCap ) && registered.FeedBack' in Error )
  }

// Register a persistent store. The operation only takes place if the role passed in has the DEFINE
// capability. Otherwise an error occurs. The operation creates a regHasPerStore relation between
// the persistent store and the registry.
op registerPerStore ( whoWantsTo : Role !, what : PersistentStore !, reg : Registry ! ) {

// Another registerPerStore operation cannot start before the current one is complete.
// all i1 : integer, t1, t2 : Time | ( theta ( begin (registerPerStore, i1, t1) &&
//    theta ( end (registerPerStore, i1, t2) && ( t2 > t1 ) )
//    -> no i2 : integer, t3 : Time | ( theta ( begin (registerPerStore, i2, t3) && ( t2 > t3 > t1 ) && ( i2 > i1 ) )
//
// A locatePerStore operation cannot start before the current registerPerStore operation completes.
// all i1 : integer, t1, t2 : Time | theta ( begin (registerPerStore, i1, t1) && theta ( end (registerPerStore, i1, t2)
//    && ( t2 > t1 ) )
//    -> no i2 : integer, t3 : Time | ( theta ( begin (locatePerStore, i2, t3) && ( t2 > t3 > t1 ) )

    ( OKtoDefine in whoWantsTo.roleUsesCap && what in reg.regHasPerStore' && registered.FeedBack' in
    OK )
    || ( ! ( OKtoDefine in whoWantsTo.roleUsesCap ) && registered.FeedBack' in Error )
  }

// Locate a persistent store. The operation only takes place if the role passed in has the USE
// capability. Otherwise an error occurs. The operation attempts to find a perStoreID relation
// between persistent stores and the identifier passed in.
op locatePerStore ( whoWantsTo : Role !, whichOne : Identifier !, reg : Registry ! ) {

// A registerPerStore operation cannot start before the current locatePerStore operation completes.
// all i1 : integer, t1, t2 : Time | theta ( begin (locatePerStore, i1, t1) && theta ( end (locatePerStore, i1, t2)
//    && ( t2 > t1 ) )
//    -> no i2 : integer, t3 : Time | ( theta ( begin (registerPerStore, i2, t3) && ( t2 > t3 > t1 ) )

    ( some perSt : ( reg.regHasPerStore ) | perSt.perStoreID = whichOne && OKtoUse in
    whoWantsTo.roleUsesCap
      && located.FeedBack' in OK )
    || ( ! ( OKtoUse in whoWantsTo.roleUsesCap ) && registered.FeedBack' in Error )
  }
}

```

7. STEP 6: TEMPORAL ISSUES

There are two temporal constraints that need to be specified for the bioinformatics system. These are:

- One piece of incremental work cannot start until the last piece of that same work has finished.

Depending on the granularity of incremental work, this requirement can be relaxed. For example, in a system where work takes place in large increments (e.g. large analyses or searches, or over large amounts of data), and each increment takes a long time to perform but is relatively independent of the work during the previous increment, it may be sufficient to specify that the next increment cannot start until all the data it needs is present.

Note that this temporal constraint is not really related to run-time configuration management of the system. It needs to be included in the bioinformatics functionality specification, but not in the run-time configuration management portion of the specification. This constraint is not included in the Alloy specification presented in Section 6.

- Operations that *change* the same data or work cannot occur concurrently, and can only occur when operations that *read* the data or work are complete. (Synchronization is needed since the system is loosely coupled, and acts in a peer-to-peer manner, with no centralized control. Synchronization must therefore be present as a part of each server operation.)

This is a reader/writer synchronization issue. Again, the constraint may be relaxed depending upon the system. The constraint can allow multiple readers

while one writer is active, or even multiple writers if the item(s) being written are sufficiently partitioned.

Note that this constraint covers broader issues than just those dealing with run-time configuration management. The only aspects of this constraint included in the Alloy specification are those dealing with run-time configuration management. These are operations that define, bind, release, activate, deactivate, instantiate, and view research threads, registry, role, work, persistence, and data. Work operations that change data are not run-time configuration management-related and are not included in the Alloy model.

RTL³⁵ can be used to specify run-time configuration management temporal constraints for the bioinformatics system.

One occurrence relation is sufficient to specify timing constraints regarding incremental operation.

The specification for the minimum time between incremental operation moves is:

$$\forall i \forall t \theta (\uparrow \text{incrWork}, i, t) \rightarrow \exists t' \mid (\theta (\downarrow \text{incrWork}, i-1, t')) \wedge (t > t')$$

$\uparrow \text{incrWork}$ is the start event of the *incrWork* operation of a particular type of work, $\downarrow \text{incrWork}$ is the end event of the *incrWork* operation, i is the i^{th} occurrence of the

³⁵ See [38] for a full description of RTL. Time is a basic element of RTL, and always continues forward. Other basic ideas of RTL are events and actions. Actions require bounded, but non-zero system resources and have durations. Events have no system resource requirements and no duration, but do occur at a particular point in time. Events are used as markers of time. There are two types of events, start and stop events that characterize an action and thus define its duration, and transition events that represent a change in a state variable. An occurrence relation is used to define the value of the time of an event. The occurrence relation is defined as $\theta(e, i, t)$ which denotes the i^{th} occurrence of the event e that happens at time t . RTL predicates are formed using the occurrence relation, equality/inequality predicates, universal and existential quantifiers, and first-order logical connectives.

start of the *incrWork* operation, and $i - 1$ is the $i^{th} - 1$ occurrence of the end of the *incrWork* operation. The relation states that for all values of i and t , the start of the i^{th} occurrence of the *incrWork* operation cannot occur until after the end of the previous $(i^{th}-1)$ occurrence of that same *incrWork* operation.

The synchronization temporal constraint is specified with the following RTL statements:

$$1) \forall i \forall t (\uparrow \text{changeOp}.i.t) \rightarrow \exists t' \mid (\theta (\downarrow \text{changeOp}.i-1.t')) \wedge (t > t')$$

This RTL relation states is that a change operation cannot start before the previous change operation ends (e.g. no multiple writers).

$$2) \forall i \forall t (\theta (\uparrow \text{changeOp}.i.t) \wedge (\theta (\downarrow \text{changeOp}.i.t') \wedge (t' > t)) \rightarrow \\ \neg \exists \tilde{t}. \tilde{t}'' \mid (\theta (\uparrow \text{readOp}.\tilde{t}.\tilde{t}'') \wedge (t > \tilde{t}'' > t'))$$

This RTL statement is that a read operation cannot start while a change operation is occurring.

$$3) \forall i \forall t (\theta (\uparrow \text{readOp}.i.t) \wedge (\theta (\downarrow \text{readOp}.i.t') \wedge (t' > t)) \rightarrow \\ \neg \exists \tilde{t}. \tilde{t}'' \mid (\theta (\uparrow \text{changeOp}.\tilde{t}.\tilde{t}'') \wedge (t > \tilde{t}'' > t'))$$

This RTL statement is that a change operation cannot start while a read operation is occurring.

Since an Alloy model consists solely of ASCII characters, symbols are not easily included. Neither are occurrence relations since Alloy has no notion of time. Therefore, these specifications are added as comments in the Alloy model, and where ever possible Alloy notation is used.³⁶ The Alloy specification presented in the previous section of the

³⁶ Thus, $\forall$ is replaced with *all* and $\exists$ with *one*, $\wedge$ with *&&*, t with *t1*, t' with *t2*, t'' with *t3*, i with *i1*, and i' with *i2* (these latter changes are made to avoid confusion between the i' and t' of the temporal expression and the notion of primed variables to indicate the ending state of a variable in Alloy). The θ symbol is

chapter has been extended with commented temporal predicates, for multiple reader/writer constraints.

8. SUMMARY

The results of each of step in the specification process described in this chapter are summarized below.

8.1 System Focus

The primary result of this step is the focus of the bioinformatics is on dynamic data, whereas the focus of the ATM network monitoring system is on run-time configuration management. The bioinformatics system analysis in this and subsequent steps identify information that will be useful in bioinformatics function specification, because of this different focus. Separating run-time configuration management from bioinformatics function aspects can be difficult, particularly during property refinement and *type* modeling steps.

8.2 Property refinement

Property refinement identifies characteristics of the incremental property and the persistence property. Only a small portion of those characteristics deal with run-time configuration management. It is tempting to include all the information identified in this step in the *type* model and the Alloy model. The extra information identified during property refinement will be useful during bioinformatics functionality specification.

replaced with *theta*, and $\uparrow$ and $\downarrow$ with functions *begin* () and *end* (), respectively. The name of an operation represents that operation called with a particular set of parameters.

8.3 Soft goals

Configuration operations are refined from the major soft goals of handling large scale, dealing with a dynamic environment, and protecting data and information in the bioinformatics system. This method identifies many operations in the system that are not part of its run-time configuration management. In fact, only the *define* operations in the incremental and persistence sub-goals deal with run-time configuration management. However, the other information identified in this step is useful for bioinformatics function specification. Only configuration management operations (those related to *define*) are included in the Alloy model.

8.5 Type Model

Information from the first three steps of the specification method, and the initial configuration matrix developed in Chapter 4 for the bioinformatics system, are used to create a *type* model. Since the system focus is on data, not run-time configuration management, property refinement and soft goals analysis do not yield enough operation information to create the *type* model. The initial configuration matrix identifies configuration operations such as *locate*, *instantiate*, *bind*, *activate*, *deactivate*, *release*, and *view*.

8.6 Alloy

The Alloy model is straightforward to create from the *type* model. As it is developed, changes are made to the *type* model to reflect increased detail present in the Alloy model. An example of this is the creation of two relations between the *Work* and *Incremental* types in the *type* model, called *piecesOfWork* and *incrStepsDone*. These relations are needed to define the *activate* operation in the *Work* type.

8.7 Temporal Logic

This step identifies functional and run-time configuration management temporal constraints. One of these constraints is the need to constrain concurrent work in the situation where one piece of incremental work somehow depends on the outcome of a previous piece of incremental work. Depending on the work and its true interdependence this constraint may be relaxed, but it does need to be in the bioinformatics function specification in order to prevent faulty system operation. This constraint is not related to run-time configuration management and is not included in the Alloy model.

Another constraint that is not strictly related to run-time configuration management is the concurrent *reader-writer* synchronization that needs to be specified for server operations. There is a run-time configuration management aspect to this constraint, namely that whenever a client requests a change to one of the configurable items in the system there needs to be a deterministic result if there are multiple concurrent requests to change the same item or if there are currently requests that are using the item. However, this constraint is equally important during normal system operation. For example, if an analysis is executing, re-defining what that analysis should do can create erroneous results. Therefore during the design of bioinformatics functionality, care must be taken to incorporate the multiple *reader-writer* constraint into the system specification.

The run-time configuration management aspects of the multiple reader-writer constraint are incorporated in the Alloy model. These constraints are included as comments in server operations. Unfortunately, there is no way to test temporal constraints using the Alloy tool.

9. CONCLUSIONS

The specification methodology and notation developed in Chapter 5 worked well to specify the run-time configuration management properties of the bioinformatics system. The methodology also identified numerous pieces of information that will be useful in the overall system specification. The issue of being able to not only specify but also test simple temporal constraints remains an issue in the bioinformatics system, as it was in the ATM network monitoring system.

The specification portion of the CMSpecV&V method is evaluated according to the criteria presented in Chapter 3:

- 1) Validation of results using system architect or user feedback: There is no feedback available to validate the results of the method.
- 2) Ability to specify all necessary requirements: The specification method allows complete specification of the bioinformatics run-time configuration management requirements.
- 3) Applicability to specification: This is a specification method.
- 4) Ability to directly support validation/verification: The techniques can directly support at least a partial validation and verification of the specification.
- 5) Ability to fit into industry:
 - 1) Automation potential: none
 - 2) Existing tools: The techniques take advantage of existing industrial notations (e.g. UML) and existing academia notations (e.g. Alloy), so there is some tool support.

- 3) Skill set needed by practitioners: The techniques do not require practitioner skills beyond the ability to deal with first order predicate logic.
- 4) Flexibility to fit into existing development processes: This method can easily fit into specification activities since it is thought-ordering and high-level documentation take place during specification.
- 5) Ability to use partially or incrementally: The method works well for partial or incremental use. In this example, it was only used to specify the run-time configuration management of the bioinformatics system.

CHAPTER 10 – APPLYING THE PROPOSED VALIDATION AND VERIFICATION METHOD TO THE CONFIGURATION MANAGEMENT SPECIFICATION OF THE BIOINFORMATICS SYSTEM

1. INTRODUCTION

This chapter validates and verifies the bioinformatics run-time configuration management specification developed in Chapter 9. This validation and verification ensures that each of the key run-time configuration management requirements is adequately represented in the specification, and that the specification is internally consistent. The CMSpecV&V method validation and verification is developed in Chapter 7.

The CMSpecV&V method uses two approaches to validate and verify a specification. The first is a modified multiple architectural view analysis. Multiple view analysis uses the standard three views of functionality, interaction protocol, and dependability, as well as additional views based on the key requirements of the system being specified. Multiple view analysis validates the existence of key run-time configuration management requirements in the specification. Multiple view analysis can verify the specification's internal consistency across the architectural views. The second approach uses the Alloy tool to test important invariants, operations, and overall system state in the Alloy specification. Alloy analysis checks for internal consistency across the specification.

The CMSpecV&V validation and verification method does allow reasoning about the existence of key requirements in a specification as well as specification internal

consistency, but it does not allow statements to be made about provable correctness of the specification, nor of any resulting implementation.

Multiple architectural view analysis is discussed in Section 2 and automated techniques, based on the Alloy specification language, are discussed in Section 3. The results of using each of these techniques are presented. Section 4 presents conclusions regarding the bioinformatics run-time configuration management requirements specification validation and verification.

2. MULTIPLE VIEW ANALYSIS

The standard architectural views used in multiple view analysis are functionality, interaction protocol, and dependability. (Issarny, Saridakis, and Zarras [33] propose the use of these views and a performance view to validate a system's consistency. Since performance may or may not be a key system requirement, this view has been dropped from the generalized technique presented in Chapter 7 in favor of views that represent the key requirements of the system being specified.)

Various aspects of each of the architectural views are compared as part of the consistency checks. Views are compared two at a time, and each is compared against all the others. The functional view of the bioinformatics system run-time configuration management specification is derived from the *type* model and Alloy textual specification presented in Chapter 9. The functional view consists of software-related items, their relationships, and operations.

The interaction view consists of the interaction protocol between clients and servers. This interaction is based on roles and capabilities in the bioinformatics system specification. The dependability view contains system structures used in fault tolerance

mechanisms. These mechanisms are persistent stores (especially for data originally derived from geographically dispersed databases) and the fact that operations are protected by roles and capabilities. (The latter is an aspect of security, but since it helps make the system dependable from a user's point of view, it is included in the dependability view.)

The multiple architectural view technique must be extended with additional views that represent key system requirements in order to validate their representation in the requirements specification. The key requirements in the bioinformatics system are the ability to handle large scale in a dynamic environment (both in terms of data and the rate at which it changes), and the ability to protect data and unpublished results. Two new architectural views need to be added to validate that these requirements are addressed in the specification. The views are scalability (including dynamic environments) and security. Only the scalability/dynamic environment view needs to be added as another architectural view, since security is an aspect of dependability in this system. Views in the modified multiple architectural view analysis are shown in Figure 10-1.

The *functional view* of the run-time configuration management specification consists of software-related items, their relationships, and the operations they perform. The functional view is shown in Figure 10-2.

The functional view in Figure 10-2 shows that the system operates in a peer-to-peer fashion. Many of the system types act as servers in some situations and as clients in others. *Data* (acting as results) and *Capability* only act as servers. *Client*, the user of the system, only acts as a client. *ResearchThread*, *Persistence*, *PersistentStore*, *Work*, *DataSource*,

Data, *Role*, and *Registry* act as both clients and servers, depending on the situation. Request paths between type operations are shown as lines between types in the figure. Table 10-1 shows the client/server interactions.

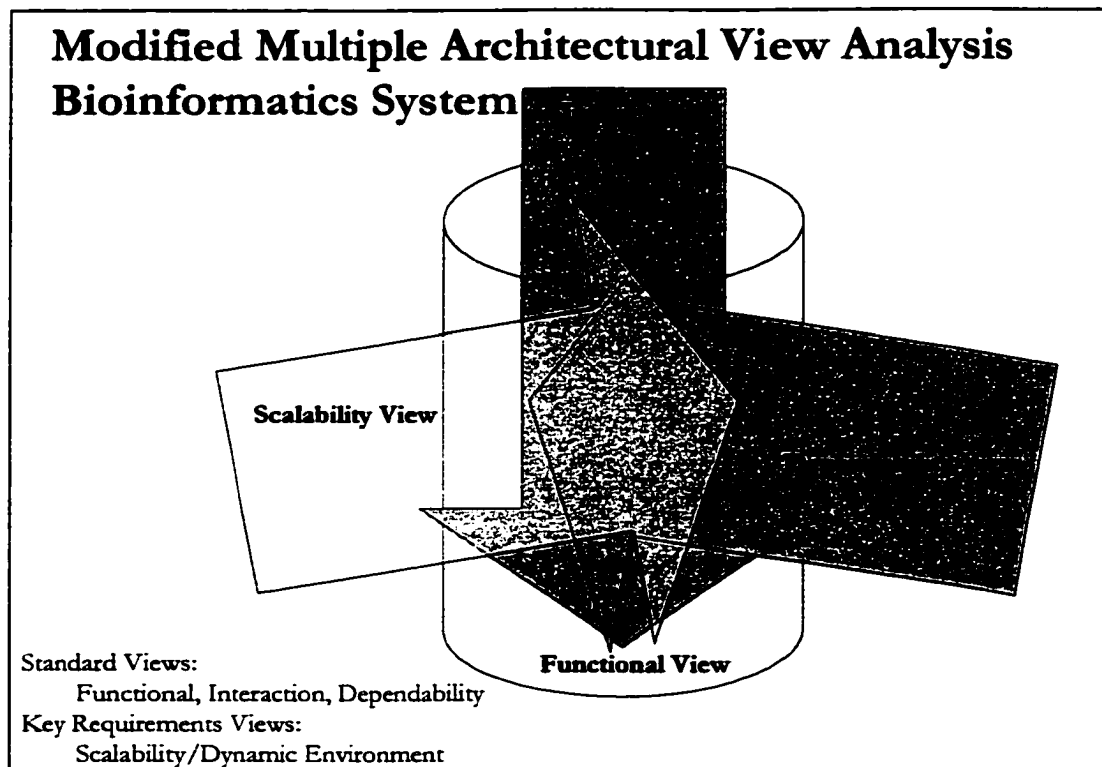


Figure 10-1. An abstract view of the modified multiple architectural view analysis of the run-time configuration management needs of the bioinformatics monitoring system. Three views are needed for any distributed system. These are functional, interaction, and dependability. The functional view of the system is the simultaneous application of all system requirements and is traditionally known as the "specification". This view is represented by the cylinder in the diagram. The interaction and dependability views are shown as intersecting arrows. These three views are compared two at a time for consistency. Views that represent key run-time configuration management requirements must also be added to validate that these requirements are adequately specified. The only additional view needed in this system is scalability (including dynamic environment), since the other key requirement of security protection is embedded in the dependability view. The scalability view is also shown as an intersecting arrow. This view must be checked for consistency against each of the three primary views (functional, interaction, and dependency) for consistency.

The *interaction view* consists of the interaction protocol between clients and servers. In particular the protocol includes the fact that each server uses Roles and Capabilities to determine how to satisfy a request, along with appropriate internal

synchronization to deal with multiple concurrent requests. This protocol is shown in Figure 10-3.

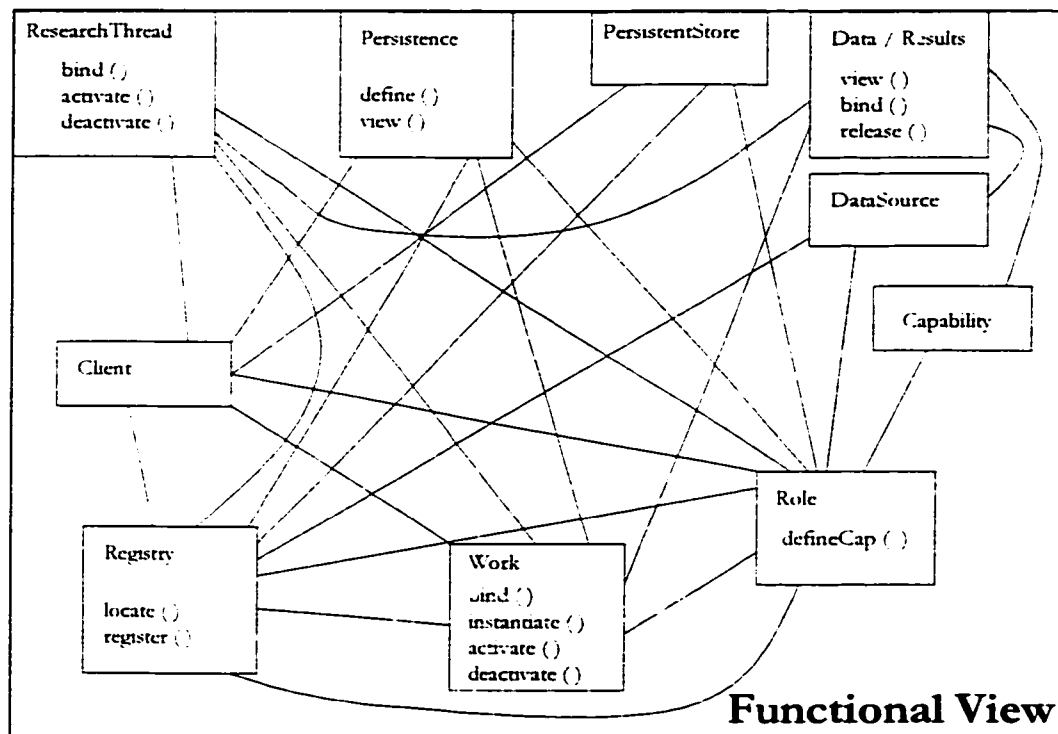


Figure 10-2. The *functional view* of software run-time configuration management in the bioinformatics system. This view shows that the overall architecture is peer-to-peer; types generally act as both clients and servers. The *functional view* shows operations callable by clients as well as server operations.

The *interaction view* shows a client/server protocol, extended to use roles and capabilities to protect system types. Synchronization occurs within servers to ensure that multiple requests return deterministic results. For example, if a server has multiple requests to read or use some run-time configuration information, it must not change that information while readers are accessing it. Similarly, readers must not start reading run-time configuration information until writers have finished changing it. (This is the classical multiple reader/writer problem. As in any reader/writer application, there are multiple correct synchronization constraints depending system requirements.) The constraints shown in Figure 10-3 (written as RTL occurrence relations) are typical of the

problem; reader operations cannot start while information is being changed, and writer operations cannot start while information is being read.

Table 10-1. Client/Server Interactions Between Types in the Bioinformatics System

Client	Server
Client	Persistence, PersistentStore, Registry, ResearchThread, Role, Work
Data	DataSource, Role
DataSource	Role
Persistence	Role
PersistentStore	Role
Registry	Role, DataSource, Work, ResearchThread, PersistentStore
ResearchThread	Work, Data (as results), Role
Role	Capability
Work	Data (and results), Persistence, Role

The *dependability view* consists of system structures used for fault tolerance. This view is shown in Figure 10-4.

Figure 10-4 shows mechanisms used for fault tolerance in the bioinformatics system. There are two main fault sources in the bioinformatics system. The first is that network errors may cause the system to be unable to access geographically dispersed external data sources. This can be alleviated using local warehousing. The second kind of fault is improper data and information protection. Although protection is a security issue, it is also a dependability issue for the bioinformatics system. Thus, information protection can be added to the dependability view. (One useful side effect of adding security to the dependability view is that there is no need for an additional security view during validation.)

The *scalability view* is developed by identifying ideas and techniques that support large scale and a dynamic environment. Large scale means the ability to deal with huge volumes of data as well as a dynamic environment in which the data regularly changes. The scalability view ensures that techniques identified to support scale are consistent across the system. Incremental work, over both time and space, support scale. Incremental work over space supports huge amounts of data, and incremental work over time supports dynamically changing data. The scalability view is shown in Figure 10-5.

The scalability view shows the incremental nature of work in the bioinformatics system. Incremental operation over space supports large volumes of data, by allowing analyses or searches to occur over parts of the data at different times. (Alternatively, parts of the analyses or searches may occur over all the data at different times.) Similarly, incremental operation over time allows the system to deal with changing data.

Figure 10-5 also shows a timing constraint. The constraint is given as an RTL¹ occurrence relation. The relation states that one piece of incremental work cannot start until the previous piece of the same work is finished. Depending on how well the work can be partitioned, and whether incremental work is completely independent (i.e. data are independent from one piece of incremental work to the next), this constraint can be relaxed or eliminated altogether.

This timing constraint has nothing to do with run-time configuration management. It deals with functionality of the bioinformatics system. Therefore it is not included in the run-time configuration specification presented in Chapter 9. This is a case where specification and validation techniques identify information that is important in the overall system specification, not just run-time configuration management

information. The temporal constraint is included in the scalability view so that it can help validate an overall system specification.

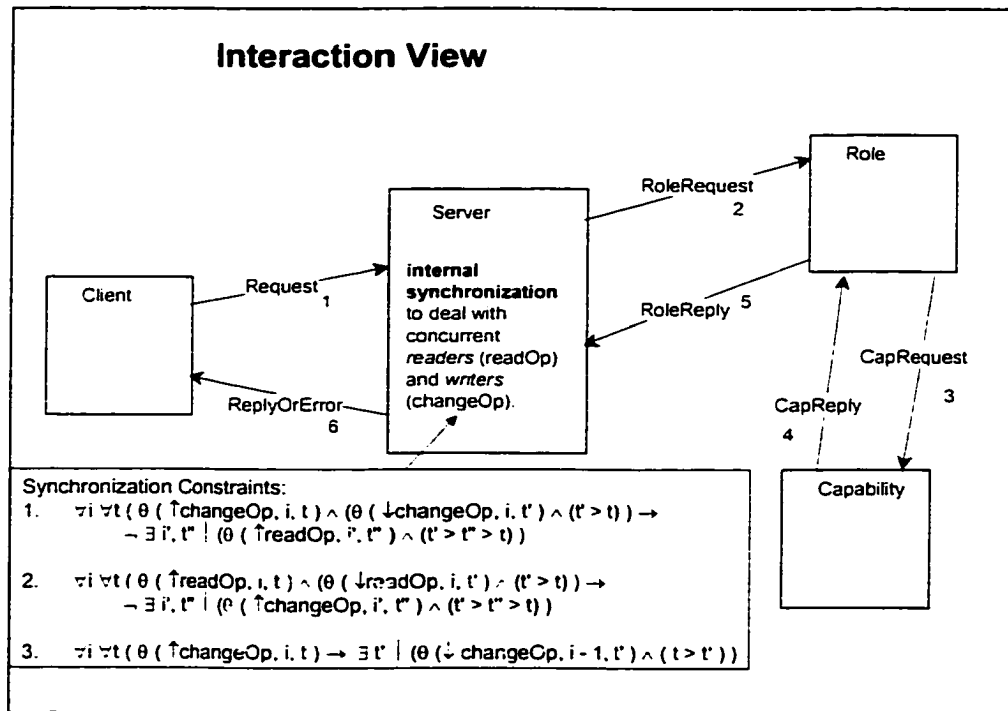


Figure 10-3. Interaction view: clients send requests to servers (1). Servers make role requests to *Role* (2), which checks capabilities (3) to make sure the role requesting the service has the appropriate capability (4, 5). If the role has the correct capability, the server replies to the request, otherwise an error occurs (6). In addition, the server has internal synchronization to deal with concurrent requests, particularly concurrent requests to read and change (i.e. write) information. Constraints associated with this synchronization are shown as RTL³⁷ occurrence relations. The first relation states that a read operation cannot start while a change operation is occurring. The second relation states that a change operation cannot start while a read operation is occurring. The third relation states that a change operation cannot start while another change operation is occurring.

³⁷ See [38] for a complete discussion of RTL. In RTL, time is a basic element of the system, and always continues forward. Other basic ideas of RTL are events and actions. Actions require bounded, but non-zero system resources and have durations, whereas events have no system resource requirements and no duration, but do occur at a particular point in time. Events are used as markers of time. There are two types of events, start and stop events, that characterize an action and thus define its duration (see [79] for a discussion of temporal logic and its application to data communication protocols), and transition events that represent a change in a state variable. An occurrence relation is used to define the value of the time of an event. The occurrence relation is defined as $\theta(e, i, t)$ which denotes the i 'th occurrence of the event e that happens at time t . RTL predicates are formed using the occurrence relation, equality/inequality predicates, universal and existential quantifiers, and first-order logical connectives (equality/inequality predicates are $=$, $<$, $>$, $\geq$, and $\leq$ and first-order logical connectives are $\neg$, $\wedge$, $\vee$, and $\rightarrow$).

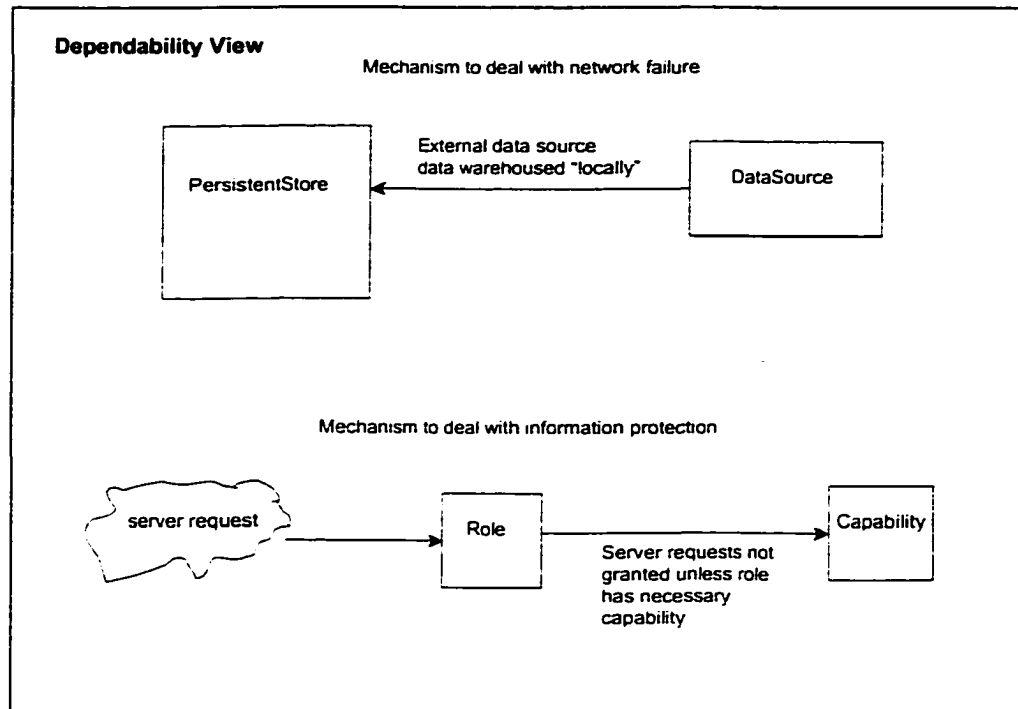


Figure 10-4. The dependability view. There are two basic dependability mechanisms. One mechanism deals with the problem of network failure to external data sources by warehousing data locally. The second dependability mechanism protects information by using roles and capabilities to determine whether server requests should be granted.

Table 10-2 shows the views, checks that must be performed across views, and consistency results. Views are checked in pairs to simplify comparisons. All views must all be checked against all the other views. The specification may be considered validated from the perspective of key requirements representation once consistency checks have been performed.

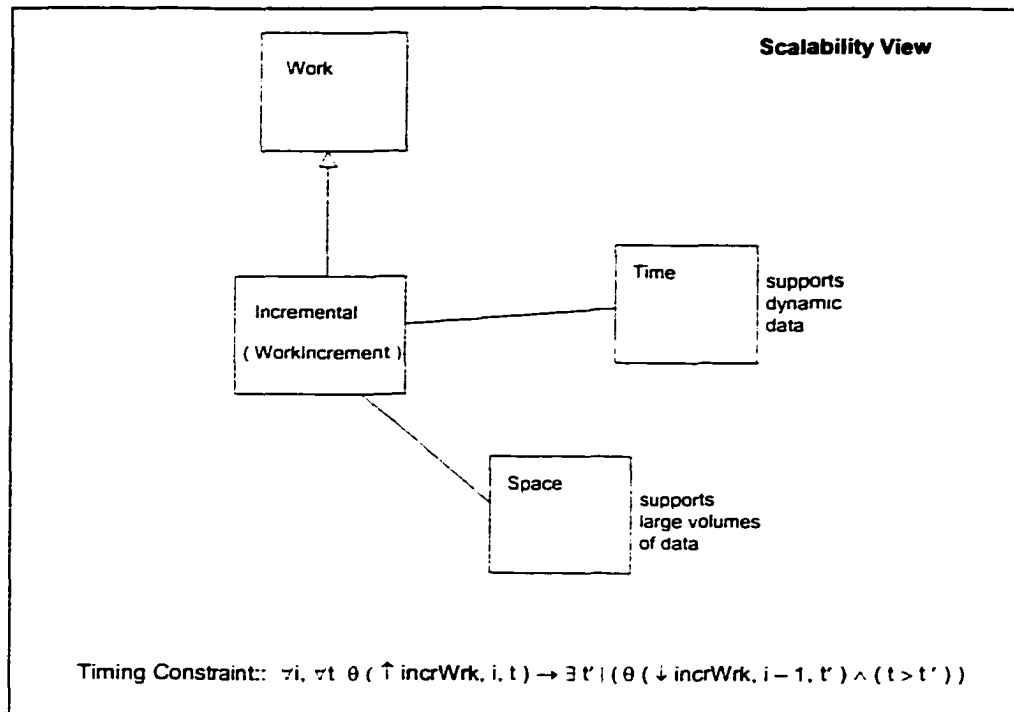


Figure 10-5. The scalability view shows mechanisms that support large scale in terms of both data volume and dynamic changes. Incremental work, both in terms of space and time supports large scale. In addition, the timing constraint dealing with incremental work is shown as an RTL occurrence relation. The relation states that a piece of incremental work cannot begin until the last piece has finished. Note that this constraint can be relaxed as appropriate for the actual work being done by the system. For example, if the work partitions completely into incremental pieces, and pieces do not depend in any way upon each other, the constraint can be eliminated.

Table 10-2. Views and their Consistency Checks for the Bioinformatics System. Consistency check results are italicized.

	Functional View	Interaction View	Dependability View
Interaction View	Do functional view components include interaction protocol? <i>yes</i>		
Dependability View	Do functional view components include dependability structures? <i>yes</i>	Do dependability components include interaction protocol? <i>yes</i>	
Scalability View	Do functional view components include scalability structures? <i>yes</i>	Do scalability components include interaction protocol? <i>yes</i>	Do dependability components include scalability structures? <i>no</i> Do scalability components include dependability structures? <i>yes</i>

2.1 Consistency Checks:

2.1.1 *Functional/ Interaction Views*

Synchronization of the interaction protocol and functional components must be consistent. The underlying interaction protocol consists of client/server requests based on role capabilities, along with internal server reader/writer synchronization. Synchronization is achieved using requests and replies along with temporal sequencing constraints on readers and writers. Clients make requests and servers are expected to respond to all requests.

All functional view operations must support this same client/server protocol for the views to be consistent.

The Alloy model in Chapter 9 must be checked to verify that the client/server protocol is supported. Specifically, each operation that acts as a server must respond to each request it receives, and must incorporate the temporal sequencing constraint. The Alloy specification must be checked to see that each operation sets the appropriate *Feedback* relation, which they do. Temporal constraints are included in all operations as comments. This consistency check validated that operations comply with the underlying interaction protocol.

2.1.2 *Interaction/ Dependability Views*

Synchronization of the interaction protocol and dependability components must be consistent. The underlying interaction protocol consists of client/server requests. Dependability components are shown in the dependability view; these are persistent stores and the use of roles and capabilities.

Since a persistent store is a functional concept that acts as a server, it has already been checked against the interaction protocol. Similarly, *Role* and *Capability* are also represented in the functional view and thus have already been checked. Therefore, the dependency components are consistent with the interaction protocol.

2.1.3 Interaction/ Scalability Views

Synchronization of the interaction protocol and the scalability components must be consistent. The underlying interaction protocol consists of client/server requests. Scalability components are shown in the scalability view. They consist of *Work* that can be *Incremental* in either time or space.

Since *WorkIncrement* is a specialization of *Work*, it shares the same functional operations as *Work*. *Work* is consistent with the interaction protocol, so scalability components are also consistent with the interaction protocol.

2.1.4 Functional/ Scalability Views

Functional components must have a scaling behavior consistent with the scalability components. Scalability components are shown in the scalability view. They consist of work that is incremental in time and space. *WorkIncrement* is shown as a specialization of *Work* in the type model (see Figure 9-6 in Chapter 9). Thus, items needed to fulfill the scalability view are present in the functional components.

2.1.5 Functional/ Dependability Views

Functional components must have a dependability behavior consistent with the dependability components. Dependability components are shown in the dependability view. They consist of persistent storage and the use of roles and capabilities. Persistent

data storage is shown in the *type* model (see Figure 9-2 in Chapter 9). Roles and capabilities are also present in the *type* model (see Figure 9-5 in Chapter 9), as well as in all server operations in the Alloy model. Thus, the functional view is consistent with the dependability view.

2.1.6 Dependability / Scalability Views

Scalability components must have a dependability behavior consistent with dependability components, and dependability components must have a scaling behavior consistent with scalability components. Scalability components are shown in the scalability view. They consist of work that is incremental in time and space. Dependability components consist of locally stored information from external data sources and the use of roles and capabilities to control server action. Scalability components are included in the functional components, so the fact that the functional components are consistent with the dependability view means that the scalability components are also consistent with dependability. Dependability components can support scalability provided that they are also able to partition work in terms of time and space. Dependability components dealing with roles and capabilities do not need this ability since the work they do is small (the work takes little time and involves little data), but the work involved with persisting external data may need to be approached in an incremental fashion in order to deal with very large amounts of data. In this case, *Work* needs to be extended to include not only analyses and searches, but also data replication. This consistency check identifies a potential consistency problem.

2.2 Consistency Check Results:

In general, the results of view comparisons are that views are consistent with each other, except for the dependability and scalability views. If the inconsistency discovered by the validation is important, *Work* can be extended to include persisting data from external sources.

2.3 Observations

Multiple architectural view analysis depends on consistency checks between pairs of views, some present in all systems (i.e. functional, interaction, and dependability) and some unique to each system. The soft goals technique and the *type* model define the functional view of the system by identifying attributes and actions of configurable items as well as their associated configuration operations.

Unique views are identified using key domain requirements.

Multiple architectural view analysis can be used to reason about how well key requirements are reflected in the specification. It can therefore be used as a validation of the specification with respect to key run-time configuration management requirements. In addition, consistency checks across views allow reasoning about the internal consistency of the specification.

The immediate result of using this technique on the bioinformatics specification is that it found a potential inconsistency between dependability concepts (i.e. external data source persistence) and the scalability concept of incremental work. Persisting external data sources may need to be incremental to support very large amounts of data. This can be accomplished by extending the *Work* to include data persistence. (The revised portion of the *type* model that shows this extension is shown in Figure 10-6.)

3. ALLOY AUTOMATED TOOL ANALYSIS

The Alloy tool can help validate and verify a specification consisting of a *type* model complemented by an Alloy model. Since the Alloy model is complete, duplicating and extending the *type* model, any verification that can be done on the Alloy model provides verification of the *type* model as well, (provided the information in each model accurately represents the other)

. The Alloy tool checks a model's state, invariants, definitions, assertions, and operations for consistency.

The Alloy tool creates a Boolean satisfaction formula from an Alloy model and assigns a scope to it. (Scope is the number of elements in each domain set in the model; scope is limited in order to create a decidable analysis.) The analysis determines if an instance of the model exists for the formula, within the scope bound. If an instance does exist, the Alloy model is consistent. The tool can also be used to look for theorem (assertion) counterexamples that indicate model inconsistency. Jackson ([34]) states that the scope is kept low in the tool (3 by default), so some consistent formulas may not have model instances at this scope, but experience has found that many formulas have model instances in a small scope. (Detailed operation of the tool is described in [34].) Note that finding an instance of a formula indicates model consistency, but not model correctness since the Alloy first order logic is undecidable. Failing to find an instance of a formula does not necessarily indicate that the model is inconsistent; it may simply have an instance at a higher scope. Similarly, failing to find a counter example to an assertion does not mean that the assertion is consistent; a counter example may exist at a higher scope.

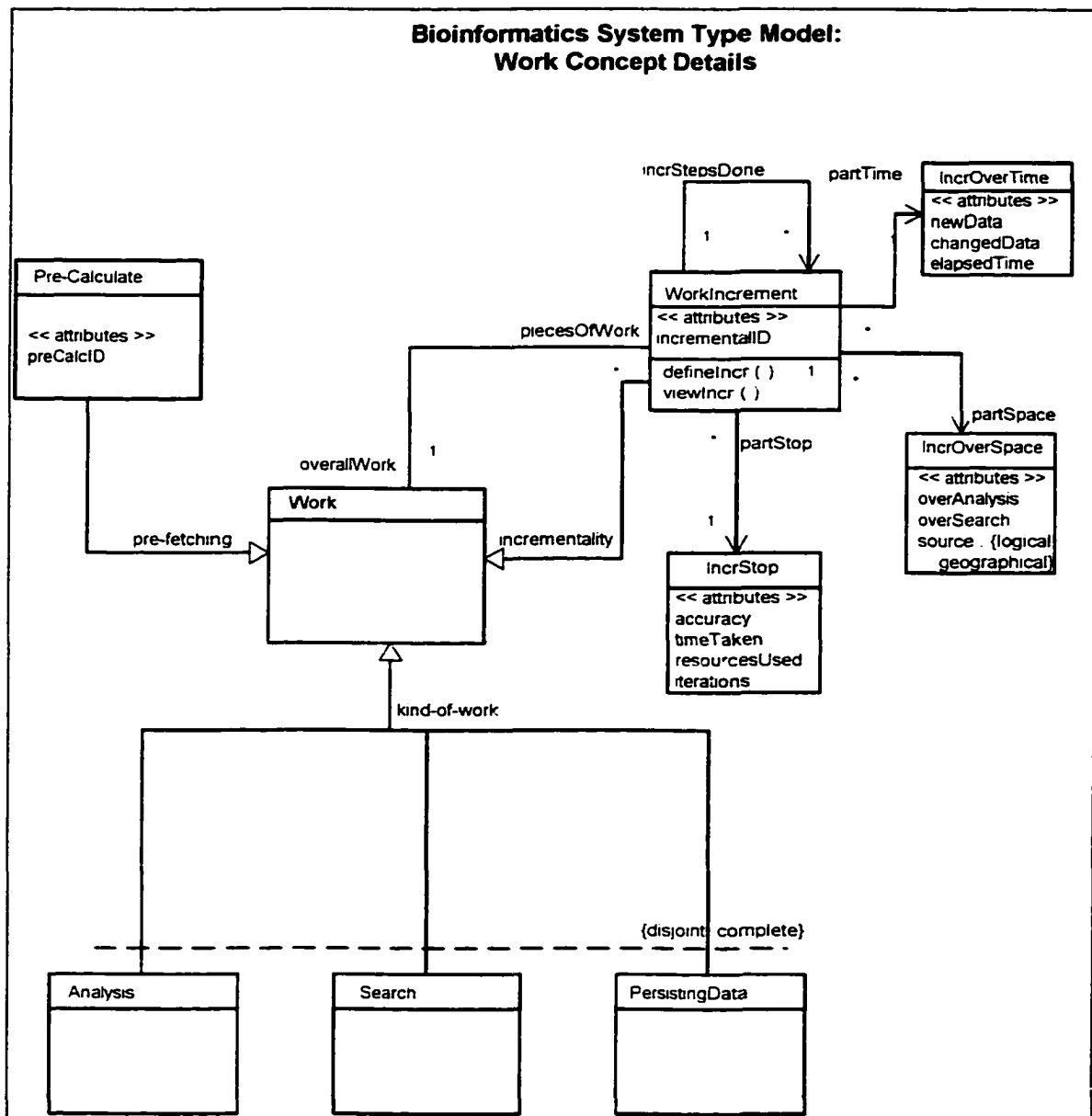


Figure 10-6. Modified *Type* model. An extension to the *Work* type to include external data source persistence.

The Alloy model presented in Chapter 9 was analyzed with the Alloy tool. The entire model compiles and its state is consistent, but checking any operation causes the tool to take too long to complete. Therefore all operations were checked using smaller models that only contain the minimum model components. The model was analyzed on

a Pentium III system with 128 Mbytes of RAM. It took 4 seconds to compile. Each state, invariant, and operation was analyzed individually, and found consistent. Selected analysis results are shown in Table 10-3.

The bioinformatics system run-time configuration management Alloy model has 33 relations, 19 domain sets, 36 subsets, 1 invariant, and 24 operations.

Table 10-3. Analysis Time for Alloy Model Components.

Component Name	Component Type	Analysis Time (seconds)	Comments
state	model state	25	Entire state schema; sets based on domain sets and their relations
bindData	operation	34	Bind data to a data source
viewData	operation	15	View data
releaseData	operation	30	Release a data source
viewIncr	operation	91	View the definition of incremental work
disjointCap	invariant	1	Capabilities OKtoUse, OKtoView, and OKtoDefine must be disjoint
registerWork	operation	205	Add a work definition to a registry
locateWork	operation	44	Locate a work definition in a registry
activateRT	operation	10	Start a research thread
deactivateRT	operation	41	Stop a research thread
bindRT	operation	17	Bind a research thread to data and results
instantiateWork	operation	49	Start a piece of work

Table 3 shows a wide variation in analysis time, depending on the complexity of the component. Jackson [34] discusses tool performance based on the number of items in the search space and the number of configurations present in the model. Each

domain state and relation increases the potential search space. The increase is related to the scope used during analysis. Since the scope used for these analyses was the default, 3, this means that each of the 19 domain sets in the model has 2^3 possible values. Jackson states that each relation increases the potential search space by a factor of 2 to the k^2 , where k is the scope used during analysis. This means that each of the 33 relations in the model has an impact of 2^9 on the search space. Thus, the number of possible states is $2^{(19 \cdot 3 + 33 \cdot 9)}$, or $3.67e^{106}$. The number of transitions is $2^{(2 \cdot (19 \cdot 3 + 33 \cdot 9))}$, or $1.35e^{213}$.

The Alloy tool analyzes operations against invariants. This type of analysis either put the tool into an unspecified state or took too long to analyze.

4. CONCLUSIONS

The CMSpecV&V validation and verification method allows reasoning about the existence of key requirements in a specification. The method also checks specification internal consistency, but it does not allow statements to be made about provable correctness of a specification, nor of any resulting implementation.

The multiple view analysis validates the existence of key run-time configuration management system requirements in a specification. This technique is also useful to identify conditions under which the key run-time configuration management requirements may not be met. For example, the specification developed in Chapter 9 is generally consistent with the bioinformatics system key requirements, except for the idea of incrementally persisting data from external data sources. This inconsistency may or may not be a real problem in the system; it depends on system expectations. The inconsistency is a system functionality issue, not a run-time configuration management

issue, so it is not addressed in detail further in this document. However, it is identified through multiple architectural view analysis and therefore can be documented as a constraint in other system functionality specifications. Incremental persistence can easily be added to the configuration management specification by extending the *Work* to include not only analyses and searches, but also persisting data. No changes in operations are necessary, so an addition to the *type* model in Figure 9-3 of Chapter 9 is sufficient to address the inconsistency. (Changes to the portion of the type model are shown in Figure 10-6.) A related addition must be made to the Alloy model, where the *Work* domain set is partitioned by an *analysis* and a *search* subset:

```
partition analysis, search, persistingData : Work
```

The validation and verification portion of the CMSpecV&V method is evaluated according to the criteria presented in Chapter 3:

- 1) Validation of results using system architect or user feedback: There is no feedback available to validate the results of the method.
- 2) Ability to validate and verify all necessary requirements: The CMSpecV&V method partially validates and verifies the specification. Temporal constraints are not verified except by manual inspection.
- 3) Applicability to validation and verification: This is a validation and verification method.
- 4) Ability to fit into industry:
 - 1) Automation potential: none
 - 2) Existing tools: The Alloy tool helps verify an Alloy model.

- 3) Skill set needed by practitioners: The techniques do not require practitioner skills beyond the ability to deal with first order predicate logic.
- 4) Flexibility to fit into existing development processes: This method can easily fit into specification activities since its specifications are validated using some method; the CMSpecV&V method can complement existing validation practices.
- 5) Ability to use partially or incrementally: The method works well for partial or incremental use. In this example, it was only used to validate and verify the run-time configuration management specification of the bioinformatics system.

In conclusion, the CMSpecV&V validation and verification method works for the bioinformatics system. It does identify one potential inconsistency in the specification. Whether or not this is really a problem is dependent on the overall system functionality, and cannot be determined from the run-time configuration management requirements perspective.

1. INTRODUCTION

The research discussed in this dissertation has several implications. It has also generated some general observations. Implications and observations are discussed in this chapter. Implications of the research are presented in Section 2, and observations are presented in Section 3.

2. RESEARCH IMPLICATIONS

There are five major implications of the research discussed in this dissertation. The first is that methods can be used to not only to identify key system requirements, but also to specify them and to validate that the requirements are adequately represented in the specification. This implication further moves the specification process toward an engineering field and away from a craft. This also implies that less experienced system architects and designers can successfully identify and specify key system requirements. The methods developed in this research are lightweight, and for the most part informal. They do not require any extraordinary skills on the part of practitioners. In addition, the methods can be used in a partial manner, expending effort only on areas of system specification that are particularly key to system acceptance.

A second implication is that the research method described in Chapter 3 leads to a useful model of specification, validation, and verification methods for distributed systems. The research method directs evaluation of several competing specification,

validating, and verification techniques in terms of their abilities to apply to an initial distributed system. Other evaluation criteria related to technique usefulness in industrial software development are also applied. The most promising aspects of the techniques are combined and extended to create the CMSpecV&V method.

The CMSpecV&V method is generalized to work with other distributed systems. The generalized model is successfully tested for its ability to address specification in another distributed system. Part of the test is to demonstrate that the method works for distributed systems that are quite different in some aspects (problem domain, size, system focus, time-value of data, and dynamic data time), and similar in other aspects (geographically distributed, large amounts of data). The two similarities lead to a class distinction, and an implication that the method will work for other systems in the same class.

The third implication is that the methods developed in this research will work for geographically distributed systems that deal with large amounts of data; the same class as the two systems used in this research. Although the size of this class is unknown, systems in this class vary widely in problem domain. Some representative examples are distributed monitoring systems in fields ranging from ecological monitoring to wireless communication monitoring, and information/analysis systems dealing with distributed data in fields ranging from scientific (e.g. bioinformatics) to enterprise financial or operation data systems.

A fourth implication of this research is that research and industrial techniques can be partially combined to create useful industrial techniques. It is not necessary to completely use a technique if the end result is not worth the effort involved. The most

useful portions of a technique can be used and still produce useful results. Examples of this situation are the application of property refinement and UML/OCL models to specification. Property refinement is described in the literature as a technique that leads to first order logic predicates specifying component functionality. If property refinement is taken to this level, the result is a system design. Property refinement in the CMSpecV&V method refines system level constraints on behavior to into more detailed statements that can be included as specification constraints in OCL, Alloy, or RTL. These constraints are abstract and do not lead to a complete system design. UML/OCL models describe a specification, as is shown in Chapter 5. However, type models comprised solely of UML types and concepts are successfully integrated with Alloy to create a partial graphical specification. The Alloy model can then be checked with the Alloy tool.

A final implication of this research is that automated tools are required to aid validating and verifying run-time configuration management specification of complex distributed systems. The specifications created in this research are complex and large, and are difficult to validate and verify.

One difficulty stems simply from their size - there are a great many constraints and operations that need to be systematically checked. Checking occurs by peer review in the absence of automated tool support. Peer review is useful as a communication forum to increase group understanding of a specification, but personal experience has shown that peer review requires careful planning to be an effective validation or verification mechanism.

Expertise in the languages used in specification notation is also required for peer review. This is especially true of UML/OCL. The UML/OCL language is continuing to develop, and tool support is not available for most recent language changes. Thus, an expert in the most recent definition of UML/OCL is required for peer review. Clearly this situation is not optimal, and it is not generally feasible to obtain experts for every peer review. The result is that complex specifications may contain notation errors that lead to misunderstanding during design and/or implementation.

Failing a group of experts to perform peer reviews, automated tools are a necessity. Automated tools will also help the wider adoption of these languages. (The situation is somewhat analogous to writing a program without ever compiling, linking, or executing it, then giving it to someone else and expecting them to understand what it should do. Depending on the knowledge and experience of both parties, the results can vary widely.).

3. GENERAL OBSERVATIONS

Most of the observations of this research stem from adapting existing research to industrial problems. Others stem from the need for automated tool support.

The primary consideration of industrial architects, designers, and implementers is getting the correct product developed and deployed to customers. Methods used to accomplish this often begin as ad-hoc best practices, and changes to them occur slowly. There usually is not a great deal of time available to improve methods and look for better ones, or ones that can be adapted, from existing research. This is the context in which I began this research.

- I found that adapting existing research to industrial problems is difficult. Most existing research is developed using example problems that are small in comparison to the size of industrial problems. (Size can be measured in terms of physical computer and network systems, the number of issues considered, the amount of system software, the amount of system data, and the amount of system results. Unlike most research problems, industrial problems of loosely coupled distributed systems tend to be large in some or all of these dimensions.) This clearly helps simplify problems encountered during research, but does not help if the technique needs to scale to large systems.
- Another difficulty is that the examples given in papers are usually not very complicated, and are often leave out details crucial to adapting them to larger problems. An example is the fuzzy set technique. The technique is described in the literature in the context of a room scheduling program. However, the paper never discusses how a mathematical relation between system requirements is obtained. It simply uses the mathematical relation in requirements tradeoffs. The results obtained from the fuzzy set technique are of limited value without such a relation, as discussed in Chapter 5.
- A final problem is a tension between the developers of a technique or notation and potential users. Researchers who develop a technique or language look at the technique in-depth, and the technique is their development goal. A potential user however, has a different development goal, and looks at the technique or language as a tool to help achieve that goal. These differing viewpoints are expressed as disagreements between those who developed the technique or

language (who care whether a technique or language is being used *correctly*) and those who use it to reach another goal (who tend to use the technique or language however they see it will help them *expeditiously* reach their goal).

- It is clear from this research that the specification of a complex loosely coupled distributed system must have automated tool support. Such a specification is large and complex, and needs to be usable by a large number of people (e.g. not only one or more architects, but also designers, implementers, testers, etc.). Drawing tools must implement all of the language, including its critical notation aspects, and must be easy to use.
- A final observation is that in a system where run-time configuration is a key focus, services or applications the system performs for a user (e.g. fraud detection in the ATM/IP network monitoring system) is an orthogonal issue, and can be completely ignored in the run-time configuration management specification. However, in systems with a different focus, run-time configuration may not be a completely orthogonal issue from main system functionality. This is the case in the bioinformatics system. Large volumes of dynamic data is the focus of this system. One of the items that must be included in the run-time configuration management specification is how the system configures incremental work on data. Some development of the specification for incremental work is needed in order to identify what aspects of that work need to be dynamically configurable.

1. INTRODUCTION

The research discussed in this dissertation develops and demonstrates a methodology to identify key system requirements and a method that specifies key system requirements for run-time configuration management of a distributed system. It has also developed and demonstrated a method to validate and verify that these key requirements are adequately represented in the specification, and that the specification is internally consistent.

The research has resulted in new methodologies and other contributions. It has also suggested several areas for further research. These are all described in this chapter. Key results of this research are discussed in Section 2. Section 3 discusses research contributions, and Section 4 proposes extensions and future work.

2. KEY RESULTS

There are four key results of this research. They consist of three methodologies and their generalization. Each is discussed below.

2.1 Key System Requirements Identification

A methodology (the KR method) is developed to identify the key system requirements of a complex, loosely coupled distributed system. The KR method draws on multiple viewpoints of the system to identify key requirements, thus the most important areas for key system stakeholders are included. Since the KR method is based

on multiple views of the system, it is likely that the critical configuration management requirements are identified correctly. These views and examples of the requirements they define are:

- The problem domain view.

The view of the problem domain leads to a need to address scale in both the ATM network monitoring system and the bioinformatics system. However, scale has slightly different meanings in these two problem domains.

Scale relates to the number of physical probes and computers in the network monitoring system as well as to the volume of data and the dynamic nature of both the physical system and data. Scale relates to the volume of data as well as to its dynamic nature in the bioinformatics system.

The difference in the meaning of scale in the two problem domains leads to different needs in the system. Scale in the network monitoring system leads to a need to physically configure the monitoring system itself, in light of large numbers of components and constant change. This leads to a key configuration requirement of automatic and incremental movement toward a desired configuration.

Scale in the bioinformatics system leads to a need to deal with large amounts of changing data. This in turn leads to a key requirement of partial and incremental work in the system. From a configuration management point of view this becomes a requirement to dynamically configure partial and incremental work.

- The view of users of the final system, their roles and tasks.

An example of a requirement identified from this view is that since operators of network monitoring systems often do not trust that the system is operating properly, constraints must be place on automatically physically changing the system while it is in operation. This leads to a requirement to allow users to direct the amount of automation in the system.

An example of a configuration management requirement in the bioinformatics domain developed from user roles and tasks is the need to allow users to protect data and unpublished results. This is the security aspect of the bioinformatics system.

- The view of generic possible configuration management needs.

Systematically studying the generic list of configuration management possibilities shown in Chapter 4 allows many to be omitted that are less important in the system being developed, while focusing on more important items, new items, or items interpreted in different ways depending on the problem domain. For example, generic possibilities in the ATM/IP monitoring system can be focused via the idea of a *configuration* that contains the most important of these items. Other generic items can be omitted from consideration in this system.

Many of the generic possibilities are not critical to the bioinformatics system (e.g. software components can easily be manually configured since there are very few computers in the system), so they can be omitted from further consideration. New ideas can be added to the set of items that must be managed (e.g. a research path used to develop, explore, and validate research hypotheses).

The results of each step in the KR method can be documented, and thus the entire process of defining key configuration management needs is traceable. A documented process is useful when system development personnel change, and can be crucial when various aspects of the system requirements change (e.g. the addition of a new type of user role with different requirements than have been taken into consideration during the initial determination of configuration management needs). Then, the new requirements can be compared to the original ones.

Finally, the outcome of the KR method is a small set of key run-time configuration management requirements. Since it is critical that only key requirements of a system are addressed during system specification and development, the KR method lends credence to the few key requirements that are actually chosen. This is in contrast to ad-hoc methods which often result in many more requirements than can actually be met.

The KR method is informally validated by system architects (in the case of the ATM network monitoring system) and by user studies (in the case of the bioinformatics system). The KR method identifies a key run-time configuration management requirement missing from the original network monitoring system. The method also identifies a key run-time configuration management requirement of the bioinformatics system according to user study results.

2.2 Specification Methodology and Notation

The specification portion of the CMSpecV&V method developed is applicable to problems of complexity and scale found in industrial use since it was developed from such a problem. It uses strengths found in existing techniques (especially notations that

exist and are already in use in industry) rather than developing everything from scratch. The method helps an architect take vague statements regarding system-wide constraints on behavior and turn them into system component constraints. It takes validation and verification into account as the specification is being developed rather than as an afterthought; this makes the validation and verification step smoother, besides helping to find problems earlier, before the specification is finished. The methodology uses existing notations along with automated tools to create a complete specification of key system configuration management requirements. These notations are UML for *type* models (see Chapter 5 for a discussion of UML *type* models), as well as Alloy. The two notations complement each other and allow a complete description of the specification to be created. (Note that alternatively, OCL can be used with UML *type* models to accomplish the same thing, except that automated validation is harder due to lack of adequate tool support.) Finally, the methodology takes into consideration the temporal constraints needed by the configuration management requirements and provides a way to specify them as well. (Note that temporal constraints can only be added as comments to a specification since the specification notation languages do not include the notion of time as a basic concept. However, temporal constraints are written as first order logic predicates, and therefore are formal and can be verified.)

The specification method results in specifications that represent all the run-time configuration management needs of the ATM network monitoring system and the bioinformatics system. The resulting specifications can be validated and verified using the CMSpecV&V method.

2.3 Specification Validation and Verification

The validation and verification portion of the CMSpecV&V method was also developed as part of this research. It is based on the extension of existing techniques (i.e. multiple architectural views) as well as on automated tools. Multiple architectural views test key system requirements for adequate representation in the specification, by including key requirement views. These multiple views are also used to informally verify internal consistency in the specification. Automated tools also verify internal specification consistency.

The CMSpecV&V validation and verification method does allow reasoning about the existence of key requirements in the specification, and specification internal consistency, but the method does not allow statements to be made about provable correctness of the specification, nor of any resulting implementation. The CMSpecV&V method also finds errors in specifications, as shown by the initial validation and verification of the ATM network monitoring system run-time configuration management specification (see Chapter 7).

2.4 Generalizations

The three methods discussed in this research are shown to generalize across two widely different problem domains of loosely coupled distributed systems. They were developed using the ATM/IP network monitoring system. The focus of this system is on physical scale, and system operation depends heavily on its run-time configuration. The methods were successfully used on a loosely-coupled distributed system in another problem domain, that of bioinformatics. The bioinformatics system focuses on data, and scale is again an issue, but this time in terms of amount of data as well as its dynamic

nature. The results of applying the CMSpecV&V method to the bioinformatics system suggest that the method can be used to specify key system requirements other than configuration management. In particular, the methodology provided insights into the requirements for partial and incremental work (e.g. searches and analyses). The methods seem to generalize not only across problem domains, but also across key system requirements, but this is a topic for future study.

3. CONTRIBUTIONS

Beyond the development of the methods discussed previously, there are four other contributions provided by this research.

- The first contribution is that the methods were developed to scale to problem sizes such as those seen in industry. This was accomplished by developing them using a complex industrial problem. The specification problem was not simplified; it deals with a critical aspect (run-time configuration management) of a commercially available system.
- The second contribution is that system-wide constraints on behavior can be methodically refined from vague statements (i.e. “configuration of the network monitoring system must occur incrementally”) to component functionality that can be specified on a per-component basis. This refinement is possible for incremental and automatic configuration management of the network monitoring system as well as for incremental and persisted work in the bioinformatics system.
- The third contribution is the use of property refinement techniques to produce problem specific definitions of incremental and automatic configuration for the

network monitoring system, and incremental and persisted work for the bioinformatics system. At present, these definitions are problem specific. That is, the definition of incremental configuration in the network monitoring system, and incremental work in the bioinformatics system are different. Future work can determine if there is a general definition of incremental operation that is specific enough to be useful, but general enough to encompass both of the definitions developed in this research. This contribution is important because non-functional requirements such as incremental and automatic operation have not been studied or defined to the same depth as non-functional requirements like reliability. In-depth study is needed to understand the behavior of these non-functional system properties.

- Finally, a fourth contribution is the inclusion of temporal constraints as needed by the network monitoring system and the bioinformatics system in their respective specifications. Sequence constraints are often included in design specifications, and are part of the UML language. Special languages have been developed to deal with the complex temporal constraints of reactive and real-time systems. However, simple temporal constraints such as those needed by the network monitoring system and the bioinformatics system either cannot be represented in a specification language (i.e. UML) or do not call for an overly rich temporal specification language (e.g. TILCO-X or an interval logic). This research uses an existing event-driven temporal logic (RTL) to express simple temporal constraints. Future work can explore including temporal logic in existing specification languages.

4. FUTURE WORK

This research suggests many areas for future research. One of the most critical is the evaluation of the KR and CMSpecV&V methods in terms of their usefulness in creating a system implementation that does exhibit the identified key requirements. This can be accomplished by taking either the network monitoring system or bioinformatics system run-time configuration management specification and continuing its development through design and implementation, then testing the implementation to ensure key requirements are present.

Generalizing the methods can be extended in two directions. The first is to continue testing the methods on loosely coupled distributed systems in other domains, in the area of their run-time configuration management needs. Systems need widely varying problem domains, as well as varying foci. (Recall that the focus of the network monitoring system is on a large scale, dynamic physical system, and the focus of the bioinformatics system is on a large volume of dynamic data.)

A second generalization direction is to test the methods on requirements other than run-time configuration management. Two possible requirements stem from the bioinformatics system, those of incremental and persistent work (e.g. analyses and searches).

Another area of work is to develop the crucial specification of tools that can support the methods developed in this research (this includes investigating the addition of RTL-style temporal logic to tools such as Alloy). Such tools are needed for industrial users to widely adopt any new methodologies. Related work is to suggest extensions to existing tools that help meet these tool specifications.

Finally, research needs to be devoted to transferring these methods to industrial software developers. This work needs to be done in terms of what training, documentation, and examples are needed to enable transfer these methods from an industrial research setting to a product development setting.

REFERENCES

1. **Abeck, S., A. Koppel, and J. Seitz.** 1998. A management architecture for multi-agent systems. *Proceedings IEEE International Workshop on System Management*:133-138.
2. **Aschemann, G. and R. Kehr.** 1998. Towards a requirements-based information model for configuration management. *Proceedings 4th International Conference on Configurable Distributed Systems*:181-188.
3. **Babaoglu, O., E. Fromentin, and M. Raynal.** 1996. A unified framework for the specification and run-time detection of dynamic properties in distributed computations. *Journal of Systems and Software* **33**(3, July):287-298.
4. **Bassett, D. E. J., M. D. Eisen, and M. S. Boguski.** 1999. Gene expression informatics -- it's all in your mine. *Nature Genetics* **21**(1, January).
5. **Bauer, M. A., Bunt, El Rayess, Finnigan, Kunz, Lutfiyya, Marshall, Martin, Oster, Powley, Rolia, Taylor, and Woodside.** 1997. Services supporting management of distributed applications and systems. *IBM Systems Journal* **36**(4):508-526.
6. **Bellini, P. and P. Nesi.** 2001. TILCO-X, an extension of TILCO temporal logic.
7. **Berghoff, J., O. Drobnik, A. Lingnau, and C. Monch.** 1996. Agent-based configuration management of distributed applications. *Proceedings 3rd International Conference on Configurable Distributed Systems*:52-59.
8. **Carver, R.** 1996. Testing abstract distributed programs and their implementations: a constraint-based approach. *Journal of Systems and Software* **33**(3, June):223-237.
9. **Chadha, R. and S.-Y. Wu.** 1996. Managing distributed systems using OSI systems management. *Proceedings IEEE International Workshop on System Management*:117-126.

10. **Cheung, S. C. and J. Kramer.** 1999. Checking subsystem safety properties using compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology* 8(1, January):49-78.
11. **Cheung, S. and J. Kramer.** 1996. Checking subsystem safety properties in compositional reachability analysis. *18th International Conference on Software Engineering*:144-154.
12. **Ciapessoni, E., A. Coen-Porisini, E. Crivelli, D. Mandrioli, P. Mirandola, and A. Morzenti.** 1999. From formal models to formally based methods: an industrial experience. *ACM Transactions on Software Engineering and Methodology* 8(1, January):79-113.
13. **Clarke, E. M. and J. M. Wing.** 1996. Formal methods: state of the art and future directions. CMU-CS-96-178. CMU Computer Science Technical Report, August.
14. **Coatta, T. and G. Neufeld.** 1994. Distributed configuration management using composite objects and constraints. *Proceedings 2nd International Workshop on Configurable Distributed Systems*:112-122.
15. **Cooper, G. M.** 1997. *The Cell*, ASM Press.
16. **Cristian, F.** 1993. Automatic reconfiguration in the presence of failures. *Software Engineering Journal* 8(2, March):53-60.
17. **Davis, A.** 1993. *Software requirements objects, functions, and states*, Prentice Hall.
18. **Dill, D. L. and J. Rushby.** 1996. Acceptance of formal methods: lesson from hardware design. *IEEE Computer*:23-24.
19. **Dillon, L., G. Kutty, P. Moser, P. Melliar-Smith, and Y. Ramakrishna.** 1994. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology* 3(2, April):131-165.
20. **Fenton, N. E. and S. L. Pfleeger.** 1997. *Software metrics; a rigorous & practical approach*. Second Edition, PWS Publishing Company.

21. **Fowler, M. and K. Scott.** 1997. UML Distilled, Addison-Wesley.
22. **France, R.** 1998. Specification and description language for software engineers.
23. **Franklin, J.** 2001. The science of conjecture, The Johns Hopkins University Press.
24. **Georg, G. and J. Bieman.** 2001. Identifying run-time configuration management requirements in distributed systems. submitted to Transactions on Software Engineering.
25. **Georg, G. and S. Seidman.** 2000. The use of architectural description languages to describe a distributed measurement system. Proceedings of Engineering of Computer Based Systems.
26. **Glass, R. L.** 1996. Formal methods are a surrogate for a more serious software concern. IEEE Computer:19.
27. **Hall, A.** 1996. Using formal methods to develop an ATC information system. IEEE Software:66-76.
28. **Hall, A.** 1996. What is the formal methods debate about? IEEE Computer:22-23.
29. **Herlea, D. E.** 1996. User's involvement in the requirements engineering process. Proceedings 10th Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW).
<http://ksi.cpsc.ucalgary.ca/KAW/KAW96/herlea/FINAL.html>.
30. **Hiltunen, M.** 1998. Configuration management for highly-customizable services. Proceedings 4th International Conference on Configurable Distributed Systems:197-205.
31. **Hinchey, M. G. and J. P. Bowen.** 1996. To formalize or not to formalize. IEEE Computer:18-19.
32. **Holloway, C. M. and R. W. Butler.** 1996. Impediments to industrial use of formal methods. IEEE Computer:25-26.

33. **Issarny, V., C. Bidan, and T. Saridakis.** 1998. Characterizing coordination architectures according to their non-functional execution properties. *Proceedings 31st Hawaii International Conference on System Science*:275-283.
34. **Jackson, D.** 2000. Automating first-order relational logic. *ACM Foundations of Software Engineering*.
35. **Jackson, D.** 2001. Alloy revisited. IFIP 2.9, Hawks Cay, Florida, February.
36. **Jackson, D., I. Schechter, and I. Shlyakhter.** 2000. Alcoa: the Alloy constraint analyzer. *Proceedings International Conference on Software Engineering*.
37. **Jackson, D. and J. Wing.** 1996. Lightweight formal methods. *IEEE Computer*:21-22.
38. **Jahanian, F., A. K. Mok, and D. A. Stuart.** 1988. Formal specification of real time systems, TR-88-25. Department of Computer Sciences, The University of Texas at Austin.
39. **Jarke, M.** 1996. Meta models for requirements engineering. *Proceedings 10th Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW)*. <http://spuds.cpsc.ucalgary.ca/KAW/KAW'96/jarke/Jarke.html>.
40. **Jmaiel, M.** 1996. Specification of communication protocols using temporal logic. *Journal of Systems and Software* **33**(3, June):299-312.
41. **Jones, C. B.** 1996. A rigorous approach to formal methods. *IEEE Computer*:20-21.
42. **Kang, D.-I., R. Gerber, and M. Saksena.** 1997. Performance-based design of distributed real-time systems. *IEEE*:2-13.
43. **Klee, R.** 1997. *Introduction to the philosophy of science: cutting nature at its seams*, Oxford University Press.
44. **Klein, M. and C. Dellarocas.** 1999. Exception handling in agent systems. *Proceedings Agents '99*:62-68.

45. **Kramer, J. and J. Magee.** 1998. Analysing dynamic change in software architectures: a case study. Proceedings 4th International Conference on Configurable Distributed Systems:91-100.
46. **Kuhn, T.** 1970. The structure of scientific revolutions. second edition, enlarged, The University of Chicago Press.
47. **Larman, C.** 1998. Applying UML and Patterns, Prentice Hall.
48. **Li, X.** 1999. Specification and simulation of a concurrent real-time system. International Symposium on Software Engineering for Parallel and Distributed Systems.
49. **Lim, A. S.** 1994. A state machine approach to dynamic reconfiguration of distributed applications. Proceedings 2nd International Workshop on Configurable Distributed Systems.
50. **Liu, X. and J. Yen.** 1996. An analytic framework for specifying and analyzing imprecise requirements. IEEE International Conference on Software Engineering:60-69.
51. **Magee, J.** 1999. Behavioral analysis of software architectures using LTSA. IEEE International Conference on Software Engineering:634-637.
52. **Magee, J. and J. Kramer.** 1996. Dynamic structure in software architectures. Proceedings SIGSOFT 96:3-14.
53. **Manna, Z. and A. Pnueli.** 1992. The temporal logic of reactive and concurrent systems, Springer Verlag.
54. **Martins, L. E. G.** 1999. An approach to software requirements elicitation using precepts from activity theory. Proceedings 14th IEEE International Conference on Automated Software Engineering (1999):15-23.
55. **McDermid, J.** 1994. On dependability, its measurement and its management. High Integrity Systems 1(1):17-26.

56. **Morris, P., M. Masera, and M. Wilikens.** 1995. Industrial workshop on requirements engineering. <http://ntsta.jrc.it/dsa/workshop1095/report.htm>, October 16-17.
57. **Moser, L. E., Y. Ramakrishna, G. Kutty, P. Melliar-Smith, and L. Dillon.** 1997. A graphical environment for the design of concurrent real-time systems. *ACM Transactions on Software Engineering and Methodology* 6(1, January):31-79.
58. **Mylopoulos, J., L. Chung, and B. Nixon.** 1992. Representing and using nonfunctional requirements: a process-oriented approach. *IEEE Transactions on Software Engineering* 18(6, June):483-497.
59. **Mylopoulos, J., L. Chung, and E. Yu.** 1999. From object-oriented to goal-oriented requirements analysis. *Communications of the ACM* 42(1, January):31-37.
60. **Nature Genetics.** 1999. 21(1, January).
61. **Nitsche, U.** 1996. Verification and behavior abstraction towards a tractable verification technique for large distributed systems. *Journal of Systems and Software* 33(3, June):273-285.
62. **O'Brien, P.** 1999. FIPA '98 draft specification: part 1 agent management. FIPA web site, July.
63. **OMG.** 2001. OMG Unified Modeling Language specification. V1.4 draft, February.
64. **Oreizy, P. and R. Taylor.** 1998. On the role of software architectures in runtime system reconfiguration. *Proceedings 4th International Conference on Configurable Distributed Systems*:61-70.
65. **Ostroff, J.** 1999. Composition and refinement of discrete real-time systems. *ACM Transactions on Software Engineering and Methodology* 8(1, January):1-48.
66. **Parallel and Distributed Computing Handbook.** 1999. Ed. A. Zomaya, McGraw-Hill.

67. **Paulter, D., S. Woods, and A. Quilici.** 1997. Exploiting domain-specific knowledge to refine simulation specifications. Proceedings 12th IEEE International Conference on Automated Software Engineering:117-124.
68. **Pires, L. F., M. v. Sinderen, and C. A. Vissers.** 1993. Advanced design concepts for open distributed systems development. IEEE:419-425.
69. **Putter, P. and J. D. Roos.** 1992. A general building block for distributed system management. South African Computer Journal:141-145.
70. **Radestock, M. and S. Eisenbach.** 1994. What do you get from a pi-calculus semantics? Proceedings PARLE 94 Parallel Architectures and Languages Europe:635-647Lecture Notes in Computer Science 817, Springer Verlag.
71. **Richards, D. and T. Menzies.** 1998. Extending the SISYPHUS III experiment from a knowledge engineering task to a requirements engineering task. Proceedings 11th Workshop on Knowledge Acquisition, Modeling and Management (KAW). <http://ksi.cpsc.ucalgary.ca/KAW/KAW98/richards1/>.
72. **Richters, M. and M. Gogolla.** 2000. Validating UML models and OCL constraints. Proceedings UML 2000.
73. **Rubin, H. and N. Natarajan.** 1994. A distributed software architecture for telecomm networks. IEEE Network.
74. **Saiedian, H.** 1996. An invitation to formal methods. IEEE Computer:16-17.
75. **Saridakis, T. and V. Issarny.** Developing dependable systems using software architecture.
76. **Sarkar, H.** 1983. A theory of method, University of California Press.
77. Scientific inquiry - readings in the philosophy of science. 1999. Ed. R. Klee, Oxford University Press.
78. **Seidman, S. and M. Rice.** 1999. Describing software architectures and architectural styles. Proceedings 1st Working IFIP Conference on Software Architecture.

79. **Simpson, H. R.** 2001. Protocols for process interaction.
80. Specification and Description Language. ITU-T Recommendation Z.100. 1996.
81. Third Edition Electronic Instrument Handbook. 2000. Ed. C. Coombs, McGraw-Hill.
82. **Thomas, B., D. Stotts, and L. Kumar.** 1998. Warping distributed system configurations. Proceedings 4th International Conference on Configurable Distributed Systems:136-143.
83. **Thomas, M.** 1994. A proof of incorrectness using the LP theorem prover: the editing problem in the Therac-25. High Integrity Systems 1(1):35-48.
84. **Thornton, J. D.** 1996. Practical description of configurations for distributed systems management. Proceedings 3rd International Conference on Configurable Distributed Systems:36-43.
85. **Tichy, W. F.** 1985. RCS - system for version control. Software - Practice and Experience 15(7, July):637-654.
86. **Tichy, W. F.** 1992. Programming-in-the-large: past, present, and future.
87. **Walker, M.** 1963. The nature of scientific thought, Prentice Hall.
88. **Wang, J. T., A. Shapiro, and D. Shasha.** 1999. Pattern Discovery in Biomolecular Data, Oxford University Press.
89. **Young, T.** 1996. Intelligent agents in systems management; a developers experience. Proceedings First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology:743-753.
90. **Zarras, A. and V. Issarny.** 1998. A framework for systematic synthesis of transactional middleware. Proceedings Middleware 98, IFIP International Conference on Distributed Systems Platforms and Open Distribution.
91. **Zave, P.** 1996. Formal methods are research, not development. IEEE Computer:26-27.

92. **Zave, P. and M. Jackson.** 1997. Four dark corners of requirements engineering. ACM Transactions on Software Engineering and Methodology **6**(1, January):1-30.
93. **Zimmermann, M. and O. Drobniak.** 1994. Specification and implementation of reconfigurable distributed applications. Proceedings 2nd International Workshop on Configurable Distributed Systems:23-34.