UNIVERSITY OF CENTRAL FLORIDA
DISSERTATION APPROVAL

The members of the Committee approve the dissertation entitled *Smart Compositional Wrappers* of Saleh Matar Mohammed Al-Hatali, defended October 21, 2002.

Gwendolyn H. Walton, Chair

Mostafa A. Bassiouni
Committee Member

Christian S. Bauer
Committee Member

Ernest W. Gibbs
Committee Member

Darrell G. Linton
Committee Member

It is recommended that this dissertation be used in partial fulfillment of the requirements for the degree of Doctor of Philosophy from the School of Electrical Engineering and Computer Science in the College of Engineering and Computer Science.

Erol Gelenbe, Director of the School of Electrical Engineering and Computer Science

Juin Liou, Acting Associate Dean for Graduate Studies

Martin P. Wanielista, Dean of College

Patricia J. Bishop
Vice Provost and Dean of Graduate Studies

The committee, the college, and the University of Central Florida are not liable for any use of the materials presented in this study.

SMART COMPOSITIONAL WRAPPERS

by

SALEH MATAR MOHAMMED AL-HATALI
B.S. University of Arizona, 1989
M.Sc. Florida Institute of Technology, 1995

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2002

Major Professor: Gwendolyn H. Walton

© 2002 Saleh Matar Mohammed Al-Hatali

ABSTRACT

Much of the software engineering community has chosen to use software components in assembling their systems in the hope of shortening development time, reducing development costs, providing more software functionality, and improving software usability. In the best case, a single ready-made component is immediately available and satisfies all of the user's requirements. Such a case rarely exists because of the different assumptions by component vendors and component users on the environment in which the component will operate.

In the absence of the best case, the approach described in the software engineering literature typically includes these areas: specification, acquisition, and integration. In the specification area, users describe needed components. The acquisition area involves finding candidate components, and selecting and acquiring the best ones that fit the user's requirements. During the integration stage, a "wrapper" code is written to encase each selected component. Such code is required to augment or restrict the behavior of the component to meet a set of the user's requirements. The integration process also involves writing the software to wire together the interfaces of the wrapped components to form a final system.

The software engineering literature is lacking guidance in each of the following areas. There are no simple and easy to use guidelines or standards for users to specify the characteristics of desired components and the objectives and constraints of target systems. Simple and easy to use guidelines are also lacking for component suppliers to describe the characteristics and behavior of the components they can provide. Additionally, there is little to no tool support for automatic acquisition and integration of components.

This dissertation provides support to each of the specification, acquisition, and integration areas by means of a new "smart compositional wrapper" approach. This approach encases an entire set of components instead of wrapping individual ones. The compositional wrapper would include: (a) the set of selected components and their associated artifacts; (b) the code necessary to: wire together the component interfaces, filter out unwanted functionality, and manage the inclusion of the compositional wrapper in host applications; and (c) a set of "smart features" to support the management and automatic upgrade and maintenance of the compositional wrapper and its components.

To support the implementation of smart compositional wrappers, the dissertation provides three contributions.

First, the dissertation describes two new formal specification languages: The Wrapper Specification Language (WSL) and the Component Specification Language (CSL). WSL is designed for system integrators who want to create smart compositional wrappers from off-the-shelf components. Component suppliers can use CSL to describe their components. The new WSL and CSL specification languages contain constructs that were chosen carefully to allow software tools (like wrapper generators) to parse and

match specifications written in the two languages easily, and to allow the software tools to automate the activities that they perform during the composition process.

The second contribution is in the description of a smart wrapper generator that takes a WSL specification for a compositional wrapper, searches and collects components whose specifications conform to CSL, acquires such components, configures them, wires them together, and generates the glue code to wire component interfaces and implement required smart features.

The third major contribution is the description of a complete composition process that can be used to produce smart compositional wrappers. In contrast to published literature accounts that only describe individual areas of component-based system development, the composition process supports the complete component-based system development lifecycle. The dissertation describes the detailed activities of each stage and the possible opportunities of automating them using wrapper generators. This research extended the discussion to some of the activities that can take place during the assembling and operation of Compositional Wrappers-Based (CWB) systems. Automation of many of the process activities were made possible by the variety and simplicity of WSL and CSL constructs; by the well-defined stages and activities of the composition process; and by the powerful built-in capabilities of the wrapper generator. The developed prototype and the included algorithms, code snippets, and example specifications illustrate the possibility of automating such activities.

A high level ROI analysis is also provided to demonstrate that this new compositional wrappers' approach is cost effective with respect to component acquisition, as-

sessments, and wrapper generation. The ROI analysis shows how organizations can save hundreds of thousands of dollars in the development of the wrapping code alone. The dissertation also shows how compositional wrappers can reduce the number of building blocks, the cost, and the legal consequences of developing and operating CWB systems.

To my ever-loving parents,
To my ever-loving and ever-supporting wife,
To all my beloved and smart children,
To all my brothers and sisters in Islam,
I dedicate this work.

ACKNOWLEDGMENTS

First and foremost, all praises are to Allah, the Lord of the universe, the most merciful, the most compassionate, for helping me to accomplish this work. My Lord: without your guidance and support, this work would not be possible. So, thank you my Lord.

Second, I would like to express my thanks to all my committee members for helping me in this work. My special thanks go to my great advisor, Dr. Gwendolyn Walton, for her kindness and dedication to bring this work to where it is now. Her insightful comments and suggestions opened new perspectives for me to look at this work.

I would like also to express my deepest gratitude to my parents, my wife, my children, and all my family and friends for their continuous support and encouragements.

TABLE OF CONTENTS

LIST OF FIGURES _____ *xv*

LIST OF TABLES _____ *xvi*

LIST OF ABBREVIATIONS _____ *xvii*

Chapter 1 : INTRODUCTION _____ *1*

1.1 Problem Statement _____ **1**

1.2 Proposed Solution _____ **3**

1.3 Contributions _____ **6**

1.4 Relationship to Other Research _____ **10**

1.5 Reader's Guide _____ **12**

Chapter 2 : BACKGROUND _____ *15*

2.1 Component-Based Software Engineering (CBSE) _____ **16**

2.2 Software Components _____ **17**

2.2.1 Architectures and Frameworks _____ 19

2.2.2 Component Marketing and Distribution _____ 22

2.3 Software Wrappers _____ **23**

Chapter 3 : COMPOSITIONAL WRAPPERS _____ *26*

3.1 The Need for Compositional Wrappers _____ **28**

3.1.1 Coding and Integration Effort _____ 29

3.1.2 Security _____ 35

3.1.3 Network Bandwidth and Disk Access _____ 36

3.1.4 Upgrade and Maintenance _____ 37

3.1.5 Performance _____ 38

3.1.6 Injection of Smart Features _____ 39

3.1.7 Monetary Cost _____ 39

3.1.8 Legal Issues _____ 40

3.2 Challenges in Using Compositional Wrappers _____ **40**

Chapter 4 : THE WRAPPER GENERATOR _____ *43*

4.1 A Data Structure to Hold the Developed Compositional Wrappers' Information ____ **45**

Chapter 5 : SPECIFICATION LANGUAGES ____ *58*

5.1 Relationship to Existing Specification Languages ____ **60**

5.2 Selection of the Specification Languages ____ **62**

5.3 Specification Languages Overview ____ **63**
5.3.1 The Languages Syntax ____ 63
5.3.2 Examples of Language Constructs ____ 65
5.3.3 Examples of Modified ADL Constructs ____ 66
5.3.4 Basic Constructs ____ 66
5.3.5 General Constructs ____ 70
5.3.6 Constructs Borrowed From ADL ____ 72
5.3.7 Interface Behaviors ____ 72

5.4 The Content of the Specification Documents ____ **75**
5.4.1 The Component Specification Document ____ 75
5.4.2 The Wrapper Specification Document ____ 77

Chapter 6 : COMMON SECTIONS ____ *82*

6.1 Specifying Behavior ____ **82**
6.1.1 Variable Declarations ____ 82
6.1.2 Data Types ____ 83
6.1.3 Inputs and Outputs ____ 92
6.1.4 Interface Behaviors ____ 97
6.1.5 Interface Stimuli ____ 103

6.2 A Complete Behavioral Specification Example ____ **105**

Chapter 7 : THE COMPONENT SPECIFICATION LANGUAGE ____ *119*

7.1 The Component Specification Constructs ____ **119**
7.1.1 The Component Provider ____ 121
7.1.2 The Component Version ____ 121
7.1.3 Used Technology ____ 122
7.1.4 Supported Operating Systems ____ 122
7.1.5 Supported Storage Databases ____ 123
7.1.6 The Development Language ____ 123
7.1.7 The Component Formats ____ 124
7.1.8 Available License Models ____ 125
7.1.9 Licensing Types ____ 125
7.1.10 The Component Cost ____ 126
7.1.11 Available Support Methods ____ 126
7.1.12 Specification of New Updates ____ 127
7.1.13 The Downloadable Artifacts ____ 128
7.1.14 Installing Downloaded Files ____ 129

7.1.15 Configuring Downloaded Files ____ 131
7.1.16 Provided Interfaces ____ 138
7.1.17 Required Interfaces ____ 139
7.1.18 Software Constraints ____ 140
7.1.19 Hardware Constraints ____ 144
7.1.20 Network Constraints ____ 145
7.1.21 Optional Add-Ons ____ 145

7.2 The Provider Specification ____ 146

7.3 A Component Specification Example ____ 147

Chapter 8 : THE WRAPPER SPECIFICATION LANGUAGE ____ 154

8.1 Importance Ranking Scheme ____ 155

8.2 The Wrapper Specification Constructs ____ 156
8.2.1 The Wrapper's Owner ____ 158
8.2.2 The Wrapper's Developers ____ 159
8.2.3 Preferred Component Suppliers ____ 160
8.2.4 Required Interfaces ____ 160
8.2.5 Personal Information ____ 161
8.2.6 Payment Information ____ 161
8.2.7 Defining a Working Directory ____ 162
8.2.8 Controlling the Wrapper Generation Process ____ 163
8.2.9 Smart Features ____ 166

8.3 Specifying the Wrapper Architecture ____ 192
8.3.1 A Complete Architecture Specification Example ____ 201

8.4 Describing People ____ 211

8.5 Specifying User Profiles ____ 212

8.6 Specifying Credit Cards ____ 213

8.7 Specifying Bank Accounts ____ 214

8.8 A Complete Wrapper Specification Example ____ 215

Chapter 9 : INTRODUCTION TO THE COMPOSITION PROCESS ____ 224

9.1 Composition Process Overview ____ 228

9.2 Architectural Design Considerations ____ 231

9.3 Building CWB Systems ____ 233
9.3.1 Synchronous Composition ____ 233
9.3.2 Asynchronous Approach ____ 238

9.4 Preparing for The Composition Process ____ 242

9.5 Specifying Needed Components ____ 245

Chapter 10 : SEARCHING FOR COMPONENTS ____ *247*

10.1 Interpreting the User Requirements ____ **249**

10.2 Searching For Components ____ **250**
10.2.1 Formulating the Search Request ____ 251
10.2.2 Specifying the Search Scope ____ 253
10.2.3 How to Interact With Search Agents? ____ 253
10.2.4 How Does the Wrapper Generator Interact With Middleware Agents? __ 266

10.3 Examining Search Results ____ **271**

10.4 How Does the Wrapper Generator React to Unsatisfied Search Requests? **272**
10.4.1 Specification Refinement ____ 273
10.4.2 Wrapping Legacy Systems ____ 274
10.4.3 Developing New Components ____ 275

Chapter 11 : PRELIMINARY EVALUATION AND CANDIDATES SELECTION *278*

11.1 Candidates Selection ____ **280**

11.2 Component Reservation ____ **281**

Chapter 12 : DETAILED EVALUATION AND SELECTION OF BEST-FIT COMPONENTS ____ *284*

12.1 Vendor Evaluation ____ **292**

12.2 Risks Evaluation ____ **293**
12.2.1 Feasibility Analysis ____ 295
12.2.2 Development Team Capability Assessment ____ 296

12.3 Testing Components ____ **296**
12.3.1 What to Test For? ____ 297
12.3.2 Test Cases ____ 298
12.3.3 Tests Adequacy ____ 299
12.3.4 Executing Tests ____ 300
12.3.5 Analyzing Test Results ____ 305
12.3.6 Displaying Analysis Results ____ 315
12.3.7 Dealing With Component Interdependencies ____ 317

12.4 Optional Features Selection ____ **317**

12.5 Selection of Best-Fit Components ____ **319**
12.5.1 How the Matching Process is Performed? ____ 321
12.5.2 An Algorithm to Assign Weights to Non-Behavioral Constructs and to Their Options ____ 324
12.5.3 An Algorithm to Assign Weights to Inputs ____ 325
12.5.4 An Algorithm to Assign Weights to Outputs ____ 326
12.5.5 An Algorithm to Evaluate Non-Behavioral Constructs ____ 327
12.5.6 An Algorithm to Evaluate Inputs ____ 334

12.5.7 An Algorithm to Evaluate Outputs ____ 336
12.5.8 An Algorithm to Check the Compatibility of Two Inputs ____ 338
12.5.9 An Algorithm to Check the Compatibility of Two Outputs ____ 342
12.5.10 An Algorithm to Check the Compatibility of Two Assertions ____ 345
12.5.11 An Algorithm to Check the Compatibility of Two Interfaces ____ 347
12.5.12 An Algorithm to Evaluate Behavioral Specifications ____ 350
12.5.13 An Algorithm to Select Best-Fit Components ____ 351

12.6 Groups Evaluation ____ 354

12.7 Calculating Additional Needed Components ____ 357
12.7.1 Matching Provided and Required Interfaces ____ 359
12.7.2 Estimating Additional Needed Components ____ 363

Chapter 13 : COMPONENTS ACQUISITION AND ADAPTATION ____ 368

13.1 Conducting Negotiations ____ 371

13.2 Downloading Components ____ 373

13.3 Installing Downloaded Components and Optional Features ____ 380

13.4 Components Adaptation ____ 380

Chapter 14 : WRAPPING COMPONENTS ____ 387

14.1 Load Balancing and Emergency Recovery ____ 393

14.2 Late Composition ____ 394

14.3 Confinement of Components ____ 396

14.4 Code Size Estimation ____ 397
14.4.1 Available Sizing Techniques ____ 398
14.4.2 A Proposed Technique for Estimating the Glue Code Size of Compositional Wrappers ____ 399

14.5 Selecting the "Best" Interfaces ____ 413

14.6 Wiring Required and Provided Interfaces ____ 415

14.7 Adding Smart Features to Generated Wrappers ____ 416
14.7.1 Managing Authentication Codes ____ 417
14.7.2 Self-Documentation ____ 418
14.7.3 Tracking ____ 420
14.7.4 Licensing ____ 422
14.7.5 State and Structure ____ 424
14.7.6 Auto-Updating ____ 425
14.7.7 Testing ____ 427
14.7.8 Security ____ 427

14.8 Specifying the Wrapping Options ____ 428

14.9 Other Guidelines for Successful Generation of Compositional Wrappers 430
14.10 Generating the Compositional Wrapper 437
14.11 Handing the Compositional Wrapper to the System Integrator 443
14.12 Wrapper Testing 444
Chapter 15 : BEYOND THE WRAPPER GENERATION PROCESS 447
15.1 System Assembly 450
15.2 System Testing 450
15.3 Maintaining the CWB System 451
15.3.1 Component Updates 453
15.3.2 Maintaining CWB Systems 459
15.4 Monitoring and Controlling CWB Systems 462
15.4.1 Abnormal Events 463
15.4.2 Recurring Events 467
Chapter 16 : RETURN ON INVESTMENT 472
16.1 Potential Savings in Auto-Generating the Compositional Wrapper's Glue Code 472
16.2 The Cost of Assembling and Generating Compositional Wrappers 475
Chapter 17 : SUMMARY AND CONCLUSIONS 479
17.1 Highlights of Work Done and Lessons Learned 480
17.2 Future Research 482
17.2.1 Enhancing the Specification Languages 483
17.2.2 Architecture Implementation 488
17.2.3 Behavioral Matching 488
APPENDIX: THE WRAPPER GENERATOR PROTOTYPE 490
LIST OF REFERENCES 495
INDEX 521

LIST OF FIGURES

Figure 1: Relationship to Other Research. _____ *11*
Figure 2: The relationship between components and objects _____ *19*
Figure 3: The development cycle that component integrators go through when they build their systems entirely from single components. _____ *31*
Figure 4: The development cycle that component integrators go through when they build their systems using compositional wrappers with the help of wrapper generators. ____ *32*
Figure 5: The wrapper generator's internal structure that will hold the state of each developed compositional wrapper _____ ***Error! Bookmark not defined.***
Figure 6: The different sections that compose component specification documents. __ *77*
Figure 7: The different sections that compose wrapper specification documents. ____ *81*
Figure 8: Specification matching for the MyBank specification example. _____ *107*
Figure 9: The complete specification for a bank example _____ *118*
Figure 10: A CSL specification example _____ *153*
Figure 11: A gas station architectural specification _____ *211*
Figure 12: A complete WSL specification for the bankAccount example _____ *223*
Figure 13: The top-level activities in a CWB system _____ *226*
Figure 14: Process Overview _____ *228*
Figure 15: Processes and interactions involved in the wrapper generation process ***Error! Bookmark not defined.***
Figure 16: Compositional wrappers development process _____ *230*
Figure 17: The synchronous interaction between the system developer and the wrapper generator in identifying and specifying compositional wrappers _____ *235*
Figure 18: The synchronous interactions between the wrapper generator and search agents in composing compositional wrappers _____ *236*
Figure 19: Asynchronous interactions between the system developer and the wrapper generator in identifying and specifying compositional wrappers _____ *239*
Figure 20: Asynchronous interactions between the the wrapper generator and search agents in composing compositional wrappers _____ *240*
Figure 21: A compositional wrapper generated by the wrapper generator. _____ *392*
Figure 22: The scientific method _____ *462*

LIST OF TABLES

Table 1: Sizes and ranges of non-numerical primitive data types defined by this specification language. _____ *84*
*Table 2: High-level breakdown of the activities involved in developing and operating a compositional wrappers-based system*_____ *225*
Table 3: Level automation possible during system design. _____ *245*
*Table 4: Detailed breakdown of the activities involved in collecting components.*___ *247*
*Table 5: Activities breakdown of the preliminary evaluation and candidate selection stage.*_____ *278*
Table 6: Detailed activities breakdown of all the activities related to the detailed evaluation process. _____ *284*
Table 7: Global variables and loop counters used to refer to the different activities in the wrapper generation process. _____ *364*
Table 8: Calculate the number of additional components that need to be obtained and evaluated to satisfy all of the interfaces required by the wrapper specification or any of the selected best-fit components. _____ *364*
Table 9: A detailed breakdown of the activities involved in components acquisition and adaptation. _____ *368*
Table 10: The detailed activities involved in the process of components' wrapping. _ *387*
Table 11: The system-wide factors that affect the glue code size estimation. _____ *401*
Table 12: Estimating the number of component updates throughout the expected life of a component. _____ *403*
Table 13: The component-specific factors that affect glue code size estimation. ____ *404*
Table 14: Estimating glue-code size of three different compositional wrappers. ____ *412*
*Table 15: Detailed activities breakdown of some of the tasks related to compositional wrappers that take place after the of the composition process.*_____ *447*

LIST OF ABBREVIATIONS

ADL	Architecture Description Language or Assertion Definition Language
AI	Artificial Intelligence
API	Application Program Interface
BNF	Backus Naur Form
BWCS	Basic-Wrapper Code Size
CDL	Component Definition Language or Contract Description Language
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off The Shelf
CSF	Component-Specific Factor
CSL	The Component Specification Language (see Chapter 7)
CSP	Component Service Provider
CWB	Compositional Wrappers-Based
DCOM	Distributed Component Object Model
EADL	Executable Architecture Definition Language
GCEF	Glue Code Expansion Factor
GOTS	Government Off The Shelf
GUI	Graphical User Interface
IDL	Interface Definition Language
ISBN	International Standard Book Number
LAN	Local Area Network
MOTS	Military-off-the-Shelf
MRE	Multi-Relation Encapsulation
OLE	Object Linking and Embedding
OMG	Object Management Group
OS	Operating System
OTS	Of The Shelf
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SF	Security Feature
SLOC	Source Lines Of Code
SWF	System-Wide Factor
WGS	Wrapper Glue-code Size
WSL	The Wrapper Specification Language (see Chapter 8)

CHAPTER 1: INTRODUCTION

1.1 Problem Statement

Software application development today is under considerable pressure. From the business perspective, it has to support the corporate world with its established and stable paradigms. Users expect to see everything available all the time, everywhere, and to everyone, through the World Wide Web. In addition, the software applications must provide the integrity and security required by the critical business processes. Talented software developers are scarce, and competitors are constantly looking for them. Most software professionals are spending a considerable amount of time 'to wrap, rewrap, and wrap again some "legacy[1]" code bequeathed to us by our predecessors' [MEYE01].

Software component integrators face various challenges. The first challenge is to decompose the design of the proposed system into manageable components. Component integrators must then specify each of the identified design component. The next task is to search for and locate pre-built candidate components that are hoped to match the specification of needed components. Integrators must then evaluate the candidate components and select the components that best fit the requirements of the developed system. The next challenge for system integrators is to evaluate the selected components and acquire

[1] One definition of "legacy" is 'a system which is no longer sufficiently flexible to be adapted in a cost-effective way' [BACH00].

the ones that pass the evaluation criteria. The last challenge is to configure the acquired components and to plug them into the developed system.

Situations often occur when component integrators require a component with a set of characteristics that cannot be satisfied by a single pre-built component. When this occurs, component integrators must either "loosen" the requirements of the sought-for component or acquire a set of components for which the union of such components' characteristics satisfies the characteristics of the target component. Sometimes, component integrators might not be willing to modify their requirements or to decompose the specification of the sought-for component into multiple ones with less-stringent constraints. The organization's policy could also require systems with a specific granularity level for their components. In addition, component integrators might not be able or not be willing to put the effort in developing the needed components because of lack of time, budget, or the expertise to carry out the component development.

Thus, there is a need for improved methods of components-based software development. The goal of this research is to provide mechanisms to support a solution to these problems. In particular, the research addresses the issues that are encountered when component integrators attempt to acquire a set of components and glue them together to form a new component or system. The list of issues include:

- The lack of simple and easy to use formal methods that component integrators can use to precisely describe the characteristics and behaviors of software components and target systems.

- The difficulty in finding a single off-the-shelf component that can satisfy a set of the component integrator's functional and non-functional requirements. Such difficulty could force component integrators to "loosen" their requirements, to split their specifications into multiple ones, to modify their standards and/or business rules, or to develop the required components from scratch.
- The lack of available tools that can automate the parsing of formal specifications and the utilization of their constructs in assembling and generating systems from off-the-shelf components.
- The high cost, long schedule, and intensive effort that are often required to assemble software systems from off-the-shelf components.

In addition to the above issues, this research addresses and attempts to solve many of the challenges faced by component integrators when attempting to build higher-order (i.e., composite) components from individual components. Examples of such challenges include the wiring of component interfaces, the selection of best-performing interfaces from a set of compatible ones, the tracking of component activities, the automatic updating of components, and the management of load-balancing and emergency-recovery situations.

1.2 Proposed Solution

This dissertation summarizes the current state of the research and practice with respect to smart compositional wrappers[2], and describes and demonstrates new methods to support the specification and development of more intelligent wrappers. The work also includes a high-level investigation of other issues related to the composition process that is used to assemble and generate compositional wrappers. Such issues include the automation of the various activities of the composition process and the analysis of the performance, cost, and upgrade and maintenance efforts of acquired components and generated compositional wrappers.

The following are the key elements that form the approach taken by this research:

- Describe and build a framework that addresses the concept of "smart compositional wrappers." The theoretical and technical issues related to compositional wrappers will be clearly defined. These issues include: the advantages and disadvantages of compositional wrappers; the relationship of compositional wrappers to composite components or system of systems [MAIE98, BOEH00]; and the conceptual and practical meaning of smart features and their applicability to compositional wrappers.

[2] A compositional wrapper is a sophisticated software encasing of one or more components. The compositional wrapper is dealt with as any other component. The word "compositional" is chosen to reflect the fact that the wrapper does not encase single components (as is addressed by some researchers), but collections of components. The notion of composition also implies that there is a process for composing the components into one wrapper. The word "smart" is used to imply the intelligent behavior that compositional wrappers could exhibit because of their ability to perform the smart features that will be discussed in Section 8.2.9. The literature is filled with articles that use the term "smart wrappers." However, wrappers in such literature accounts are shell scripts or source code fragments that are developed to wrap only single components. The concept of "supercomponent" proposed by [ORFA96] wraps single objects that can roam across machines. The supercomponent includes many smart properties that allow the objects to live in networks. The concept of "megacomponent" is used to refer to collections of frameworks that can be specialized to work with certain devices, such as video tape recorders, refrigerators, etc.

- Develop a Component Specification Language (CSL) that is suitable for describing the syntax and behavior of software components, as well as other attributes required by the tools that will parse such specifications. CSL will allow component suppliers to specify their components for use by available component integration tools.

- Develop a Wrapper Specification Language (WSL) that can be used to describe the desired characteristics of compositional wrappers. WSL will contain needed constructs to support available software tools in the assembly and generation of compositional wrappers.

- Formalize a composition process that can be used to build compositional wrappers from pre-built components. The process should include all the required stages to build and generate compositional wrappers. Such stages could include developing a WSL specification to describe the characteristics of target compositional wrappers and their encased components; searching for components whose specifications must conform to CSL and whose characteristics match those of the WSL specification; evaluating found components; selecting, acquiring, and configuring candidate components; and selecting and wrapping the best-fit components along with any desired smart features.

- Discuss the design and implementation of a wrapper generator that must use the composition process to guide it in assembling and generating compositional wrappers. The wrapper generator must have built-in capabilities to parse and interpret

CSL and WSL specifications, utilize available search and middleware agents, to support user-defined code templates, and to control the wrapper generation process. The wrapper generator should be able to instantiate code templates to accomplish the various compositional wrapper tasks.

1.3 Contributions

This dissertation provides an analysis of the potential effects of using compositional wrappers. In addition, the two new formal specification languages (CSL and WSL) can be used to describe the characteristics of software components and compositional wrappers. The prototype that is described in the Appendix and the many algorithms, code snippets, and example specifications that can be found throughout this dissertation demonstrate the simplicity and usefulness of CSL and WSL. Users of CSL and WSL are no longer required to use the lengthy and cryptic constructs found in many of the traditional formal specification languages. The two languages offer many powerful constructs to allow component suppliers and system integrators to capture the syntactic and semantic behaviors of components. Examples of the powerful constructs in CSL include the ability to formally describe components' functional (e.g., inputs, outputs, behavior, and stimuli) and non-functional (e.g., costs, environment settings, and installation and configuration options) properties. Examples of such constructs in WSL include the ability to specify compositional wrappers' operating constraints (e.g., software, hardware,

and network constraints), the inclusion of smart features, controlling the wrapping and generation process, and the wrapper's architectural specification.

The following bullets list specific benefits of using compositional wrappers. The list also includes the application of the two new specification languages and the composition process, and the utility of wrapper generators.

- The use of compositional wrappers can reduce the costs and efforts of developing and operating Compositional Wrappers-Based (CWB) systems by reducing the number of their building blocks (see Section 3.1.1 for a thorough analysis of this topic.) Such reduction can have great impact on development organizations' market credibility since their systems are more likely to be delivered on-schedule and within budget. In addition, the systems may be more robust and reliable than those developed using only individual components.

- The use of compositional wrappers can increase the number of components that one can integrate in one system by up to six times the limit suggested by [GARL95a] and [ABTS00b]. [GARL95a] suggested that four components are the limit to have a manageable integrated system. [ABTS00b] raised that limit to six. Taking their suggestions by allowing each compositional wrapper to encase up to six components and by limiting the CWB system to integrate a maximum of six compositional wrappers, results in multiplication of potential components usage by a factor of six. Such increase in the number of components that can be encased in one system will help build more complex and sophisticated systems. (Note that the increase in the total number of encased components in a system does not imply

a corresponding increase in the costs and efforts of building and operating such systems. This is because the components that are encased within compositional wrappers are managed by the wrappers themselves.)

- The use of compositional wrappers can reduce the legal obligations of an organization. With fewer components (mostly compositional wrappers) in a CWB system, an organization will have fewer licenses to acquire and fewer agreements to abide by. In addition, increased robustness and reliability of the developed systems can reduce potential legal consequences by clients.

- The use of compositional wrappers can reduce the overall cost of CWB systems. Compositional wrappers can cost less than the cumulative cost of the components they encase. The return-on-investment analysis presented in Chapter 16 shows that, for each generated compositional wrapper, development organizations can save at least $100,000 of the development costs alone. The expected automation of activities in other areas of the composition process can further reduce the overall cost of CWB systems.

- The use of compositional wrappers can enforce the life cycle model adopted by an organization. Without compositional wrappers, the functionality provided by off-the-shelf components can force an organization to use specific life cycle models. This is because in normal components-based development, system requirements can only be determined after knowing what functionality is available through acquired components. Using compositional wrappers, as many components as nec-

essary can be wrapped to deliver the functionality required to match the requirement of the used life cycle model. Compositional wrappers are responsible for stripping off any extra component functionality that is not consistent with the user's requirements. Additionally, compositional wrappers can be configured to account for any industrial or organizational standards adopted by the organization.

- The use of compositional wrappers can shift the development paradigm from being user-centric to become vendor-centric. Integrators' role will focus on the identification, search, selection, acquisition, and plugging of components. Vendors' role will concentrate on specifying, designing, building, and dispatching compositional wrappers.
- The use of compositional wrappers can support the enhancement of effort, cost, quality, and reliability estimation models to account for the new development paradigm presented in this dissertation.
- It is expected that the introduction of the two new specification languages will widen the research spectrum of formal specification languages and can motivate researchers to come up with new or enhanced constructs that can address many of the issues faced by system integrators and composite systems.

For many years, the software engineering community has focused research on enhancing life-cycle models to produce better software through traditional development paradigms. Recently, the focus shifted to building systems from components. Compo-

nents may be off-the-shelf (OTS), reengineered, reused, or newly developed. Using pre-built components and compositional wrappers will further enhance the composition process by supporting new techniques for composing components. Such enhancements could result in new client-vendor models and new, more intelligent component and wrapper features.

It is hoped that the research of this dissertation will also stimulate researchers in the area of composition processes and in other related areas, such as wrapper composition and generation languages, code generation problems and techniques, wrapper interaction facilities, and wrapper development and usage environments. Other research areas that may benefit from this work could include code optimization techniques, distributed wrapper generation, wrapper service providers, and software agents dedicated for composing and generating wrappers.

1.4 Relationship to Other Research

Figure 1 illustrates the relationship of this work to related work of other researchers. The inner circle is the goal of this research. The middle circle contains approaches similar to the research reported here. The outer circle contains useful, but not very related, techniques. The outer box contains approaches that are not related to this work.

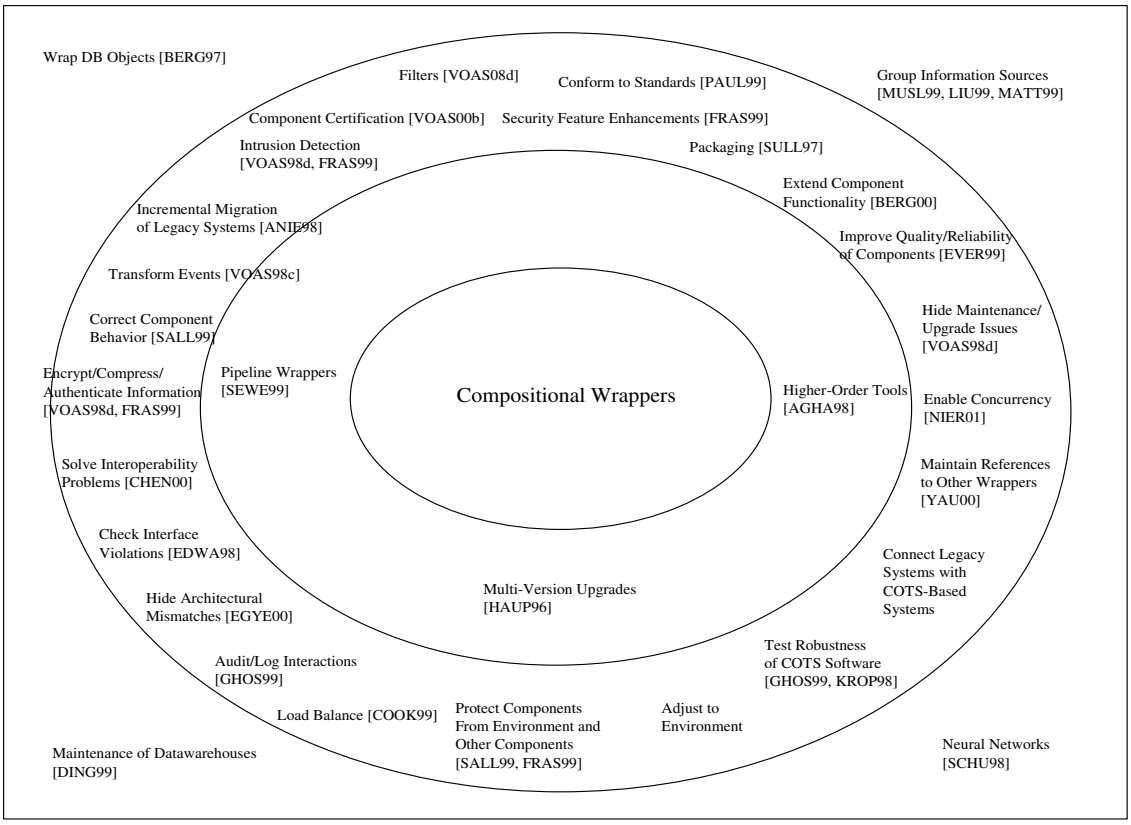

Figure 1: Relationship to Other Research

The research of this dissertation built on previous work by other researchers and introduced new concepts to yield a more complete solution for assembling and auto-generating component aggregates, rather than being limited to polishing individual components. Specifically, this work was influenced by the concept of building higher-order tools from complete applications that was proposed by [AGHA98] and the concept of system of systems discussed by [MAIE98, BOEH00]. This work allows a compositional wrapper to encase pre-built components of any size and complexity. Many of the concepts presented by [ACHE00, BATO96, CANA97, KLOU00, MICH00, REID00, SEWE00, SUCC00] and by other researchers were also of great benefit in formulizing

the notion of compositional wrappers. The work on smart features presented in this dissertation benefited from the concepts and propositions offered by many researchers (for example, ORFA96, GAO01, HAUP96, and RAKI00.) The work on specifications and specification languages in this dissertation was influenced by the concepts that were presented by many researchers (for example, CHAN99a, MEYE85, PARI96, PENI97b, and ZARE97.)

1.5 Reader's Guide

The remainder of this document is organized as follows:

- Chapter 2 through Chapter 4 introduce basic terminology and concepts about software components and wrappers. The following information is provided: basic concepts from the software engineering literature (Section 2.1), software components (Section 2.2), wrappers (Section 2.3), compositional wrappers (Chapter 3), and wrapper generators (Chapter 4).
- Chapter 5 to Chapter 8 discuss specification languages to support compositional wrappers. Chapter 5 introduces specification languages in general and presents the criteria used in this dissertation to form the WSL and CSL specification languages. Chapter 5 also introduces the constructs that form the basis for many other constructs in these two languages. Chapter 6 presents the constructs and sections

that are common to both of WSL and CSL. Chapter 7 and Chapter 8 present the details of CSL and WSL, respectively.

- Chapter 9 to Chapter 14 outline the various stages in the proposed composition process. After the general process is introduced in Chapter 9, each stage of the process is discussed in detail. For example, Chapter 10 presents the translation of user specifications to search requests and discusses how components that satisfy such requests can be found. Chapter 11 discusses the issues of preliminary evaluation of the components that resulted from the search process. That chapter also discusses the selection of candidate components. Chapter 12 discusses issues related to detailed evaluations that are needed to select best-fit components from the candidate components. Evaluations of component groups are also discussed in this chapter. Chapter 13 discusses the issues of component acquisition and adaptation. Chapter 14 discusses the wrapper generation and testing issues.

- Chapter 15 highlights some of the issues related to compositional wrappers that do not fit within the composition process. These issues include the assembling of compositional wrappers into complete systems; the testing of CWB systems; and the upgrade and maintenance issues related to compositional wrappers and CWB systems. The chapter also discusses the activities and events that could occur during the operation of CWB systems.

- Chapter 16 presents a brief return-on-investment analysis for compositional wrappers. The chapter provides discussions on potential cost and effort reductions that

can result from using wrapper generators to generate compositional wrappers. The emphasis is on the potential levels of automation of the various activities in the composition process by the wrapper generator.

- Chapter 17 presents summaries and conclusions.
- The Appendix lists and explains the content of the enclosed CD.

CHAPTER 2: BACKGROUND

Early research in software components (see, for example, BAIL91, STER96, and SCHN99) focused on reuse and reengineering issues. In the early 1990s, research focused on development issues related to components (see, for example, BATO92, BRAC92, ELLI95, and SEGA93.) Marketing and legal issues have been addressed in a few recent papers and workshop (see, for example, CHAV98, BASI01, and GUNT00.)

The idea of wrapping components has been an area of interest to researchers in the last six to seven years. Many of these researchers have focused on security issues [FRAS99, VOAS98d, KHAN00, SALL99]. Other researchers used wrappers to extend component functionality, to augment or transform the information flowing in and out of components, and to test the reliability and robustness of software components that wrappers are developed to encase [GOSH99, KROP98, BERG00, VOAS98c].

One area that requires additional research is component composition. Early research on software composition focused on building architectures and frameworks to guide the composition process [GARL95b, ALLE92, INVE95]. Building composite component structures is a recent topic and still even lacks a clear definition of its meaning and purpose [ACHE99, BATO96, CANA97, CARN00, MICH00]. This section introduces basic terminologies and concepts about software components and wrappers.

2.1 Component-Based Software Engineering (CBSE)

CBSE has significantly different characteristics from traditional, custom system development [SEAC99]. For example, system architectures and frameworks take an additional importance in a CBSE process. CBSE is concerned with the rapid assembly of systems from components [BACH00].

CBSE tries to reduce the complexity of today's feature-packed software applications by enabling the creation of interchangeable components of software functionality. While object-oriented technology was a big step in shaping a new paradigm of software development, objects are too fine-grained and their deployment requires too much understanding of an object's working [ASAI00]. The component-based approach, on the other hand, encourages the reuse of existing assets and reshaping of the development process. The goals are to cut costs, shrink development schedules, minimize software defects, improve the productivity of software teams, and enhance the quality of software products.

Components are used as the building blocks of software applications without regards of their origin. Components could come from various sources, such as existing programs (reused or reengineered components), templates (generated components), commercial vendors (COTS components), or as new development. Published reports [JACO97] have demonstrated that the use of components can provide software that is more usable, more flexible, and available sooner.

It has been estimated that by 2003, 70% of new applications will be deployed as a combination of pre-assembled and newly created components integrated to form complex business systems [BROW98a], and that the worldwide market for component construction and assembly tools will be in excess of $8 billion, with an additional $2 billion spent on acquiring components [BACH00]. [HEIN01] quoted a prediction by PriceWaterhouseCoopers which says that 'the open market for software components will be worth $1 Billion US dollars by 2002.' Unfortunately, as [BACH00] noted, many of these predictions are tainted by a lack of agreement about what software components are, and how they are used to design, develop and field new systems.

2.2 Software Components

The term *component* indicates that one artifact (the component) is part of another artifact (the developed system or another component.) The term *software component* is interpreted differently depending on the level of abstraction considered. Advocates of software reuse equate components to anything that can be reused; practitioners using commercial-off-the-shelf (COTS) software equate components to COTS products; software methodologists equate components with units of project and configuration management; and software architects equate components with design abstractions [BACH00]. The following is the definition of a software component that is used in this dissertation:

> *A component is one or more lines of code acting as a unit, in source or object form written in a compiled or interpreted language, possibly with its*

own artifacts, capable of performing one or more tasks as part of the execution of a bigger task through a controller program.

From the plethora of literature accounts that talk about software components (see for example, ACHE00, BAIL91, BATO96, ERIK00, HALL00, HOEK98, KOZA00, and SZYP00a) one can extract many properties of software components. For example, a component consists of many functional and declarative details to deliver a specific business function or technical task. A component has well defined behavior delivered through its interfaces**.** Interfaces are the only mean by which components communicate with the outside world. Components' behavior is constrained using preconditions and post-conditions. Preconditions must be satisfied by any system that uses the component. If the precondition is true when the component begins executing, the component is guaranteed to terminate in a state where the post-condition is true [PENI97b]. There are no restrictions or guarantees on the behavior of the component when the precondition does not hold. Post-conditions describe what is known to be true because of an interface invocation. Interfaces allow the component to be used in multiple combinations and to interact with other systems or components. Components are interoperable and multiple instances of the same component can exist at the same time. A component can also exist in multiple versions and variations and can be dynamically placed or replaced in a system.

A component is a convenient way to package object implementations and to make them available for assembly into a larger software system. Figure 2 illustrates a component as one or more object implementations within the context of a component model.

The component model defines the set of rules that must be followed by the component to make those object implementations accessible to others. Furthermore, the model describes a set of standard services that can be assumed by components and assemblers of component-based systems (e.g., naming of components and component interfaces, security of access to those interfaces, transaction management, and so on) [SZYP98].

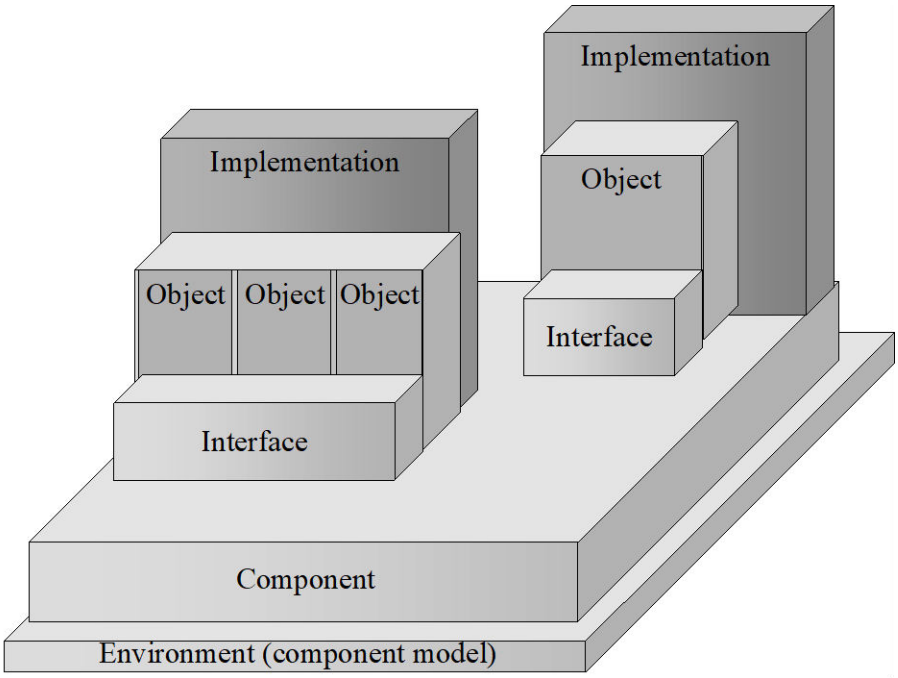

Figure 2: The relationship between components and objects

2.2.1 Architectures and Frameworks

For the various software components to coexist and cooperate in a component-based application, a well-defined architecture is required. The term software *architecture*

has been recently adopted referring to the level of software design in which the system is represented as a collection of computational and data elements, or components, interconnected in certain way [CANA97]. Architectural Description Languages (ADLs) [CANA97] appeared in the 90's to address the need for expressive notation in architectural design. The goal of an ADL is to provide precise descriptions of the *glue* for combining components into larger systems. The basic entities in architecture description languages are components, interfaces, and connectors [HOA98].

Software architectures involve (1) descriptions of the elements from which systems are built, (2) interactions of those elements, (3) patterns that guide their composition, and (4) constraints on these patterns [RAKI00]. [GARL95b] indicated that principled use of software architecture can have positive impact on at least five aspects of software development:

- Comprehending large systems by presenting them at a level of abstraction at which high-level design can be understood;
- Supporting reuse of large components and frameworks into which components can be integrated;
- Exposing the dimensions along which a system is expected to evolve in terms of performance, interoperability, prototyping, and reuse;
- Providing new opportunities for analysis, including high-level forms of system consistency checking, conformance to an architectural style, conformance to quality attributes, and analysis of domain-specific features; and

- Managing the development of new software through a new paradigm that focuses on the capability, anticipated growth, and the feasibility of the developed system.

Ensuring the consistent handling of system characteristics across all components is one of the important roles of component architectures. Designers must remove sources of misfit among components [THOM00b]. This misfit occurs on two levels: misfit between the user's needs and the available components, and misfit among components that cannot be seamlessly integrated. [GACE98a] suggested the use of conceptual features to detect architectural mismatches. Some of the other approaches that may be used for mismatch risk minimization include [EGYE00]: using a different set of components with differing characteristics to support same required functionality; changing the means of component interaction (for example, using connectors [MEHT99, VIGD98]); varying the features of the components to be built; and introducing wrappers, bridges [GARL95a], or adapters [ALLE01].

One way to avoid mismatches in composite components is by using frameworks [SZYP00b]. A component framework is a software entity that supports components conforming to certain standards and allows instances of these components to be “plugged” into the component framework. An important role of a framework is its regulation of the interactions that the parts of the framework can engage in [SZYP98]. Examples of component frameworks include Microsoft's VisualBasic [SZYP00b], OpenDoc [JACO97], WaterBeans [PLAK99], and BlackBox [OBER02a] for visual components and JBed [OBER02b] for real-time control applications.

Component frameworks essentially define *architectural styles* in that applications built using the same framework will exhibit similar architectural structure and use the same kinds of collaborations among components [NIER95b]. The style describes how the components, connectors, and interconnection topology are organized into a system that provides the required services. Examples of traditional architectural styles include pipe and filter, real-time data feeds, event-driven architecture, layered, blackboard, client-server, distributed processes, main-subroutine, multithreaded, software bus, closed-loop feedback control, logic programming, real-time, rule-based, transactional database, and Internet distributed entities. Other recent styles include GenVoca [BATO96], C2 [OREI98], ACME [GARL97], SADL [MORI97, DINC00], and Darwin [MAGE95].

2.2.2 Component Marketing and Distribution

Despite well-known component technology benefits, the software industry has been unable to mobilize an infrastructure of production for reusable software components [SILV00]. The software community appears to blame this on missing, insufficient, or incompatible interaction standards [GUNT00]. Analysts have attributed the relative lack of an open market for components to the tight coupling between delivery and revenue collection [SIMM93]. To be useful in practice, the revenue collection mechanism must be safe, flexible, efficient, and transparent to the user.

There are two principal arrangements possible between the producer and the consumer of a component. Pay-per-use allows the consumer to pay the producer for as much

of the producer's software as the consumer's software actually uses. Pay-per-user permits the consumer's software to use the producer's software for a fixed period in return for a fixed fee, independent of whether or how much the consumer's software actually uses the producer's software.

2.3 Software Wrappers

Software wrappers are a family of technologies that force components to behave in specified, desirable ways. Wrappers typically work by restricting a component's input or output information, both of which alter the component's functionality [VOAS98b]. Wrappers are sometimes injected as an additional software layer between the component-based applications and their operating system or between the applications themselves [FRAS99]. Wrappers are also used to audit/log the interactions that take place between the component and the operating system and between the component and other components.

Wrappers are used for various things, such as addressing incompatibility, security, and dependability problems [VOAS98d]. Protecting applications and operating systems was the reason that motivated researchers to consider wrappers in the first place. In this sense, a wrapper performs corrective actions to the behavior being produced by the wrapped component. Wrappers are also used in access control or intrusion detection by identifying data generated by the component that is known to cause harm [FRAS99]. If the component carries illegal actions, then a wrapper can place the system in a safe state

instead of letting it crashes [SALL99]. With the same token, a wrapper can protect components from the hosting application that tries to hack into their internal structures.

On the testing side, wrappers can be used to test the robustness of software components [GHOS99]. Fault injection [KROP98, VOAS98b] is used to test software components by artificially inducing a fault and observing the system's response. Wrappers are used to check and correct interface-mismatches [EDWA98] or to solve interoperability problems [CHEN00]. The wrapper will have the same interface as the component it encases and delegates the work involved in carrying out each operation to the component itself. The interface checks can be turned on or off at runtime.

From the development perspective, wrappers can glue together known services, or can add new services while leaving others undisturbed. In this situation, wrappers can reduce the set of required services by the wrapped components as some of these services are provided by other components in the composition [AGHA98]. Wrappers are also used to extend components functionality, as will be discussed in Chapter 9 to Chapter 14. Wrappers are also suitable for encasing components generated from design patterns [YAU00]. The wrapper's responsibility is to maintain references to other wrappers and the interactions with their components.

A wrapper can also be used to encase more than one instance of the same component version or multiple versions of the same component [RAKI00, COOK99, MOSE98]. If any component within that wrapper fails, the wrapper switches operation to another instance. This mechanism ensures continuity of operation. Similarly, if a component falls below a performance threshold, the wrapper can invoke another instance of the same

component and delegate some of the first component services to the second component. However, [VOAS97] pointed out that, unlike hardware, duplicating components does not guarantee a better operation.

Wrappers are sometimes used as interfaces to connect legacy systems to systems built using software components that use new technologies and protocols [AGHA98, ANIE98]. Other, non-common uses of wrappers include conformance to standards [THOM00b, FLYN98, PAUL99, SCHU98], wrapping web documents [LIU99, MATT99, MUSL99], certifying components [HEIN01, VOAS00b], and interfacing components with middleware technologies [PENI00, KLOU00].

CHAPTER 3: COMPOSITIONAL WRAPPERS

Composition is a technique for specifying a large system by combining the specifications of smaller, simpler pieces – the system components [OLAW00]. The real benefit of composable software systems lies in their increased flexibility. It becomes easy to recompose such systems to address new requirements [NIER95a]. The main theme of this dissertation is to introduce the notion of *compositional wrappers*. Compositional wrappers are higher-order components built from pre-built components. Compositional wrappers allow a set of components to interact with each other and with the outside world through the wrapper's interfaces.

A compositional wrapper provides its users with a mechanism for performing smart actions through its built-in *smart features*. Smart features are a set of utilities built in the compositional wrapper and perform intelligent actions. Without such smart features, the wrapper users will have to perform such actions manually. For example, the built-in smart features can protect the compositional wrapper users from the encased components' undocumented or unwanted functionality. Smart features can also filter, augment, or transform inputs to and outputs of the wrapper components to fit the user requirements. The smart features built-into compositional wrappers will be responsible for selecting the best interfaces from the union of provided interfaces of the composition-

al wrapper's encased components (see Section 14.5) to export them as provided interfaces of the compositional wrappers. The built-in smart features will also be responsible for matching the required and provided interfaces of the selected components (see Section 14.6.) The smart features are also responsible for performing maintenance and upgrade of encased components.

Literature accounts that discuss properties, such as these smart features, give the components with such built-in properties "higher-level" names to reflect their nature. For example, [ORFA96] call them "supercomponents" while [SUCC01] refer to them as "packages". Other researchers use terms, such as "megablocks" and "megacomponents", to refer to these feature-packed components. [ORFA96], for example, define a supercomponent as:

> *'A "supercomponent" adds support for: security, licensing versioning, life-cycle management, support for visual assembly, event notification, configuration and property management, scripting, metadata and introspection, transaction control and locking, persistence, relationships, ease of use, self-testing, semantic messaging, and self-installation.'*

In contrast to the work of other researchers (for example, PENI97b and MART02), the work presented in this dissertation assumes that component integrators will acquire compositional wrappers as single components. Since compositional wrap-

pers are higher-order building blocks, the developed system would have smaller number of components if compositional wrappers were used instead of individual components.

3.1 The Need for Compositional Wrappers

The goals of using compositional wrappers to integrate components are as follows:

- Minimize the cost of the overall system.
- Minimize the effort required to acquire, customize, and plug the components in the developed system.
- Minimize the number of components in the system. The number of components is directly related to the complexity, cost, maintenance, performance, and reliability of the finished system.
- Minimize the security breaches that could result from plugging foreign components in new environments and from the interactions of such components. As stated in [KHAN00], "the security properties of a component in isolation may differ from the compositional impact of the same component with an application system."

The following subsections present some of the benefits and justifications of using compositional wrappers instead of individual components.

3.1.1 Coding and Integration Effort

From the system integrators' point of view, the process of acquiring a compositional wrapper is similar to that of acquiring individual components. However, the effort spent on acquiring and gluing together individual components will multiply as the number of components increases. After studying thirteen COTS-based products, [ABTS00b] found that, for each of three of the thirteen products, the size of the glue code was between 25K to 50K lines of code. For another two, the glue code was between 100K and 500K lines of code.

These results provide support for an argument that compositional wrappers require extra effort in writing the wrapper code and making sure the encased components work as intended. This argument would be valid if component integrators would write the wrapper code. However, this argument is not valid for compositional wrappers that are provided as ready-made components by using wrapper generators. The wrapper generator described in this dissertation is a sophisticated tool that generates compositional wrappers from pre-built components whose specifications must conform to the Components Specification Language discussed in Chapter 7. The composition process, as well as the characteristics of the generated compositional wrappers, is controlled using the Wrappers Specification Language discussed in Chapter 8.

A system built from individual components requires glue code to wire the components together. The size of this glue code, and hence the cost of developing it, would be

much higher than the glue code size of the compositional wrapper because of the number of components involved.

Another merit of using compositional wrappers instead of individual components becomes apparent when one considers the *required* and *provided* interfaces of individual components. When two or more components have similar provided interfaces, the component integrator could have a difficult task of selecting their "best" interface. In this case, voting schemes [COOK99] have been used to decide which output is correct. Similarly, the component integrator will need to make sure that all required interfaces by the individual components are provided by other components within the developed system. While dealing with complete compositional wrappers might not completely solve this problem, it would definitely reduce it.

To justify a claim that the use of compositional wrappers can reduce the number of steps (and hence the effort), requires understanding of the process used by component integrators to build their systems. Figure 3 illustrates the basic development process when systems are built entirely from separate components. In contrast, Figure 4 illustrates a development process in which systems are integrated using compositional wrappers and wrapper generators.

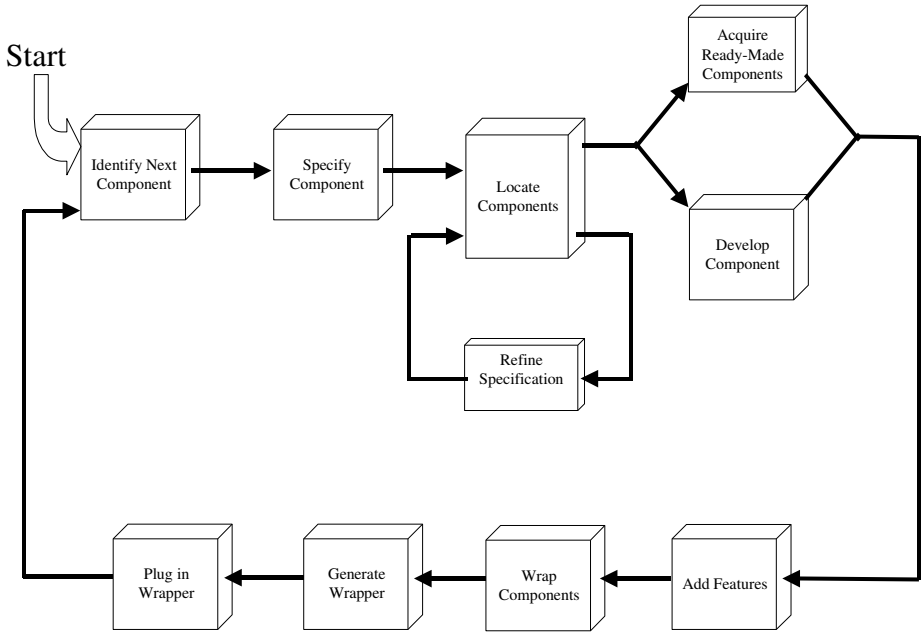

Figure 3: The development cycle that component integrators go through when building systems entirely from single components

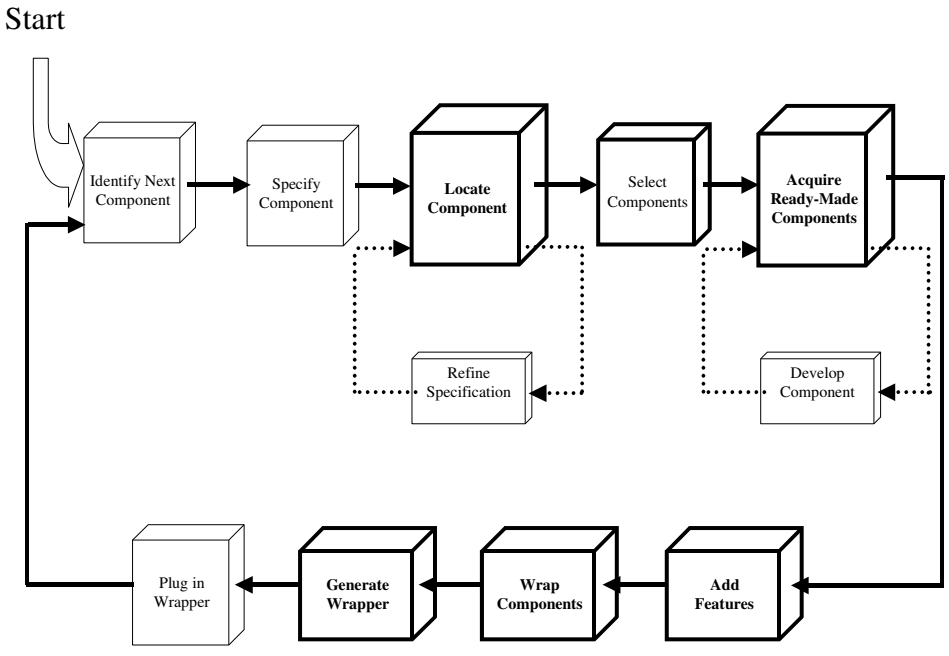

Figure 4: The development cycle that component integrators go through when building systems using compositional wrappers and wrapper generators

The boxes with darker boundaries in Figure 4 represent steps that are carried out by the wrapper generator. Dotted lines represent pathways to steps needed only when the wrapper generator cannot find ready-made components. The process shown in Figure 4 is not always linear. Most of the process steps require going back to the specification and maybe doing some fine-tuning to it. Such fine-tuning of the specification could result in components with lower quality, less-stringent constraints, or fewer delivered functions. The dotted lines to the new component development activity hint that developing new components would rarely be needed. This can be true in two cases. The first case is when there exists a component for every possible user requirement. The second case is when the system integrator prefers to refine the specification to avoid such new development. This second case could be the only feasible solution for organizations that do not have the infrastructure for new developments. However, the dotted lines in Figure 4 assume that specification refinement would also be rarely needed when using a wrapper generator. If one assumes the existence of an infinite number of components, this assumption is justified. The idea behind such justification is that for every user requirement, a component can be found that satisfies such requirement. This, of course, is not always the case. Component repositories are limited in the number of delivered components and addressed domains. Moreover, users cannot change the functionality of third-party components. Therefore, users are sometimes forced to modify their requirements by modifying the specifications of the sought-for components.

The following observations can be drawn from Figure 3 and Figure 4:

- In both processes, the system integrator has to identify and specify each needed component. When using single components (other than compositional wrappers), the specified components have to be fine- to medium-grained. Otherwise, the integrator could go through many cycles of specification refinement before being able to locate a component with the specified granularity. This situation is expected to occur less often when using compositional wrappers, because the wrapper generator will search for multiple components that, as a whole, match the specification.
- In the process described by Figure 3, the development team could be forced to develop components from scratch. This situation occurs when the integrator refines the specification to a level where new development becomes more practical than acquiring and managing many very fine-grained components. The development team could also be forced to develop the component from scratch if no single component is found to satisfy the complete set of requirements. While this situation could also occur with compositional wrappers, it would be less likely to occur, since, with compositional wrappers, multiple components would collaborate to deliver the required functionality.
- The steps carried out by the wrapper generator, illustrated by the dark boundary boxes in Figure 4, can save the development team a lot of effort in acquiring each component and in wrapping the component to account for security and standardization requirements. However, the development team would still need to select

single components or groups of components to be acquired by the wrapper generator. This selection process consists of nothing more than reading the details of component specifications and, based on the requirements, making a decision to include or reject each component.

- Each component comes with its set of provided interfaces and required interfaces. Using a wrapper generator, the wiring of these two types of interfaces can be done automatically, requiring no effort from the system integrator. However, with the absence of a wrapper generator, the development team must develop all the necessary wiring code and must be responsible for subsequent upgrading and maintenance of the wired components. This is not a simple task. Reports [MORI00, ABTS00d] show that glue code can skyrocket to hundreds of thousands or even millions of lines of code.
- After acquiring needed components, the development team might need to inject extra functionality or include some of the smart features (see Section 8.2.9.) Using a wrapper generator, this can be done when the system integrator specifies the wrapper. Without a wrapper generator, the development team would have to code each feature manually. Such a manual effort could take prohibitively long time. A further difficulty explained in Section 8.2.9 is that many smart features are not practical, or sometimes not applicable, for use in single components.
- The plugging of components (compositional wrappers, wrapped legacy systems, and newly developed components) phase occurs in both processes, as shown in Figure 3 and Figure 4. However, using compositional wrappers would require the

system integrator to plug in fewer components than when using single components.

Considering the potential areas of effort reduction in the development cycle, if one assumes that each compositional wrapper will encase at least two simpler components, one can justify a claim that using compositional wrappers can greatly lower the overall effort required to develop a component-based system. Obviously, the reduction increases when compositional wrappers are used to encase more than two components.

3.1.2 Security

Security has become a major concern both to component vendors and to system integrators. Vendors worry about someone hacking into their components to steal intellectual properties. Component integrators worry about vulnerabilities to attacks and hidden functionality that components might bring with them. Compositional wrappers can be used to partially address this problem. If vendors encrypted their components to minimize hacker attacks into them, the compositional wrapper could be responsible for decrypting each component before executing them. On the other hand, regardless of whether or not compositional wrappers are used, to protect the integrated system from attacks by the components would require that each component be wrapped and confined within its intended domain. However, wrapping few components (i.e., compositional wrappers) would require much less effort than wrapping individual components.

3.1.3 Network Bandwidth and Disk Access

Loading or downloading speed is of concern to component integrators whether components are accessed from their original sources (which could be anywhere in the Internet) or from copies stored on local disk drives. As the file size of a component increases, the loading or downloading time increases. One solution to this problem is to compress the components. However, unless the vendor sells compressed components, the component integrator cannot do much about compressing components before downloading them if components are downloaded from their original sources at runtime. If the components were loaded from local disks, then the integrator would have to compress the components before loading them into the system. The integrator would have to provide additional code to uncompress each component before the component can be invoked within the system.

Using compositional wrappers can support a vendor who wishes to compress components before making them available to clients. In this situation, it is the compositional wrapper's responsibility to uncompress the components before calling them. The component integrator would not need to worry about such problem. In this case, the total size of all of the compositional wrapper files and the files of its compressed components will be significantly less than the total size of all of the individual component files.

3.1.4 Upgrade and Maintenance

Upgrading one component (the compositional wrapper) should be a less-difficult task than upgrading a set of individual components. In addition, the potential problems induced by the upgrade process should eventually be less in the compositional wrapper case than when upgrading individual components. The same can be said about the maintenance of the compositional wrapper versus the maintenance of individual components. If the updates are delivered via insecure channels, updates could be modified in transit and system developers could be fooled into installing fraudulent updates [LIND98]. The probability of such problems occurring increases with the number of components downloaded.

Compositional wrappers can affect the upgrade and maintenance of their components. For example, though it is unlikely that the individual components encased within a compositional wrapper will need to be upgraded or maintained at the same time, a compositional wrapper might need to be upgraded (with possible recoding and recompilation) when one of its components is upgraded. Maintaining a compositional wrapper could be problematic since it might be difficult to pinpoint the individual component that is causing a problem. However, if the compositional wrapper implements component replacement techniques [COOK99], the compositional wrapper itself could take care of updating its components. In addition, the compositional wrapper could be responsible for detecting problems in its components and for solving them (see Section 14.1 for a discussion on encasing multiple versions or instances of the same component). Such a capability could relieve the wrapper users from keeping track of new updates and of the effort of recoding

and recompiling some parts of the system. The use of the auto-update smart feature (see Section 8.2.9.6) would further reduce the efforts required by the wrapper users during the upgrading process.

3.1.5 Performance

Performance of the developed system is affected by the size of the component. It takes less time to load or download small individual components than to load or download a larger component (i.e., the compositional wrapper.) Using compositional wrappers will typically yield better performance than using a set of small individual components since less swapping in and out of memory will be required (because the compositional wrapper is used as a single component.)

Performance is also affected if encryption and/or compression are used. It is possible that the compositional wrapper's code to decompress and/or decrypt encased components will be more optimized for performance than the code written by individual component integrators for the same task.

In addition, it is unusual for two or more components to supply the same interface in terms of functionality. However, the provided interfaces can differ from component to component in several attributes, such as quality, reliability, and speed. When such a situation exists, component integrators would be responsible for manual selection or for writing the code to select the "best" interface from the compatible ones. Performance degradation can occur when the system tries to select the best interface among the individual

components or when the system tries to match required interfaces with provided ones. This would not be an issue if compositional wrappers were used.

3.1.6 Injection of Smart Features

Today's systems are continuously increasing in complexity and in demand for automation. The smart features and the opportunities for their automation that are discussed in Section 8.2.9 can be used to counter such complexity. Because it can require great and careful effort to craft these features, including them in single components would typically not be justifiable. On the other hand, a wrapper generator can be equipped with the necessary logic to include such features in generated compositional wrappers.

3.1.7 Monetary Cost

The cost of a compositional wrapper might be greater than the cost of each of its individual components. However, the compositional wrapper is expected to cost less than the total cost of all of the individual components because of the expected savings in assembling and auto-generating the compositional wrapper. In addition, additional cost savings can occur when there are many individual components involved. In particular, there can be a lot more effort involved in finding, negotiating, evaluating, acquiring, and configuring each individual component than the effort required to do the same tasks for a

compositional wrapper. Therefore, choosing compositional wrappers over individual components can yield long-term savings for the organization.

3.1.8 Legal Issues

Commercial components come with legal restrictions (for example, copyright, licensing agreements, upgrade and maintenance agreements.) Building systems using fine-grained components can result in increased legal restrictions. While systems built entirely from compositional wrappers could face similar legal restrictions, the number of compositional wrappers in the developed system is less and, consequently, would result in fewer legal obligations.

3.2 Challenges in Using Compositional Wrappers

There are several challenges to be addressed in the use of compositional wrappers. As with any composite system, a compositional wrapper can be complex, incomplete, and unreliable. These problems can be in the wrapper software itself or inherent in one or more of its components. The potential for such problems increases when component suppliers do not supply reliability and quality measures of their components and/or when the documentation is not provided or is poorly written.

The first challenge to using compositional wrappers is to decide on the appropriate number of components to encase in a compositional wrapper. Too many or too few components in the wrapper can cause problems. Consider the attributes of COTS components: Beyond a certain point, an increase in the number of components in a system may actually reduce the system's overall economic life span rather than increase it [ABTS00a]. Anecdotal evidence [ABTS00c] suggests that generally, though not universally, the more software components a system includes, the shorter the economic life will be of that system. In fact, the more components a system uses, the more severe the consequences of the incompatibilities among its components can become. Though there are no practical data to indicate the correct number of components to encase, [BOEH99] suggested that six components are the maximum that should be integrated. However, the selection of such a maximum will depend on a variety of attributes, such as: the granularity of the encased components, the richness of their interfaces, the components' quality and reliability measures, and the vendor-client relations.

Another problem with using compositional wrappers is due to the nature of the individual components that they encase. For example, components may adhere to different, incompatible component technologies (such as CORBA, DCOM, or JavaBeans.) Components may also be implemented in various programming languages. Additionally, the granularity of the components may be too coarse or too fine for the needs of a given system. This problem has great impact on performance, security, etc.

Components may not provide the exact functionality required in an interface. This results either in abandoning the component and looking for another one, if available,

or in letting the compositional wrapper provide the missing functionality. Another problem that is inherent in components is that different components may assume different types of interaction within the system (e.g., procedure calls versus asynchronous event notifications.) Even worse is the situation when some components support only sequential processing while others are event-driven or multi-threaded.

Finally, compositional wrappers can suffer from *compositional mismatches* [SCHN99]. This can occur when using a component in an environment that does not match the component's assumptions about the structure of the integrated system. The nature of a compositional mismatch can be: i) incompatible assumptions about the architectural styles of the underlying component frameworks (i.e., architectural mismatch); ii) different data formats (i.e., interoperability mismatch); iii) different synchronization schemes; and many more. In such a situation, some kind of glue code is needed to overcome a compositional mismatch and to adapt the "foreign" component to the new environment.

CHAPTER 4: THE WRAPPER GENERATOR

This chapter will start by introducing wrapper generators. Then, it will present a proposed data structure to hold the specific information about each developed compositional wrapper. The various aspects of this data structure (i.e., its elements, its content, and its use) will be discussed throughout the remaining chapters of this dissertation.

The need to build complex systems with less effort drives a need for wrapper generators. However, only few examples of wrapper generators have been proposed in the software engineering literature. Typically, the generators described in the literature focus on wrapping source code modules and generating scripts to allow using such modules in other environments. One impediment to the emergence of widespread use of wrapper generators is the need for robust and easy to use specification languages for such generators.

Wrapper generators are domain-specific for a variety of reasons. There are many difficulties in writing a general-purpose wrapper generator, including the need to target the wrapper generator to deal with specific languages, technologies, and user needs; and the need to follow specific component standards. For example, if the wrapper generator is forced to use a specific vendor's API, then the wrapper generator has to match the language in which that API is written. (Otherwise, it will not be able to take advantage of

such API.) In addition, it is difficult to write a general-purpose wrapper generator that can capture all the aspects of the many component technologies that exist in the market today and the many others that are emerging. Such a generator would need ways to deal with specifying, searching, evaluating, acquiring, and composing components in every individual technology. [DOMI02] proposed the OVIBUS platform to allow interoperability among various component technologies. OVIBUS wraps components from one technology to make it compatible with the technology with which it wants to communicate. Such an approach can be integrated with wrapper generators to allow the generators to take advantage of the various available technologies.

The wrapper's GUI would have to address each organization's specific mechanism for specifying and handling requirements. The generator would need also to match the user's resource requirements and needs. Thus, it is difficult, if not impossible, to build a generator that can satisfy all organizations.

Some techniques are available to support working around these limitations. For example, the generator can use generic templates for its requirement specification, its presentation, and its communication with its environment. In this situation, the integrator's role would be to fill the templates with appropriate information for the integrated system's platform and environment. The templates can provide a standard that is understood both by the generator (or its developers) and the system integrators. By encoding domain-specific intelligence into the wrapper generator, the generated compositional wrappers can perform automatic self-customization and optimization.

According to [BATO97], there are at least three qualifications that must be satisfied for generator technologies to be applicable: the domain in which software is to be generated must be mature; generators are expensive to build and therefore their costs must be justifiable; it must be feasible, both technically and politically to build such wrapper generators. Of these three qualifications, political feasibility may be the most difficult, since many organizations are not yet prepared to automate the development of software that they are manually producing today.

A complete wrapper generator solution will consist of three parts: the system integrator's side, the middleware, and the vendor's side. On the system integrator's side, there are two entities: the wrapper generator and the system integrator. On the vendor's side, there are also two entities: the vendor and its server software. The middleware will consist of commercial or newly developed software that is outside the scope of this dissertation.

4.1 A Data Structure to Hold the Developed Compositional Wrappers' Information

Table 1 shows the data structure that will hold the specific information of each compositional wrapper developed using the wrapper generator discussed in this dissertation. The wrapper generator will store in its repository an instance of this structure for each developed compositional wrapper. Most of the information in this structure will come directly from the wrapper's specification. The wrapper generator will insert the remaining information in this structure as the composition process continues (as de-

scribed in Chapter 9 to Chapter 14.) The attribute names in Table 1, as well as the actual data that each attribute will hold, conform to the syntax of the two specification languages that will be presented in Chapter 7 and Chapter 8.

Table 1: The wrapper generator's internal structure that will hold the state of each developed compositional wrapper

Name A unique name (within the developed system) for the current compositional wrapper. The wrapper generator will use this name to reference the compositional wrapper in its screens and reports. The generator will get the value of this field from the *name* clause in the wrapper specification.
Description A brief narrative description of the compositional wrapper and its role in the system. The *description* clause in the wrapper specification will supply the content of this field.
Required_Interfaces The list of interfaces required by the compositional wrapper or by any of its components. The list will hold pointers to structures that describe the interfaces to be provided by acquired components. Not all interfaces provided by acquired components will be exported as provided interfaces of the compositional wrapper (see a discussion of this topic in the introduction of Chapter 14.) The generator will build the list of required interfaces from the interface specifications in the wrapper specification. Each element in ***Required_Interfaces*** will point to a structure that contains complete information about one required interface. The information stored for each required interface would include:

Name

Each required interface in a compositional wrapper would be identified by a unique name within the same wrapper. However, uniqueness of interface names is not guaranteed across applications or networks. Section 14.9 discusses a technique that can be used to guarantee uniqueness of such interface names within each compositional wrapper.

Specification

This attribute holds the semantic specification of the required interface. This specification is needed when searching for desired components. The wrapper generator will also use this *Specification* during the specification-matching process described in Section 12.5.11.

Inputs and *Outputs*

Each of these attributes is a list that will hold the description of the inputs and outputs expected by the compositional wrapper from its components. Each element in these lists will contain the following information:

- *Name*. The input/output name.
- *Type*. The data type of the input/output that should conform to the `<Type>` construct (see Section 6.1.2) in the specification languages.
- *Size*. The size of the passed or returned information according to the `<Size>` construct (see Section 6.1.2) in the specification languages.
- *Target*. The source from which to expect the input or the destination to which to send the output.
- *Format*. The expected format of valid inputs and outputs. It should conform to the `<ValidData>` construct (see Section 6.1.2) in the specification languages.

Constraints

Constraints are necessary in searching for and acquisition of components that must satisfy the specification of the developed compositional wrapper. Most of these constraints will come from the specification of the compositional wrapper. The following is the minimum set of constraints that will be needed throughout the composition process (described in Chapter 9 to Chapter 14.)

	max_components The maximum number of components that can be encased in the compositional wrapper. Since this attribute has no corresponding construct in the wrapper specification, a default value for it (for example, six components is suggested by [ABTS00b]) can be set by the system integrator.
	min_provided_interfaces_per_component The minimum number of provided interfaces (in *each* acquired component) that should match one or more of the interfaces specified in the compositional wrapper specification. This attribute has no corresponding construct in the wrapper specification language and, therefore, can default to 1.
	max_unmatched_interfaces_per_component The maximum number of extra (not matched) provided interfaces that *each* acquired component can have. This attribute has no corresponding construct in the wrapper specification language. Therefore, the wrapper generator implementation can use a very large number for this constraint. Alternatively, the wrapper generator developers can choose not to use this attribute in the matching process.
	min_total_provided_interfaces The minimum number of *all* provided interfaces (including matching and non-matching interfaces) in all acquired components. This attribute has no corresponding construct in the wrapper specification language. Therefore, a default value for it (for example, the number of interfaces defined in the wrapper specification) can be set by the system integrator.
	max_total_provided_interfaces This attribute limits the maximum number of *all* provided interfaces (including matching and non-matching interfaces) in *all* acquired components. However, since there is no corresponding construct in the wrapper specification language for this attribute, the wrapper generator would either not use it as a constraint in the matching process or would assume a very large number in its place.

	max_required_interfaces_per_component *max_total_required_interfaces* The *max_required_interfaces_per_component* attribute limits the number of interfaces that *each* acquired component can require from its environment. The *max_total_required_interfaces* restricts the maximum number of interfaces that *all* acquired components can require. Both of these attributes have no corresponding constructs in the wrapper specification language and will either be not used by the wrapper generator in the matching process or will be taken as very large numbers.
	max_license_cost *max_upgrade_cost* *max_maintenance_cost* *max_support_cost* *max_installation_cost* *max_training_cost* These attributes specify the maximum amount of money that clients are willing to pay for *each* acquired component for the various needs of the developed system. Each of these attributes is, in fact, an array that corresponds to each license model (single, developer, site, etc.) supported by CSL (see Chapter 7) and WSL (see Chapter 8).
	preferred_suppliers This attribute will hold a partial list of component suppliers preferred by the development organization. Wrapper generator users would normally fill this list in the first use of the wrapper generator (see Section 9.4.) If the generator user selected one or more preferred suppliers, then only those selected suppliers will be contacted during the searching process. Otherwise, the search heuristics will use information from the *supplier_types* constraint (discussed next) to select relevant supplier categories for each needed compositional wrapper.
	supplier_types A prioritized list of one or more of the following: ORIGINAL_DEVELOPER, INTERMEDIARY_SUPPLIER, PUBLIC. This constraint limits the type of component suppliers to be considered in the search process. The generator will use these supplier types in the absence of a *preferred_suppliers* list.

	preferred_technologies In some situations, it is best for organizations to use specific technologies, such as OMG's CORBA or Microsoft's DCOM. Such practice will provide common components that can be shared and reused in different projects throughout the organization. This will also aid in building a common knowledge base that can be shared among development groups. On the other hand, committing the organization to one or few technologies could force projects to use technologies that are not suitable for them.
	preferred_operating_systems This constraint restricts the operating system environments that the compositional wrapper is targeted for.
	preferred_languages If the system integrator specified a preference on the programming languages in which the found components were written, then such preference will be stored here. This constraint is applicable to components both in source-code or in binary formats. For binary formats, the original programming language in which the component was written, as well as the compiler that was used to generate the object code, has significant impact on the performance, size, and quality of the generated component.
	preferred_component_formats This attribute constrains the format of the sought-for components. The component could be in source-code (i.e., SOURCE), binary (i.e., EXECUTABLE), or object code (or OBJECT) formats.
	software_constraints *hardware_constraints* *network_constraints* Each of these constraints is, in fact, a structure that holds the constraints of the potential operating environment. The corresponding sections in CSL and WSL give a hint on how the structure to hold these constraints could be implemented.

	preferred_licensing_models A prioritized list that contains one or more of the following licensing models: FREE, DEMO, PER_USE, PER_USER, and TIMED. Only vendors that support the models stored in this list will be considered during the searching process.
	license_count The maximum number of licenses required when the PER_USE or PER_USER license types is selected. This attribute has no corresponding construct in the wrapper specification language. Therefore, the wrapper generator can either ignore using this constraint in the matching process or can set it to a very large number.
	license_duration The maximum duration (in months) for which the system integrator wishes to obtain TIMED licenses. This attribute has no corresponding construct in WSL and will either be ignored during the matching process or will be taken as a very large number.
	preferred_payment_methods A prioritized list of one or more of the following: CHECK, CREDIT_CARD, DIRECT_DEBIT, MONEY_ORDER, or WIRE_TRANSFER. Only vendors that support these payment methods will be considered in the searching process.
	preferred_download_methods A prioritized list of one or more of the following: DOWNLOAD_ANYTIME, DOWNLOAD_AT_RUNTIME, NODOWNLOAD. The DOWNLOAD_AT_RUNTIME option requires downloading the component on the first invocation of any of the component interfaces. The NODOWNLOAD option will not download the component to the client's premise. Instead, the component will be executed at the vendor's premise. These download options are discussed in Section 14.2.

Reservations

The list of reservations that are made during the preliminary evaluation stage (as discussed in Section 11.2.) The following information will be kept for each reservation made:

	Reservation_ID A reference code or a reservation number provided by the component provider after a reservation is made.
	Component_ID The name or ID (if the component is a compositional wrapper) of the component for which the reservation is made.
	Vendor_ID The component vendor's name or ID.
	Reservation_Date The date and time on which the reservation is made.
	Expiration_Date The last date after which the reservation becomes invalid.
	Paid_Fees The amount of money paid to the component vendor by the developing organization in return of reserving the component.

found_components

This attribute holds a list of pointers to the components that were found during the search process. Information stored for each found component will include:

	ID The unique identifier (if the component is a compositional wrapper) that is assigned to the component by its supplier. Currently, CSL has no construct to require this attribute.
	Name The popular name of the component taken from the *name* clause in the component specification.
	Description A narrative description of the component taken from the *description* clause in the component specification.
	Vendor_Profile A structure to hold information about the component's vendor. Information in this structure will come from the *provider* section in the component specification.
	Reservation_ID A reservation number or code that the component vendor can supply to the search agent if a reservation to this component was made. See Section 11.2 for a discussion of component reservations.
	Version The component's version number taken from the *version* clause in the component specification.
	Technologies The list of technologies supported by the component. This list is taken from the *technologies* clause in the component specification.
	Operating_Systems The list of operating systems supported by this component. This list is obtained from the *operating_systems* clause in the component specification.

	Databases The list of databases supported by this component as listed in the *databases* clause in the component specification.
	Languages The list of development languages in which the component can be provided. This list is taken from the *languages* clause in the component specification.
	Component_Formats The list of formats (SOURCE, OBJECT, or EXECUTABLE) in which the component can be supplied. This information is taken from the *formats* clause in the component specification.
	License_Models The list of licensing models supported by the component vendor for this particular component. This information is taken from the *license_models* clause in the component specification.
	License_Types The list of licensing types (i.e., single, developer, or site licensing) supported by the vendor of this component for this particular component. The information for this list comes from the *license_types* clause in the component specification.
	License_Price *Upgrade_Price* *Maintenance_Price* *Support_Price* *Installation_Price* *Training_Price* Each of these attributes is an array to hold the component license cost for the various activities. Each element in the array will hold the activity's price of one supported license type. Elements of these arrays will come from the corresponding cost clauses in the component specification.

	Support_Methods The list of support methods provided by the component supplier for this particular component. Such information will be taken from the *support_methods* clause in the component specification.
	Payment_Methods The forms of payment supported by the component vendor. Information for this attribute will come from the *payment_methods* clause in the *provider* section of the component specification.
	New_Updates A URL pointing to a remote location in the component supplier's website in which new updates will reside. This URL will be taken from the *new_updates* clause in the component specification.
	Downloads The list of downloadable artifacts for this component. Each element in this list is a structure that will hold the artifact's name and other related information (as described in Section 7.1.13.)
	Installation_Commands This list will contain the commands outlined in the *installation* clause of the component specification. These commands will be used during the actual installation of the component in the user's environment as described in Section 7.1.14.
	Configuration_Commands The list of commands (from the *configuration* clause in the component specification) that are needed to configure the component for use with the client's system.

	Provided_Interfaces *Required_Interfaces* These attributes hold, respectively, the list of interfaces provided or required by the component. Each element of these lists will be a structure similar to that of the *Required_Interfaces* of the compositional wrapper. Information for the *Provided_Interfaces* will be extracted from the *Interface* or *Module* specification sections in the component specification. Information for the *Required_Interfaces* will come from the *requires* clause in the component specification.
	Software_Constraints *Hardware_Constraints* *Network_Constraints* These lists of constraints have the same structure as those for the compositional wrapper. Information for these lists will come from clauses by the same name in the component specification.
	Optional_Features The list of optional features, plug-ins, or add-ons provided by the component suppliers to enhance or extend the functionality of the component. These information will be obtained from the *optional* clause in the component specification.
candidate_components This attribute holds a partial list of ***found_components***. Only the components that pass the preliminary evaluation (see Chapter 11) will be added to this list. ***candidate_components*** have the same structure as that of ***found_components***.	
matching_components This attribute holds a partial list of the ***candidate_components*** list. Only the components that pass the detailed evaluation (see Chapter 12) will be added to this list. Each ***matching_component*** must have at least one interface that either match a required interface in the wrapper specification or a required interface in one of the other ***matching_components***.	

matching_groups Each element of this attribute holds a partial list of ***matching_components***. Components in some of the ***matching_groups*** will collectively match the complete specification of the current compositional wrapper. Other ***matching_groups*** will partially match the specification of the current compositional wrapper. The ***matching_groups*** will be presented to the system integrator for groups' evaluation (see Section 12.6.) At the end of the groups' evaluation process, the integrator either will select one entire group or will select a set of components from one or more ***matching_groups***. The selected components are the ones that will be encased in the compositional wrapper.
selected_components This attribute holds pointers to one or more components from the ***matching_groups*** lists. The components in ***selected_components*** are selected by the system integrator from among matching components and groups. The ***selected_components*** will be the one that the wrapper generator will acquire, configure, and wrap.

CHAPTER 5: SPECIFICATION LANGUAGES

Researchers have been concerned with specification and semantic[3] languages for decades. There are a plethora of languages and specification methods from which to choose. However, the choice must come from an understanding of the goals. For the research of this dissertation, the selection is based on simplicity of development process and ease of its use.

Since this research deals with commercial components for which most vendors will hide implementation details, a mechanism is needed to encode the semantics of components and make them public through the component interfaces. The semantics representation should have the following properties:

- ***Simple***. To encourage vendors to build component factories, the semantics representation should be as simple and easy to understand as possible. Unfortunately, at present, most of the available representations that are capable of representing the semantics of "things" are difficult to understand by humans and difficult to deal with by tools. Most of these representations are not aimed at describing the semantics of wrappers or even components.

[3] Semantics: the actual meaning of things beyond their outer form or appearance [SZYP98].

- ***Lightweight***. Since the goal of having interfaces to components is to simplify their structure and make them portable objects, the semantics structure should be very small. Otherwise, the side effects, such as network bandwidth and storage consumption, incurred during the transportation of components from vendors' premise to the developers' premise, will make it unfeasible to build software from commercial components.

- ***Safe***. A representation that discloses the internal implementation details of components is considered to violate the privacy and the trade secrets of component vendors. However, at the same time, the representation should assure clients that it exactly represents the components internal implementation and nothing more or less.

This part of the dissertation presents two new specification languages that were developed for this dissertation as one part of a complete solution to the concept of smart compositional wrappers. Section 5.2 gives an overview of the criteria used to formulate the two specification languages. Chapter 7 focuses on the Component Specification Language (CSL) and Chapter 8 on the Wrapper Specification Language (WSL). While each of the languages is *new*, each uses a few constructs from the Assertion Definition Language (ADL) [OBAY98, SANK94a, and SANK94b].

The languages developed for this dissertation are easy to use and understand, but have constructs that are sufficiently formal to describe the syntactic and semantic aspects of the target wrapper and the encased components. WSL gives specification writers the

ability to seamlessly integrate user-selected features into the generated wrapper's code. WSL includes constructs that can support the management of the wrapped components, including: tracking component use and performance, tracking component licenses, and updating the compositional wrapper and its components when necessary.

5.1 Relationship to Existing Specification Languages

There are many well-researched and well-established formal, semi-formal, and informal specification languages. However, there are drawbacks to using these languages to express the syntactic and semantic aspects of compositional wrappers. The richness and complexity of some of the languages can make it difficult to comprehend their constructs. As a result, it can be difficult to write parsers to translate specifications written in those languages into a notation appropriate for wrapper generation. Some of the languages were developed for specific application domains and are not generally applicable. In addition, while the existing languages typically contain constructs that are not needed in the intended use of compositional wrappers, the wrapper generation process can require many constructs that are not addressed by these languages.

The following criteria were used to select a language to augment the support of the wrapper's composition and generation process: the language should be (a) widely accepted, (b) easy to use and understand, and (c) targeted for use with software components. Based on these criteria, the decision was made to exclude languages that use encrypted formalism and symbols that can be difficult to understand and apply. Thus lan-

guages such as VDM [FITZ98], Z [SPIV92] and Larch [GUTT93] and their object-oriented extensions, such as OOZE [ALEN92], VDM++ [DURR94], and Object-Z [DUKE96], were not considered. Similarly, languages that use predicate logic, process algebra, or logic calculi to express their notations were not considered (see for example, PENI00, MAGE95, and CANA97.)

The Interface Definition Language (IDL) [RESE95], Rapide [LUCK96, KENN96], and the Assertion Definition Language (ADL) [OBAY98, SANK94a, SANK94b] each met the criteria. The remainder of this section gives a brief description of each of these three languages. The next section will describe the rationale for the decision to use some of the general constructs from the ADL language.

IDL was developed by the Object Management Group and is widely used in industry. IDL is flexible and platform independent with respect to distributed systems. Using IDL, a developer can define only the required interfaces to services without the need to understand many of the details of implementing the services. However, IDL focuses on the syntactic part of distributed objects and does not include mechanisms for describing components' behavior.

Rapide is an architectural definition language developed at Stanford University as part of a project that focuses on developing a new technology for building large-scale, distributed multi-language systems. This technology is based upon a new generation of computer languages, called Executable Architecture Definition Languages (EADLs), and includes an innovative toolset supporting the use of EADLs in evolutionary development and rigorous analysis of large-scale systems. Rapide provides a capability to allow a

programmer to describe required interfaces for a component, offering a clear mechanism for specifying varieties of behavioral specifications of interfaces. However, because the Rapide compiler generates executable code, it would be difficult to integrate Rapide with a wrapper generator developed in a language other than Rapide. In addition, Rapide is less widely used in industry than IDL or ADL.

ADL is a formal specification language developed jointly by The Open Group and the SunTest division of Sun Microsystems Laboratories and funded by the Japanese MITI's Information-technology Promotion Agency. The language is aimed at specifying software units written in C, C++, IDL, and Java. Inputs to ADL are in the form of formal specification files. Assertions are written as a description of how the function behavior is affected by the input state, and the resulting changes on the output state.

5.2 Selection of the Specification Languages

Based on the strengths and weaknesses of these three languages, it was decided that the Component Specification Language and the Wrapper Specification Language will be developed from scratch, borrowing from ADL the constructs that are common to many programming and specification languages. In particular, the borrowed ADL constructs are those needed for building expressions, statements, loops, and conditions. As needed, some of the borrowed ADL constructs were modified to fit the constructs' structure of the two new languages.

Though the borrowed constructs could come from many other languages, ADL was selected for a number of considerations: ADL is based on IDL, an industry standard for specifying the *syntax* of component interfaces. ADL includes most of the constructs needed to specify the general constructs. In addition, ADL's constructs are formal enough to support specification of the borrowed general constructs without ambiguities. At the same time, ADL's constructs are simple enough to be easy to understand.

5.3 Specification Languages Overview

This section provides an overview of CSL and WSL. The provided material covers the syntax and building blocks of the two languages.

5.3.1 The Languages' Syntax

The syntax of WSL and CSL constructs follows the standard Backus-Naur Form (BNF.) The following explains the BNF syntax:

- The vertical bar "`|`" represents a choice between different expansions. Hence, "`A | B | C`" represents either `A`, `B`, or `C`.
- Square brackets "`[ ... ]`" indicate optional constructs. Hence "`A [ B ] C`" is the same as "`ABC | AC`".

- Parentheses "`( ... )`" are used for grouping constructs. Hence "`A ( B ) C`" is the same as "`ABC`" and "`A ( B | C ) D`" is the same as "`ABD | ACD`".
- "`( ... )*`" is used to represent zero or more occurrences of the group, and "`( ... )+`" is used to represent one or more occurrences of the group. Hence, "`A ( B )* C`" is the same as "`AC | ABC | ABBC | ABBBC |` *etc.*" and "`A ( B )+ C`" is the same as "`ABC | ABBC | ABBBC |` *etc.*".
- Lexical tokens and reserved words may appear literally within single quotations (for example, '`name`'). The name of a lexical token may appear in angle brackets (for example <`QuotedString`>.)
- The left hand and the right hand sides of productions are separated by the symbol "`::=`".

Other general formatting considerations for the new languages are as follows:

- Use of white spaces and comments in these two languages is consistent with their use in ADL and other specification and programming languages.
- Multi-line constructs (as in specifying postal addresses) can be broken in multiple lines.
- Productions and lexical tokens will be shown using a `Courier` font. Code snippets will be shown using the Arial font.
- Reserved words and identifier names are case-insensitive.

5.3.2 Examples of Language Constructs

Some lexical tokens can be expressed using regular expressions. For example, a <Digit> defines a numerical digit from 0 to 9. It is expressed in regular expressions as:

```
<Digit> ::= /\d/
```

Constructs can also be formed from other constructs. For example, the <TwoDigits> construct is used in specifying months and days and consists of two consecutive <Digit> constructs as follows:

```
<TwoDigits> ::= <Digit><Digit>
```

More complex constructs can be constructed from simple constructs, lexical tokens, and, possibly, some of the BNF notations. For example, the <Size> construct, used to express the data size of some of the data types and data structures, is expressed as follows:

```
<Size> ::=
            'any'
      |     <Digits>
      |     <Range>
      |     <PositiveRange>
      |     <IntegerClass>
      |     <CharClass>
```

This says that <Size> can be either the reserved keyword 'any' or any one of the five listed constructs.

5.3.3 Examples of Modified ADL Constructs

The syntax of some of the constructs that were borrowed from ADL was modified to fit the need of the CSL and WSL languages and the use by the wrapper generator that was discussed in Chapter 4. For example, a for-loop in ADL is expressed using the following syntax:

```
ForStatement ::= "for" "(" [ ForInit ] ";" [ Expression ] ";" [ Fo-
rUpdate ] ")" Statement
```

The syntax of a for-loop in CSL and WSL is defined as follows:

```
<ForEvaluator> ::= 'for' '(' [ <ForInit> ] ';' [ <Expression> ] ';'
      [ <ForUpdate> ] ')' '{' <CallAssertionsGroup> '}'
```

The syntax of <ForInit>, <Expression>, and <ForUpdate> is the same as that of ADL.

5.3.4 Basic Constructs

This section lists some of the basic constructs that are used to build other constructs. These basic constructs are expressed either by using the regular expression notations or as quoted terminals, as follows:

`<Digit> ::= /\d/` A numerical digit from 0 to 9.
`<Digits> ::= /\d+/` One or more numerical digits.
`<HexDigit> ::= /(\d|[ABCDEF])+/i` A hexidecimal digit.
`<NonZeroDigit> ::= /[1-9]/` A positive, non-zero digit.
`<Identifier> ::= /[a-zA-Z_][a-zA-Z0-9_]*/` An identifier starts with a letter or an underscore and is followed by any sequence of letters, numbers, or an underscore.
`<Sign> ::= '-' | '+'` A mathematical sign.
`<Char> ::= /\w/` Any printable character.
`<BooleanLiteral> ::= /True|False/i` A Boolean literal is either true or false.
`<YesNo> ::= /Yes|No/i` A Yes/No construct.
`<QuotedString> ::= /"([\w\W][^\n])+?"/` A quoted string is any sequence of characters enclosed between two double quotes. The double-quotes themselves are not allowed within the string. An end-of-line marker also terminates the quoted string.
`<MultiLineString> ::= /\{([\w\W)+?\}/` Multi-line strings are used for constructs that span multiple lines, such as addresses or descriptions. The multi-line string must be enclosed between two curly brackets.

Though it is not required, it is recommended to split the multi-line constructs into multiple lines. This will help the specification readers to clearly spot the supplied information, as can be seen from the following example.

```
address = {
        1 N. First Drive
        Suite # 9999
        New Florida, FL 77777
};
```

```
<ColorName> ::= 'black' | 'blue' | 'cyan' | 'green' | 'red' | 'yellow'
| 'white'
```

This construct lists examples of color names. Implementations of the specification languages can list the complete set of color names.

```
<ScreenName> ::= /"\w+(\/\w+)*?"/
```

A screen name is a sequence of strings separated by a forward slash. The complete name must be enclosed between double quotes.

```
<Domain> ::= ( <Name> '.')+ <WebsiteType> ['.' <CountryCode> ]
<WebsiteType> ::= 'com' | 'net' | 'org' | 'edu' | 'mil'
<CountryCode> ::= 'ca' | 'om' | 'uk' | 'us'
```

These constructs specify an Internet domain or sub-domain name. Implementations of the two specification languages can include the full list of website types and country codes.

The following is a list of the remaining basic constructs that are defined by CSL and WSL and are used to build many other constructs. This set of constructs is built using the basic constructs shown above.

```
<TwoDigits> ::= <Digit><Digit>
```

This construct is used in specifying months and days.

```
<FourDigits> ::= <TwoDigits><TwoDigit>
```

This construct is used for specifying years and credit-card numbers.

```
<Name> ::= <Identifier> ( '.' <Identifier> )*
```

This says that a <Name> can be either a single identifier, such as *x*, or a dotted identifier, such as *x.y*. Dotted identifiers are used to refer to interfaces within certain components (for example, *mycomponent.its_interface*.)

```
<PositiveInteger> ::= [ '+' ] <Digits>
```

A positive integer is any sequence of 0-9 without trailing scientific notation symbols. Only the plus sign is allowed to precede the number.

```
<Integer> ::= [ <Sign> ] <Digits>
```

An integer is any sequence of digits that can be preceded by a sign.

```
<Float> ::= [ <Sign> ] <Digits>[ '.' <Digits> ]
```

A floating-point number is any sequence of digits, as with <Digits>. It can optionally be followed by a period and another sequence of digits. A sign can precede the floating-point number.

```
<PositiveFloat> ::= [ '+' ] <Digits>[ '.' <Digits> ]
```

A positive floating-point number is the same as <Float>. However, only the plus sign is allowed to precede the number.

```
<Time> ::= <Hour>':'<Minute>[':'<Second> ] ('am' | 'pm')
<Hour> ::= <NonZeroDigit> | '10' | '11' | '12'
<Minute> ::= <ZeroToFiftyNine>
<Second> ::= <ZeroToFiftyNine>
<ZeroToFiftyNine> ::= <NonZeroDigit> |
                      ( '1' | '2' | '3' | '4' | '5' )<Digit>
```

These constructs specify how a time can be entered. A time will consist of an hours-part, a minutes-part, and, optionally, a seconds-part. A colon separates the parts. A trailing AM/PM designator can be appended to the time. The time is specified in a 12-hour basis.

```
<Day> ::= <NonZeroDigit> | ( '1' | '2' )<Digit> | '3'( '0' | '1' )
```

A <Day> specifies the day of the month and can fall in three categories. The first category is a one-digit number for days between 1 and 9. The second category is a two-digits number for days between 10 and 29. The third category is for days 30 and 31st.

```
<Month> ::= <NonZeroDigit> | '1'( '0' | '1' | '2' )
```

A <Month> specifies the month number and can fall into two categories. The first category is a one-digit number for months between 1 and 9. The second category is a two-digit number for months 10, 11, and 12.

`<Year> ::= <FourDigits>` A <Year> is defined here as just a four-digits number. No restriction on the year is stated. However, implementations of the two languages can define a year more precisely.
`<Date> ::= <Month> [ '-' | '/' ] <Day> ['-' | '/' ] <Year>` A <Date> is specified as a two-digits month followed by a two-digits day and, then, a four-digits year. A dash or a slash can separate the parts.

`<Path>`
`<FileName>`
`<Directory> ::= '"' <Path> '"'`
`<Url>`
`<Email>`
 	
These constructs are left to implementations of the specification languages to specify them according to their environment settings.

5.3.5 General Constructs

This section lists some of the frequently used constructs. The following list of constructs are used in composing many of the CSL and WSL constructs that will be outlined in the remaining sections of this chapter and in Chapter 6 to Chapter 8.

`<Data> ::= <Char>+` This construct defines a stream of raw characters that is used in inputs and outputs.
`<NameDeclaration> ::= 'name' '=' <QuotedString> ';'` A <NameDeclaration> construct is used to define names of components, wrappers, and people.
`<DescriptionDeclaration> ::= 'description' '=' <MultiLineString> ';'` This construct is used to explain in more detail the features and behaviors of components

and wrappers.

```
<UrlDeclaration> ::= 'url' '=' <Url> ';'
```

This construct gives the syntax for expressing Internet URLs.

```
<EmailDeclaration> ::= 'email' '=' <Email> ';'
```

This construct gives the syntax for expressing email addresses.

```
<PhoneDeclaration> ::= 'phone' '=' <QuotedString> ';'
```

This construct defines a telephone number as simple a quoted string. Implementation of these specification languages can give more restrictive-syntax of expressing telephone numbers.

```
<FaxDeclaration> ::= 'fax' '=' <QuotedString> ';'
```

This construct defines a fax number as simple a quoted string. Implementation of these specification languages can give more restrictive-syntax of expressing fax numbers.

```
<AddressDeclaration> ::= 'address' '=' <MultiLineString> ';'
```

This construct defines the syntax for a postal address.

```
<OtherDeclaration> ::= 'other' '=' <QuotedString> ';'
```

This <OtherDeclaration> construct is used to enter narrative descriptions of various entities, such as software, hardware, and network constraints.

```
<StorageDesignator> ::= 'KB' | 'MB' | 'GB'
```

This construct lists the abbreviations that usually follow a memory or disk size.

```
<Size> ::= 'any' | <Digits> | <Range> | <PositiveRange> | <Integer-
Class> | <CharClass>
<Range> ::= [ <Sign> ]<Digits> '..' <Digits> | <Char> '..' <Char>
<PositiveRange> ::= <Digits> '..' <Digits> | <Char> '..' <Char>
<CharClass> ::= '[' <Char> ( ',' <Char> )* ']'
<IntegerClass> ::= '[' [ <Sign> ]<Digits> ( ',' [ <Sign> ]<Digits> )*
']'
<FloatClass> ::= '[' [ <Sign> ]<Float> ( ',' [ <Sign> ]<Float> )* ']'
```

These constructs list the various forms of expressing data sizes. Many of the data structures that will be outlined in Section 6.1.2, as well as inputs and outputs, use this general form of expressing sizes. Some of the data structures use restricted forms for defining the size of their data types.

```
<FileMode> ::= 'create' | 'current' | 'append'
```

This construct lists three possible actions for modifying files. The `create` keyword means to create the file if it does not exist or to overwrite its content if it exists. The `current` keyword means to insert the new information at the current insertion point in the file. The `append` keyword says to insert the new information at the end of the existing file or to create the file if it does not exist.

5.3.6 Constructs Borrowed From ADL

The following constructs are examples of the list of constructs that are borrowed from ADL. The constructs have the same syntax as that defined by ADL and are used in some of CSL and WSL constructs.

```
<ArgumentList>
<Arguments>
<Assignment>
<Block>
<ComparisonOperator>
<ConditionalExpression>
<Expression>
<ForInit>
<ForLoop>
<ForUpdate>
<FormalParameters>
<IfStatement>
<RelationalExpression>
<WhileLoop>
```

5.3.7 Interface Behaviors

The discussion on behavior specification applies equally to CSL and WSL. However, while component suppliers would use behavior specification constructs to precisely describe the behavior of their components, wrapper specification writers use the behavior

specification to express the general behavior expected from their compositional wrappers. Specification writers can use one of two ways to express interface behaviors of software components: (a) define the complete interface using the <InterfaceSpecification> construct, or (b) group interfaces into modules using the <ModuleSpecification> construct. The syntax for these two constructs is given below:

```
<ModuleSpecification> ::= 'module' <Identifier>
      '{' ( <InterfaceSpecification> | <BehavioralSpecification> )+ '}'

<InterfaceSpecification> ::= 'interface' <Identifier>
      '{' <BehavioralSpecification>+ '}'
```

The <ModuleSpecification> construct has no semantic value in CSL or WSL. Rather, it is used for grouping interfaces that share other semantic constructs, such as inputs, outputs, variables, and functions. If a <ModuleSpecification> is not used, each interface must be declared individually with its inputs, outputs, variables, and functions. The disadvantage of defining stand-alone interfaces (i.e., without enclosing them in a module) is that redundant definitions of inputs, outputs, variables, and functions could occur in the same specification document.

The identifier in each of these constructs is referenced in the *provides* clause in the component specification language (see Section 7.1.16) and by the *interfaces* clause in the wrapper specification language (see Section 8.2.4)

Additionally, the <Name> in the *provides* and *interfaces* clauses points to the <Identifier> of a stand-alone <InterfaceSpecification> section or to an interface embedded within a <ModuleSpecification> section. For this latter case, <Name> is formed as "<Identifier>.<Identifier>", where the first <Iden-

tifier> refers to the module name of the <ModuleSpecification> section and the second <Identifier> refers to one of the interface names that are embedded within the <ModuleSpecification> section. For example, in the following declarations, interface outer is not embedded within a module declaration. Therefore, the interface is referenced by its name alone (for example, provides = outer;). However, to refer to any of the two interfaces, first and second, that are embedded within the InterfacesGroup1 module, the interface name must be prefixed with the module name and a dot (for example, provides = InterfacesGroup1.first, InterfacesGroup1.second;).

```
interface outer {
        . . .
}

Module InterfacesGroup1 {
        i:int;

        interface first {
                . . .
        }
        interface second {
                . . .
        }
}
```

The <BehavioralSpecification> is defined as follows:

```
<BehavioralSpecification> ::=
      <VariableDeclaration> | <InputDeclaration> |
      <OutputDeclaration> | <BehaviorDeclaration> |
      <StimulusDeclaration> | <FunctionDeclaration>
```

The next chapter will provide the details for each of the constructs included in the above production.

5.4 The Content of the Specification Documents

This section describes the minimum content of CSL and WSL specification documents. For each specification element, a reference is provided to the applicable section in the dissertation.

5.4.1 The Component Specification Document

The main production of the component specification language looks like the following:

```
CSL_Specification ::= <ComponentSpecification>
      ( <ComponentSpecification> | <ProviderSpecification> |
      <ModuleSpecification> | <InterfaceSpecification> )*
```

Figure 5 shows the various sections that a component specification document could contain. Each oval in this figure corresponds to one of the sections that are defined in this chapter or in Chapter 7. The heavily shaded ovals reflect the sections that must be present in each component specification document. The lightly shaded ovals are sections that exist in the specification document to fulfill the need of other required sections. The unshaded ovals represent those sections in the component specification that are left (intentionally or unintentionally) by component specification writers. The "left-over" sec-

tions would include the *component*, *module*, *interface*, or *provider* specification sections. (The "left-over" sections serve no purpose for the wrapper generator.)

As indicated by Figure 5, the component specification document should contain the following:

- A component specification for describing the main component. The specification document can also contain component specifications for any of the component identifiers listed in the *optional* clause (see Section 7.1.21) or the *requires* clause (see Section 7.1.17) of the main component or any other included component specifications. The first component specification in the specification document *must* be for the main component to which the specification document is written.
- At least one specification for the provider listed in the *provider* clause (see Section 7.1.1) of the main component specification. Other components defined in the specification document can introduce more provider specifications if such components reference providers that are not yet defined in the specification document.
- One or more `<ModuleSpecification>` or `<InterfaceSpecification>` sections for each interface identifier listed in the *provides* clause (see Section 7.1.16.) The specification document can optionally contain one or more `<ModuleSpecification>` or `<InterfaceSpecification>` sections for any of the interface identifiers listed in the *requires* clause (see Section 7.1.17.)

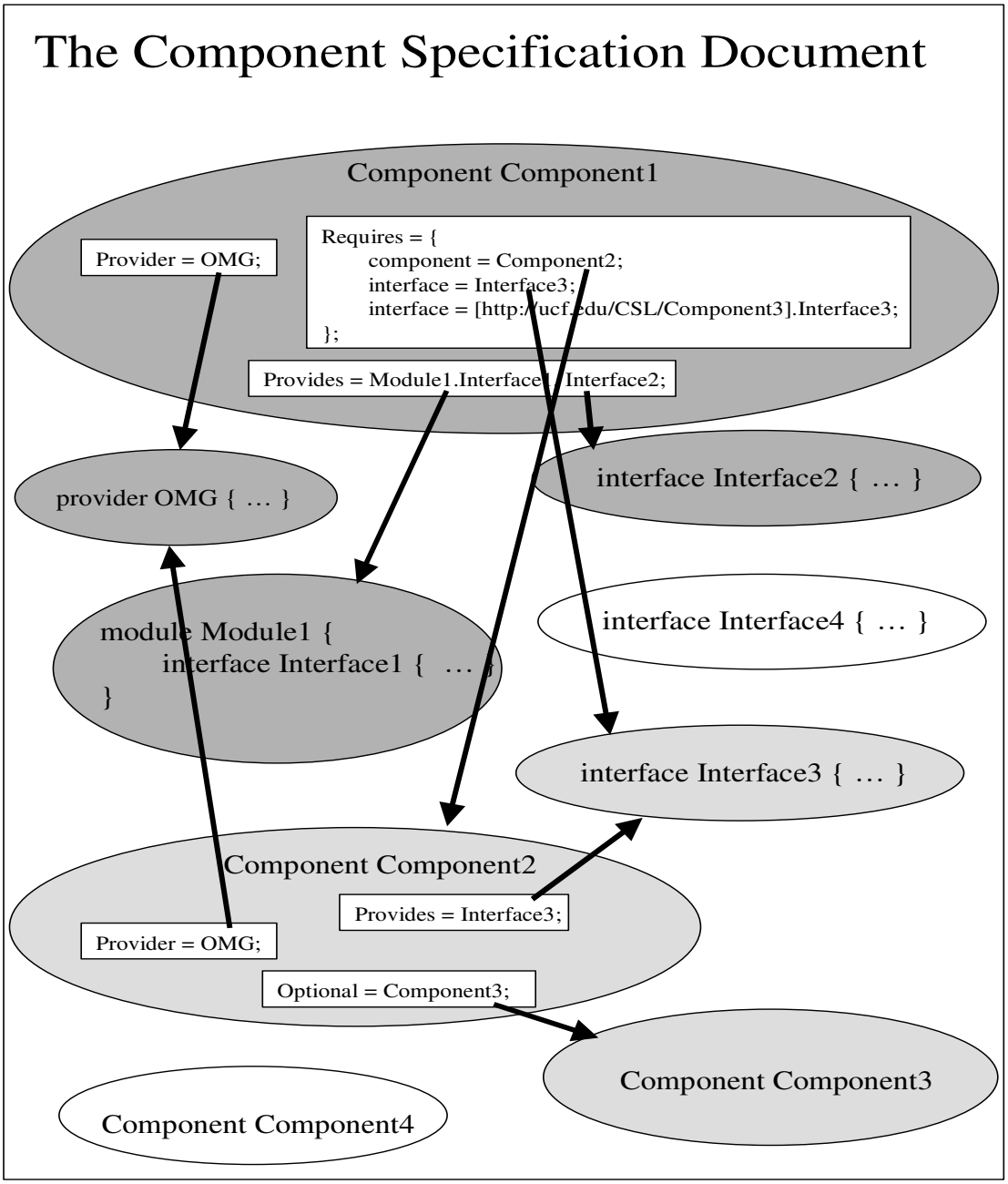

Figure 5: The various sections that compose component specification documents

5.4.2 The Wrapper Specification Document

The main production of the wrapper specification language looks like the following:

```
WSL_Specification ::= <WrapperSpecification> |
        ( <ModuleSpecification> | <InterfaceSpecification> |
          <PersonSpecification> | <ProfileSpecification> |
          <CreditCardSpecification> | <BankAccountSpecification> )* |
        <ArchitectureSpecification>
```

Figure 6 shows the various sections that compositional wrapper specification documents could contain. Each oval in this figure corresponds to one of the sections that are defined in this chapter or in Chapter 8. The heavily shaded ovals reflect sections that must be present in each wrapper specification document. The lightly shaded ovals are sections that exist in the specification document to fulfill the need of other required sections. The unshaded ovals represent those sections in the wrapper specification that are left by the wrapper specification writers. The "left-over" sections include the *module, interface, person*, *profile*, *credit_card*, or *bank_account* specification sections. Such "left-over" specification sections may be of help to specification writers in future modifications of the specification document. However, these "left-over" sections serve no purpose to the wrapper generator.

As indicated by Figure 6, the wrapper specification document should contain the following:

- One `<WrapperSpecification>` section to specify the wrapper that needs to be generated.

- One or more <ModuleSpecification> or <InterfaceSpecification> for each component identifier listed in the *interfaces* clause (see tion 8.2.4.)

- One or more <PersonSpecification> sections (see Section 8.4) for the wrapper owner listed in the *owner* clause (see Section 8.2.1) and for each, other than owner, identifier listed in the *developers* clause (see Section 8.2.2.) One or more <PersonSpecification> sections must also be included for each identifier listed in the *contacts* clause in user profiles (see Section 8.5.)

- One or more <ProfileSpecification> sections (see Section 8.5.) There must be one <ProfileSpecification> section defined in the wrapper specification whose name matches that listed in the *personal_info* clause (see 8.2.5.) Additional <ProfileSpecification> sections can be present in the specification. However, the wrapper generator will ignore such unused profile specifications.

- One or more <CreditCardSpecification> sections (see Section 8.6.) If the *payment_info* clause (see Section 8.2.6) lists a credit card identifier, then there must be one <CreditCardSpecification> section defined in the wrapper specification whose name matches that listed in the *payment_info* clause. Additional <CreditCardSpecification> sections can be present in the specification. However, the wrapper generator will ignore them.

- One or more <BankAccountSpecification> sections (see Section 8.7.) If the *payment_info* clause lists a bank account identifier, then there must be one <BankAccountSpecification> section defined in the wrapper specification whose name matches the identifier listed in the *payment_info* clause. Additional <BankAccountSpecification> sections can be present in the specification. However, the wrapper generator will ignore such "left-over" sections.
- One <ArchitectureSpecification> section (see Section 8.3.)

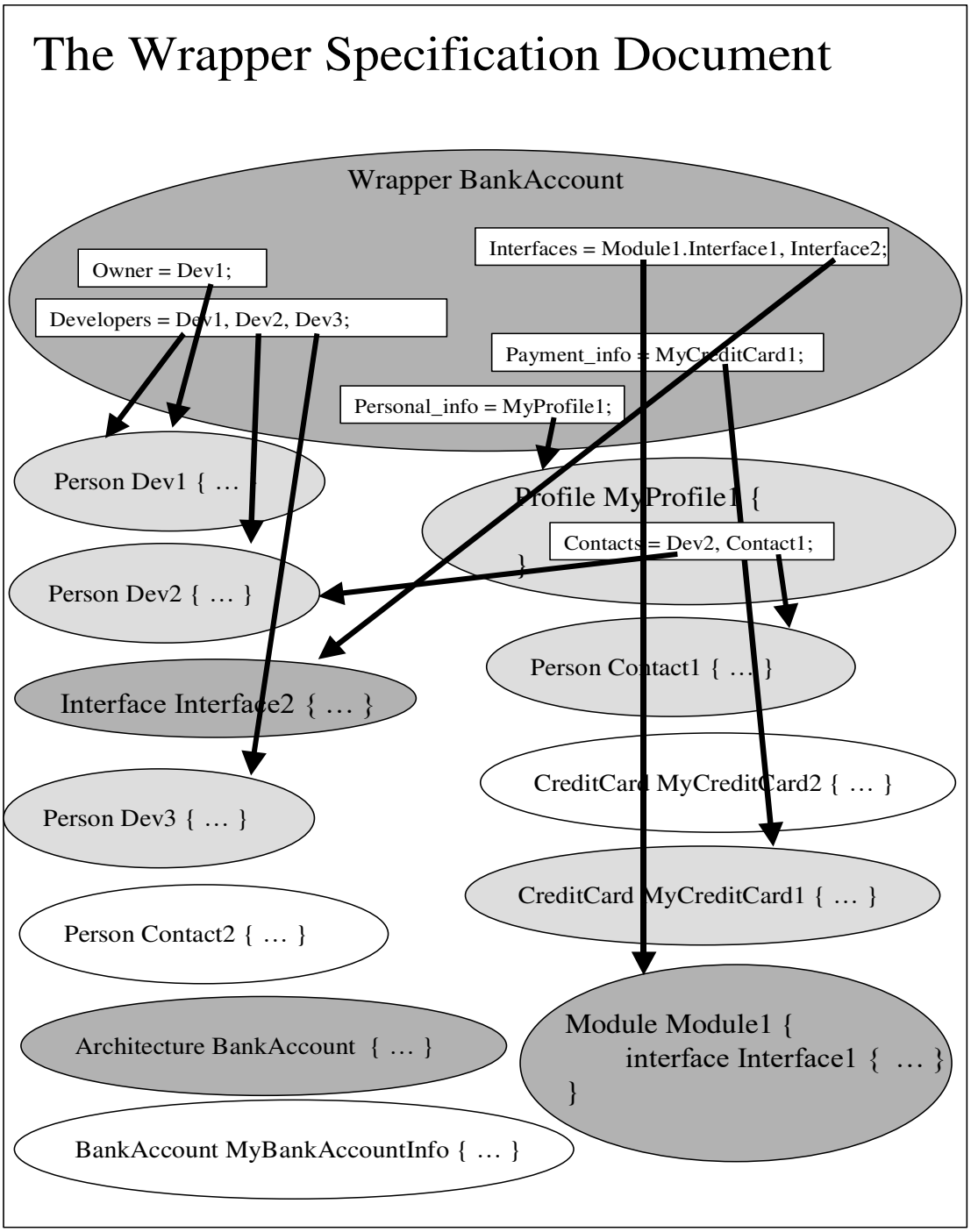

Figure 6: The various sections that compose wrapper specification documents

CHAPTER 6: COMMON SECTIONS

This chapter contains the sections that are common to both specification languages (CSL and WSL.) Section 6.1 describes the specification of behavior, including variable declarations, data types, inputs and outputs, interface behaviors, and interface stimuli. Section 6.2 provides a complete behavioral specification example.

6.1 Specifying Behavior

The following subsections discuss the elements that can be used to specify the behavior of compositional wrappers and off-the-shelf components. (The general structure of interface and module specifications is given in Section 5.3.7.)

6.1.1 Variable Declarations

Both CSL and WSL allow the declaration of variables and constants. Such declarations are used within interfaces, modules, and architecture definitions.

```
<VariableDeclaration> ::= <VariableDeclarator>
      ( ',' <VariableDeclarator> )* ';'

<VariableDeclarator> ::= ['const'] ( <Identifier>
      ( ',' <Identifier> )* ':' <Type> [ '=' <VariableInitializer> ]
```

```
<VariableInitializer> ::= <Expression>
```

The following example defines two constants, S and D, as short integers that can be used as global constants.

```
const S: int, const D: short int;
```

6.1.2 Data Types

Both specification languages support most of the data types that are common to most programming and specification languages. New data types have also been added to give more flexibility in expressing component behaviors. The following construct lists all the data types supported by the CSL and WSL languages:

```
<Type> ::= <PrimitiveType> | <Name> | <InputType> | <OutputType> |
      <EnumType> | <RangeType> | <ArrayType> | <ListType> |
      <StructureType> | <FileType> | <EventType> | <DatabaseType> |
      <ObjectType> | <ReferenceType> | <MessageType> |<PrintoutType> |
      <StateType> | <HardwareType> | <DeviceType>
```

A <PrimitiveType> consists of <NumericalPrimitiveType>s, which can be prefixed by a plus or minus sign, and of <OtherPrimitiveType>s.

```
<PrimitiveType> ::= ( [ <Sign> ]<NumericalPrimitiveType> ) |
      <OtherPrimitiveType>
```

The <NumericalPrimitiveType> lists all the numerical primitive data types supported by this specification language. The <OtherPrimitiveType> lists the other primitive data types whose sizes and ranges are given in Table 2.

```
<NumericalPrimitiveType> ::= 'byte' | ( [ 'signed' | 'unsigned' ]
      [ 'short' | 'long' ] 'int' ) | 'float' | 'double'

<OtherPrimitiveType> ::= 'char' | 'string' | 'boolean' | [ 'small' |
      'medium' | 'long' ] 'text'
```

Table 2: Sizes and ranges of non-numerical primitive data types defined by the CSL and WSL specification languages

Char	Any valid ASCII character
String	Can hold up to 255 characters
Small text	Can hold up to 16K characters
Medium text	Can hold up to 32K characters
Long text	Can hold up to 64K characters
Boolean	Either *true* or *false*

The <Name> clause is used for variable declarations. Its syntax was given in Section 5.3.4.

An <InputType> and <OutputType> are used when passing inputs or outputs as parameters to functions, as will be discussed later in this section. These constructs have the following syntax:

```
<InputType> ::= 'input' '(' <Name> ')'
<OutputType> ::= 'output' '(' <Name> ')'
```

The <Name> in these two constructs refers to a named input or output that was declared using the <InputDeclaration> or <OutputDeclaration> constructs.

The <EnumType> is used to group a list of items under one name. It has the following syntax:

```
<EnumType> ::= '{' <Data> ( ',' <Data> )* '}'
```

For example, one can list all color names under the enumerated type of colors as follows:

colors: { black, red, green, blue, white }

A <RangeType> defines a range of <NumericalPrimitiveType>s or a range of characters.

```
<RangeType> ::= 'range' '(' ( <NumericalPrimitiveType> | 'char' ) ')'
```

For example, the declaration (S: range(int);) defines a range of integers. In this case, S = 0..10; and S = -20 .. 50; are all valid assignments of S.

An <ArrayType> is a structure that is similar to that defined in many programming and specification languages. However, in CSL and WSL, a range can have an absolute size, a range of sizes, or a discrete set of valid sizes. It has the following syntax:

```
<ArrayType> ::= ( <PrimitiveType> | <Name> ) '[' <Size> ']'
```

For example, to declare SSN to be an array of 9 positive numbers and Telephone to be a string of only 7, 10, or 11 digits, the following declarations can be used.

SSN: +int[9];
Telephone: +int[7,10,11];

The plus sign before int in the two declarations explicitly states that only positive integers are allowed. Thus, the assignments SSN = 123456789; and Telephone =

4071234567; are valid ones, but the assignments SSN=-123456789;, SSN=345;, and Telephone = 407123456789; are invalid.

A <ListType> defines a list structure that can grow indefinitely. Optionally, the list size can be limited to a fixed size or a range of sizes. The syntax of <ListType> looks like the following:

```
<ListType> ::= 'list' '(' <Type> [ ‘,’ <Size> ] ')'
```

The following declarations declare A as a list of integers of any size, B as a list of 10 boolean values, and C as a list of 5 lists of characters of any size. C is a list of lists, similar to two-dimensional arrays in some languages.

```
A: list (int);
B: list(boolean, 10);
C: list (list(char), 5);
```

A <StructureType> defines a structure that can contain any number of elements of any type. This structure's definition is similar to that defined in the C language. <StructureType> has the following syntax:

```
<StructureType> ::= 'structure' '{' <VariableDeclaration>
      ( ',' <VariableDeclaration> )* '}'
```

An example of a structure declaration is:

```
Address: structure { first_line, second_line, city: string, state: string[2], zipcode: +int[5] };
```

A <FileType> declaration associates an identifier (a file handle) with a file name or with any other identifier of type <FileType>. For example, F1: file ("temp/abc.txt"); and F2: file(F1); are all valid assignments. The syntax of <File-Type> looks like the following:

```
<FileType> ::= 'file' '(' <FileSpecifier> ')'
<FileSpecifier> ::= <Name> | ( [ <Path> ]<FileName> )
```

An <EventType> captures most of the possible event types. It has the following syntax:

```
<EventType> ::= 'event' '(' <EventChoice> ( ',' <EventChoice> )* ')'
```

An <EventChoice> defines most of the possible event types.

```
<EventChoice> ::= <MouseEventType> | <KeyboardEventType> |
      <MenuEventType> | <TimeEventType> | <ProgramEventType>
```

The <MouseEventType> lists three possible mouse events as follows:

```
<MouseEventType> ::= 'MouseRightClick' | 'MouseLeftClick' |
   'MouseDoubleClick'
```

The <KeyboardEventType> is used to define events generated by pressing any of the keyboard's normal or extended keys. (Extended keys are any key pressed while pressing any of the alt, control, or shift keys.)

```
<KeyboardEventType> ::= 'keyboard' '(' ( '\''<Char>'\'' |
      '\''<ExtendedKey>[ '-'<ExtendedKey> ]
      '-'<Char>'\'' | <CharClass> ) ')'

<ExtendedKey> ::= 'Alt' | 'Ctrl' | 'Shift'
```

The <MenuEventType> declares an event that can be generated by a specific menu item. The <ScreenName> construct is defined in Section 5.3.4.

```
<MenuEventType> ::= 'menu' '(' <ScreenName> ')'
```

A <TimeEventType> defines a specific time event that can occur at certain time (and optionally at certain date.) There are three time event choices defined here. A recurrent time event occurs every specific number of seconds, minutes, etc. A time event can also occur at a specific day of the week or month. The third option is to have a recurrent time event that occurs on a specific time of a specific day of the week or the month.

```
<TimeEventType> ::= 'time' '(' [ <Date> ',' ] ( <Time> |
      <TimeEventChoice> ) ')'

<TimeEventChoice> ::=
      'Every' <Digits> ('seconds' | 'minutes' | 'hours' | 'days' |
              'weeks' | 'months' | 'years') |
      'On' <Digits> 'of' 'every' ('week' | 'month') |
      'At' <Time> of <Digits> 'of' 'every' ('week' | 'month')
```

A <ProgramEventType> allows one to declare a system command as an event. This means that once the indicated system command (or program) launches, it generates an event that can be captured by interested entities. A set of passed parameters or options can be specified to differentiate between the many possible invocations of the system command or program.

```
<ProgramEventType> ::= 'exec' '(' <QuotedString>
      [ ',' <PassedParameters> ] ')'

<PassedParameters> ::= '-' /\w+/ ( ' -' /\w+/ )*
```

The /\w+/ regular expression means any sequence of printable characters.

A <DatabaseType> is used to declare variables that are pointers to database schemas. The database is tagged with a specific type, which is the record type that the database is composed of. For example, the definition EmployeesDatabase: database (Employee); states that EmployeeDatabase is a database of Employee structures.

```
<DatabaseType> ::= 'database' '(' <Type> ')'
```

An <ObjectType> declaration defines a variable to be an instance of a certain object. If the this keyword is used as the object type, then it means an object of the same type as the enclosing object.

```
<ObjectType> ::= 'object' '(' ( 'this' | <Name> ) ')'
```

A <ReferenceType> defines a pointer to any type. It has the following syntax:

```
<ReferenceType> ::= 'reference' '(' <Type> ')'
```

For example, the following lines declare A as a pointer to an integer number, B to be a pointer to an Employee structure, and C to be a pointer to a database of Employee structures:

```
A: reference(int);
B: reference (Employee);
C: reference (database(Employee));
```

The <MessageType> helps in defining the specifics of messages that can be sent to other components or applications. The definition specifies the message type, the

message destination, and an optional data stream to be sent. A message can be sent to a server, a database, or a specific component interface. The name of the recipient can be specified as an identifier or as a quoted string. The actual message data can be included in the message definition.

```
<MessageType> ::= 'message' '(' <Type> ',' <MessageDestination>
      [ ',' '"' <Data> '"' ] ')'

<MessageDestination> ::= ( 'server' | 'interface' | 'database' ) ':'
      ( <Url> | <QuotedString> | <Name> )
```

The <PrintoutType> is used to declare report structures that can be saved to global variables, sent to screens, or written to files. The actual report data can be of any type. Its content can optionally be included within the definition.

```
<PrintoutType> ::= 'printout' '(' <Type> ',' <PrintoutDestination>
      [ ',' [ <FormatOptions> ] [ ',' '"' <Data> '"' ] ] ')'
```

If screen is used as the printout destination, then one can optionally specify where in the screen to start printing that output (both rows and columns can be specified.) If no screen position is given, then printing begins at the position where the last printout ended. If file is the destination, then a file mode can be specified. This tells the component whether to create a new file, print at the current position in the file, or append the printout to the end of the file.

```
<PrintoutDestination> ::= 'global' '(' <Identifier> ')' |
      'screen' | 'file' '(' <FileSpecifier> [ ',' <FileMode> ] ')'
```

The third part in the <PrintoutType> definition is an optional formatting of the printed information. These formatting options give the specification writers full con-

trol on specifying reports. `pos`, `col`, `row`, and `filter` are applicable only to files and screens. `color` is only applicable to screen outputs.

```
<FormatOptions> ::= '{' <FormatOption> ( ';' <FormatOption> )* '}'

<FormatOption> ::= <PositionDeclaration> | <ColumnDeclaration> |
      <RowDeclaration> | <ColorDeclaration> | <FilterDeclaration>

<PositionDeclaration> ::= 'pos' '=' <PositiveInteger>
      [ ',' <PositiveInteger> ]

<ColumnDeclaration> ::= 'col' '=' <PositiveInteger>
<RowDeclaration> ::= 'row' '=' <PositiveInteger>
<ColorDeclaration> ::= 'color' '=' <ColorChoice>
<ColorChoice> ::= '#'<HexDigit> | <ColorName>
<FilterDeclaration> ::= 'filter' '=' <FormattingCode>
```

`pos` has a required row number (for screens) or line number (for files) and an optional column number. The `<FormattingCode>` is similar to that used in the C language.

```
<FormattingCode> ::= '%'<Float>('f' | 'd' | 'l' | 's')
```

A `<StateType>` definition specifies a certain state to which an executing thread should go next. A `stop` state will immediately terminate execution of the thread. The `wait` state will let the thread enter into a waiting state until a certain amount of time (specified in milliseconds) elapsed or a certain event occurred. The `sleep` state will put the thread in a no-activity state for a certain amount of time (specified in milliseconds.) Finally, the `return` state tells the executing thread to return to its caller.

```
<StateType> ::= 'state' '(' <States> ')'
<States> ::= 'stop' | 'wait' '(' <PositiveInteger> | <EventType> ')' |
      'sleep' '(' <PositiveInteger> ')' | 'return'
```

The <HardwareType> allows one to specify variables that bear the characteristics of known hardware items. The <HardwareItems> construct below lists only examples of hardware items. Implementations of CSL and WSL can declare their own applicable list of hardware items and the logic necessary to deal with them.

```
<HardwareType> ::= 'hardware' '(' <HardwareItems> ')'
<HardwareItems> ::= 'money' | 'check' | 'coin' | 'paper'
```

A <DeviceType> declaration attaches a variable to a certain device. The definition requires one to specify the filename for which the device is described fully (for example, ATM = device ("ATM.dvi");.) The format of such device files is implementation-dependent. Depending on how these specification languages are implemented, the device file can be a stream of characters, a list of (keyword, value) pairs, or any other format that is best suited to describe the device.

```
<DeviceType> ::= 'device' '(' [ <Path> ]<FileName> ')'
```

6.1.3 Inputs and Outputs

Input and output declarations are used to describe the inputs that components are expected to require and the outputs that components are expected to generate. Inputs and outputs can be defined using the following constructs:

```
<InputDeclaration> ::= 'input' [ <Identifier> ] '=' '{' ('none' |
      ([ <QuotedString> ','] <InputDeclarationBody> )) '}' ';'

<OutputDeclaration> ::= 'output' [ <Identifier> ] '=' '{' ( 'none' |
      'dontcare' | ( [ <QuotedString> ',' ]
      <OutputDeclarationBody> ) ) '}' ';'
```

The <QuotedString> in both declarations serves as a comment to describe inputs and outputs. The <QuotedString> can also be used to represent the input or output name. Wrapper specification writers can use the keyword none to state that the components to be included in their wrappers must not have inputs or outputs other than those described in the wrapper specification. Component suppliers can use the none keyword to assure component users that their components will not generate any output not listed in the component specification. Additionally, wrapper specification writers can use the dontcare keyword to indicate that their compositional wrappers will ignore any other outputs that components might generate and that are not stated in the component specification. The dontcare keyword is not applicable for use in component specifications.

The following two productions show that input and output declarations are composed of four parts. The first part specifies the data type to be sent to the component (as input) or received from the component (as output.) The second part specifies the size of the passed or received data. The any keyword can be used to indicate that the input or output will be of any size. The third part specifies the list of valid data or data classes. The any keyword can also be used here to indicate that the input or output can have any kind of data (of the specified type.) The fourth part tells the source from which the component can expect the input or the destination to which the component can send the output.

```
<InputDeclarationBody> ::=
      <Type> ',' <Size> ',' <ValidData> ',' <InputSource>
```

```
<OutputDeclarationBody> ::=
      <Type> ',' <Size> ',' <ValidData> ',' <OutputDestination>
```

The syntax for specifying the valid data, the input source, and the output destination is as follows:

```
<ValidData> ::= 'any' | <Digits> | <Range> | <DataClass>
<DataClass> ::= <CharClass> | <IntegerClass> | <FloatClass>
<InputSource> ::= 'parameter' | <EventType>
<OutputDestination> ::= 'return' | 'screen' | 'file' | 'device'
```

Examples of input and output definitions: The following two input examples are taken from the behavioral specification section of a sequence generator component (available in the enclosed CD.) The sequence generator can be initialized to start from a specific seed, defined as input Seed. It can count upward or downward depending on the value of a direction flag, defined as input Flag. Seed and Flag are passed to the sequence generator when it is instantiated.

```
input Seed = { "seed", int, any, any, parameter };
input Flag = { "direction", int, 1, [1, -1], parameter };
```

The following example declares PIN as an input of any positive integer of 4 digits that can be passed as a parameter:

```
input PIN = { "PIN #", +int, 4, any, parameter };
```

The following two examples declare NextNumber to be either 1 or 2 digits in length and AccountID to be a reference to an object of type AccountsDatabase. The returned values of NextNumber should be within the 0..99 range. The last part in the

two outputs says that the output is expected to be returned in the name of one of the interfaces (for example, next = SequenceGenerator.Next();)

```
output NextNumber = { "next sequence number", unsigned short int, 1..2, 0..99, return };
output AccountID = { "Account ID", reference, any, AccountsDatabase, return };
```

6.1.3.1 Specifying Input Constraints

Input constraints provide the wrapper specification writer with a mechanism to express restrictions on certain types of inputs that could be required by candidate components. The specification matchers (for example, wrapper generators) can use such constraints to select the best-fit components from the candidate ones. The symbols to express such constraints (shown below) follow the input name and are enclosed between parentheses. Section 12.5.3 describes the use of input constraints in component evaluation and in wrapper generation.

- "Forbidden" inputs are designated by "-". For example, the following is interpreted as "No component should request its users to supply it with their Social Security Numbers":

```
input SSN(-) = { "SSN", +int, 9, any, parameter };
```

- "I would rather not supply" type of inputs are designated by "*", followed by a criticality number (lower criticality numbers are interpreted as a higher objection to supplying the input.) For example, the following two input declarations state

that if a component requires the user to supply either a Social Security Number or a PIN number, then the user would rather supply the PIN number and not the SSN.

```
input SSN(*1) = { "SSN", +int, 9, any, parameter };
input PIN(*9) = { "PIN #", +int, 4, any, parameter };
```

- "OK" inputs are those that users have no objection on supplying them to components. Such inputs are designated by O, or are left without any designator. For example:

```
input SSN(o) = { "SSN", +int, 9, any, parameter };
input PIN = { "PIN #", +int, 4, any, parameter };
```

6.1.3.2 Specifying Output Constraints

Wrapper specifications can also include constraints on expected outputs from candidate components. Output constraints provide a mechanism for specifying the relative importance of expected outputs to the specification writer. Section 12.5.4 describes the use of output constraints in component evaluation and in wrapper generation.

- Any "forbidden" output is designated by "-". For example, the following is interpreted as "The wrapper generator should immediately reject any component that has an interface that returns an object of type BankAccount."

```
output BankAccountObject(-) = { "Bank Account Object", object, this, return };
```

- "Must have" outputs are designated by "*". For example, the following is interpreted as "At least one interface must return a pointer to the BankAccount object."

```
output BankAccountObject(*) = { "Bank Account Object", object, this, return }
```

- Important outputs are designated by "+" followed by an importance rank (lower numbers mean higher importance.) For example, the following two output declarations state that an interface that returns a reference to the AccountsDatabase is more important (to the development organization) than an interface that returns an object of type BankAccount.

```
output BankAccountObject(+2) = { "Bank Account Object", object, this, return };
output AccountID(+1) = { "Account ID", reference, any, AccountsDatabase, return };
```

- Optional outputs are designated by "O" followed by an importance rank similar to that of important outputs. For example, the following output declarations say that an interface that returns an object of type BankAccount object is more preferable than an interface that returns a reference to the AccountsDatabase.

```
output BankAccountObject(O2) = { "Bank Account Object", object, this, return }
output AccountID(O123) = { "Account ID", reference, any, AccountsDatabase, return };
```

6.1.4 Interface Behaviors

CSL and WSL support three ways of specifying the details of the behavioral part of an interface. These are the `new` and `call` clauses and function calls as can be seen from the following syntax:

```
<BehaviorDeclaration> ::= <New> | <Call> | <FunctionDeclaration>
```

`new` gives a way to express an instantiation of the component in which the interface is defined. This is similar to `new` in `C` and in many other languages or to a constructor in `C++`. Arguments can be passed to `new`. The body of `new` can contain any combination of variable declarations or behavioral assertions.

```
<New> ::= 'new' [ <Arguments> ] ':' '{' <NewAssertionsGroup>* '}'
<NewAssertionsGroup> ::= <VariableDeclaration> | <BehavioralAssertion>
```

The following is an example of a `new` statement for a sequence generator component. The example states that during the instantiation of the component, new will be called with two arguments that were defined using the `<InputDeclaration>` construct. The first argument, Seed, is the initial sequence number (the seed.). The second argument, Flag, is the direction flag that will be used to count upward or downward. new is expected to return an object reference to the sequence generator instance. The first input will be stored in the global constant, S. The second output will be stored in the global constant, D. These global constants are defined before new is called and will be accessed by other interfaces in the component. Section 6.1.1 presented the constant dec-

larations of S and D, and Section 6.1.3 presented the input declarations of Seed and Flag.

```
new (Seed, Flag): { S: int = Seed; D: input(Flag) = Flag; return (O1) == object(this); }
```

The `call` construct expresses what happened in a call to the interface. A call can be parameterized. Each call can be designated by a call number (if the call sequence is known), or by the asterisk "*" or the letter `N` to mean any call number. If `N` is used as the call number, then `N` can be referenced in the `call` body. The body of `call` will contain any initial declaration followed by one or more behavioral assertions. All the assertions return either `true` or `false`. Assignment statements, by convention, will always succeed (i.e., return `true`.) Function calls can only return `true` or `false`. The result of the last assertion in a loop or an if-statement will be returned as the result of the whole loop or if-statement.

```
<Call> ::= 'call' '(' <CallParameters> ')' ':'
      '{' <CallAssertionsGroup> '}'

<CallParameters> ::= ( <PositiveInteger> | '*' | 'N' )
      [ ',' <Arguments> ]

<CallAssertionsGroup> ::= [ <VariableDeclaration>* ]
      <BehavioralAssertion> ( '&&' <BehavioralAssertion> )* ';'

<BehavioralAssertion> ::= <ConditionalExpression> | <Assignment> |
      <FunctionCall> | <ForEvaluator> | <IfEvaluator> |
      <ReturnEvaluator>

<FunctionCall> ::= <Name> [ <Arguments> ]
```

The body of `new` and `call` can contain any number of `<BehavioralAssertion>` constructs. Every assertion under this construct must return either `true` or `false`. The syntax for `<ConditionalExpression>` and `<Assignment>` is the same as that defined in the ADL language [OBAY98, SUNT98] and will not be repeated here. Function calls could refer to *dummy* functions or inline functions. Dummy functions are placeholders for operations performed by servers, agents, databases, etc. or performed on lists, stacks, queues, etc. Inline functions are functions defined within the same interface or module specification.

For-loops have the same structure as that defined in many languages, such as `C`, `C++`, and `Perl`. However, the body of the loop must have the syntax of `<CallAssertionsGroup>`. The syntax of `<ForInit>`, `<Expression>`, and `<ForUpdate>` is the same as that in ADL. The syntax of the for-loops is as follows:

```
<ForEvaluator> ::= 'for' '(' [ <ForInit> ] ';' [ <Expression> ] ';'
      [ <ForUpdate> ] ')' '{' <CallAssertionsGroup> '}'
```

The syntax for an `<IfEvaluator>` is similar to if-statements in many programming and specification languages. The only difference is that the body of the if-evaluator must be a `<CallAssertionsGroup>`. The syntax for the `<IfEvaluator>` is as follows.

```
<IfEvaluator> ::= 'if' '(' <Expression> ')' <CallAssertionsGroup>
[ 'else' <CallAssertionsGroup> ]
```

The <ReturnEvaluator> is used to evaluate results of returned interface calls. It has the following syntax:

```
<ReturnEvaluator> ::= 'return' '(' <Identifier> ')'
      <ReturnExpression> | '(' <Name> '=' 'return'
      '(' <Identifier> ')' ')' <ReturnExpression>

<ReturnExpression> ::= ( '==' | '!=' ) ( <InExpression> |
      <ObjectType> | <RelationalExpression> )

<InExpression> ::= 'in' `(` <DataClass> `)'
```

Expressions to be tested by a `return` evaluator can be expressed as a list of some valid data items using the <InExpression> construct, as an object of a certain type, or as any relational expression. The syntax for <RelationalExpression> is the same as that in ADL. The <Identifier> in the <ReturnEvaluator> construct must be an output name.

The following two `call` expressions are part of the specification of a sequence number generator that can generate sequence numbers upwardly or downwardly depending on a direction flag, D. The direction flag, D, and the initial seed, S, will be passed to the sequence generator when it is instantiated using the `new` expression. The following expression says that the first call should return a value with the same characteristics as that defined by output NextNumber. Further, the returned value's content must match the value saved in S.

```
call (1): { return (NextNumber) == S; }
```

The following expression defines the result for subsequent calls. If D is positive, then it should return the next (larger) number. Otherwise, it should return the next (smaller) number.

```
call (N): {
        if (D == 1) { return (NextNumber) == S+N-1; }
        else if (D == -1) { return (NextNumber) == S-N+1; }
}
```

Inline functions are defined similar to other programming and specification languages with the restriction that functions that are called from a `call` expression must be declared as `boolean` (i.e., return either `true` or `false`.) The following is the syntax for inline function declarations. The `<FormalParameters>` construct has the same syntax as that defined in ADL.

```
<FunctionDeclaration> ::= <Type> <Name> ['(' <FormalParameters> ')']
      '{' <BehavioralSpecification> '}'
```

For example, the following function receives two lists of integers, list1 and list2 of size S elements, and returns `true` if the elements in both lists match or `false`, otherwise.

```
boolean sameElements (list1: list(int, S), list2: list(int, S))
{
        i: int, j: int;

        for (i=0; i<S; i++) {
                found: boolean = false;

                for (j=0; j<S; j++) {
                        if (list1[i] == list2[j]) { found = true; }
                }
                if (!found) { return (false); }
        }

        return (true);
}
```

6.1.5 Interface Stimuli

The last part on the set of constructs that specify interface behaviors is the definition of stimuli that can cause an interface to be active. The following are three cases that can stimulate an interface:

- Case 1: The first case is when the interface receives a direct call from another interface. The interface will usually be inactive waiting for calls from other interfaces. The interface will get stimulated (i.e., becomes active) when it receives a call from another interface.

- Case 2: A state change occurs. In this case, the interface is already active. The interface might be in a loop or a wait state waiting for:

- Another interface to finish
- A specific time to come
- A global variable to be created or modified (possibly to a certain value)
- An environment variable to be created or modified (possibly to a certain value)
- An object to be created or modified (possibly to have certain properties)
- A file to be created or modified (possibly to contain certain information)
- The available memory to go below or above a certain value
- The available disk space to go below or above a certain value
- A process or thread to be created or terminated
- A certain device to go on or off
- A loop counter to reach a specific value. This is similar to modifying a global variable to a certain value.
- A certain event to occur

- Case 3: An event occurs. The interface is in idle, wait, or sleep state. It will be stimulated by the occurrence of any or specific event.

The above activation scenarios are captured in the following syntax.

```
<StimulusDeclaration> :: 'stimuli' '=' '{' <StimulusChoice>
      ( ',' <StimulusChoice> )* '}'
```

```
<StimulusChoice> ::= <CallStimulus> | <EventStimulus> |
      <TimeStimulus> | <ObjectStimulus> | <SpaceStimulus> |
      <DeviceStimulus>

<CallStimulus> ::= 'call' [ '(' <Name> ')' ]
<EventStimulus> ::= <EventType>
<TimeStimulus> ::= 'time' '(' [ <Date> ]<Time> ')'

<ObjectStimulus> ::= 'object' '(' <Name> ',' <StimulusObjectType>
      <StimulusStateMode> [ ',' <Data> ] ')'

<StimulusObjectType> ::= 'global' | 'process' | 'thread' | 'file' |
      'environment' | 'interface' | 'object'

<StimulusStateMode> ::= 'created' | 'modified' | 'vanished'

<SpaceStimulus> ::= 'space' '(' ( 'memory' | 'storage' )
      <ComparisonOperator> <PositiveFloat> <SpaceDesignator> ')'

<DeviceStimulus> ::= 'device' '(' <Identifier> <DeviceMode> ')'
<DeviceMode> ::= 'ON' | 'OFF' | 'IDLE' | 'BUSY'
```

An example of a stimuli definition is shown below. This example is interpreted as follows: the interface in which this expression is defined will become active if called from another interface. If the user selects the "File/New" menu item or presses the Control-N on the keyboard, then that will also activate this interface. The `stimuli` expression restricts the events that can stimulate the interface to those listed stimuli. All other stimuli will be ignored.

```
stimuli = {
        call,
        event (menu ("File/New"), keyboard ('Ctrl-N'))
}
```

6.2 A Complete Behavioral Specification Example

Shown in Figure 8 is the *behavioral* specification for a bank module. This module will usually be part of a complete behavioral and non-behavioral specification of a compositional wrapper. This module contains all the necessary interfaces needed to manage bank accounts. The compositional wrapper that will contain the interfaces defined by this module will be responsible for maintaining the integrity of all the bank accounts. Users will be able to deposit, withdraw, or get balance accounts. Bank representatives are the only ones allowed to open new accounts, generate reports of all accounts, or close accounts. The compositional wrapper should not allow withdrawal if the withdrawn amount exceeds the current balance. The module is only used to group interfaces into one entity. The module can have its own global variables and function declarations. At implementation time, the defined interfaces could come from a single component or from multiple ones.

Some of the called interfaces, such as exists or insert, are not actual function invocations, but rather dummy ones or placeholders for operations that are performed by servers, databases, operating systems, or acted upon on lists, stacks, queues, etc. Each dummy function says that the interface in which the dummy function is defined must do the function calls as part of the interface's internal behavior. The dummy function calls are used in the wrapper specification to make the sequencing of actions logically correct.

Figure 7 shows how the matching can be performed between the specification of the MyBank compositional wrapper and the specifications of candidate components. The right side of this figure shows the interfaces that are present in the *MyBank* specification. The left side of this figure shows the interfaces of two components whose union of pro-

vided interfaces matches the set of interfaces listed in the *MyBank* wrapper specification. The arrows show the matched interfaces.

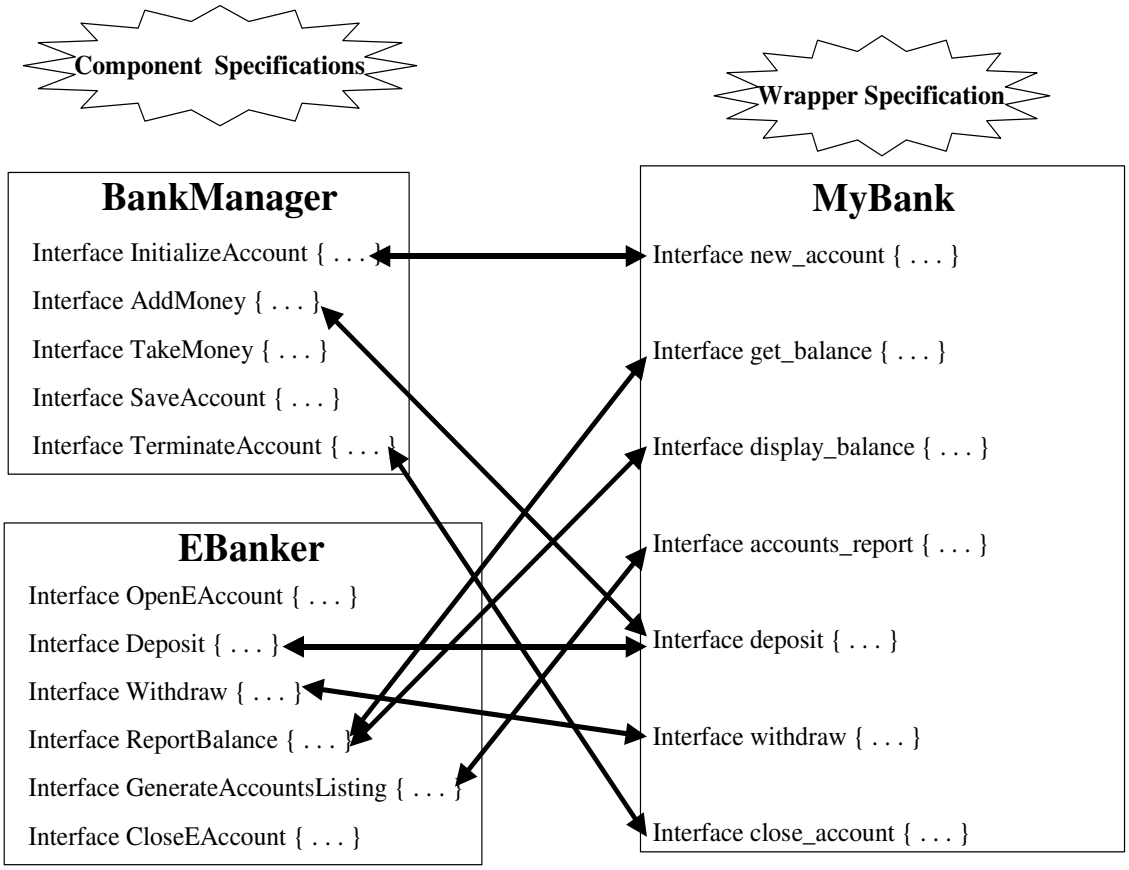

Figure 7: Matching the specification for the MyBank compositional wrapper with the specifications of the two commercial components

A few remarks about the objects in this figure are:

- It took two components to satisfy the complete behavioral specification of the compositional wrapper. In some cases, more than two component specifications would be needed to satisfy one wrapper specification.

- The specification matcher matches the behavioral parts of the interfaces in the wrapper specification and the components specifications without any regards to interface names. For example, Figure 7 shows that interface *deposit* in the wrapper specification matches an interface with the same name in component *EBanker* and another with a different name (*AddMoney*) in component *BankManager*.

- Some of the component interfaces will not match any interface from the wrapper specification (for example, interfaces *TakeMoney* and *SaveAccount* in component *BankManager* and interfaces *OpenAccount* and *CloseAccount* in component *EBanker*.) The compositional wrapper of Figure 7 will not make use of such interfaces. However, a different wrapper specification could take advantage of such unused interfaces.

- More than one interface from the wrapper specification (for example, *get_balance* and *display_balance*) could match the same interface specification in one of the components (for example, *ReportBalance* in component *EBanker*.) This could happen when the component interface performs multiple functions depending on a parameter that will be passed to it.

```
Module MyBank {
  // ATM is a device (ATM machine) whose information can be found in
  // the "ATM.dvi" file.
  ATM: device ("ATM.dvi");

  Address: structure { first_line: string, second_line: string,
           city: string, state: string[2], zipcode: string[5] };

  Bank: structure { name: string, branch: string, address: Address,
        phone: string, email: string };
```

```
// Employee is a structure that holds the bank employee's personal
// information
Employee: structure { name: string, ssn: string, address: Address,
          phone: string, email: string, bank: Bank,
          salary: +float, join_date: date };

// Customer is a structure that holds the new customer's personal
// information
Customer: structure { name: string, ssn: string, address: Address,
          phone: string, email: string };

// This structure holds the account information
Account: structure { bank: Bank, customer: Customer, number: string,
         pin: int[4], balance: int };

// This input can have any positive integer that is 9 digits long
// and will be passed as a parameter
input RepresentativeID = { "Bank representative SSN", +int, 9, any,
      parameter };

// This input will be any positive integer of 4 digits and will be
// passed as a parameter
input PIN = { "PIN #", +int, 4, any, parameter };

// This defines a database whose records are of type Employee
EmployeesDatabase: database (Employee);

// This is a database of all bank accounts
AccountsDatabase: database (Account);

// This is a global reference to a row ID in the AccountsDatabase
accountID: AccountsDatabase(row);

// This output should be an object reference to the BankAccount
// component
output BankAccountObject = { "Bank Account Object", object, this,
       any, return }

interface new_account {

  // This input will be a structure of type Account and will be
  // passed as a parameter
  input Account = { "Account information", Account, any, any,
        parameter };

  // The output should be a reference to a database object of type
  // AccountsDatabase.
  // I expect the output to be returned in the function name
  output AccountID = { "Account ID", reference, any,
         AccountsDatabase, return };

  // This interface should not generate any other outputs
  output = { none };
```

```
behavior = {
  call (*, RepresentativeID, Account, PIN): {

    // "exists" is a dummy database operation.
    // It will compare the "ssn" column in every record
    // of database EmployeesDatabase to see if it contains
    // a record with the same value as that of
    // RepresentativeID.  This check is not necessary for
    // the wrapper specification, but it is here to tell
    // the component that it should make sure to make such
    // check

    exists (RepresentativeID, EmployeesDatabase (ssn)) &&

    // "insert" is a database operation.  This tells the
    // database to insert a new record
    // with the Account information and the supplied PIN
    // number and to return a reference
    // to the row ID of the new record or null, otherwise.

    (accountID = return (AccountID)) ==
      insert (AccountsDatabase, Account, PIN) &&

    // It makes sure the returned accountID is not null

    accountID <> null &&

    // And since accountID is a reference to a
    // database(Account), it makes sure
    // that one of the row IDs of database
    // AccountsDatabase is the same as accountID

    exists (accountID, AccountsDatabase) &&

    // Finally, it makes sure that the accounted
    // references a record with the same
    // information as that of Account (including bank,
    // customer, and pin# information)

    sameCustomer (accountID);
  }

  boolean sameCustomer
   (accountID: reference(database(Account))) {

    sameCustomer =
      exists (input(Account), accountID->account) &&
      exists (input(Customer), accountID->customer) &&
      exists (input(PIN), accountID->pin);
  }
}
```

```
    stimuli = {
      call,
      event (menu ("File/New"), keyboard ('Ctrl-N'))
    }
  }

  interface get_balance {

    // The output defines the returned balance which should be any
    // floating point number
    // I expect the output to be returned in the function name
    output Balance = { "Current balance", float, any, any, return };

    // It should not generate any other outputs
    output = { none };

    behavior = {
      call (*, accountID, PIN): {

        // Make sure the supplied accountID is not null and
        // is a reference to Account

        accountID <> null &&
        accountID == reference(Account) &&

        // Make sure the supplied PIN# is the same as that of
        // accountID

        accountID->pin == PIN &&

        // This will make sure the supplied accountID exists
        // in the AccountsDatabase

        exists (accountID, AccountsDatabase) &&

        // This tells the database AccountsDatabase to get
        // the current value in the
        // balance column of the accountID record.  The
        // returned value should match output Balance

        return (Balance) == fetch (accountID (balance)) &&
      }
    }

    stimuli = {
      call
    }
  }

  interface display_balance {

    // This output is defined as a printout that will be sent to the
    // screen
```

```
    output Balance = { "Current balance", printout (float, screen,
           {pos=10,20; color=red}), any, screen };

    // It should not generate any other outputs
    output = { none };

    behavior = {
      call (*, accountID, PIN): {

        // Make sure the supplied accountID is not null and
        // is a reference to Account

        accountID <> null &&
        accountID == reference(Account) &&

        // Make sure the supplied PIN# is the same as that
        // of accountID

        accountID->pin == PIN &&

        // This will make sure the supplied accountID exists
        // in the AccountsDatabase

        exists (accountID, AccountsDatabase) &&

        // This says that the returned value must match that
        // returned by the get_balance interface.
        // The returned value should have already be displayed
        // in the screen at position 10,20 and
        // with the "red" color.

        return (Balance) == get_balance (accountID, PIN) &&
      }
    }
    stimuli = {
      call,
      event (menu ("Tools/Balance"))
    }
  }

  interface accounts_report {

    // Report defines the formating string that will applied to each
    // record returned from the database
    Report: string = "%20d\t%12s\t\%30s\t%20.2f\n";

    // This defines a variable of type "file" whose name is
    // "accounts.rpt"
    DestinationFile: file ("accounts.rpt");

    // This is the name of the destination file that will passed to
    // the accounts_report interface.
    input Destination = { "Destination File", DestinationFile, null,
```

```
          any, parameter };

    // This output is defined as a file whose records are formated as
    // in Report.  The output will be sent to a file.
    output ReportFile = { "Accounts Report", file(Report), any, any,
           file };

    // It should not generate any other outputs
    output = { none };

    behavior = {
      call (*, RepresentativeID, Destination): {

        exists (RepresentativeID, EmployeesDatabase (ssn)) &&

        // This says that the returned output must match
        // that of ReportFile

        return (ReportFile) == true &&

        // "exists" here is a file operation that checks if
        // the supplied file handle exists

        exists (DestinationFile) &&

        matchedFormat (DestinationFile);
      }

      boolean matchedFormat (file: file(string)) {
        while ((line = read(file))) {
          if (line !~ Report) {
            return (false);
          }
        }
        return (true);
      }
    }
    stimuli = {
      call,
      event (menu ("Reports/All Accounts"), time(every 12 hours))
    }
  }

  interface deposit {
    balance: float;

    // This defines a list of hardware items that will be used in
    // making deposits
    DepositItems: hardware (money, check, coin);

    // This defines the receipt that the customer will get when
    // making a deposit
    Receipt: hardware (paper);
```

```
// This input will be any positive floating point number and will
// be passed as a parameter
input Amount = { "Deposited amount", +float, any, any,
      parameter };

input Deposit = { "Deposited items", DepositItems, any, any,
       device };

// This output will be in the form of a printed receipt
output Receipt = { "Printed Receipt", Receipt, any, any, device };

// The output should be true.
// I expect the output to be returned in the function name
output O1 = { "success or failure of operation", boolean, null,
       true, return };

// It should not generate any other outputs
output = { none };

behavior = {
  call (*, accountID, PIN, Amount): {

    // Make sure the supplied accountID is not null and
    // is a reference to Account

    accountID <> null &&
    accountID == reference(Account) &&

    // Make sure the supplied PIN# is the same as that
    // of accountID

    accountID->pin == PIN &&

    // This will make sure the supplied accountID exists
    // in the AccountsDatabase

    exists (accountID, AccountsDatabase) &&

    // This will save the balance before the deposit
    // is made

    balance = get_balance (accountID) &&

    // This asks the ATM machine to accept the deposited
    // items from its deposit slot
    // and to return "true" if successful or "false",
    // otherwise.  Assume the ATM has the intelligence
    // to count total amount of the deposited items, it
    // will then return true if that total amount matches
    // Amount.  "accept" is a special method defined
    // just for this device in its device definition file.
```

```
        return (O1) == ATM.accept (Deposit, Amount) &&

        // This tells the database AccountsDatabase to update
        // the balance column of the
        // accountID record with the value (balance+Amount).
        // The update operation should be successful

        return (O1) ==
          update (accountID (balance), balance+Amount) &&

        // Now make sure the balance actually reflects
        // the new deposit

        get_balance (accountID) == balance+Amount &&

        // This requests from the ATM machine to print a
        // receipt for the user's deposit

        return (O1) == ATM.printReceipt (deposit);
      }
    }
    stimuli = {
      call,
      event (menu ("Tools/Deposit"), keyboard ('Ctrl-D'))
    }
  }

  interface withdraw {
    balance: float;

    // This defines a list of hardware items that will dispensed at
    // withdrawals
    DispensedItems: hardware (money);

    // This defines the receipt that the customer will get when
    // making a deposit
    Receipt: hardware (paper);

    // This input will be any positive floating point number and will
    // be passed as a parameter
    input Amount = { "Withdrawn amount", +float, any, any,
          parameter };

    output Cash = { "Dispensed items", DispensedItems, any, any,
         device };

    // This output will be in the form of a printed receipt
    output Receipt = { "Printed Receipt", Receipt, any, any, device };

    // The output should be true.
    // I expect the output to be returned in the function name
    output O1 = { "success or failure of operation", boolean, null,
           true, return };
```

```
// It should not generate any other outputs
output = { none };

behavior = {
  call (*, accountID, PIN, Amount): {

    // Make sure the supplied accountID is not null and
    // is a reference to Account

    accountID <> null &&
    accountID == reference(Account) &&

    // Make sure the supplied PIN# is the same as that
    // of accountID

    accountID->pin == PIN &&

    // This will make sure the supplied accounted
    // exists in the AccountsDatabase

    exists (accountID, AccountsDatabase) &&

    // This will save the balance before the deposit
    // is made

    balance = get_balance (accountID) &&

    // Make sure there is enough balance for the
    // withdrawn amount

    balance >= Amount &&

    // This asks the ATM machine to dispense withdrawn
    // money through its money dispenser slot
    // and to return "true" if successful or "false",
    // otherwise.  It will return successful
    // only if the dispensed amount equals Amount.
    // "dispense" is a special method defined just
    // for this device in its device definition file.

    return (O1) == ATM.dispense (Cash, Amount) &&

    // This tells the database AccountsDatabase to
    // update the balance column of the
    // accountID record with the value (balance-Amount).
    // The update operation should be successful

    return (O1) == update (accountID (balance),
       balance-Amount) &&

    // Now make sure the balance actually reflects
    // the new withdrawal
```

```
      get_balance (accountID) == balance-Amount &&

      // This requests from the ATM machine to print
      // a receipt for the user's deposit

      return (O1) == ATM.printReceipt (deposit);
    }
  }
  stimuli = {
    call,
    event (menu ("Tools/Withdraw"), keyboard ('Ctrl-W'))
  }
}

interface close_account {

  // The output should be true.
  // I expect the output to be returned in the function name
  output O1 = { "success or failure of operation", boolean, null,
        true, return };

  // It should not generate any other outputs
  output = { none };

  behavior = {
    call (*, RepresentativeID, accountID): {

      // Make sure the supplied accountID is not null
      // and is a reference to Account

      accountID <> null &&
      accountID == reference(Account) &&

      // This will make sure the supplied accountID exists
      // in the AccountsDatabase

      exists (accountID, AccountsDatabase) &&

      // This will make sure the RepresentativeID is one
      // of the "ssn" entries in the
      // EmployeesDatabase database

      exists (RepresentativeID, EmployeesDatabase (ssn)) &&

      // This tells the database AccountsDatabase to delete
      // the accountID record
      // and the delete operation should be successful

      return (O1) == delete(accountID, AccountsDatabase) &&

      // accountID should not exist anymore in the
      // AccountsDatabase
```

```
        !exists (accountID, AccountsDatabase) &&
      }
    }
    stimuli = {
      call,
      event (menu ("Tools/Close"))
    }
  }
}
```

Figure 8: The complete WSL behavioral specification for a compositional bank wrapper

CHAPTER 7: THE COMPONENT SPECIFICATION LANGUAGE

In addition to the constructs that are presented in Chapter 5 and Chapter 6, this chapter discusses the additional constructs that compose the Component Specification Language (CSL). The complete set of CSL constructs allow component suppliers to specify the syntax and semantic of their components in a way that can be understood by wrapper generators.

Pre-built software components usually reside either in central repositories owned by their original producers or by third-party suppliers. The goal of presenting CSL in this dissertation is to allow a wrapper generator to be able to read and parse the specifications of such components to facilitate searches through component repositories. Specially designed software agents can download component specifications from such repositories, or system integrators can download the specifications manually from their original sources.

7.1 The Component Specification Constructs

At a minimum, a CSL specification document must contain the specification section that describes the main component. The syntax for the component specification section looks like the following:

```
<ComponentSpecification> ::= 'component' <Identifier>
      '{' <ComponentBody> '}'

<ComponentBody> ::= <ComponentDeclarationUnit>+
```

The <ComponentBody> construct can contain one or more <ComponentDeclarationUnit> constructs that can show up in any order. However, each <ComponentDeclarationUnit> construct may appear only once in each <ComponentSpecification> section. Some of the constructs are required; others are optional.

The syntax for the <ComponentDeclarationUnit> is shown below. The syntax for the <NameDeclaration> and <DescriptionDeclaration> constructs are given in Section 5.3.4. The <NameDeclaration> construct gives the popular name of the component. It must be present in each <ComponentSpecification> section. Only one occurrence of this clause can appear in a <ComponentSpecification>. The required <DescriptionDeclaration> construct will be beneficial to the component user to assist in the decision of whether to consider acquiring a component.

```
<ComponentDeclarationUnit> ::= <NameDeclaration> |
      <DescriptionDeclaration> | <ProviderDeclaration> |
      <VersionDeclaration> | <TechnologyDeclaration> |
      <OperatingSystemsDeclaration> | <DatabasesDeclaration> |
      <LanguageDeclaration> | <FormatsDeclaration> |
      <LicenseModelsDeclaration> | <LicenseTypesDeclaration> |
      <CostDeclaration> | <SupportMethodsDeclaration> |
      <NewUpdatesDeclaration> | <DownloadsDeclaration> |
      <InstallationDeclaration> | <ConfigurationDeclaration> |
      <ProvidesDeclaration> | <RequiresDeclaration> |
      <SoftwareConstraintsDeclaration> |
      <HardwareConstraintsDeclaration> |
      <NetworkConstraintsDeclaration> | <OptionalDeclaration>
```

The following subsections will discuss these constructs in more details.

7.1.1 The Component Provider

```
<ProviderDeclaration> ::= 'provider' '=' <Identifier> ';'
```

Each component specification must contain a <ProviderDeclaration> expression to list the name of the <ProviderSpecification> section (see Section 7.2) where the detailed vendor's information is given. There must be one <ProviderSpecification> section somewhere in the specification document whose name matches the <Identifier> in the above <ProviderDeclaration> construct. Multiple components defined in the same specification document can refer to the same provider section.

7.1.2 The Component Version

```
<VersionDeclaration> ::= 'version' '=' <VersionString> ';'

<VersionString> ::=
      <PositiveInteger>'.'<PositiveInteger>['.'<PositiveInteger> ]
```

A *version* clause must be present in each component specification. A version number can consist of three parts. The first part identifies the major revision number.

The second part identifies the minor revision number. The (optional) third part tells about the current service level. For example, the specification can state the component's version either as version = 1.0; or version = 2.5.3;.

7.1.3 Used Technology

```
<TechnologyDeclaration> ::= 'technology' '=' <TechnologyChoice> ';'

<TechnologyChoice> ::= 'COM' | 'CORBA' | 'DCOM' |
      'EJB' | 'J2EE' | 'JavaBean' | 'MTS' | '.NET'
```

These constructs specify the underlying technology of the component. There must be only one <TechnologyChoice> listed in the *technology* clause and only one *technology* clause in each component specification. If the component supports multiple technologies, then the vendor must create a separate specification document for each technology.

7.1.4 Supported Operating Systems

```
<OperatingSystemsDeclaration> ::= 'operating_systems' '='
      '(' ( 'ALL' | ( <OperatingSystemsChoice>
      ( ',' <OperatingSystemsChoice> )* ) ) ')' ';'

<OperatingSystemsChoice> ::= 'WINNT4.0' | 'WIN2000' 'WINXP' |
      'MACINTOSH' |'LINUX2.4.7' | 'SOLARIS7' | 'SOLARIS8'
```

The *operating_systems* clause lists the operating systems in which the component can operate properly. There must be one *operating_systems* clause per component specification. Component vendors can list one or more of the operating system keywords (for example, operating_systems = (Macintosh, WinNT4.0, SOLARIS7);.) A catch-all keyword `ALL` is also provided to be used if a component is so general that no operating system restriction applies.

7.1.5 Supported Storage Databases

```
<DatabasesDeclaration> ::= 'databases' '=' '(' ( 'ALL' | 'NONE' |
      ( <DatabaseChoice> ( ',' <DatabaseChoice> )* ) ) ')' ';'

<DatabaseChoice> ::= 'FILE' | 'MYSQL' | 'ACCESS' |
      'MSSQLSERVER' | 'ORACLE'
```

The `<DatabasesDeclaration>` construct allows component suppliers to list the storage formats supported by their components (for example, databases = (File, MySQL);.) The `NONE` keyword can be used if the component does not do any storage activities. If the component can support any possible storage format, then the `ALL` keyword can be used. (The `FILE` keyword refers to the file system within the supported operating systems.)

7.1.6 The Development Language

```
<LanguageDeclaration> ::= 'language' '=' <LanguageChoice> ';'

<LanguageChoice> ::= 'C' | 'C++' | 'JAVA' |
      'PAYTHON' | 'PERL' | 'PHP' | 'TCTKL'
```

These constructs give the development language in which the component was written. (While this construct has no significance to the wrapper generation process, it could be useful to the development team during the component evaluation process.) There can only be one development language specified per component specification (for example, language = Java;.)

7.1.7 The Component Formats

```
<FormatsDeclaration> ::= 'formats' '=' '(' <FormatChoice>
      ( ',' <FormatChoice> )* ')' ';'

<FormatChoice> ::= 'SOURCE' | 'OBJECT' | 'EXE'
```

The required <FormatsDeclaration> construct tells the wrapper generator the supported formats of the main component files that are listed in the *downloads* clause (see Section 7.1.13.) A component can be supplied in any format. (For example, formats = (SOURCE, OBJECT, EXE);). The *downloads* clause can include all the supported formats. The generator needs to know the component format(s) so that compilation and wrapper generation decisions can be made properly.

7.1.8 Available License Models

```
<LicenseModelsDeclaration> ::= 'license_models' '='
      '(' <LicenseModelChoice> ( ',' <LicenseModelChoice> )* ')' ';'

<LicenseModelChoice> ::= 'FREE' | 'DEMO' | 'SHAREWARE' | 'PER_USE' |
      'PER_USER' | 'TIMED' | 'UNLIMITED'
```

License models specify the vendors' supported mechanisms for charging for their components. A vendor can specify one or more license models per *license_models* clause. (For example, license_models = (FREE, DEMO);.) Component users will decide on what of these models to use.

7.1.9 Licensing Types

```
<LicenseTypesDeclaration> ::= 'license_types' '=' '('
      <LicenseTypeChoice> ( ',' <LicenseTypeChoice> )* ')' ';'

<LicenseTypeChoice> ::= 'SINGLE' | 'SITE' | 'DEVELOPER'
```

Vendors can offer more than one licensing option. `SINGLE` licenses can be acquired for individual users or single workstations. Companies and organizations may prefer to obtain `SITE` licenses instead of `SINGLE` licenses. Component vendors may offer special versions of their components to developers through their `DEVELOPER` licenses. Any combination of these licensing options can appear within the *license_types* clause. (For example, license_types = (SINGLE, SITE);.)

7.1.10 The Component Cost

```
<CostDeclaration> ::= <LicenseTypeChoice>'.'<CostType>'Cost'
      '=' <CurrencySymbol> <PositiveFloat> ';'

<CostType> ::= 'License' | 'Upgrade' | 'Maintenance' | 'Support' |
      'Training' | 'Installation'

<CurrencySymbol> ::= 'R.O.' | '$' | '£' | '¥'
```

There are six different types of component costs defined by the component specification language: (1) the cost of the basic component license (`LicenseCost`), (2) the cost of upgrades (`UpgradeCost`), (3) the maintenance cost (`MaintenanceCost`), (4) the support cost (`SupportCost`), (5) the training cost (`TrainingCost`), and (6) the installation cost (`InstallationCost`). Each of these cost types must be prefixed by the license type (`SINGLE`, `SITE`, or `DEVELOPER`) and be followed by a period. (For example, Single.LicenseCost = $100;.) None of these cost types are required. However, the presence of a cost type can aid in the component evaluation process. The component vendor can specify any combination of cost types. Each of these expressions can appear at most once within each component specification.

7.1.11 Available Support Methods

```
<SupportMethodsDeclaration> ::= 'support_methods' '=' '('
      <SupportMethodChoice> ( ',' <SupportMethodChoice> )* ')' ';'

<SupportMethodChoice> ::= 'ONSITE' | 'ONLINE' | 'TELEPHONE' | 'EMAIL' |
```

```
'FORUM' | 'SUPPLIEDDOC' | 'ONLINEDOC'
```

The *support_methods* clause allows a vendor to list the type of support offered to their component users. It is not required to specify this clause. However, if present, it can be used during the component evaluation process. Multiple support options can be listed. (For example, support_methods = (SUPPLIEDDOC, EMAIL, FORUM);).

7.1.12 Specification of New Updates

```
<NewUpdatesDeclaration> ::= 'new_updates' '=' <Url> ';'
```

This construct helps the wrapper generator check for new updates using the <Update_ComponentsAutoUpdate> feature (discussed in Section 8.2.9.6.) The construct requires a valid URL that points to a CSL specification document that contains the description of the new update. When checking for new updates, the wrapper generator compares the date on the current update specification file to that of the previously downloaded update specification (if one exists.) If the date is newer, then it checks the version number of the new update specification. If the version number in the new update specification is also newer than the version number in the previously downloaded update specification, then the wrapper generator performs the steps for obtaining the new updates as discussed in Section 8.2.9.6.

7.1.13 The Downloadable Artifacts

```
<DownloadsDeclaration> ::= 'downloads' '=' '(' <DownloadsList> ')' ';'

<DownloadsList> ::= <ArtifactSpecifier> '=' <Url>
      ( ',' <ArtifactSpecifier> '=' <Url> )*

<ArtifactSpecifier> ::= ['*']<Identifier> '(' ( 'ALL' |
      ( <ArtifactChoice> ( ',' <ArtifactChoice> )* ) ) ')'

<ArtifactChoice> ::= <FormatChoice> | 'DOC' | 'OTHER'
```

This complex <DownloadsDeclaration> construct can help the wrapper generator in accessing the referenced components. The wrapper generator cannot make use of the component specification if this clause is not present in it.

The <DownloadsDeclaration> construct tells the wrapper generator what files are included with the main component. Included files can be binary files (OBJECT, EXE,) source files (SOURCE,) documentation files (DOC,) or any other miscellaneous files (OTHER) used by the component. The ALL keyword can be used to indicate the inclusion of all supported artifact types. The list of downloads *must* include the file of the specified component.

One or more valid URLs can appear between the right and left parenthesis of the *downloads* clause. Each URL must be preceded by one of the supported artifact type keywords. The <Identifier> in the <ArtifactSpecifier> construct assigns a unique identifier to each file within the *downloads* clause. This unique identifier will be used during the component's installation (see Section 7.1.14.) An asterisk must precede the identifier that corresponds to the file of the specified component.

The following is an example of a *downloads* clause:

```
downloads = (
        *File1(object)="ftp://ucf.edu/CSL/bank/bankAccount.class",
        File2(object)="ftp://ucf.edu/CSL/bank/insufficientFundsExc.class",
        File3(object)="ftp://ucf.edu/CSL/bank/negativeAmountExc.class"
);
```

All the files specified by this clause will be downloaded to the temporary working directory "`$WRAPPER_TEMP/<Identifier>`" where `<Identifier>` refers to the one listed in the `<ComponentSpecification>` section of the current component. `$WRAPPER_TEMP` is the environment variable that holds the wrapper generator's temporary working directory (see Section 14.9.)

7.1.14 Installing Downloaded Artifacts

```
<InstallationDeclaration> ::= 'installation' '='
      '{' <InstallationCommand>* '}' ';'

<InstallationCommand> ::= <MkdirDeclaration> | <MoveDeclaration> |
      <RunDeclaration> | <UseDeclaration> | <OtherDeclaration>

<MkdirDeclaration> ::= 'mkdir' <Directory> ';'
<MoveDeclaration> ::= 'move' <Identifier> <Directory> ';'
<RunDeclaration> ::= 'run' <QuotedString> ';'
<UseDeclaration> ::= 'use' <QuotedString> ';'
```

The required `<InstallationDeclaration>` construct guides the wrapper generator on what to do with the downloaded artifacts. Without this construct, the generator would only download the artifacts into a temporary directory and leave them there.

Any combination of the installation options can show up within the *installation* clause, as can be seen from the following example:

```
installation = {
        mkdir   "_temp_bank_";
        move    File1 "_temp_bank_";
        run     "gunzip _temp_bank_/bank.tar.gz";
        run     "tar -xvf _temp_bank_/bank.tar";
        mkdir   "bank";
        run     "mv _temp_bank_/* bank/*";
        run     "rm -r _temp_bank_";
        use     "import bank.*;";
};
```

Some important points about the use of the installation options:

- A `<Directory>` must be a valid directory name for the target operating system. If an operating system has no formal mechanism for assuring existence or absence of the directories in their file systems, vendors cannot assume existence or absence of a directory. Vendors can, however, use the environment variables `$GENERATOR_HOME`, `$WRAPPER_TEMP`, or `$WRAPPER_INSTALL` (see Section 14.9) in their directory specifications to help them in referencing subdirectories descending from the directories specified in these three environment variables.

- The installation commands will be executed in the same order in which the commands appear in the *installation* clause. Thus, vendors need to ensure that the installation commands are listed in the correct order.

- Unless qualified fully from the root directory, directory names will be interpreted as subdirectories of the installation directory specified in the `$WRAPPER_INSTALL` environment variable.

- File identifiers within the `<MoveDeclaration>` construct refer to the corresponding file identifiers specified using the `<ArtifactSpecifier>` construct (see Section 7.1.13.)
- The `use` statement lists the code that would need to be included in host applications or wrappers. Such code would be responsible for making the host application or wrapper aware of the installed component and its interfaces.
- The `<OtherDeclaration>` construct can be used to catch all other installation instructions that cannot be enumerated using the other installation constructs. The instructions included in the *other* clause would usually be in narrative form. Thus, the wrapper generator would not be able to automate such instructions.

7.1.15 Configuring the Downloaded Component

The constructs that will be presented in the following subsections can greatly help in automating the component configuration process by guiding the wrapper generator (or the development team) on what to do with the installed components before using them. The configuration consists of a series of actions, each of whose targets are either: component files or screens within the downloaded components, or other popular applications (for example, Microsoft Office applications.)

7.1.15.1 Configuration options

```
<ConfigurationDeclaration> ::= 'configuration' '='
      '{' <ConfigurationOption>* '}' ';'

<ConfigurationOption> ::= <ConfigurationInsert> |
      <ConfigurationUpdate> | <ConfigurationDo> | <OtherDeclaration>
```

Configuring components can be a very complex activity. This research provides only a startup of the process. CSL currently allows inserting new information into a specific location in a file or a field in the screen (the `<ConfigurationInsert>` construct), updating existing information in a file or screen with new information (the `<ConfigurationUpdate>` construct), or performing a set of simple actions that work as a group (the `<ConfigurationDo>` construct). A configuration action that cannot be specified using one of the above three constructs, can be specified using the `<OtherDeclaration>` construct.

The development team will be responsible for performing the adaptation activity specified using the `<OtherDeclaration>` construct. It might be possible for the wrapper generator to automate the actions described in the other three constructs.

7.1.15.2 The *insert* Configuration Option

```
<ConfigurationInsert> ::= 'insert' '(' <QuotedString> ','
      (<ScreenName> | <FileNameWithWidlcards>)
      ['{' <GeneralFilePosition> '}'] <InfoChoice> ')' ';'

<FileNameWithWildcards> ::= <FileName> |
      <Path> /[a-zA-Z0-9\.\*\?\-\_]+/
```

```
<GeneralFilePosition> ::= ( <Digits> | '*') [ ‘,’ <FieldOrColumn> ]

<FieldOrColumn> ::= <Digits> |            // Column #
      <QuotedString>                      // Field name

<InfoChoice> ::= <Integer> | <Float> | <BooleanLiteral> |
      <YesNo> | <Url> | <Email> | <QuotedString>
```

The above set of productions describes the insertion of new information into files or screens. The <QuotedString> in the <ConfigurationInsert> production will hold either: the actual information to be inserted, or the instructions to the component user concerning the nature of the inserted information. The second argument to the insert command can either be a screen name (for example, “File/Open”) or a file name, possibly with wildcards, <FileNameWithWildcards>. The third optional argument (the <GeneralFilePosition> construct) can be used to specify the line number and, optionally, the column number for files. Alternatively, the <GeneralFilePosition> construct can be used to specify the field number in a screen. An asterisk in a <GeneralFilePosition> construct means to place the information anywhere in the file or screen. The last argument in the insert command holds the actual inserted information. Currently, CSL supports inserting the following types of information: integers, floats, Booleans (true or false), yes/no, URLs, and email addresses. If the inserted information is not one of these defined ones, then a description of it can be enclosed in double quotes using the <QuotedString> construct. The following is an example of an insert command:

```
insert ("Telling the component about your email", "customer.info", {5},
      "put your email here");
```

This says that the purpose of this insert command is to instruct the system integrator to provide an email address to the component. To do so, the system integrator must open the customer.info file and place the email address in line 5.

7.1.15.3 The *update* Configuration Option

```
<ConfigurationUpdate> ::= 'update' '(' <QuotedString> ','
      <FileNameWithWidlcards> ‘,’
      ['{' <RestrictedFilePosition> '}' ‘,’ ] <InfoChoice> ')' ';'

<RestrictedFilePosition> ::= <Digits>[ ‘,’ <FieldOrColumn> ]
```

The *update* command works in a similar manner to the *insert* command with the exception that file or screen position must be *absolute* (because the *update* command changes existing information which usually resides in a *specific* location in the file or screen.)

7.1.15.4 The *do* Configuration Option

```
<ConfigurationDo> ::= 'do' '{' <ConfigurationDoAction>+ '}' ';'

<ConfigurationDoAction> ::= <DoTargetType> <DoTarget> <DoAction>
```

The <ConfigurationDo> construct is used to enclose a set of actions that work as a group. Each action is composed of three parts: the action type (<DoTarget-

Type>), the action target (<DoTarget>), and the action itself (<DoAction>.) These parts are described as follows:

- The Action *type*:

```
<DoTargetType> ::= 'TargetType' '=' ('screen' | 'file') ';'
```

  Currently, the CSL language defines configuration action targets to be either screens or files.

- The Action *target*:

```
<DoTarget> ::= 'target' '=' ( <FileName> | <ScreenName> ) ';'
```

  Depending on what target type is selected for the action, the above production requires entering the name of the target screen or file. (Both of the <FileName> and <ScreenName> are defined in Section 5.3.5.)

- The *do action*

```
<DoAction> ::= 'action' '=' '(' [ <ActionItem>
      ( ( 'or' | 'and' ) <ActionItem> )* ] ')' ';'
```

  The actual configuration action is a series of one or more action items connected together with the logical "and" or "or." All the action items must be enclosed between right and left parentheses. As can be seen from the above construct, the *do* action can be empty. This occurs when the action's *target* terminates the component (for example, when the action *target* is the "File/Exit" menu item.)

- The *do action* Items

```
<ActionItem> ::= '{' <Action> [ ',' <TargetObjectName> ] ','
      <InfoChoice> [ ',' <InfoChoice> ] '}'
```

```
<TargetObjectName> ::= 'File' | 'Folder'

<Action> ::= ‘open’ | ‘close’ | ‘exit’ | ‘navigate’ | 'enter' |
      'select' | 'insert' | 'delete' | 'change' | 'modify' |
      'click' | 'leftclick' | 'rightclick' | 'doubleclick' |
      'press' | 'tab'
```

An action item is composed of several parts. The first part, <Action>, states the name of the action. Currently, the only defined actions (listed in the <Action> production) are simple actions that can be performed easily by system integrators and that can possibly be automated by wrapper generators. The second part, <TargetObjectName>, is optional and is only used if the action target is a file or folder. The second optional <InfoChoice> is used for actions that act on two pieces of information in the same action (for example, the *modify* action replaces an old information with a new one as in { modify, “my name”, “your name” }).

The following example instructs system integrators (or wrapper generators) to open a file. The example lists three options for opening the file: by selecting the file from the list, by typing the file name in the provided field, or by double-clicking on the file name from the displayed list of files.

```
action=({select, File, "customer.info"} or
        {enter, "customer.info"} or
        {doubleclick, File, “customer.info”});
};
```

7.1.15.5 The *other* Configuration Option

The configuration constructs described above capture only the basic activities involved in the component adaptation process. Configuration instructions that cannot be described using these constructs can be specified using the `<OtherDeclaration>` construct. However, it should be noted that to fully automate the adaptation process requires the wrapper specification writers to think of all the possible constructs that all kinds of component vendors can require from their clients for the process of adapting various kinds of components to work properly with various sorts of environments and for various sorts of user requirements. Even if this were possible, complete automation may not be feasible since a component may have adaptation actions that require explicit user interventions.

7.1.15.6 A Configuration Example

The following example demonstrates the use of the do command to simulate the opening of a template file using WordPad, replacing all occurrences of <Customer-Name> with the component user's name, saving the file to a new name and/or directory, and finally exiting WordPad.

```
configuration = {
        do {
                target_type=screen;
                target="File/Open";
                action=({select, File, "customer.info"} or
                        {enter, "customer.info"} or
```

```
                    {doubleclick, File, “customer.info”}
                );
        };
        do {
                target_type=screen;
                target="Edit/Replace";
                action=({enter, "<CustomerName>"} and {tab, 1} and
                        {enter, "Your name"} and
                        {click, "Replace All"}
                );
        };
        do {
                target_type=screen;
                target="File/Save As";
                action=({navigate, Folder, "C:\Bank\Data"} and
                        {enter, "customer.dat"} and {tab, 2} and
                        {click, "Save"}
                );
        };
        do {
                target_type=screen;
                target="File/Exit";
                action=();
        };
};
```

7.1.16 Provided Interfaces

```
<ProvidesDeclaration> ::= ‘provides’ ‘=’ <Name> ( ‘,’ <Name> )* ‘;’
```

A wrapper generator communicates with components by means of interfaces. Interfaces are also the focus of the user’s functional requirements. The provided interfaces clause lists one or more interface identifiers. (For example, provides = deposit, withdraw;.) (Interface identifiers correspond to function names or method names in the target component, depending on the language terminology used.)

The <ProvidesDeclaration> construct is required in each <ComponentSpecification> section. Detailed specifications of each interface can be given

using the <InterfaceSpecification> section or the <ModuleSpecification> section (see Section 5.3.7.)

7.1.17 Required Interfaces

```
<RequiresDeclaration> ::= 'requires'
      '{' <ComponentRequirement>+ '}' ';'

<ComponentRequirement> ::= ( ( 'component' '=' <ComponentSpecifier> ) |
      ( 'interface' '=' <InterfaceSpecifier> ) ) ';'

<ComponentSpecifier> ::= <Identifier> | <Url> | <QuotedString>

<InterfaceSpecifier> ::= <Name> | ( '[' <Url> ']' '.' <Identifier> ) |
      <QuotedString>
```

The required <RequiresDeclaration> construct can assist both the component user and the wrapper generator by listing all constraints on the proper operation of the component. The <ComponentSpecifier> and the <InterfaceSpecifier> list all the components and interfaces that are required to be within the reach of the component when it is instantiated. Components usually would not function properly if the required interfaces or components were missing.

Component suppliers can specify required components or interfaces in various ways. For example, the full specification of a required component or interface can be included within the specification document of the main component using the <ComponentSpecification> construct for components and the <InterfaceSpecification> or <ModuleSpecification> constructs for interfaces. A vendor can also

point to a "remote" component or interface specification file using the remote file's explicit URL. The content of the remote specification file must conform to CSL's syntax as described in this dissertation.

If the specification of any required component or interface cannot be included within the specification document of the main component and cannot be pointed to using an explicit URL, component suppliers can describe the component or interface in narrative form. Such a narrative description, while inadequate for precise unambiguous specifications, may give the component user a hint on where to find (or how to identify) the required component or interface. Alternatively, the narrative description may only list the general functionality of the component. In this situation, it would be the component user's responsibility to find the described component or interface. The following is an example of using the *requires* clause:

```
requires {
        component = "ftp://ucf.edu/CSL/bank/OnlineBanking.csl";
        component = "Any component that supports online banking";
        interface     = OnlineBanking.deposit;
        interface     = ["ftp://ucf.edu/CSL/bank/EBills.csl"].PayOnline;
};
```

7.1.18 Software Constraints

Vendors may need to enforce certain restrictions on the use and operation of their components. For example, a vendor might require the existence or absence of certain features in the host component, system, or environment to ensure that the vendor's component can perform according to its specification. There are discussions in the software

engineering literature (see, for example, [BAGG97, BROW00, SZYP98]) about the specification of properties that must exist in the environment to allow components to function properly. Examples of these properties include: access to certain files or directories, fast Internet connection, use of specific middleware architecture, or existence of certain utilities or features in the host operating system.

In some situations, a component might function properly only in the presence or absence of other components. Additionally, a component may need to specify required operational parameters, such as the amount or percentage of its binary image that needs to reside in memory, or the place where the remaining of its binary image is to reside.

Component specifications can also address properties that must *not* exist in the environment due to the conflicts or interferences that such properties could have on the operation of the component. Examples of such properties could include: virus scanners, operating systems that do not support concurrency, or other components that are known to the vendor to cause problems with their component.

Some of these constraints on the operation of a component can be captured using the following set of constructs.

```
<SoftwareConstraintsDeclaration> ::= 'software_constraints' '=' '{'
      <SoftwareConstraint>+ '}' ';'

<SoftwareConstraint> ::= <SoftwareProgramDeclaration> |
      <SoftwareNotPresentDeclaration> |
      <SoftwareNotLoadedDeclaration> | <OtherDeclaration>

<SoftwareProgramDeclaration> ::= 'program' '=' '(' <QuotedString>
      [ ',' <VersionString> ] ')' ';'

<SoftwareNotPresentDeclaration> ::= 'notpresent' '='
      '(' <NotPresentChoice> ( ',' <NotPresentChoice> )+ ')' ';'
```

```
<NotPresentChoice> ::= <Identifier> | <Url> | <QuotedString>

<SoftwareNotLoadedDeclaration> ::= 'notloaded' '='
      '(' <NotLoadedChoice> ( ',' <NotLoadedChoice> )+ ')' ';'

<NotLoadedChoice> ::= 'VIRUS'
```

The <SoftwareProgramDeclaration> construct specifies required programs (and optionally, their version numbers.) For example, certain programs may be required during the download of the component, during its installation, when configuring the component, or at component runtime.

The <SoftwareNotpresentDeclaration> construct lists programs or other components that are known to cause conflicts with the operation of the component. The list of programs or components can be expressed in three ways:

- If components are listed as identifiers, a separate <ComponentSpecification> section is used to describe each component.
- If identifiers are not used, URLs can be used to point to the location of programs or component specifications that conflict with the component being specified.
- If it is not possible to point to the location of programs or component specifications using URLs, then a narrative description of such programs or components can be given using the <OtherDeclaration> construct. This form is only useful to the development team; the wrapper generator would not be able to directly apply it.

The <SoftwareNotLoadedDeclaration> construct allows listing categories of programs (such as anti-virus programs) that are known to interfere with the component's operation.

The collection of all of these constraints, along with other constraints that could be listed using the <RequiresDeclaration> construct (see Section 7.1.17,) represents the *minimum* that must be met in order for the component to work properly. The following is an example of how a *software_constraints* clause could be used:

```
software_constraints = {
        program    = ("iexplore.exe", 5.0);
        notpresent = (ManualBanking, "http://ucf.edu/TerminateAccount.csl");
        notloaded  = (VIRUS);
        other      = "This component must be instantiated before any other electronic
                      banking components";
};
```

Assume the component in which the above *software_constraints* clause resides is called ***main***. This example states that in order for ***main*** to function properly, version 5.0 or above of the application "iexplore.exe" (Microsoft Internet Explorer) must be present when ***main*** is instantiated. Additionally, at the instantiation time of ***main***, the *Manual-Banking* component (described in the same specification file of ***main***) cannot be present in the same application in which ***main*** exists. Similarly, the component whose specification can be reached at the URL *http://ucf.edu/TerminateAccount.csl* cannot be present in the same application as that of ***main***. The *notloaded* clause states that no virus-scanning programs should be running during the operation of ***main***. Finally, the *other* clause gives a constraint in the form of an instruction to the system integrator to instantiate ***main*** before any other component that deals with electronic banking. Since this final constraint

could cause conflicts with other candidate components (during component evaluation), the system integrator could be forced to abandon considering ***main*** in favor of other candidate components or vice versa.

7.1.19 Hardware Constraints

```
<HardwareConstraintsDeclaration> ::= 'hardware_constraints' '=' '{'
      <HardwareConstraint>+ '}' ';'

<HardwareConstraint> ::= <HardwareRAMDeclaration> |
      <HardwareStorageDeclaration> | <OtherDeclaration>

<HardwareRAMDeclaration> ::= 'RAM' '='
      <PositiveFloat> <StorageDesignator> ';'

<HardwareStorageDeclaration> ::= 'storage' '=' <PositiveFloat>
      <StorageDesignator> ';'
```

The <HardwareConstraintsDeclaration> construct allows component suppliers to list any hardware restrictions or requirements that must be met for their component to work properly. (Examples of such constraints might include available memory and storage space.)

For all constraints other than specifying memory and storage requirements, the <OtherDeclaration> can be used. These constraints represent the *minimum* hardware constraints that must be met in order for the component to work properly.

7.1.20 Network Constraints

```
<NetworkConstraintsDeclaration> ::= 'network_constraints' '=' '{'
      <NetworkConstraint>+ '}' ';'

<NetworkConstraint> ::= <OtherDeclaration>
```

These constructs are to complete the set of constructs that give restrictions on the use of the component. With this construct, any known network restriction can be listed using the <OtherDeclaration> construct. The constraints listed using this construct represent the *minimum* network constraints that must be met in order for the component to work properly.

7.1.21 Optional Add-Ons

```
<OptionalDeclaration> ::= 'optional' '='
      ( <ComponentSpecifier> | <InterfaceSpecifier> )
      ( ',' ( <ComponentSpecifier> | <InterfaceSpecifier> ) )* ';'
```

This construct allows component suppliers to list additional component plug-ins or add-ons. Only component or interface identifiers can be listed in the *Optional* clause. The CSL specification document must contain a separate <ComponentSpecification> or <InterfaceSpecification> section for each listed component or interface identifier.

7.2 The Provider Specification

A component specification document can include one or more provider specification sections. The constructs for specifying component providers follow similar structure as those for specifying components. The syntax for the <ProviderSpecification> construct is as follows:

```
<ProviderSpecification> ::= 'provider' <Identifier>
      '{' <ProviderBody>+ '}'
```

The provider's <Identifier> is the one that need to be listed in the <ProviderDeclaration> construct of each component specification. The provider identifier does not generally have to correspond to the actual name of the provider (because identifiers follows more strict syntax than provider names.) Provider names can be entered in their own clause within the body of the <ProviderSpecification> section. The <ProviderBody> is defined as follows:

```
<ProviderBody> ::= <NameDeclaration> | ( <UrlDeclaration> |
      <EmailDeclaration> | <PhoneDeclaration> | <FaxDeclaration> )* |
      <AddressDeclaration> | <PaymentMethodsDeclaration>
```

The <AddressDeclaration> construct supplies the provider's business address. All of the options listed in the right side of the <ProviderBody> construct are optional, except for <NameDeclaration> (the provider's name) and <PaymentMethodsDeclaration>. Additionally, each of the options, except <NameDeclaration>, <AddressDeclaration>, and <PaymentMethodsDeclara-

tion> can show up more than once in the same <ProviderSpecification> section. The syntax for all the constructs in <ProviderBody>, except for <Payment-MethodsDeclaration>, is given in Section 5.3.5. The syntax for the <Payment-MethodsDeclaration> is given below:

```
<PaymentMethodsDeclaration> ::= 'payment_methods' '=' '('
      <PaymentMethodChoice> ( ',' <PaymentMethodChoice> )* ')' ';'

<PaymentMethodChoice> ::= 'CHECK' | 'DEBIT' | 'CREDIT_CARD' |
      'MONEY_ORDER' | 'WIRE_TRANSFER' | 'PAYPAL'
```

The <PaymentMethodsDeclaration> construct is very important for wrapper generators to know if automated payment submissions for purchased components are possible. Wrapper generators will also make use of the information provided in this construct when they try to perform automatic upgrades of components. More than one payment method can be specified (for example, payment_methods = (CHECK, CREDIT_CARD, PAYPAL);).

7.3 A Component Specification Example

Figure 9 shows an example of a CSL specification document for an online bank. The example contains most of the behavioral and non-behavioral constructs discussed in this chapter. The interface specifications in this example are not complete but included only to give the reader a feeling on their placement in the specification file. Only the *name* clause is included in the specifications of the optional *OnlineBanking* and *EBills*

components. The presence of these two optional components in this example is to show the reader how the component identifiers can be referenced in the *requires* clause. The reader should note the difference between the component identifiers and the component names.

```
component MyBank {

  name = "Bank Accounts Manager";
  description = {
    This component is a simulation of a simple bank.
    It consists of two interfaces: deposit and withdraw.
    The component is written using Java 1.1.  Both interfaces
    throw exceptions.
  };

  provider = TheBankersGroup;
  version = 1.0.13;
  technology = JavaBean;
  operating_systems = ( WIN95, win98, winme, win2000, winnt4.0 );
  databases = ( NONE );
  language = Java;
  formats = (source);
  license_models = (FREE, DEMO);
  license_types = (SINGLE, SITE);
  single.LicenseCost = $673.30;
  single.UpgradeCost = $20;
  site.MaintenanceCost = $430.12;
  site.SupportCost = $230.33;
  developer.TrainingCost = $250;
  developer.InstallationCost = $100;
  support_methods = (EMAIL, ONLINE, FORUM);
  new_updates = "ftp://ucf.edu/CSL/bank/update.csl";

  downloads = (
    *File1(object)="ftp://ucf.edu/CSL/bank/bankAccount.class",
    File2(object)="ftp://ucf.edu/CSL/bank/insufficientFundsExc.class",
    File3(object)="ftp://ucf.edu/CSL/bank/negativeAmountExc.class"
  );

  installation = {
    mkdir "bank";
    move  File1 "bank";
    move  File2 "bank";
    move  File3 "bank";

    mkdir  "_temp_bank_";
```

```
    move  BigFile "_temp_bank_";
    run  "gunzip _temp_bank_/bank.tar.gz";
    run  "tar -xvf _temp_bank_/bank.tar";
    mkdir  "bank";
    run  "mv _temp_bank_/* bank/*";
    run  "rm -r _temp_bank_";

    use  "import bank.*;";
  };

  configuration = {
    insert ("Telling the component about your email",
            "customer.info", {5}, "put your email here");
    do {
      target_type=screen;
      target="File/Open";
      action=({select, File, "customer.info"} or
              {enter, "customer.info"} or
              {doubleclick, File, "customer.info"}
      );
    };
    do {
      target_type=screen;
      target="Edit/Replace";
      action=({enter, "<CustomerName>"} and {tab, 1} and
              {enter, "Your name"} and
              {click, "Replace All"}
      );
    };
    do {
      target_type=screen;
      target="File/Save As";
      action=({navigate, Folder, "C:\Bank\Data"} and
              {enter, "customer.dat"} and {tab, 2} and
              {click, "Save"}
      );
    };
    do {
      target_type=screen;
      target="File/Exit";
      action=();
    };
  };

  provides deposit, withdraw;

  requires {
    component = OnlineBanking;
    component = "ftp://ucf.edu/CSL/bank/OnlineBanking.csl";
    component = "Any component that supports online banking";
    interface = OnlineBanking.deposit;
    interface =
          ["ftp://ucf.edu/CSL/bank/OnlineBanking.csl"].PayOnline;
```

```
    interface = Deposit;
  };

  software_constraints = {
    program = ("iexplore.exe", 5.0);
    notpresent = (ManualBanking,
                  "http://ucf.edu/TerminateAccount.class");
    notloaded = (VIRUS);
    other = "This component must be instantiated before any other
             electronic banking components";
  };

  hardware_constraints = {
    RAM = 64MB;
    storage = 30MB;
    other = "Pentium IV or compatible processor";
  };

  network_constraints = {
    other = "Support for the TCP/IP protocol";
  };

  optional = (OnlineBanking, EBills);
}

interface Deposit {
  balance: float;

  DepositItems: hardware (money, check, coin);
  // This defines a list of hardware items that will be used in
  // making deposits

  Receipt: hardware (paper);
  // This defines the receipt that the customer will get when
  // making a deposit

  input Amount = { "Deposited amount", +float, any, any, parameter };
  // This input will be any positive floating point number and
  // will be passed as a parameter

  input Deposit = { "Deposited items", DepositItems, any, any,
                    device };

  output Receipt = { "Printed Receipt", Receipt, any, any, device };
  // This output will be in the form of a printed receipt

  output O1 = { "success or failure of operation", boolean,
                null, true, return };
  // The output should be true.
  // I expect the output to be returned in the function name

  output = { none };
  // It should not generate any other outputs
```

```
  behavior = {
    call (*, accountID, PIN, Amount): {

      accountID <> null && accountID == reference(Account) &&
      // Make sure the supplied accountID is not null and is
      // a reference to Account

      accountID->pin == PIN &&
      // Make sure the supplied PIN# is the same as that of accountID

      exists (accountID, AccountsDatabase) &&
      // This will make sure the supplied accountID exists in
      // the AccountsDatabase

      balance = get_balance (accountID) &&
      // This will save the balance before the deposit is made

      return (O1) == ATM.accept (Deposit, Amount) &&
      // This asks the ATM machine to accept the deposited items
      // from its deposit slot and to return "true" if successful
      // or "false", otherwise.  Assume the ATM has the intelligence
      // to count total amount of the deposited items, it will then
      // return true if that total amount matches Amount.  "accept"
      // is a special method defined just for this device in its
      // device definition file.

      return (O1) == update (accountID (balance), balance+Amount) &&
      // This tells the database AccountsDatabase to update the
      // balance column of the accountID record with the value
      // (balance+Amount). The update operation should be successful

      get_balance (accountID) == balance+Amount &&
      // Now make sure the balance actually reflects the new deposit

      return (O1) == ATM.printReceipt (deposit);
      // This requests from the ATM machine to print a receipt for
      // the user's deposit
    }
  }
  stimuli = {
    call,
    event (menu ("Tools/Deposit"), keyboard ('Ctrl-D'))
  }
}

interface withdraw {
  balance: float;

  DispensedItems: hardware (money);
  // This defines a list of hardware items that will dispensed
  // at withdrawals
```

```
Receipt: hardware (paper);
// This defines the receipt that the customer will get when
// making a deposit

input Amount = { "Withdrawn amount", +float, any, any, parameter };
// This input will be any positive floating point number and
// will be passed as a parameter

output Cash = { "Dispensed items", DispensedItems, any,
                any, device };

output Receipt = { "Printed Receipt", Receipt, any, any, device };
// This output will be in the form of a printed receipt

output O1 = { "success or failure of operation", boolean,
              null, true, return };
// The output should be true.
// I expect the output to be returned in the function name

output = { none };
// It should not generate any other outputs

behavior = {
  call (*, accountID, PIN, Amount): {

    accountID <> null && accountID == reference(Account) &&
    // Make sure the supplied accountID is not null and is a
    // reference to Account

    accountID->pin == PIN &&
    // Make sure the supplied PIN# is the same as that of accountID

    exists (accountID, AccountsDatabase) &&
    // This will make sure the supplied accountID exists in the
    // AccountsDatabase

    balance = get_balance (accountID) &&
    // This will save the balance before the deposit is made

    balance >= Amount &&
    // Make sure there is enough balance for the withdrawn amount

    return (O1) == ATM.dispense (Cash, Amount) &&
    // This asks the ATM machine to dispense withdrawn money
    // through its money dispenser slot and to return "true" if
    // successful or "false", otherwise.  It will return successful
    // only if the dispensed amount equals Amount.  "dispense" is
    // a special method defined just for this device in its device
    // definition file.

    return (O1) == update (accountID (balance), balance-Amount) &&
    // This tells the database AccountsDatabase to update the
    // balance column of the accountID record with the value
```

```
      // (balance-Amount). The update operation should be successful

      get_balance (accountID) == balance-Amount &&
      // Now make sure the balance actually reflects the new
      // withdrawal

      return (O1) == ATM.printReceipt (deposit);
      // This requests from the ATM machine to print a receipt for
      // the user's deposit
    }
  }
  stimuli = {
    call,
    event (menu ("Tools/Withdraw"), keyboard ('Ctrl-W'))
  }
}

provider TheBankersGroup {
  name = "The Bankers Group";
  url = "http://www.TheBankersGroup.org";
  email = "webmaster@TheBankersGroup.org";
  phone = "407-999-9999";
  phone = "321-999-9999";
  fax = "407-999-9999";
  fax = "321-999-9999";
  address =  {
        1 N. First Drive
        Suite # 9999
        New Florida, FL 77777
  };

  address =  {
      P.O.Box 777
        London, AL 55555-4444
  };
  payment_methods = (CHECK, PAYPAL);
}

component OnlineBanking {
  name = "Online Banking";
}

component EBills {
  name = "Electronic Bill Payments";
}
```

Figure 9: A complete CSL specification document for an online banking component

CHAPTER 8: THE WRAPPER SPECIFICATION LANGUAGE

This chapter discusses WSL, the specification language specifically designed for describing compositional wrappers. The constructs presented for this language have similar syntax to that of CSL though the constructs of this language refer to different entities. For example, the wrapper's name uses the same syntax as that of the component's name. The same can be said about the wrapper's description and version number.

Sometimes, despite similarity in the structure, the intended meaning of using a construct in WSL might be different from that of CSL. For example, though the constructs to specify the software, hardware, and network constraints have similar syntax in the two languages, they are interpreted in CSL as the minimum constraints that are required for proper operation of the component. However, in WSL, these constructs are interpreted to mean the maximum constraints that system integrators are to expect component suppliers to require for their components.

Note that constructs that are similar in both languages are only discussed briefly in this chapter since such constructs have already been discussed in the previous chapter. Whenever, possible a reference to the relevant CSL section is provided.

8.1 Importance Ranking Scheme

To help specification writers to express their preferences on the various language constructs, a ranking scheme can be used to tag constructs that are of different importance from other constructs in the specification. Such a scheme can be used by the wrapper generator during the component evaluation process, and during the matching between required and provided interfaces. The following bullets provide a guideline for using this ranking scheme:

- Ranking is defined on an ordinal scale.
- Any construct can be followed by a relative importance number with respect to other constructs. A lower ranking number denotes greater importance. For example, if the `operating_systems` clause is more important to the system integrator than the `databases` construct, and the `databases` is more important than the `languages` clause, then these clauses can be defined in the following way:

```
operating_systems (2) = ( WIN95, win98, winme, win2000, winnt4.0 );
databases (20) = ( FILE, MySQL );
languages (55) = (Perl, PHP);
```

- Constructs that have no ranking following them are considered to have the same importance as any other unranked constructs. All unranked constructs are considered to have lower importance than all ranked ones.
- Each option within a construct can be ranked. Thus, the above example can be modified as follows:

```
operating_systems (2) = ( WIN95, win98, winme(3), win2000(1), winnt4.0(4) );
databases (20) = ( FILE(20), MySQL(1) );
```

languages (55) = (Perl(3), PHP(16));

- Options that have no ranking following them are considered to have the same importance as any other unranked options within the same construct. All unranked options are considered to have lower importance than all other ranked options in the same construct.
- Options can be ranked even if the construct that such options belong to is not ranked.
- The order in which ranked or unranked constructs appear within the wrapper specification has no effect on the ranking scheme. Similarly, the order in which options are listed within a construct has no effect on the ranking scheme of the options.

8.2 The Wrapper Specification Constructs

At a minimum, the wrapper specification document must contain the expression for describing the compositional wrapper that the system integrator is seeking to generate. The syntax for the wrapper expression is as follows:

```
<WrapperSpecification> ::= 'wrapper' <Identifier> '{' <WrapperBody> '}'
<WrapperBody> ::= <WrapperDeclarationUnit>+
```

When the wrapper generator parses the <WrapperSpecification> construct, then it creates an environment variable called $WRAPPER_NAME that contains the value specified in <Identifier> of this construct. The generator will use the envi-

ronment variable when performing the actual downloads, installation, and configuration of the selected components. The <WrapperBody> construct can contain one or more <WrapperDeclarationUnit> constructs. The syntax for the <WrapperDeclarationUnit> construct is as follows:

```
<WrapperDeclarationUnit> ::= <NameDeclaration> |
      <DescriptionDeclaration> | <VersionDeclaration> |
      <OwnerDeclaration> | <DevelopersDeclaration> |
      <SuppliersDeclaration> | <TechnologiesDeclaration>|
      <OperatingSystemsDeclaration> | <DatabasesDeclaration> |
      <LanguagesDeclaration> | <FormatsDeclaration> |
      <LicenseModelsDeclaration> | <LicenseTypesDeclaration> |
      <CostDeclaration> | <SupportMethodsDeclaration> |
      <PaymentMethodsDeclaration> | <PersonalInfoDeclaration> |
      <PaymentInfoDeclaration> | <InterfacesDeclaration> |
      <SoftwareConstraintsDeclaration> |
      <HardwareConstraintsDeclaration> |
      <NetworkConstraintsDeclaration> |
      <TempDirectoryDeclaration> | <DestinationDirectoryDeclaration> |
      <WrapperFormatDeclaration> |
      <WrappingOptionsDeclaration> |
      <SmartFeaturesDeclaration>
```

The above constructs can show up in any order. However, each construct can appear at most once in each wrapper specification. Most of the constructs defined in the above production have the same syntax as that of the components specification language. These include <NameDeclaration>, <DescriptionDeclaration>, <VersionDeclaration>, <OperatingSystemsDeclaration>, <DatabasesDeclaration>, <LicenseModelsDeclaration>, <LicenseTypesDeclaration>, <CostDeclaration>, <SupportMethodsDeclaration>, <SoftwareConstraintsDeclaration>, <HardwareConstraintsDeclaration>, and <NetworkConstraintsDeclaration>.

The <PaymentMethodsDeclaration> construct is the same as that defined in the <ProviderSpecification> section (see Section 7.2.)

All of the constructs that are listed in the right hand side of the <WrapperDeclarationUnit> are optional, except for the wrapper's name. The <VersionDeclaration> is useful for organizations that want to make use of current technology trends but do not want to limit themselves to the vendors that supplied the components in their existing wrappers. These organizations can revise their wrapper specifications, give the modified specifications newer version numbers, and then repeat the wrapping generation process. The software, hardware, and network constraints are to be interpreted as the *maximum* that system integrators expect components to require. The <TechnologiesDeclaration> and <LanguagesDeclaration> constructs are similar to that of CSL except that in WSL, more than one option can be listed in each construct. (This approach is similar to that of other multi-option constructs such as operating_systems and databases.)

The following subsections will discuss the constructs that are new to WSL.

8.2.1 The Wrapper's Owner

```
<OwnerDeclaration> ::= 'owner' '=' <Identifier> ';'
```

This construct specifies the wrapper's original owner. Only an identifier is specified in this construct. The actual owner's personal information should be given in a sepa-

rate <PersonSpecification> section whose name matches the <Identifier> in the <OwnerDeclaration> construct. (The information provided by the <Owner-Declaration> construct and the corresponding <PersonSpecification> section is used in the implementation of some of the smart features that will be discussed in Section 8.2.9.)

8.2.2 The Wrapper's Developers

```
<DevelopersDeclaration> ::= 'developers' '=' '(' <Identifier>
      ( ',' <Identifier> )* ')' ';'
```

This construct specifies the wrapper's list of developers (i.e., the people that write and maintain the wrapper's specification, perform the components evaluations and testing, and manage the overall wrapper generation process.) Only identifiers are specified in this construct. The actual personal information of each developer should be given in separate <PersonSpecification> sections with the same name as the <Identifier>s listed in the above <DevelopersDeclaration> construct. When the wrapper's owner is also one of the developers, then only one <PersonSpecification> section is needed to describe the owner. The <Identifier> that appears in the <OwnerDeclaration> will be one of the <Identifier>s that are listed in the <DevelopersDeclaration>. At the same time, the same <Identifier> will appear as the name of the <PersonSpecification> section for the owner. (The list

of developers is used in the implementation of some of the smart features that will be discussed in Section 8.2.9.)

8.2.3 Preferred Component Suppliers

```
<SuppliersDeclaration> ::= 'suppliers' '=' '(' <SupplierChoice>
      ( ',' <SupplierChoice> )* ')' ';'

<SupplierChoice> ::= <Url> | <QuotedString>
```

This construct is optional and allows system integrators to specify a list of preferred suppliers. When searching for components, the wrapper generator will try to adhere to the list of suppliers provided by this construct. Integrators can specify a supplier by either listing the supplier's popular name or the URL to the supplier's official website (for example, suppliers = ("http://www.omg.org", "Microsoft Corporation");).

8.2.4 Required Interfaces

```
<InterfacesDeclaration> ::= 'interfaces' '=' <Name> ( ',' <Name> )* ';'
```

Interfaces are the means by which the wrapper generator can communicate with its components. This construct lists one or more interface identifiers. The *interfaces* clause must list at least one required interface. Formal specification of the listed interfac-

es will need to be given in separate <InterfaceSpecification> or <ModuleSpecification> sections.

8.2.5 Personal Information

```
<PersonalInfoDeclaration> ::= 'personal_info' '=' <Identifier> ';'
```

This required construct allows specifying a personal profile to be used by wrapper generators during the process of acquiring selected components (and possibly when making component reservations.) The actual profile information will reside in a separate <ProfileSpecification> section (see Section 8.5) whose name matches the <Identifier> in the <PersonalInfoDeclaration> construct. The <ProfileSpecification> section can reside anywhere within the specification document. During the components' acquisition process and upon the vendor's request, the information in the designated <ProfileSpecification> will be passed to the vendor (using specific APIs available in the vendors' systems or through message passing techniques like RPC or RMI.)

8.2.6 Payment Information

```
<PaymentInfoDeclaration> ::= 'payment_info' '='
      '(' <PaymentInfoChoice> ',' <Identifier> ')' ';'
```

```
<PaymentInfoChoice> ::= 'CREDIT_CARD' | 'BANK_ACCOUNT'
```

This required construct allows specifying a payment profile to be used by wrapper generators during the process of acquiring selected components (and possibly when making component reservations.) The actual profile information will reside in either a <CreditCardSpecification> section or a <BankAccountSpecification> section depending on the payment type (CREDIT_CARD or BANK_ACCOUNT) entered using the <PaymentInfoChoice> construct. The name of the <CreditCard-Specification> or <BankAccountSpecification> section must match the <Identifier> in the <PaymentInfoDeclaration> construct. The <CreditCardSpecification> or <BankAccountSpecification> section can reside anywhere within the specification document. During the components' acquisition process and upon the vendor's request, the information in the designated <CreditCardSpecification> or <BankAccountSpecification> section will be passed to the vendor (using specific APIs available in the vendors' systems or through message passing techniques like RPC or RMI.)

8.2.7 Defining a Working Directory

```
<TempDirectoryDeclaration> ::= 'temp_directory' '=' <Directory> ';'
```

This construct specifies a directory that can be used by wrapper generators to save their temporary files and the files of the downloaded component. (The `<Directory>` construct in the above production can refer to the `$GENERATOR_HOME` environment variable which is set by the system integrator before launching the generator (see Section 14.9)).

8.2.8 Controlling the Wrapper Generation Process

Three constructs are defined in WSL to allow the development team full control of the working and outcome of the wrapper generation process. These constructs are discussed in the following subsections.

8.2.8.1 Specifying the Wrapper Format

```
<WrapperFormatDeclaration> ::= 'wrapper_format' '=' <FormatChoice> ';'
```

This construct allows system integrators to specify the format (`SOURCE`, `OBJECT`, or `EXE`) of generated compositional wrappers. The usability of this construct depends on the implementation of the wrapper generator. If the generator supports generating more than one format, then this construct is useful to the generator. However, if the generator supports only one format, then the generator will ignore this construct and gen-

erate the compositional wrapper into the supported format. If this construct is missing, then the generator can use the default format preset by its developers.

8.2.8.2 Specifying the Destination Directory

```
<DestinationDirectoryDeclaration> ::= 'destination_directory' '='
      ( <Url> | [ <Path> ]<FileName> ) ';'
```

The purpose of this construct is to allow the system integrator to specify the directory where wrapper generator can put the generated compositional wrapper and all of the wrapper's associated component files. If this construct is missing, then the wrapper generator can use a default directory and point the system integrator to it when the generation process completes.

8.2.8.3 Controlling the Generated Wrapper's Content

```
<WrappingOptionsDeclaration> ::= 'wrapping_options' '='
      '(' <WrappingChoice> ( ',' <WrappingChoice> )* ')' ';'

<WrappingChoice> ::= 'EncryptWrapper' | 'EncryptComponents' |
      'EncryptAll' | 'CompressWrapper' | 'CompressComponents' |
      'CompressAll' | 'FastLoading' | 'PreserveMemory'
```

The options listed in the `<WrappingOptionsDeclaration>` construct tells the wrapper generator what to do when it is time to wrap the components and all other generated code. The first three options (`EncryptWrapper`, `EncryptComponents`,

and `EncryptAll`) tell the wrapper generator whether to encrypt just the generated wrapper's code, encrypt the component files, or encrypt both. The next three options (`CompressWrapper`, `CompressComponents`, and `CompressAll`) tell the wrapper generator whether to compress just the generated wrapper's code, compress component files, or compress both. The `FastLoading` option will cause the wrapper generator to split the code of the compositional wrapper into self-contained modules that can be loaded separately and only when needed. The `PreserveMemory` option instructs the wrapper generator to optimize the generated code so that it requires the least amount of memory.

The following are some points to consider when specifying these options:

- Using modern cryptographic techniques to encrypt the compositional wrapper and/or its components would provide the following assurances to the compositional wrapper users [MOOR94]:
    - To ensure that no third party can make unauthorized modifications to the wrapper.
    - To ensure that the wrapper's owner cannot modify its wrapper after its unique identifier is assigned and generated (see Section 14.9 for a discussion on generating such unique identifiers.)
    - To ensure that no one can masquerade as another provider.

- Encrypted code will either be put in separate file(s) or be embedded within the *enclosing shell* (see section 14.9) depending on the wrapper generator implementation.
- The `EncryptWrapper` option does not imply the encryption of components. Conversely, the `EncryptComponents` does not imply the encryption of the wrapper.
- Components' code can be left in separate files or embedded within the wrapper's generated code, depending on the wrapper generator's implementation.
- If encryption and compression are both specified, then encryption will be done first, followed by compression. (Note that the reverse order might also be satisfactory.)
- Encryption and compression will make loading of the hosting application a bit slower when the application tries to instantiate the wrapper.
- If `FastLoading` is used with encryption and/or compression, then the controlling script (the small shell or the wrapper responsible for the decryption and decompression, see section 14.9) will work as usual by decrypting and/or decompressing each module when the wrapper is instantiated.

8.2.9 Smart Features

The need to add smart features (see the beginning of Chapter 3 for a definition of smart features) to generated compositional wrappers was one of the major reasons in introducing CSL and WSL. Without exhibiting some kind of intelligence, the compositional wrappers described in this research would not be different from other types of composite components being discussed in the literature. Despite the claims of [BASI01] that more than half the features in large components-based software products go unused, including smart features in compositional wrappers is needed in order to support the automation of many of the steps in the composition process.

In addition to automation support, including smart features into compositional wrappers has several advantages. For example, building smart properties into individual components can overload components with too much, and in many cases unnecessary, functionality that will not only yield "fat" components, but more expensive ones. Such "fat" components have several disadvantages:

- Fat components have bigger file sizes and, hence, slower loading and downloading time. This will make the system using them less efficient.
- Since each individual component will be doing many "hidden" operations in the background that, in most part, have no direct benefits to the system user; system performance can be affected greatly. In addition, many of these operations will be redundant if each component implements similar or identical smart features.
- It will be difficult (if not impossible) to have all component vendors abide by agreed-upon standards that guide the implementation, customization, and use of

such features. This will result in less-efficient code fragments implementing each feature.

- The cost of feature-loaded components (and therefore the resultant systems) will be higher since developers will not include their features free of charge.
- Fat components put an additional burden on component developers to implement such features and on application assemblers that customize each of these components. Such extra effort will make components more expensive and take longer to produce.

The second benefit of implementing the smart features into compositional wrappers rather than implementing them in individual components is to relieve component developers from worrying about minute details when designing and implementing these features in their components. Component vendors can utilize the time spent in designing and implementing smart features to produce more innovative and robust components. In addition, because integrated systems will eventually require fewer numbers of auto-generated compositional wrappers than single components, the inclusion of such features in compositional wrappers is more justifiable than including the smart features in single components. Furthermore, because the generated wrapper's code implementing such features will be designed and coded by expert developers (when developing the wrapper generator), the generated code will typically be more optimized and less prone to errors than the code to implement similar features but by individual developers. In addition, it will be easier to enhance and manage such features in wrappers than in individual com-

ponents because of the provided interfaces that will be included with each feature to perform the management and control of such features.

The third benefit is that it is easier (through the provided interfaces) to switch individual features or group of features on and off, as needed, in a wrapper level than in each individual component. This is because of the fewer number of compositional wrappers that would compose a complete system compared to systems built from individual components. Thus, having this capability built into compositional wrappers will give more flexibility and confidence both to wrapper vendors and users. Wrapper vendors may desire to turn features on or off for various reasons, such as when vendors discover bugs in the implementation of such features or when they want to upgrade their wrappers with enhanced versions of such features. Wrapper users may also desire to turn features on or off to speed up their systems or when they suspect security breaches because of availability or absence of certain features.

The following construct would help specification writers to include the smart features they desire to have in their compositional wrappers.

```
<SmartFeaturesDeclaration> ::= 'SmartFeatures' '=' '{'
      [ <SmartFeatures_AuthenticationCodesDeclaration> ]
      ( 'ALL' | <SmartFeatures_List>* ) '}' ';'
```

At wrapper generation time, the wrapper generator will include only the code for the selected features. To include all the code for all of the supported smart features, a specification writer can use the `ALL` keyword.

The <SmartFeatures_AuthenticationCodesDeclartion> construct can be included to specify the codes needed to authenticate requests to activate or manage some of the features. When implementing these features in the wrapper generator, there will be interfaces to allow other components or applications to access the various options of each feature. Other interfaces might be included to help wrapper users in managing these features. The interfaces for the various smart features options are listed and discussed in Section 14.7.

The <SmartFeatures_List> construct is defined as:

```
<SmartFeatures_List> ::= <SmartFeatures_SelfDocumentationDeclaration> |
      <SmartFeatures_TrackingDeclaration> |
      <SmartFeatures_LicensingDeclaration> |
      <SmartFeatures_StateDeclaration> |
      <SmartFeatures_UpdateDeclaration> |
      <SmartFeatures_TestingDeclaration> |
      <SmartFeatures_SecurityDeclaration>
```

The following subsections give detailed explanation about each of these constructs. Section 14.7 gives guidelines on implementing such features into compositional wrappers.

8.2.9.1 Authentication Codes

```
<SmartFeatures_AuthenticationCodesDeclaration> ::=
      <AuthenticationCodes>*

<AuthenticationCodes> ::= ( 'OwnerAuthenticationCode' |
        'DeveloperAuthenticationCode' |
        'UserAuthenticationCode' ) '=' <QuotedString> ';'
```

These constructs allow specification writers to enter default authentication codes for the generated wrapper's original owner, the wrapper's developer(s), or the wrapper's current owner. The generated compositional wrapper will use these codes to authenticate requests to activate the interfaces of some of the smart features that will be outlined in the following sections. These codes act like passwords to protect against malicious use of such features. Three separate authentication codes are provided (along with interfaces to manage them) to allow each administrative authority (original owner (owner), wrapper developer(s), or current owner (user)) to access the features that are relevant to their needs. If the wrapper specification document is missing any of the authentication codes and if the wrapper specification contains some of the smart features that require authentication, then the wrapper generator will auto-generate such missing codes when generating the compositional wrapper.

The wrapper licensor is another party that can have access to some of the smart features that will be discussed in the next sections. The wrapper licensor would be the previous wrapper owner, or the original owner in the case when the wrapper has not been licensed to another party. The wrapper generator will track its last owner and the authentication code of that owner. There will be separate interfaces to allow the licensor to retrieve or change its authentication code. Additionally, compositional wrappers will have built-in code to check for the validity of these authentication codes and for allowing each party to change its own authentication code. Wrapper generator implementations can

encrypt authentication codes before including them in the generated wrapper. Such an approach will give an added security to accessing internal features of generated wrappers.

8.2.9.2 Self-Documentation

As emphasized by [RUTT98], component self-documentation is needed to support the evaluation of component usability and stability. Current technologies supporting component-based development provide self-documentation services at the component level [SZYP00a]. For example: COM products require a *QueryInterface* method in each COM component to display the component interfaces and to describe how to connect to the interfaces; The JavaBeans *BeanDescriptor* provides a similar mechanism; CORBA provides an object query service that supports locating instances of objects and a trader service that supports locating services by description; The WaterBeans model [PLAK99] requires a *Display()* method in each component to enable the component to represent itself in the user's screen.

A compositional wrapper will have the capability to generate formal or natural language self-documentation upon receiving authorized requests. The documentation can be presented in textual or graphical format depending on the environment settings or the user needs. The disclosed documentation will serve as a *syntactic* and *semantic contract* [SZYP98] between the wrapper's vendor and its current user.

At a minimum, the generated static documentation will include:

a) Identification of the compositional wrapper and its components. This includes the wrapper's name, a brief description of the components it encases and their license information, and the list of interfaces (services and features) provided by the wrapper.

b) Iconic and/or GUI representations of the compositional wrapper. Such representations could be made public to allow application assemblers easy handling of assembled wrappers.

c) Description of each provided interface and the format for calling them.

d) A list of pre-conditions necessary for a compliant operation of the wrapper with its specification. These would include environment settings, such as storage quotas and communication protocols, as well as availability or absence of other interfaces.

e) Any available vendor-supplied estimates of the quality attributes (reliability, survivability, etc.) for each provided interface.

f) Self-test and status reporting capability for each interface, including a description of the capability and the test suite (inputs, procedures, and expected outputs) used to demonstrate the compliance of the interface implementation to its specification.

It should be noted that compositional wrappers would retrieve most of the above-mentioned information from its individual components through their provided interfaces. Therefore, other restricted information, such as the internal data structures of the wrapper

and the source and/or object code of the wrapper's components, could also be made available to authorized requesters. Depending on the system requirements, compositional wrappers could also generate some forms of dynamic documentation. Examples include:

- Representation of the wrapper's current state, including memory usage and active instances.
- Actual values for quality attributes of each service based on recent usage and collected statistics.
- Usage history and timing statistics for each service.

In general, it is expected that the wrapper will disclose dynamic information only to authorized requesters. The compositional wrapper will be responsible for authenticating such requests using the authentication codes presented in the previous section.

The following constructs implements most of the above-mentioned self-documentation features. (Section 14.7.2 lists the interfaces that will be provided in generated compositional wrappers to support these features.)

```
<SmartFeatures_SelfDocumentationDeclaration> ::=
      'selfdoc' '=' '(' ( 'ALL' | ( <SelfDocumentation_Option>
      ( ',' <SelfDocumentation_Option> )* ) ) ')' ';'

<SelfDocumentation_Option> ::= <SelfDocumentation_DisplayIcon> |
      <SelfDocumentation_PrintDescription> |
      <SelfDocumentation_PrintState> |
      <SelfDocumentation_PrintHistory> |
      <SelfDocumentation_SendState> | <SelfDocumentation_SendHistory> |
      <SelfDocumentation_ListComponentsInfo> |
      <SelfDocumentation_ListInterfaces>

<SelfDocumentation_DisplayIcon> ::=
```

```
      'DisplayIcon' ['(' [ <Path> ]<FileName> ')']

<SelfDocumentation_PrintDescription> ::= 'PrintDescription'
<SelfDocumentation_PrintState> ::= 'PrintState'
<SelfDocumentation_PrintHistory> ::= 'PrintHistory'
<SelfDocumentation_SendState> ::= 'SendState'
<SelfDocumentation_SendHistory> ::= 'SendHistory'
<SelfDocumentation_ListComponentsInfo> ::= 'ListComponentsInfo'
<SelfDocumentation_ListInterfaces> ::= 'ListInterfaces'
```

Options in the <SmartFeatures_SelfDocumentationDeclaration> construct will expose the wrapper's internal workings to requested parties. Because the <SelfDocumentation_DisplayIcon>, <SelfDocumentation_PrintDescription>, <SelfDocumentation_ListComponentsInfo>, and <SelfDocumentation_ListInterfaces> are public, their corresponding interfaces can be called by any application without passing them authentication codes. The interface for the <SelfDocumentation_PrintHistory> construct requires the authentication code of the current wrapper owner. The interfaces for the other constructs would need to be accompanied by one of the four authentication codes discussed in Section 8.2.9.1. The <SelfDocumentation_PrintHistory> and <SelfDocumentation_SendHistory> constructs are only applicable if the <Tracking_TrackHistory> construct (discussed in Section 8.2.9.3) is specified. The following is an example of defining the self-documentation feature:

selfdoc = (DisplayIcon ("mywrapper.gif"), ListComponentsInfo, ListInterfaces);

8.2.9.3 Tracking Wrapper Activities

Compositional wrappers can have built-in capabilities to track the various aspects of their activities and the activities performed by their components. For example, when the compositional wrapper is used for the first time, it would inform its original owner that its use has started and then it would pass to the owner several pieces of information, such as the wrapper's name and ID, the wrapper's version number, and specific information about the environment in which the wrapper is executing. The original owner's server interface would check whether a valid license exists for the user in which the wrapper resides. The owner's server interface would also advise the licensed wrapper about actions to be taken in case of license agreement violation. (For example, the wrapper may switch to demo mode or offer a grace period if no valid license exists or if the license has expired.) The wrapper vendor's server may periodically send event notifications to the wrapper, which must reply by reporting its usage statistics.

These tracking activities require that compositional wrappers have built-in mechanisms to capture execution traces, store them, and re-execute them. Execution traces (also known as execution profiles, execution histories, or execution logs) are the list of states that a component passes through from its initiation to its termination. [VOAS00c] suggested having operational profiles' tracking enabled in as earlier versions as possible. This would help wrapper users in testing their wrappers and in deciding when to stop testing. [GAO00b] listed several consequences for lacking tracking-ability in software components. Among these consequences are high costs of performance testing and com-

ponents' tuning; difficulty in tracking, isolating, and debugging errors; difficulty in validating resource usages; and difficulty to monitor and control components behavior.

Software components can generate six types of traces. They are operation, performance, error, state, GUI events, and communication traces [GAO00a]. Execution traces are important in detecting whether a component passes through any states that violate the list of permissible states outlined in the component's specification. Execution traces are useful in detecting bugs or failures and in measuring component quality, performance, and reliability. Execution traces can also be used to identify better-performing components. Such information are needed in load-balancing or emergency recovery (see Section 14.1.)

Execution profiles (or traces) are usually very long lists of values in machine- or human-readable formats. Storing and retrieving such traces can be problematic because of the volume of information to be stored and the performance considerations to be considered when retrieving them (especially if retrieval is done while the system is in operation.) For fault identification and correction, hosting systems (for example, compositional wrappers) may need to go back in history and re-execute a portion of the trace to isolate the buggy component.

Compositional wrappers should also be able to track their current and previous owners. As stated in [STER96], such a feature can be used to support the distribution, use, maintenance, and upgrading of the wrappers and their components.

The following constructs outline some of the tracking capabilities built into WSL. The interfaces for implementing these features in the wrapper generator are presented in Section 14.7.3.

```
<SmartFeatures_TrackingDeclaration> ::= 'Tracking' '=' '(' ( 'ALL' |
      ( <Tracking_Option> ( ',' <Tracking_Option> )* ) ) ')' ';'

<Tracking_Option> ::= <Tracking_TrackCalls> | <Tracking_TrackOwners> |
      <Tracking_TrackLicenses> | <Tracking_TrackHistory> |
      <Tracking_TrackAdminActivities> |
      <Tracking_PrintTrackingInfo> | <Tracking_SendTrackingInfo>

<Tracking_TrackCalls> ::= 'TrackCalls'
<Tracking_TrackOwners> ::= 'TrackOwners'
<Tracking_TrackLicenses> ::= 'TrackLicenses'
<Tracking_TrackHistory> ::= 'TrackHistory'
<Tracking_TrackAdminActivities> ::= 'TrackAdminActivities'
<Tracking_PrintTrackingInfo> ::= 'PrintTrackingInfo'
<Tracking_SendTrackingInfo> ::= 'SendTrackingInfo'
```

The above constructs allow wrapper administrators to monitor the behavior of the compositional wrapper and its components. Interfaces for all of the tracking features require valid authentication codes. Applications calling their corresponding interfaces would need to accompany the interface call by one of the four authentication codes discussed in Section 8.2.9.1. The <Tracking_TrackHistory> is required to use the <SelfDocumentation_PrintHistory> and <SelfDocumentation_SendHistory> constructs (discussed in Section 8.2.9.2.) The <Tracking_TrackCalls> will monitor all calls received by the wrapper from the hosting application or from other components. The <Tracking_TrackOwners> construct will track all the wrapper owners. The <Tracking_TrackLicenses> construct will monitor licensing activities performed by the wrapper or by its components. Tracked licensing activities would include tracking licenses' expiration dates, costs, and renewal

decisions. The <Tracking_TrackHistory> option will monitor all the activities performed by the compositional wrapper or by its components. It will save the history of such activities for retrieval by the interfaces of the <SelfDocumentation_PrintHistory> and <SelfDocumentation_SendHistory> constructs (discussed in Section 8.2.9.2) or by the interfaces of the <Tracking_PrintTrackingInfo> and <Tracking_SendTrackingInfo> constructs. The <Tracking_TrackAdminActivities> will monitor calls to the smart features' restricted interfaces. Such interfaces would include the ones for manipulating authentication codes, wrapper's state, history, licenses, etc. The <Tracking_PrintTrackingInfo> allows the current wrapper owner to display all the tracking activities gathered. The <Tracking_SendTrackingInfo> will enable any of the four administrative parties to receive a copy of the tracking activities through their server agents or email addresses.

8.2.9.4 Managing Licenses

A license enables an entity that owns property rights in something (the licensor) to grant to a third party (the licensee) the right to use those property rights. A software component licensor owns copyright, trade secrets, or patent rights in the software component. The copyright protects only one particular representation, plus its derivatives and descendants [GUNT00]. As mentioned in [CHAV98], licenses typically apply only to

large-granularity components that consist of many lines of code and constitute a significant part of the finished software product.

As discussed in [APPL98, SIMM93], available software licensing models use various software approaches to impose licensing protection on components. However, these licensing models are not directly applicable to all environments and there is limited support for them within programming languages. Enforcing such licensing models requires sophisticated techniques and technologies to track privileges and monitor the nature of their use. Because of these issues, compositional wrappers are more suitable for implementing licensing capabilities than individual components.

In addition to basic licensing capabilities, the vendor of a component or compositional wrapper may need to be able to suspend or terminate operation, or delete all instances and uses of a component. For example, if the vendor decided to stop supporting a component (perhaps because a replacement component has been made available that uses a different technology, corrects a security risk, etc.), the vendor might want to suspend operation of the old component to force the component clients to replace the old component with the new one. On the other hand, in the case of unauthorized use, the vendor might decide to suspend or terminate the component, or to "withdraw" the component from all systems suspected of using it. Because this could have negative effect on the system's operations (and, hence, on vendor/client relationships), a protocol for authenticating the legitimacy of these actions must be established in advance. In DCUP [HOA98], the suspend/resume feature is used to dynamically update components at runtime.

To support selective suspension or termination of components or wrappers from the systems of specific clients requires: the use of unique identifications of components and wrappers; a messaging system to facilitate communication between the vendor and each client; and pre-established protocols for notification, suspension, and termination of components. A compositional wrapper can act very intelligently in these situations. The wrapper can be responsible both for authenticating actions and for finding and installing components to replace suspended or withdrawn components. Such intelligent behavior can help to ensure smooth operation of the host application.

The following constructs implement some of the mentioned licensing features. Interfaces to be provided by the wrapper generator to support these features will be presented in Section 14.7.4.

```
<SmartFeatures_LicensingDeclaration> ::= 'Licensing' '='
      '(' ( 'ALL' |
      ( <Licensing_Option> ( ',' <Licensing_Option> )* ) ) ')' ';'

<Licensing_Option> ::= <Licensing_PrintLicenseInfo> |
      <Licensing_PrintCurrentOwner> | <Licensing_PrintOwners> |
      <Licensing_SendOwners> | <Licensing_Suspend> | <Licensing_Resume>

<Licensing_PrintLicenseInfo> ::= 'PrintLicenseInfo'
<Licensing_PrintCurrentOwner> ::= 'PrintCurrentOwner'
<Licensing_PrintOwners> ::= 'PrintOwners'
<Licensing_SendOwners> ::= 'SendOwners'
<Licensing_Suspend> ::= 'Suspend'
<Licensing_Resume> ::= 'Resume'
```

These constructs allow tracking wrapper owners and getting current licensing information. Interfaces for all of the licensing options, except the ones for <Licensing_PrintCurrentOwner>, will require valid authentication codes. The <Li-

censing_Suspend> and <Licensing_Resume> options work as a pair and must be specified together. Their corresponding interfaces can only be accessed by the wrapper licensor. This means that the original owner, the wrapper developer(s), and the current wrapper owner cannot suspend the operation of the wrapper. The <Licensing_Suspend> will put the wrapper in an inactive state. During this inactive state, the wrapper will still be accessible by the hosting application or by other components. However, the wrapper will provide no functionality to them. Its only function during the inactive state is to respond "politely" to callers and to save calls information (if the <Tracking_TrackCalls> or <Tracking_TrackAdminActivities> are specified.) The interface for <Licensing_PrintLicenseInfo> will only print the current licensing information. Previous licensing activities can be obtained through the interfaces for the <Tracking_Licenses>, <Tracking_PrintTrackingInfo>, and <Tracking_SendTrackingInfo> constructs.

8.2.9.5 Managing the Wrapper's State

Built-in scheduler. There are situations when compositional wrappers need to schedule the tasks that they perform. For example, many of the smart features presented in this dissertation require the wrapper to send reports or interact with its owners and/or developers. Such interactions will normally take a significant amount of time from the system, due to the amount of information to be exchanged between the wrapper and its

owners and/or developers. Because this can greatly degrade system throughput, it is often best if such interactions are scheduled for certain periods of the day and/or week. In this situation, a scheduler must monitor scheduled tasks and execute them at required times. Because of its complicated structure and requirements, the scheduler is best implemented at the wrapper level. This way the scheduler can schedule its own tasks as well as the tasks for all of its components.

Implementing schedulers in compositional wrappers has many advantages, including:

- Scheduling downtime periods to enable the wrapper to interact with its owner(s) or developer(s).
- Scheduling downtime periods to check for new updates of the compositional wrapper and of its components.
- Synchronizing the compositional wrapper's activity or downtime periods with those of the hosting system.
- Enforcing certain licensing models and monitoring their validity.

Another potential application of the scheduler is to report to other components or wrappers information concerning the periods when the smart wrapper is *active*. (A wrapper is said to be active when it can be called or "considered" by other components or wrappers.) Information of interest about each scheduled period includes: *active start time*, *active end time*, *initialization script* (to be executed at the beginning of each peri-

od), *inactivation script* (to be executed at the end of each period), and *inactive script* (to describe how the wrapper responds to the outer world when it is not active).

Changing roles. Another smart feature to consider in this category is to allow the compositional wrapper to change roles. For example, a student taking an instructional course may also serve as a teaching assistant and need to occasionally take the role of an instructor. To support such behavior, the compositional wrapper would need to monitor its state and to react to state changes according to a pre-planned schedule. Implementing such behavior might require the use of Artificial Intelligence or Knowledge-Based techniques. It would also require experts to encode knowledge about the domain in which the wrapper operates. If a wrapper is to react to changes in its state (attributes), it must be able to react when any of its attributes is added, modified, or deleted.

To support the capability of role changing, each attribute might have scripts attached to it such as: *if-changed* (to be executed when the attribute's value changes); *if-added* (to be executed when the attribute is first created); and *if-removed* (to be executed when the attribute is removed). The *if-added* and *if-removed* scripts can be made global to the wrapper level.

Another form of changing role is to make the wrapper's functionality evolve based on the specific needs of its environment or the host application [HEIN01]. Wrappers that support mathematical and statistical algorithms are candidates for implementing this feature. Such wrappers will change behavior with the introduction of new or modified algorithms.

Intelligent response. When something goes wrong, smart compositional wrappers should respond more intelligently than just raising exceptions. For example, when another component or wrapper communicates with a smart wrapper, the smart wrapper might respond "politely" to the call by informing the caller that it cannot provide the requested information. If a directory service were available to the wrapper, it might be able to suggest other components or wrappers that might be able to service its requests. The smart wrapper might also report such a call to the owners and/or developers of the wrapper or of the wrapper components. The owners and/or developers could use this information to support decisions about the services to provide in subsequent versions of the components or wrapper.

The smart wrapper should also have some mechanism of "conversing" with callers and storing information passed from them. Passed information could include the wrapper's and callers' domains of expertise and services provided. Other components or wrappers could use such a mechanism to contact a smart wrapper and "introduce" themselves to it.

Intelligent restart. Compositional wrappers should be able to store their current state and restore it upon subsequent invocations by the system(s) that are using them. In addition, compositional wrappers should be able to resume execution from where they left.

The following constructs implement some of the features listed in this section. Corresponding interfaces for each of the following constructs will be presented in Section 14.7.5.

```
<SmartFeatures_StateDeclaration> ::= 'State' '=' '(' ( 'ALL' |
      ( <State_Option> ( ',' <State_Option> )* ) ) ')' ';'

<State_Option> ::= <State_Schedule> | <State_PrintSchedule> |
      <State_Pause> | <State_Resume>

<State_Schedule> ::= 'Schedule'
<State_PrintSchedule> ::= 'PrintSchedule'
<State_Pause> ::= 'Pause'
<State_Resume> ::= 'Resume'
```

These constructs allow the current wrapper's owner to schedule the periods when the wrapper is active and able to deliver its functionality. The constructs also allow hosting applications to pause the wrapper's activities and resume them at later time. The <State_Schedule> will program the wrapper to become active at certain periods and inactive at other periods. The <State_Pause> and <State_Resume> options work as a pair and must be specified together. The <State_Pause> will put the wrapper in an inactive state. Unlike the <Licensing_Suspend>, the wrapper in the <State_Pause> inactive state will not be accessible by the hosting application or by any other components.

8.2.9.6 Managing Wrapper and Component Updates

[SEGA93] noted that, in order for a system to update its components, the system's logic should be preserved. This statement could also be applied to components and wrappers, requiring components to be updated in the correct order and at the correct time. The smart wrapper could assure such a mechanism by synchronizing the updates at fixed intervals. If a component lacks this "self-updating" capability, the smart wrapper can acquire any updates on the components' behalf.

```
<SmartFeatures_UpdateDeclaration> ::= 'Update' '=' '(' ( 'ALL' |
      ( <Update_Option> ( ',' <Update_Option> )* ) ) ')' ';'

<Update_Option> ::= <Update_SelfAutoUpdate> |
      <Update_ComponentsAutoUpdate>

<Update_SelfAutoUpdate> ::= 'SelfAutoUpdate' '(' <Url> ','
      <UpdateFrequency> ',' <Update_AutoUpdateOptions> ')'

<UpdateFrequency> ::= 'WhenAvailable' | <TimeEventChoice>

<Update_ComponentsAutoUpdate> ::= 'ComponentsAutoUpdate' '('
      <Identifier> ',' <Identifier> ',' <UpdateFrequency>
      ',' <Update_AutoUpdateOptions> ')'

<Update_AutoUpdateOptions> ::= '{' (
      'PreReminder' '(' <When> ',' <ReminderFormat> ')' |
      'PreAuthorization' |
      'PostReminder' '(' <When> ',' <ReminderFormat> ')' |
      'PostAuthorization' | 'DownloadOnly' |
      'AutoInstall' '(' <When> ',' <InstallationFormat> ')' ) '}'

<When> ::= <Digits> ('days' | 'weeks' | 'months') | 'immediately'

<ReminderFormat> ::= 'AlertBox' | 'Email'

<InstallationFormat> ::= 'Overwrite'
```

These complex constructs will help the compositional wrappers to perform self-updating of their files and the files of its components. The above constructs offer many options to help in performing the updates easily and correctly. The constructs allow a

compositional wrapper to check for new updates, download them, and install them (all without any user intervention.) However, to give wrapper owners full control on the updating process, some constructs are provided to keep the owner informed and in control. When the self-auto updating smart feature is specified, the specification writer is required to enter an initial URL to the new updates specification file. This URL can later be changed using one of the interfaces that wrapper generators will provide. The update specifications for the components will be determined by consulting the new updates' URL of each component that was provided through the <NewUpdatesDeclartion> construct (discussed in Section 7.1.12.) The first <Identifier> in the <Update_ComponentsAutoUpdate> construct refers to the name of a <ProfileSpecification> section and the second <Identifier> refers to the name of a <CreditCardSpecification> or a <BankAccountSpecification> section. The information in the personal and payment profile sections will be sent to component vendors when performing the components auto-update.

The following is an example of using the <SmartFeatures_UpdateDeclaration> construct:

```
Update = (ComponentsAutoUpdate (MyProfile, MyCard,
            Every 2 months,
            { PreReminder (1 week, email),
              PreAuthorization,
              AutoInstall (immediately, overwrite) } )
);
```

This statement requires the generated wrapper to perform a bi-monthly check for new updates of each of the components that the wrapper encases. One week before per-

forming the check for new updates, the generated wrapper should send an email reminder to the wrapper user. This gives the user a chance to prepare for the new updates, such as setting the environment or allocating the staff needed to do the actual update. The generated wrapper will use the new updates' URLs specified through the `<NewUpdatesDeclartion>` construct (discussed in section 7.1.12.) If it detects a new update, then the wrapper will get an authorization (from its current owner) to download the new update. Once it is authorized, it downloads the new update and immediately installs the update files in their designated folders, overwriting any existing files with the same names as the files in the new update. When performing the downloading of the new update, the compositional wrapper uses the `MyProfile` personal information and the `MyCard` payment information. `MyProfile` and `MyCard` must match the name of the `<ProfileSpecification>` and `<CreditCardSpecification>` sections, respectively.

8.2.9.7 Debugging Faulty Wrappers

Wrappers could fail because of several reasons. For example, their operation might be inconsistent with their specification after the wrappers or their components are updated. Wrappers might also fail if new components are added to the hosting application or when the environment settings are altered. An intelligent wrapper should be able to monitor its behavior and should have a recovery plan to allow it to correct or ignore the fault. [BAGG97] discussed integrating fault management functionality into generated

wrappers. The basic process of fault management can be described as follows: *monitor* the system's behavior to detect potential failure symptoms as early as possible; *diagnose* and *trace* the fault symptom back to the faulty components to identify the failure scenario; and perform *fault correction* by acting on the faulty components or on the application structure to eliminate the fault.

Fault-injection techniques [KROP98, VOAS98f] can also be used to detect faults and isolate problematic components. These techniques work by injecting "garbage" into the interfaces between components and then observing how the garbage propagates through the system. [ATKI02] proposed building tests into components so components can check their environment at instantiation time. [COOK99] encouraged the use of redundant or replicated components to protect against hardware and/or network faults or to provide better service in terms of speed and availability. When implementing any testing capability into compositional wrappers, one must observe the following issues [KOPP01]: heterogeneity of languages, platforms, and architectures; monitoring and control mechanisms available for distributed software testing; and deadlocks and race conditions that could occur in the tested software.

Simulated execution. Re-execution of a trace often causes problems in other parts of the systems. Thus, facilities for simulated execution may be necessary. (For example, one might simulate executing by logging the states visited without changing system state.) Simulated execution can be of value in debugging mission-critical systems for which live testing might be costly or even impossible.

The following constructs implement only the simulated execution feature. Interfaces for accessing this feature in compositional wrappers will be presented in Section 14.7.7.

```
<SmartFeatures_TestingDeclaration> ::= 'Testing' '=' '(' ( 'ALL' |
      ( <Testing_Option> ( ',' <Testing_Option> )* ) ) ')' ';'

<Testing_Option> ::= <Testing_Pretend>
<Testing_Pretend> ::= 'Pretend'
```

The ALL keyword is provided to account for debugging and testing features that could be added to WSL in the future. Currently, WSL offers only a "pretending" feature. Through the interfaces provided for this smart feature, the user can direct the wrapper to stop performing real functions and to start "pretending" to do such functions. In this "pretending" state, the wrapper only prints informative messages about the calls received from the hosting application or from other components and the recipient components of such calls. With that information, the user can detect any suspicious calls. When the user corrects the problem, it can bring the wrapper back into its normal operation using the StopPretending interface.

8.2.9.8 Limiting Wrapper Access

```
<SmartFeatures_SecurityDeclaration> ::= 'Security' '=' '(' ( 'ALL' |
      ( <Security_Option> ( ',' <Security_Option> )* ) ) ')' ';'

<Security_Option> ::= <Security_LimitCallers>

<Security_LimitCallers> ::= 'LimitCallers' '('
      <CallersSpecifier> ( ',' <CallersSpecifier> )* ')'
```

```
<CallersSpecifier> ::= 'domain' '=' '"' <Domain> '"' |
      'id' '=' <QuotedString>
```

The <Security_LimitCallers> construct allows wrapper users to limit access to their wrappers. The ALL keyword is provided to account for new security options to be added in the future. By restricting the components that can call the wrapper, one can control access to the wrapper. Access restrictions can be specified in two ways (for example, Security = (LimitCallers (domain="ucf.edu", id="1234577888"));.) A user can specify a domain or sub domain from which calls to this wrapper can originate. This is useful if the user has distributed applications where components in different sites talk to each other. The other restriction method is through using explicit component IDs. Only calls from components with such IDs will be honored. Other callers will be rejected.

8.3 Specifying the Wrapper Architecture

A wrapper specification language would not be complete without giving specification writers the ability to express the interactions that could take place among wrapped components and the sequence of those interactions. WSL defines some useful constructs for these purposes. An <ArchitectureSpecification> construct is defined as follows:

```
<ArchitectureSpecification> ::=
      'architecture' <Name> '{' [ <InitializationDeclaration> ]
      [ <CaseRulesDeclaration> ] <CaseDeclaration>+ '}'
```

An architecture specification in WSL consists of three parts. The first part (the <InitializationDeclaration>) defines constructs for making any necessary initialization that can help in expressing the other architecture constructs. Additionally, the initialization section can define constants and initialize data structures. (Simulators could use such initializations when tracing the architecture.) The second part (the <CaseRulesDeclaration>) tells how to expect the various interactions (or scenarios) that could occur among the system components. Here, one can define the sequencing, the concurrency, or the constraints that can restrict the occurrence of each scenario. The third part (the <CasesDeclaration>) defines the individual scenarios that can occur in the system. Each scenario expresses one complete case of a successful operation of the system. (It is assumed that *one and only one* of the cases will occur at runtime, unless specifically noted using the <CaseRulesDeclartion> statement *before* defining the various cases using the <CasesDeclaration> construct.) These three parts are defined in the following constructs:

```
<InitializationDeclaration> ::= 'initialization' '=' <Block>
```

A <Block> is defined in the same way as it is in ADL. Constants or variables that are global to all the architecture can be defined at the beginning of the architecture or before their first use. An initialization section can be used at the beginning of the architecture definition. This initialization section can contain initialization of used structures

or lists, setting default values for used variables, or creating and initializing objects that will be used in the architecture. The following is an example of an initialization block:

```
initialization = {
        OperatorsCount: unsigned int = 2;
        i: int;

        for (i=0; i<OperatorsCount; i++) {
                Operator[i].busy = false;
        }
}
```

```
<CaseRulesDeclaration> ::= 'CaseRules' '='
      '{' <InteractionsList>* '}' ';'
```

The <CaseRulesDeclaration> defines when and how the various cases appear during the system operation. This specification language defines six types of interactions, as can be seen from the following syntax:

```
<InteractionsList> ::= <AtomicInteraction> | <GroupedInteractions>|
      <RepeatedInteractions> | <OptionalInteractions> |
      <UnorderedInteractions> | <AlternateInteractions>
```

An atomic interaction consists only of a name that refers to one of the named cases in the architecture.

```
<AtomicInteraction> ::= [ <Block> ] <Name> [ <Block> ]
```

An interaction can optionally be preceded or followed by a block of code that is used to restrict when and how the case will show up in the system. For example, the PayLater case of the GasStation example, shown in Figure 10, can be restricted to be valid only between 6:00am and 8:00pm using the following block:

```
PayLater  { time >= "0600" && time <= "2000"; }
```

Grouped interactions are one or more of the interaction types grouped together between parentheses. The grouping is used for easy reading of the constructs and for repetition purposes (as will be shown later.)

```
<GroupedInteractions> ::= '(' <InteractionsList> ')'
```

Repeated interactions are one or more interactions that could repeat a specific number of times. The set of interactions can repeat zero or more times if followed by "*", one or more times if followed by "+", zero or one times if followed by "?", or a specific number of times if followed by a positive integer.

```
<RepeatedInteractions> ::= <GroupedInteractions> <RepetitionOption>
<RepetitionOption> ::= '*' | '+' | '?' | <Digits>
```

Optional interactions are one or more interactions that are not required by the system. Optional interactions can occur multiple times similar to repeated interactions.

```
<OptionalInteractions> ::=
      '[' <InteractionsList> ']' [ <RepetitionOption> ]
```

The `or` keyword can be used to separate unordered interactions. This definition of unordered interactions has two implications. The first is that all of the interactions will definitely take place. The second is that it is not known in advance, and in most cases unnecessary to know, which of these interactions will occur first and which one will occur next.

```
<UnorderedInteractions> ::=
      <InteractionsList> ( 'or' <InteractionsList> )*
```

A group of interactions are said to form alternative interactions when it is known that *only* one of the interactions will definitely occur, but it is not known (or it is unimportant to know) which of the interactions will occur.

```
<AlternateInteractions> ::=
      <InteractionsList> ( 'else' <InteractionsList> )*
```

The following is a complete `<CaseRulesDeclaration>` taken from the `GasStation` Example that is shown in Figure 10.

```
CaseRules = { (PrepaidCash else PrepaidCard else
   PayLater  { time >= "0600" && time <= "2000"; } else PayAtPump )*
};
```

The `<CasesDeclaration>` construct forms the heart of the architecture. Each case lists all the transactions that can take place in one scenario. Execution of a case starts when an event that occurred in the system matches the case stimulus. For every event that occurs in the system, the system first checks the next action of the currently executing cases. If it finds a match, then it passes the event to that case. However, if the system does not find an action that matches the event, then it checks the execution plan outlined in the `<CaseRulesDeclaration>` expression. For all cases that can fire in response to the event, the system checks the stimuli of those candidate cases. When it finds one whose stimulus matches the event, the system starts that case. Once a case is started, it is expected that the flow of execution of the system will continue from one transaction to the next until the system reaches the end of the case.

```
<CaseDeclaration> ::= 'case' ( <Name> | <Digits> ) <CaseStimulus> ':'
      '{' <TransactionsList>* '}'

<CaseStimulus> ::= '(' <Name> [ '[' [ <Name> ] ']' ] '.' <Name> ')'
```

Usually, case stimuli will be interface invocations of certain component instances. If the exact instance is known, then it can be specified, as in Operator[OperatorID].AuthorizePump. However, if the stimulus can come from any component instance, then the stimulus will be expressed as in Operator[].AuthorizePump. In order for a case to be selected, the component interface that is listed as a case stimulus must have generated an event of the same type as the current event. If multiple instances of the same component generated the same event, then more than one instance of the stimulated case will be invoked.

Before explaining transactions, brief discussions are provided for other entities of which transactions are composed. Any construct in the form of e.x is called an action x performed by entity e. The entity e is usually another component in the system. x is an interface that is capable of sending or receiving messages.

```
<Action> ::=
      [ <Name> '=' ] <Name> [ '[' <Name> ']' ] '.' <Name> <Arguments>
```

Actions can have their own arguments (for example, Operator.AuthorizePump(Amount)). Arguments are information sent or received by the interface. Action results can be saved to a variable so that results can be referenced in other transactions. For example,

```
ValidationResult = CardAuthorizationAgent.ValidateCard (Card);
```

Any construct in the form of e1.x -> e2.y is called a message that is sent from entity e1 to entity e2. Entity e1 uses interface x to send the message. Entity e2 uses interface y to receive the message.

```
<ActionDeclaration> ::= <MessageAction> | <CodeAction>
<MessageAction> ::= <Action> '->' <Action> | <Action>
<CodeAction> ::= <Block>
```

Code actions are blocks of code used to perform required initialization or setup for the next transactions, process transferred information, or generate new information or structures. If code is enclosed between { }, then all actions in the transaction will fire only if the code returns `true`. By convention, assignment statements within the code block will always yield `true` even if the assigned value is `false` (for example, ID = false;). The following example states that if a customer approaches a specific operator to prepay for gas, then that operator will be busy interacting with the customer. The operator cannot do other actions until it is done with that customer.

```
Customer[CustomerID].PrePay(OperatorID, Amount)
{ Operator[OperatorID].busy = true; };
```

This specification language defines six types of transactions. The transactions follow the same structure as that of interactions.

```
<TransactionsList> ::= <AtomicTransaction> | <GroupedTransactions> |
      <RepeatedTransactions> | <OptionalTransactions> |
      <UnorderedTransactions> | <AlternateTransactions>
```

Transactions are series of actions that take place in sequence and in a cause-and-effect fashion. This means that the first action in the transaction immediately causes the

second action to take place and that the second action immediately causes the third action to take place, and so on. "Pauses" between actions are usually the places to split the group of actions into two ore more transactions. Pauses usually occur when humans are the doers of the next action. However, sometimes devices cause pauses as well. For example, it might take a printer a long time (with respect to other computer actions) to finish printing a report before it sends a "finished" message. In this case, the printing of the report will be part of one transaction and the "finished" message action will be part of the next transaction. When a pause occurs, then it is the place where the actions should be split in two groups. All the actions before the pause will form one transaction. The remaining actions can have one or more transactions.

Any series of messages or code actions that ends with a semicolon form an atomic transaction. This kind of transaction is performed by the system as one unit. If any part of this transaction (either the sender or receiver) fails, then the whole transaction fails. Transaction failure causes the whole case to fail.

```
<AtomicTransaction> ::= <ActionsList> ';'
<ActionsList> ::= <ActionDeclaration>+
```

Transactions can be grouped together to make their specifications readable. Transactions can also be grouped for repetition purposes.

```
<GroupedTransactions> ::= '(' <TransactionsList> ')'
```

To repeat a transaction or a series of transactions a specific number of times, use repeated transactions.

```
<RepeatedTransactions> ::= <GroupedTransactions> <RepetitionOption>
```

To repeat a set of transactions *zero* or more times, enclose such transactions within parenthesis and put an asterisk after the right parenthesis. For example,

(Customer.PumpGas or Pump.UpdateDisplay)*

To repeat a transaction or a series of transactions *one* or more times, enclose such transactions within parenthesis and put a plus sign after the right parenthesis. For example,

(Customer.PumpGas or Pump.UpdateDisplay)+

To repeat a transaction or a series of transactions exactly *N* times, enclose such transactions within parenthesis and put the number of repetition times after the right parenthesis. For example, to repeat the following transaction five times, put the number 5 after the transaction:

(Customer.PumpGas or Pump.UpdateDisplay)5

Optional actions or transactions (those whose occurrence is unknown) are enclosed between brackets (for example, [Customer.TakeReceipt;]).

```
<OptionalTransactions> ::=
      '[' <TransactionsList> ']' [ <RepetitionOption> ]
```

If the order of two or more actions within a transaction is not relevant, then such actions are joined with the `or` keyword (for example, Customer.PumpGas or Pump.UpdateDisplay.) This does not mean that either the Customer.PumpGas or the

Pump.UpdateDisplay action will occur and not both. Rather, this means that both of the actions will definitely occur, but, from the architectural perspective, it does not matter which of the actions takes place first.

```
<UnorderedTransactions> ::=
      <TransactionsList> ( 'or' <TransactionsList> )*
```

If unordered actions occur among other actions in a transaction, then one must enclose such actions between parentheses. For example,

```
Customer.LiftNozzle ->
(Customer.PumpGas or Pump.UpdateDisplay)* ->
Customer.ReplaceNozzle;
```

If only one of a set of transactions will occur, then such transactions are joined with the `else` keyword.

```
<AlternateTransactions> ::=
      <TransactionsList> ( 'else' <TransactionsList> )*
```

If-statements can be used to control the execution of transactions. Loops (for-loops or while-loops) can also be used to iterate through a set of transactions a number of times. The <IfStatement>, <ForLoop>, and <WhileLoop> have the same syntax as that defined in ADL.

8.3.1 A Complete Architecture Specification Example

Shown in Figure 10 is the complete architectural specification for a gas station. The gas station problem is a classic example that was originally proposed by Helmbold and Luckham [HELM85] in 1985. Recently, various researchers (for example, [CHEU96, LUCK97, LEMO01]) used this example to discuss architecture and concurrency among software components. In the architecture specification of Figure 10, there are three components interacting. The first component is an *Operator* component, representing a human cashier or the system that the human cashier uses to control interactions with the gas pumps or to swipe customers' credit cards. The *Operator* component could be split into two components: a pure human component and a pure software component. If that to be done, then some of the transactions in the specification of Figure 10 would need to be rearranged. The second component is the *Customer*, representing any human coming to the gas station for the purpose of pumping gas. The third component is the gas *Pump*. There can be any number of instances of the three components active at the same time. Instances of the three components will be saved in arrays referenced using global variables. Note that this example uses a question mark "?" before some variables (for example, ?CustomerID.) This notation says that, at the point where the question mark is used, the value of CustomerID might be different from value of CustomerID before that point. To understand how such designation could be used, let us think about a customer that prepaid an operator for gas. The operator authorized a pump that the customer can use. Now, from the *Pump* component's perspective, the customer that lifts its nozzle and start pumping gas could be different from the one that prepaid the operator. Prefixing variables with question marks are used only for cosmetic purposes and have no effect on

the architecture's content or logic. A variable with or without a question mark is functionally the same.

```
architecture GasStation {
  initialization = {
    OperatorsCount: unsigned int = 2;
    MaxCustomers: unsigned int = 100;
    PumpsCount: unsigned int = 20;

    for (i=0; i<OperatorsCount; i++) {
      Operator[i].busy = false;
    }
    for (i=0; i<MaxCustomers; i++) {
      Customer[i].ID = null;
    }
    for (i=0; i<PumpsCount; i++) {
      Pump[i].busy = false;
    }
  }


  CaseRules = { (PrepaidCash else PrepaidCard else
        PayLater  { time >= "0600" && time <= "2000"; } else
        PayAtPump )* };


  // Case 1: Customer prepaid operator in cash

  case PrepaidCash (Customer[].PrePay): {

    // RandomUnused will return a number that is not currently
    // assigned to ID field of each element in the Customer array

    const CustomerID = RandomUnused(Customer[]{ID}),
          OperatorID, PumpID: unsigned int;

    const Amount, Refund: float;

    { Customer[CustomerID].ID = CustomerID; }

    {
      for (i=0; i<OperatorsCount; i++) {
        if (!Operator[i].busy) {
          OperatorID = i;
          break;
        }
      }
    }
```

```
Customer[CustomerID].PrePay(OperatorID, Amount)
{ Operator[OperatorID].busy = true; };

Operator[OperatorID].ReceivePayment(CustomerID, Amount)
{
  for (i=0; i<PumpsCount; i++) {
    if (!Pump[i].busy) {
      PumpID = i;
      break;
    }
  }
};

Operator[OperatorID].AuthorizePump(PumpID, Amount)
  -> Pump[PumpID].ReceiveAuthorization(OperatorID, Amount)
  { Pump[PumpID].busy = true;
    Operator[OperatorID].busy = false; };

// The ?CustomerID says that this might be the customer with
// CustomerID or it might be someone else.  The pump doesn't
// care, but is tied with amount paid by CustomerID

Customer[?CustomerID].LiftNozzle(PumpID) ->
  Pump[PumpID].ResetDisplay;

( Customer[?CustomerID].RequestGas(PumpID) ->
  Pump[PumpID].DispenseGas
  -> ( ( Total = Pump[PumpID].CalculateCharges;
         Pump[PumpID].UpdateDisplay (Total); )
       or Customer[?CustomerID].PourGas
     );
)*;

Customer[?CustomerID].ReplaceNozzle(PumpID)
or
( Pump[PumpID].StopGas { Total == Amount; };
  Customer[?CustomerID].ReplaceNozzle;
);

( Pump[PumpID].PrintReceipt
  or
  Pump[PumpID].NotifyOperator(OperatorID, Total)
);

Operator[OperatorID].ReceiveNotification (PumpID, Total);

// Customer might wish not to take the receipt
[ Customer[?CustomerID.TakeReceipt; ]

// Customer might have pumped all the gas he prepaid for and so
// there is no refund to get back.  If he has pumped less than
// what he has prepaid for, he still might wish not to take
// refund.
```

```
  [
    Customer[?CustomerID].RequestRefund;

    {
      for (i=0; i<OperatorsCount; i++) {
        if (!Operator[i].busy) {
          ?OperatorID = i;
          break;
        }
      }
      Operator[?OperatorID].busy = true;
    };

    Refund = Operator[?OperatorID].CalculateRefund (Amount, Total);

    Operator[?OperatorID].GiveRefund (Refund) { Refund > 0; };

    [ Customer[?CustomerID].ReceiveRefund (Refund)
      { Refund > 0; }; ]
  ]

  {
    Pump[PumpID].busy = false;
    Operator[OperatorID].busy = false;
    Customer[CustomerID].ID = null;
  }
}


// Case 2:  Customer prepaid operator using credit or debit card

case PrepaidCard (Customer[].GiveCard): {


  // RandomUnused will return a number that is not currently
  // assigned to ID field of each element in the Customer array

  const CustomerID = RandomUnused(Customer[]{ID}),
        OperatorID, PumpID: unsigned int;
  const Card: CreditCard;

  { Customer[CustomerID].ID = CustomerID; }

  {
    for (i=0; i<OperatorsCount; i++) {
      if (!Operator[i].busy) {
        OperatorID = i;
        break;
      }
    }
  }
```

```
Customer[CustomerID].GiveCard(OperatorID, Card)
{ Operator[OperatorID].busy = true; };

Operator[OperatorID].ReceiveCard(CustomerID, Card);

(
  Operator[OperatorID].GetAuthorization (Card)
    -> CardAuthorizationAgent.ReceiveAuthorizationRequest
       (OperatorID, Card);

  ValidationResult = CardAuthorizationAgent.ValidateCard (Card);

  CardAuthorizationAgent.SendValidationResult
      (OperatorID, Card, ValidationResult);

  ValidationResult =
      Operator[OperatorID].ReceiveAuthorizationResult (Card,
      ValidationResult);

  // This will cause the enclosing loop to re-iterate if the
  // tested condition is true

  { repeat if (ValidationResult != SUCCESS); }
)+;


// This section is optional because the customer might wish not
// get gas from this station

if (ValidationResult == SUCCESS) {
[       {
    for (i=0; i<PumpsCount; i++) {
      if (!Pump[i].busy) {
        PumpID = i;
        break;
      }
    }
  }

  Operator[OperatorID].AuthorizePump(PumpID)
  -> Pump[PumpID].ReceiveAuthorization(OperatorID)
  { Pump[PumpID].busy = true;
    Operator[OperatorID].busy = false; };

  Customer[?CustomerID].LiftNozzle(PumpID) ->
      Pump[PumpID].ResetDisplay;

  // The ?CustomerID says that this might be the customer with
  // CustomerID or he might be someone else.  The pump doesn't
  // care, but is tied with amount paid by CustomerID


  ( Customer[?CustomerID].RequestGas(PumpID) ->
```

```
        Pump[PumpID].DispenseGas
        -> ( Pump[PumpID].UpdateDisplay or
             Customer[?CustomerID].PourGas )
      )*;

      Customer[?CustomerID].ReplaceNozzle(PumpID) ->
        Amount = Pump[PumpID].CalculateCharges
        -> Pump[PumpID].NotifyOperator(OperatorID, Amount)

      Operator[OperatorID].ReceiveNotification (PumpID, Amount)
        -> Operator[OperatorID].RequestAmount (Card, Amount)
        -> CardAuthorizationAgent.ReceiveChargeRequest
           (OperatorID, Card, Amount);

      ConfirmationCode = CardAuthorizationAgent.ChargeCard
                         (Card, Amount);

      CardAuthorizationAgent.SendConfirmationCode
          (OperatorID, Card, ConfirmationCode);

      Operator[OperatorID].ReceiveConfirmationCode
        (Card, ConfirmationCode)
        -> Operator[OperatorID].PrintSlip;

      Operator[OperatorID].HandOnSlip;

      Customer[CustomerID].SignSlip;

      [ Customer[CustomerID].TakeReceipt; ]
    ] };

    {
      Pump[PumpID].busy = false;
      Operator[OperatorID].busy = false;
      Customer[CustomerID].ID = null;
    }
  }


  // Case 3:  Customer pumps gas first and then goes inside to pay
  // operator

  case PayLater (Customer[].LiftNozzle): {

    // RandomUnused will return a number that is not currently
    // assigned to ID field of each element in the Customer array

    const CustomerID = RandomUnused(Customer[]{ID}),
          OperatorID, PumpID: unsigned int;
    const Card: CreditCard;

    { Customer[CustomerID].ID = CustomerID; }
```

```
Customer[CustomerID].LiftNozzle(PumpID)
{
  for (i=0; i<OperatorsCount; i++) {
    if (!Operator[i].busy) {
      OperatorID = i;
      break;
    }
  }
}
-> Pump[PumpID].GetAuthorization (OperatorID);

Operator[OperatorID].ReceiveAuthorizationRequest(PumpID)
  -> Operator[OperatorID].AuthorizePump (PumpID)
  -> Pump[PumpID].ResetDisplay;

( Customer[CustomerID].RequestGas(PumpID)
  -> Pump[PumpID].DispenseGas
  -> ( Pump[PumpID].UpdateDisplay or
       Customer[CustomerID].PourGas )
)*;

Customer[CustomerID].ReplaceNozzle(PumpID)
  -> Total = Pump[PumpID].CalculateCharges
  -> Pump[PumpID].NotifyOperator(OperatorID, Total)
  -> Operator[OperatorID].ReceiveNotification (PumpID, Total)

{
  for (i=0; i<OperatorsCount; i++) {
    if (!Operator[i].busy) {
      OperatorID = i;
      break;
    }
  }
}

(
  Customer[CustomerID].PayCash(?OperatorID, Amount)
  { Operator[?OperatorID].busy = true; };

  Operator[?OperatorID].ReceivePayment(CustomerID, Amount);

  Refund = Operator[?OperatorID].CalculateRefund (Amount, Total);

  Operator[?OperatorID].GiveRefund (Refund) { Refund > 0; };

  Operator[?OperatorID].PrintReceipt
  or
  [ Customer[CustomerID].ReceiveRefund (Refund) { Refund > 0; }; ]

  [ Customer[CustomerID].TakeReceipt; ]
)
else
(
```

```
      Customer[CustomerID].GiveCard(?OperatorID, Card)
      { Operator[?OperatorID].busy = true; };

      Operator[?OperatorID].ReceiveCard(CustomerID, Card);

      (
        Operator[?OperatorID].RequestAmount (Card, Total)
        -> CardAuthorizationAgent.ReceiveChargeRequest
           (?OperatorID, Card, Amount)
        -> ValidationResult = CardAuthorizationAgent.ValidateCard
                              (Card);

        if (ValidationResult == SUCCESS) {
          ConfirmationCode = CardAuthorizationAgent.ChargeCard
                             (Card, Total);

          CardAuthorizationAgent.SendConfirmationCode
            (?OperatorID, Card, ConfirmationCode);

          Operator[?OperatorID].ReceiveConfirmationCode
            (Card, ConfirmationCode)
        }

        // This will cause the enclosing loop to re-iterate if the
        // tested condition is true

        { repeat if (ValidationResult != SUCCESS); }
      )+;

      Operator[?OperatorID].PrintSlip;

      Operator[?OperatorID].HandOnSlip;

      Customer[CustomerID].SignSlip;

      [ Customer[CustomerID].TakeSlipCopy; ]

    };

    {
      Pump[PumpID].busy = false;
      Operator[?OperatorID].busy = false;
      Customer[CustomerID].ID = null;
    }
  }


  // Customer uses his/her credit or debit card to charge for gas
  // at the pump

  case PayAtPump (Customer[].SwipeCard): {

    // RandomUnused will return a number that is not currently
```

```
// assigned to ID field of each element in the Customer array

const CustomerID = RandomUnused(Customer[]{ID}),
      OperatorID, PumpID: unsigned int;
const Card: CreditCard;

{ Customer[CustomerID].ID = CustomerID; }

{
  for (i=0; i<PumpsCount; i++) {
    if (!Pump[i].busy) {
      PumpID = i;
      break;
    }
  }
}

(
  Customer[CustomerID].SwipeCard(PumpID, Card)
    { Pump[PumpID].busy = true; }
    -> Pump[PumpID].GetAuthorization (Card)
    -> CardAuthorizationAgent.ReceiveAuthorizationRequest
      (PumpID, Card);

  ValidationResult = CardAuthorizationAgent.ValidateCard (Card);

  CardAuthorizationAgent.SendValidationResult
     (PumpID, Card, ValidationResult);

  ValidationResult = Pump[PumpID].ReceiveAuthorizationResult
                     (Card, ValidationResult);

  // This will cause the enclosing loop to re-iterate if the
  // tested condition is true

  { repeat if (ValidationResult != SUCCESS); }
)+;

{
  for (i=0; i<OperatorsCount; i++) {
    if (!Operator[i].busy) {
      OperatorID = i;
      break;
    }
  }
}

Pump[PumpID].ResetDisplay -> Pump[PumpID].NotifyOperator
     (OperatorID)
  -> Operator[OperatorID].ReceiveNotification (PumpID);

Customer[CustomerID].LiftNozzle(PumpID);
```

```
    ( Customer[CustomerID].RequestGas(PumpID)
      -> Pump[PumpID].DispenseGas
      -> ( Pump[PumpID].UpdateDisplay or
         Customer[CustomerID].PourGas )
    )*;

    Customer[?CustomerID].ReplaceNozzle(PumpID)
      -> Amount = Pump[PumpID].CalculateCharges
      -> Pump[PumpID].RequestAmount (Card, Amount)
      -> CardAuthorizationAgent.ReceiveChargeRequest
        (PumpID, Card, Amount);

    ConfirmationCode = CardAuthorizationAgent.ChargeCard
                       (Card, Amount);

    CardAuthorizationAgent.SendConfirmationCode
      (PumpID, Card, ConfirmationCode);

    {
      for (i=0; i<OperatorsCount; i++) {
        if (!Operator[i].busy) {
          OperatorID = i;
          break;
        }
      }
    }

    Pump[PumpID].ReceiveConfirmationCode (Card, ConfirmationCode)
      -> Pump[PumpID].PrintReceipt
      -> Pump[PumpID].NotifyOperator (?OperatorID)
      -> Operator[OperatorID].ReceiveNotification (PumpID);

    [ Customer[CustomerID].TakeReceipt; ]
  }
}
```

Figure 10: A gas station architectural specification

8.4 Describing People

This section describes how to enter the personal details of individuals. The wrapper specification will contain at least one occurrence of this section to describe the wrap-

per's owner and developers. The wrapper's specification could also contain more occurrences of this section to describe the wrapper specification developers and the contact people who are listed in the wrapper owner's profile. The listed identifiers in these various constructs must match the <Identifier> of the <PersonSpecification> section shown below. The main productions of this section are defined as:

```
<PersonSpecification> ::= 'person' <Identifier> '{' <PersonBody>+ '}'

<PersonBody> ::= <NameDeclaration> | ( <EmailDeclaration> |
      <PhoneDeclaration> | <FaxDeclaration> )*
```

The syntax of all the constructs on the right side of the <PersonBody> production are discussed in Section 5.3.5.

8.5 Specifying User Profiles

```
<ProfileSpecification> ::=
      'profile' <Identifier> '{' <ProfileBody>+ '}'
```

Wrapper generators use user profiles during the component acquisition process. Information included in this specification is sent to component suppliers upon their request. The <ProfileBody> construct is defined as:

```
<ProfileBody> ::= <NameDeclaration> | ( <UrlDeclaration> |
      <EmailDeclaration> | <PhoneDeclaration> | <FaxDeclaration> )* |
      <BusinessAddressDeclaration> | <Maililng_addressDeclaration> |
      <ContactsDeclaration>

<BusinessAddressDeclaration> ::= 'business_address' '='
      <MultiLineString> ';'
```

```
<MailingAddressDeclaration> ::= 'mailing_address' '='
      <MultiLineString> ';'

<ContactsDeclaration> ::= 'contacts' '='
      <Identifier> ( ',' <Identifier> )* ';'
```

The only new constructs in this section are the <BusinessAddressDeclaration>, <MailingAddressDeclaration>, and the <ContactsDeclaration>. These constructs refer to the development organization's business and mailing addresses and the organization's contacts list, respectively. The syntax for all the other constructs was given in Section 5.3.5. The <ContactsDeclaration> construct helps vendors in contacting the right people in development organizations that are using their components. For each of the identifiers listed in the <ContactsDeclaration> construct, there needs to be a separate <PersonSpecification> section within the specification document. The organization's name and business address are the only required constructs in this section. All other constructs are optional.

8.6 Specifying Credit Cards

```
<CreditCardSpecification> ::= 'CreditCard' <Identifier>
'{' <CreditCardBody>+ '}'
```

Wrapper generators use credit card information during the component acquisition process if specified by the system integrator as the method of payment to acquired components. Information included in this specification is sent to vendors when finalizing the

component acquisition process. Specification writers can include as many <CreditCardSpecification> sections in the same wrapper specification document. That way, organizations can specify different credit cards at different times. The <CreditCardBody> construct is defined as:

```
<CreditCardBody> ::= <CreditCardTypeDeclaration> |
      <CreditCardNumberDeclaration> |
      <CreditCardExpirationMonthDeclaration> |
      <CreditCardExpirationYearDeclaration> |
      <NameDeclaration> | <AddressDeclaration>
```

The <NameDeclaration> and <AddressDeclaration> constructs were given in Section 5.3.5. <AddressDeclaration> refers to the card holder's billing address. The syntax for the other constructs is given below. All of the constructs in this section are required and must show up only once in a <CreditCardSpecification> section.

```
<CreditCardTypeDeclaration> ::= 'type' '=' <CreditCardChoice> ';'

<CreditCardChoice> ::=
      'Visa' | 'MasterCard' | 'DinersClub' | 'AmericanExpress'

<CreditCardNumberDeclaration> ::= 'number' '='
   <FourDigits>['-']<FourDigits>['-']<FourDigits>['-']<FourDigits> ';'

<CreditCardExpirationMonthDeclaration> ::=
      'expiration_month' '=' <Month> ';'

<CreditCardExpirationYearDeclaration> ::=
      'expiration_year' '=' <FourDigits> ';'
```

8.7 Specifying Bank Accounts

The <BankAccountSpecification> is similar to the <CreditCard-Specification> section. It describes the organization's bank account information. All constructs are required except for those that describe the bank's emails, phone numbers, and fax numbers. These three constructs can appear more than once in the same <BankAccountSpecification> section. All other constructs can appear only once. The syntax for the <BankAccountSpecification> section is shown below:

```
<BankAccountSpecification> ::= 'BankAccount' <Identifier>
      '{' <BankAccountBody>+ '}'

<BankAccountBody> ::= <BankAccountNumberDeclaration> |
      <NameDeclaration> | <BankNameDeclaration> |
      <BankAddressDeclaration> | <BankEmailDeclaration> |
      <BankPhoneDeclaration> | <BankFaxDeclaration>

<BankAccountNumberDeclaration> ::= 'number' '=' <QuotedString> ';'
<BankNameDeclaration> ::= 'bank_name' '=' <QuotedString> ';'
<BankAddressDeclaration> ::= 'bank_address' '=' <MultiLineString>';'
<BankEmailDeclaration> ::= 'bank_email' '=' <Email> ';'
<BankPhoneDeclaration> ::= 'bank_phone' '=' <QuotedString> ';'
<BankFaxDeclaration> ::= 'bank_fax' '=' <QuotedString> ';'
```

8.8 A Complete Wrapper Specification Example

Aside from the various constructs and sections that a wrapper specification can contain, the following example also demonstrates the use of ranking scheme that was discussed in Section 8.1. Unlike the example presented in Section 6.2, the interfaces defined in this example are defined separately (i.e., not enclosed within a module.) Some of the constructs that are shown in this example will be matched to the corresponding constructs in component specifications. For example, taking the example component specification that was shown in Figure 9, the *technologies* clause in Figure 11 will be matched

to the *technology* clause in Figure 9. The two clauses will match since *JavaBean* exists in both clauses. Similarly, the *databases* clause in both examples will match since the one in the component specification of Figure 9 does not require any kind of storage medium. There are clauses in both examples that are specific to compositional wrappers or to components. For example, the *owner* and *developer* clauses in the WSL specification of Figure 11 is specific to compositional wrappers while the *new_updates* in the CSL specification of Figure 9 is only applicable to components. Such kind of clauses will not be used in the matching process.

The category of clauses that gives narrative information, such as the *name* and *description* clauses in wrapper and component specifications, will not be used in the matching process.

The *interface* section exists in both specification languages but have different meanings. Interface specifications in the WSL specification describe the overall user requirements. Such interfaces do not need to match interfaces with the same name and characteristics in component specifications. On the other side, the *interface* sections in CSL specifications describe actual interfaces in components and serve as a contract between the component vendor and the component users. Any deviation of the component's behavior from that stated in such interfaces will be considered a violation of such contracts.

```
wrapper MyBank {

  name = "Bank Accounts Manager";
  description = {
    This component is a simulation of a simple bank.
```

```
    It consists of two interfaces: deposit and withdraw.
    The component is written using Java 1.1.  Both interfaces
    throw exceptions.
  };
  version = 1.3;
  owner = Me;
  developers = (Developer1, contact1);

  suppliers = ("The Object Management Group", "http://www.omg.org");
  technologies = (JavaBean, VBX);
  operating_systems (2) = ( winme(3), win2000(1), winnt4.0(4) );
  databases (20) = ( FILE(20), MySQL(1) );
  languages (55) = (Java(3), PHP(16));
  formats = (source, object);
  license_models = (FREE, DEMO);
  license_types = (SINGLE, SITE);
  single.LicenseCost = $673.30;
  single.UpgradeCost = $20;
  site.MaintenanceCost = $430.12;
  site.SupportCost = $230.33;
  developer.TrainingCost = $250;
  developer.InstallationCost = $100;
  support_methods = (EMAIL, ONLINE, FORUM);
  payment_methods = (CHECK, PAYPAL);

  interfaces = deposit, withdraw;

  software_constraints = {
    other = "No constraints";
  };

  hardware_constraints = {
    RAM = 64MB;
    storage = 30MB;
    other = "No additional constraints";
  };

  network_constraints = {
    other = "No constraints";
  };

  SmartFeatures = {
    OwnerAuthenticationCode = "112233445566";
    selfdoc = (DisplayIcon ("mywrapper.gif"),
       ListComponentsInfo, ListInterfaces);
    Tracking = (TrackCalls, SendTrackingInfo, TrackHistory);
    Licensing = (PrintCurrentOwner, PrintOwners, PrintLicenseInfo);
    State = (Schedule, PrintSchedule);
    Update = (ComponentsAutoUpdate (MyProfile, MyCard,
      Every 2 months),
      {PreReminder (1 week, email),
         PreAuthorization,
         AutoInstall (immediately, overwite)
```

```
      }
    );
    Testing = (Pretend);
    Security = (LimitCallers
        (domain="ucf.edu",
        id="1234577888")
    );
  };

  personal_info = Profile1;
  payment_info = (CREDIT_CARD, credit1);
  temp_directory = "$GENERATOR_HOME/temp";
  installation_directory = "C:\MyApps\Bank";
  wrapping_options = (EncryptWrapper, EncryptComponents,
                      CompressWrapper, CompressComponents,
                      FastLoading, PreserveMemory);
}

interface deposit {
  balance: float;

  DepositItems: hardware (money, check, coin);
  // This defines a list of hardware items that will be used in
  // making deposits

  Receipt: hardware (paper);
  // This defines the receipt that the customer will get when
  // making a deposit

  input Amount = { "Deposited amount", +float, any, any, parameter };
  // This input will be any positive floating point number and
  // will be passed as a parameter

  input Deposit = { "Deposited items", DepositItems, any, any,
                    device };

  output Receipt = { "Printed Receipt", Receipt, any, any, device };
  // This output will be in the form of a printed receipt

  output O1 = { "success or failure of operation", boolean,
                null, true, return };
  // The output should be true.
  // I expect the output to be returned in the function name

  output = { none };
  // It should not generate any other outputs

  behavior = {
    call (*, accountID, PIN, Amount): {

      accountID <> null && accountID == reference(Account) &&
      // Make sure the supplied accountID is not null and is a
      // reference to Account
```

```
      accountID->pin == PIN &&
      // Make sure the supplied PIN# is the same as that of accountID

      exists (accountID, AccountsDatabase) &&
      // This will make sure the supplied accountID exists in the
      // AccountsDatabase

      balance = get_balance (accountID) &&
      // This will save the balance before the deposit is made

      return (O1) == ATM.accept (Deposit, Amount) &&
      // This asks the ATM machine to accept the deposited items
      // from its deposit slot and to return "true" if successful
      // or "false", otherwise.  Assume the ATM has
      // the intelligence to count total amount of the deposited
      // items, it will then return true if that total amount
      // matches Amount.  "accept" is a special method defined
      // just for this device in its device definition file.

      return (O1) == update (accountID (balance), balance+Amount) &&
      // This tells the database AccountsDatabase to update the
      // balance column of the accountID record with the value
      // (balance+Amount). The update operation should be successful

      get_balance (accountID) == balance+Amount &&
      // Now make sure the balance actually reflects the new deposit

      return (O1) == ATM.printReceipt (deposit);
      // This requests from the ATM machine to print a receipt for
      // the user's deposit
    }
  }
  stimuli = {
    call,
    event (menu ("Tools/Deposit"), keyboard ('Ctrl-D'))
  }
}

interface withdraw {
  balance: float;

  DispensedItems: hardware (money);
  // This defines a list of hardware items that will dispensed
  // at withdrawals

  Receipt: hardware (paper);
  // This defines the receipt that the customer will get when
  // making a deposit

  input Amount = { "Withdrawn amount", +float, any, any, parameter };
  // This input will be any positive floating point number and
  // will be passed as a parameter
```

```
output Cash = { "Dispensed items", DispensedItems, any,
                any, device };

output Receipt = { "Printed Receipt", Receipt, any, any, device };
// This output will be in the form of a printed receipt

output O1 = { "success or failure of operation", boolean,
              null, true, return };
// The output should be true.
// I expect the output to be returned in the function name

output = { none };
// It should not generate any other outputs

behavior = {
  call (*, accountID, PIN, Amount): {

    accountID <> null && accountID == reference(Account) &&
    // Make sure the supplied accountID is not null and is a
    // reference to Account

    accountID->pin == PIN &&
    // Make sure the supplied PIN# is the same as that of accountID

    exists (accountID, AccountsDatabase) &&
    // This will make sure the supplied accountID exists in the
    // AccountsDatabase

    balance = get_balance (accountID) &&
    // This will save the balance before the deposit is made

    balance >= Amount &&
    // Make sure there is enough balance for the withdrawn amount

    return (O1) == ATM.dispense (Cash, Amount) &&
    // This asks the ATM machine to dispense withdrawn money
    // through its money dispenser slot and to return "true" if
    // successful or "false", otherwise.  It will return successful
    // only if the dispensed amount equals Amount.  "dispense" is
    // a special method defined just for this device in its device
    // definition file.

    return (O1) == update (accountID (balance), balance-Amount) &&
    // This tells the database AccountsDatabase to update the
    // balance column of the accountID record with the value
    // (balance-Amount). The update operation should be successful

    get_balance (accountID) == balance-Amount &&
    // Now make sure the balance actually reflects the new
    // withdrawal

    return (O1) == ATM.printReceipt (deposit);
```

```
      // This requests from the ATM machine to print a receipt for
      // the user's deposit
    }
  }
  stimuli = {
    call,
    event (menu ("Tools/Withdraw"), keyboard ('Ctrl-W'))
  }
}

profile Profile1 {
  name = "The Compositional Wrappers Group";
  url = "http://www.TheCompositionalWrappersGroup.com";
  email = "webmaster@TheCompositionalWrappersGroup.com";
  phone = "407-999-9999";
  phone = "321-999-9999";
  fax = "407-999-9999";
  fax = "321-999-9999";
  business_address =  {
    1 N. First Drive
    Suite # 9999
    New Florida, FL 77777
  };

  mailing_address =  {
    P.O.Box 777
    London, AL 55555-4444
  };
  contacts = (contact1, contact2);
}

person Me {
  name = " Me Tom ";
  email = "MeMe@TheCompositionalWrappersGroup.com";
  phone = "407-999-9999";
  phone = "321-999-9999";
  fax = "407-999-9999";
}

person Developer1 {
  name = "The Compositional Wrappers Developer";
  email = "developer@TheCompositionalWrappersGroup.com";
  phone = "407-999-9999";
  phone = "321-999-9999";
  fax = "407-999-9999";
}

person contact1 {
  name = "Ms. Johnson";
  email = "Johnson@TheCompositionalWrappersGroup.com";
  phone = "407-999-9999";
  phone = "321-999-9999";
  fax = "407-999-9999";
```

```
}

person contact2 {
  name = "Ed Steward";
  email = "ESteward@TheCompositionalWrappersGroup.com";
  phone = "407-999-9999";
  phone = "321-999-9999";
  fax = "407-999-9999";
}

CreditCard card1 {
  type = VISA;
  number = 9999-9999-9999-9999;
  expiration_month = 5;
  expiration_year = 2010;
  name = "My name";
  billing_address =  {
    P.O.Box 777
    London, AL 55555-4444
  };
}

CreditCard card2 {
  type = MasterCard;
  number = 9999-9999-9999-9999;
  expiration_month = 10;
  expiration_year = 2003;
  name = "My name";
  billing_address =  {
    P.O.Box 777
    London, AL 55555-4444
  };
}

BankAccount account1 {
  number = "000000-00000";
  name = "My name";
  bank_name = "Bank name";
  bank_address =  {
    P.O.Box 777
    London, AL 55555-4444
  };
  bank_email = "webmaster@CompositionalWrappersBank.com";
  bank_phone = "407-999-9999";
  bank_phone = "321-999-9999";
  bank_fax = "407-999-9999";
}

BankAccount account2 {
  number = "000000-00000";
  name = "My name";
  bank_name = "Bank name";
  bank_address =  {
```

```
    P.O.Box 777
    London, AL 55555-4444
  };
  bank_email = "webmaster@CompositionalWrappersBank.com";
  bank_phone = "407-999-9999";
  bank_phone = "321-999-9999";
  bank_fax = "407-999-9999";
}
```

Figure 11: A complete WSL specification for the *MyBank* compositional wrapper

CHAPTER 9: INTRODUCTION TO THE COMPOSITION PROCESS

This research supports a vision of an integrated environment where component integrators command wrapper generators to search for components, acquire them, configure them, wrap them, and, possibly, plug them into their system. This environment is equipped with a powerful and smart wrapper generator along with specialized software agents to carry out the various tasks. Such an environment would require component integrators and wrapper generators to perform many tasks when assembling components into compositional wrappers.

This chapter provides a high-level introduction to the process of assembling and generating compositional wrappers. Such a composition process consists of several well-defined stages. Each stage of the process consists of several activities that build on the activities of the stages that precede it. Some parts of this composition process can be automated; others will require humans to perform the tasks that are beyond the current capabilities of wrapper generators or software agents.

Table 3 provides a high-level breakdown of the list of activities' involved in developing and operating Compositional Wrappers-Based (CWB) systems. [MINK01] proposed a taxonomy of activities that is similar to that presented here. Their taxonomy was used in a COTS' cost estimation project.

Table 3: High-level breakdown of the activities involved in developing and operating a Compositional Wrappers-Based (CWB) system

0.0 Develop and operate a compositional wrappers-based (CWB) system		
	1.0 Design the CWB system (Chapter 9)	
	2.0 Build the CWB system (Chapter 10 to Chapter 14)	
		2.1 Collect components (Chapter 10 to Chapter 13)
		2.2 Wrap components (Chapter 14)
		2.3 Deliver compositional wrapper (Section 14.11)
		2.4 Assemble components into the CWB system (Section 15.1)
		2.5 Test the CWB system (Section 15.2)
	3.0 Operate the CWB system (Sections 15.3 and 15.4)	

Figure 12 shows the top-level activities involved in developing and operating CWB systems. The activities are shown in three different shadings, depending on their relevance to the work of this dissertation. The heavily shaded activities form the heart of this dissertation. While the lightly shaded activities are outside the focus of the composition process, such activities will be discussed in the relevant chapters of the composition process. Activities that are not shaded are outside the scope of this work. Such activities will only be discussed to the degree required to support the work of this dissertation.

Developing a compositional wrapper will often involve activities other than assembling the wrapper from pre-built components. These activities can include: developing new components from scratch, reengineering components from existing systems, and wrapping existing pieces of software or complete systems so they can be used as components. However, the focus of the composition process described in this dissertation is on assembling compositional wrappers completely from pre-built components. Other types

of components will only be considered when pre-built components are not available or not feasible to acquire.

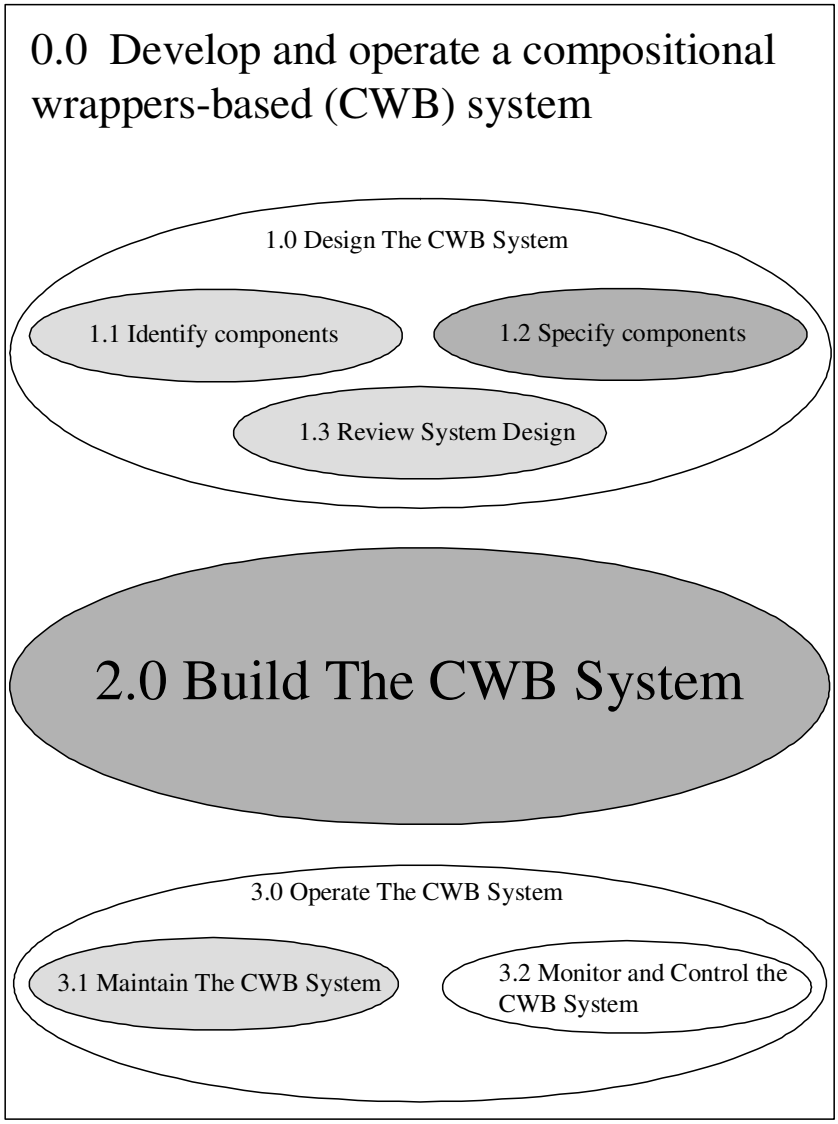

Figure 12: The top-level activities in a CWB system

Subsequent chapters of this document provide a breakdown of the activities of each stage of the composition process and discuss possibilities for the automation of such

activities. Each chapter distributes the expected level of automation of its respective stage, both with and without the use of a wrapper generator, into four levels: no automation (can only be done manually), low level of automation, medium level of automation, and high level of automation. The following are two observations that need to be made about the activities list provided at the beginning of the following chapters.

- The amount of automation in each level is judgmental and depends on understanding the number of actions needed to accomplish each activity. The automation distribution also depends on the weight that the automated actions carry to the total effort needed to perform the activity. For example, a low level of automation could mean less than 20% automation of all the actions in the activity. A medium level of automation could mean the possibility of automating from 20% to 50% of all the actions in the activity. The high level of automation could mean that 50% or more of the actions in the activity can be automated. [ABTS00d] cautioned against an erroneous conclusion that the effort (and hence the automation level) in each stage can be discussed without regard to the activities in other stages. The reality is that the steps are intricately intertwined with the rest of the development process. The activities in one stage may support or be supported by many other process activities [ERDO00].

- The automation of the activities in each stage or the actions that compose those activities could come from various sources. The major contributor of such automation will be the wrapper generator. However, automation could also come from

search agents, middleware agents, server API's, operating system utilities, or any other software with which the wrapper generator can interact.

In the following sections, the composition process is discussed in more detail. Some of the issues involved are outlined and the alternative solutions, if any, are discussed. Each stage of the process is then discussed in separate chapters.

9.1 Composition Process Overview

Figure 13 shows the three parties involved in the composition process

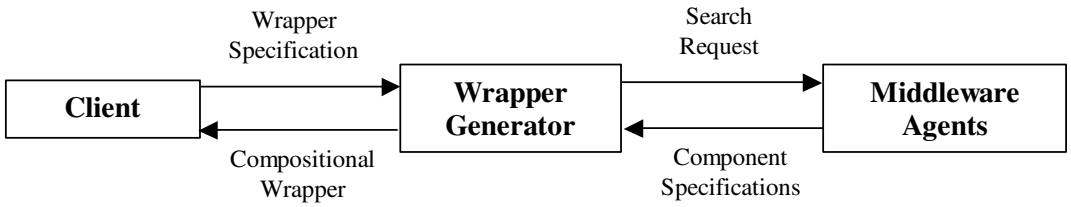

Figure 13: Composition process overview

The *client* consists of the application developer and associated integrated environment. The developer (usually, a human) identifies and specifies needed compositional wrappers. After deciding on the characteristics of the compositional wrapper, the developer passes a request to the *wrapper generator* to locate components that satisfy the wrapper specification. Then, the developer can either wait for the generator to come back

with a result or can continue to work. The wrapper generator will be responsible for providing the compositional wrappers specified by the client. In most cases, the generator will command *middleware agents* to locate and retrieve a set of component specifications for which the union of the provided interfaces of such specifications matches the required interfaces of the wrapper specification. The middleware agents will be responsible for locating components, retrieving the component specifications, comparing the component specifications with the wrapper specifications or the search requests supplied by the wrapper generator, negotiating deals with component suppliers, and handing the component specifications over to the wrapper generator. Upon the receipt of the component specifications, the wrapper generator will notify the developer and submit the result. The developer should have some mechanism on interpreting the received results. If the results look promising to the developer's needs, the developer will command the wrapper generator to start negotiating with the suppliers of the candidate components. The wrapper generator will delegate such task to the middleware agents. If an agreement is reached, the wrapper generator will perform (with the help of the developer and the middleware agents) a detailed evaluation on the candidate components. The generator will then acquire and configure the components that passed the detailed evaluations. After that, the generator wraps the components and generates the compositional wrapper. Upon receiving the compositional wrapper from the wrapper generator, the developer plugs the wrapper in the developed application.

Figure 14 shows how the wrapper generator takes the developer's specification for a new compositional wrapper, collects needed components that satisfy the user's re-

quirement for the new compositional wrapper, and then generates the compositional wrapper. The left side of the diagram illustrates development work. The right side illustrates the component wrapping process. The numbers attached to the activities in the diagram are the same as the numbers assigned to the various activities in each stage of the composition process. Such activities' breakdown will be shown at the beginning of the following chapters.

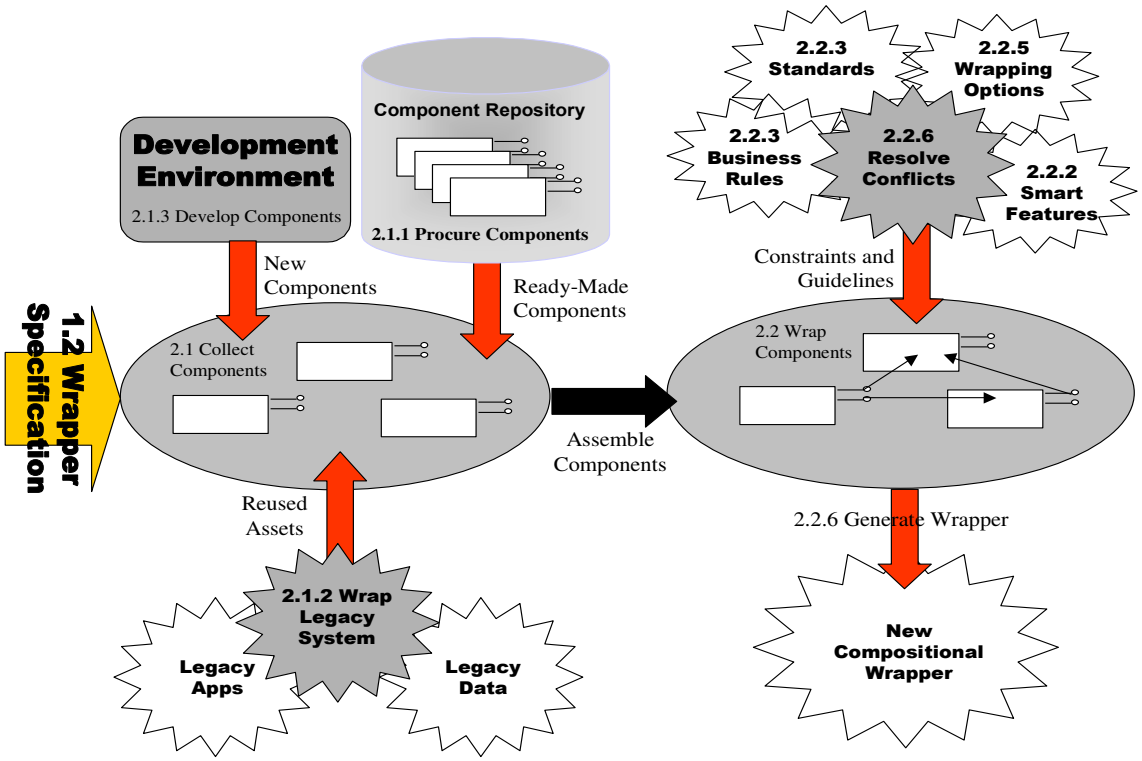

Figure 14: The compositional wrappers development process

9.2 Architectural Design Considerations

During design time, the proposed system can be subdivided into one or more components. The granularity of the components can vary from fine to coarse, as desired by the designers. Fine-grained components require more work at design time while coarse-grained components shift much of the work to the composition stage. Such design work will involve the decomposition of coarse-grained components into finer ones and the specification of the new components. Using coarse-grained components also has the effect of shifting much of the project's cost and time from the design stage to the composition stage. For example, [WINT02] used a coarse-grained approach in their PECOS project. PECOS starts by identifying general-matching, course-grained components. Then, the found components are decomposed into more fine-grained ones. The process continues until the established decomposition is fine-grained enough to be realized.

Specification writers must then develop a complete specification for each compositional wrapper. The specification writer can use the wrapper specification language, discussed in Chapter 8, to provide the specification of the target compositional wrapper.

Architecture design of a system decomposes the system into components and specifies how the components communicate [GRAH96a]. Consider the complete system as composed of one or more compositional wrappers. The application developer starts with a high level design that outlines the compositional wrappers that the developed system will have and the possible interactions among them. This way, the developer starts by formulating decisions on the basic "shape" of the solution. The resulting architecture must be examined with respect to the required functional (i.e., what the system is ex-

pected to do) and non-functional (e.g., fault-tolerance, usability, portability, and performance) attributes. The specific application domain, the user's needs, and what the user is willing to pay usually determine the required attributes.

The developer has then to consider the interfaces among the major architectural pieces (the components) and how these interfaces will interact. Component communication, collaboration, and synchronization are the major issues to be considered at this point. Other issues that need to be considered include replication of services (for load balance or to minimize downtimes), tolerance to certain faults, and unavailability of data. These pieces of information can be captured using the provided specification constructs and the *architecture* section in the wrapper specification language.

Next, the developer has to decide on the specific products, languages, and platforms, etc., that will make the system work. Such choices should be based on the required performance, reliability, ease of use, and the available skills and vendor market credibility. Some of these attributes (such as language and platform characteristics) can be easily captured in the wrapper specification. Other attributes (such as the wrapper's implementation, deployment, and runtime management) are managed by the wrapper generator.

The architectural design process occurs as part of the work to analyze the requirements and determine the design specifications for the system to be developed. The process would need to be done for each identified component in the developed system design. Such design components will translate to compositional wrappers when their specifications are fed to the wrapper generator at later stages.

9.3 Building CWB Systems

After the architectural considerations are worked out, the developer could start with the complete system as one compositional wrapper and then refine its specification as needed. Such recursive decomposition is similar to that of the Kobra [ATKI00] approach, where every component (regardless of its granularity or location in the decomposition hierarchy) is manipulated with the same set of concepts.

Application developers can follow either of two approaches to their system composition process: synchronous and asynchronous composition. Both of these approaches have many advantages and disadvantages. The parties involved (developers, management, and clients) will need to weigh these advantages and disadvantages in the context of the project constraints (size, cost, time, etc.) to decide whether the disadvantages of the chosen approach can be tolerated in favor of its advantages. The following two sections discuss these two approaches in more detail.

9.3.1 Synchronous Composition

In the synchronous composition approach, the developer starts with one of the identified compositional wrappers in the system, supplies its specification to the wrapper generator, waits until the generator delivers the compositional wrapper, plugs it into the

developed system, and continues with the next wrapper. Figure 15 shows a high-level diagram of this approach. Figure 16 shows the wrapper generator and the search agents' process activities. Note that the search agents also work synchronously. The key aspect in this approach is that the developer works with one wrapper at a time.

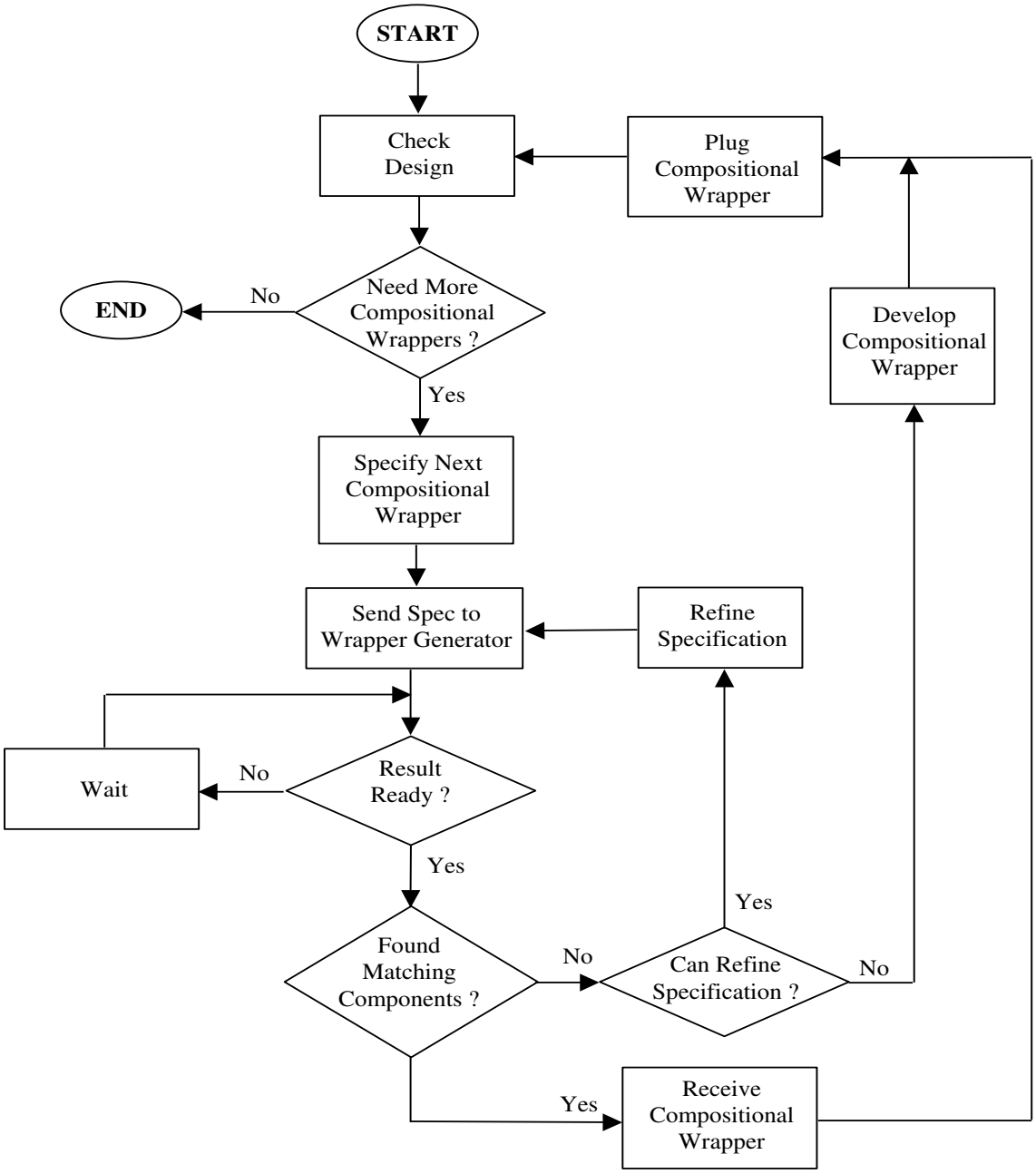

Figure 15: The synchronous interaction between the system developer and the wrapper generator in identifying and specifying compositional wrappers

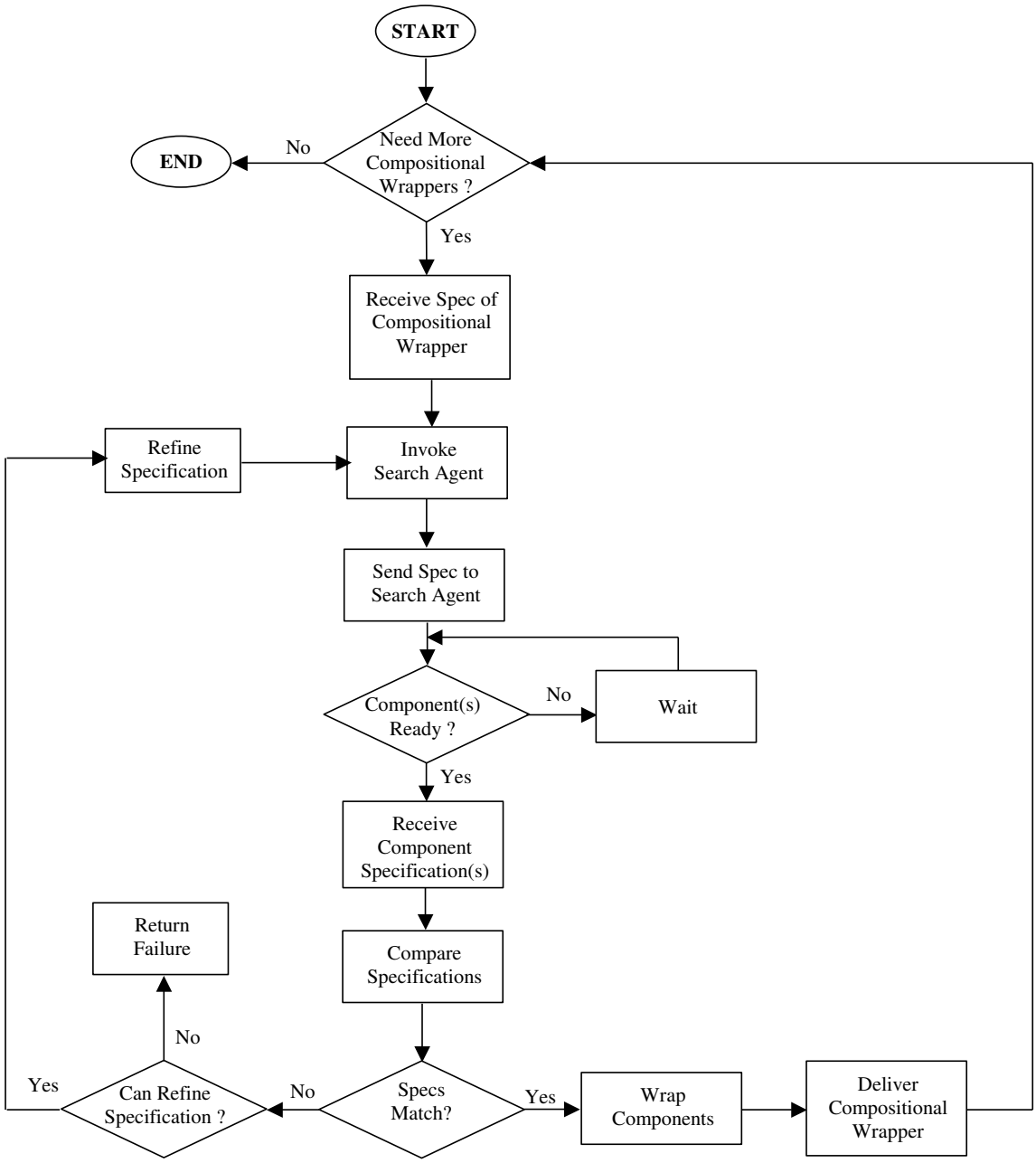

Figure 16: The synchronous interactions between the wrapper generator and search agents in composing compositional wrappers

This approach has several advantages:

- The developer can design the system as it builds up: integrating the needed compositional wrappers and the refinement of the wrapper specifications.
- Since the developer starts with the complete system as one compositional wrapper and refines its specification, the developer can, when necessary, take advantage of bigger and more complex components not known of in advance. This can reduce the overall development time and cost of each compositional wrapper, and the system in general, and will make the final system easier to maintain and extend. In addition, the total risk of system failure will likely be lower since fewer compositional wrappers will be involved.
- This approach is suited for projects with small team size. Team members can work together on the decomposition and refinement of needed compositional wrappers.
- Clients can work hand-in-hand with the developers and will be more pleased to track the progress of the developed system.
- Development chaos is limited since the system is built one compositional wrapper at a time.

The disadvantages of this approach include:

- Difficult to implement for large and complex systems. Such systems will usually follow a more structured design where most of the compositional wrappers will be identified during design time.

- Not suitable for large teams since most of the team members will be idle waiting for the wrapper generator to return with the compositional wrappers.
- Difficult to assess in advance the budget and schedule requirements and its needed resources. This is because it is normally impossible to determine if a particular component can be used to provide a required capability until that component has been evaluated. Lack of such insight would be a problem for both the management and the client.
- Less utilization of the manpower and resources involved with the project since most of the development time will be wasted waiting for wrappers.
- Only one search agent can be utilized at a time because developers can work with only one wrapper at a time.

9.3.2 Asynchronous Composition

The asynchronous approach to sending requests to the wrapper generator, shown in Figure 17, is typically preferred to the synchronous approach. After sending the request to the generator, the developer can continue working with other components. When the compositional wrapper is ready, the developer plugs it in and continues other work. Figure 18 illustrates the wrapper generator and the search agent interactions, restructured to take advantage of the asynchronous approach.

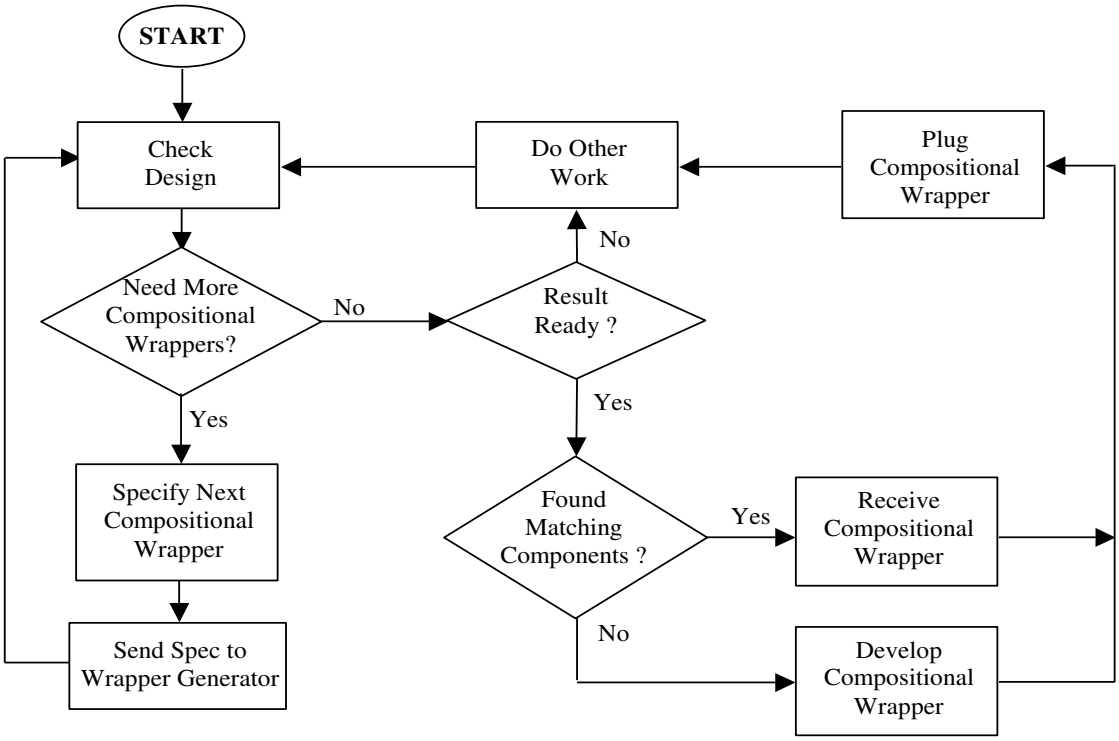

Figure 17: Asynchronous interactions between the system developer and the wrapper generator in identifying and specifying compositional wrappers

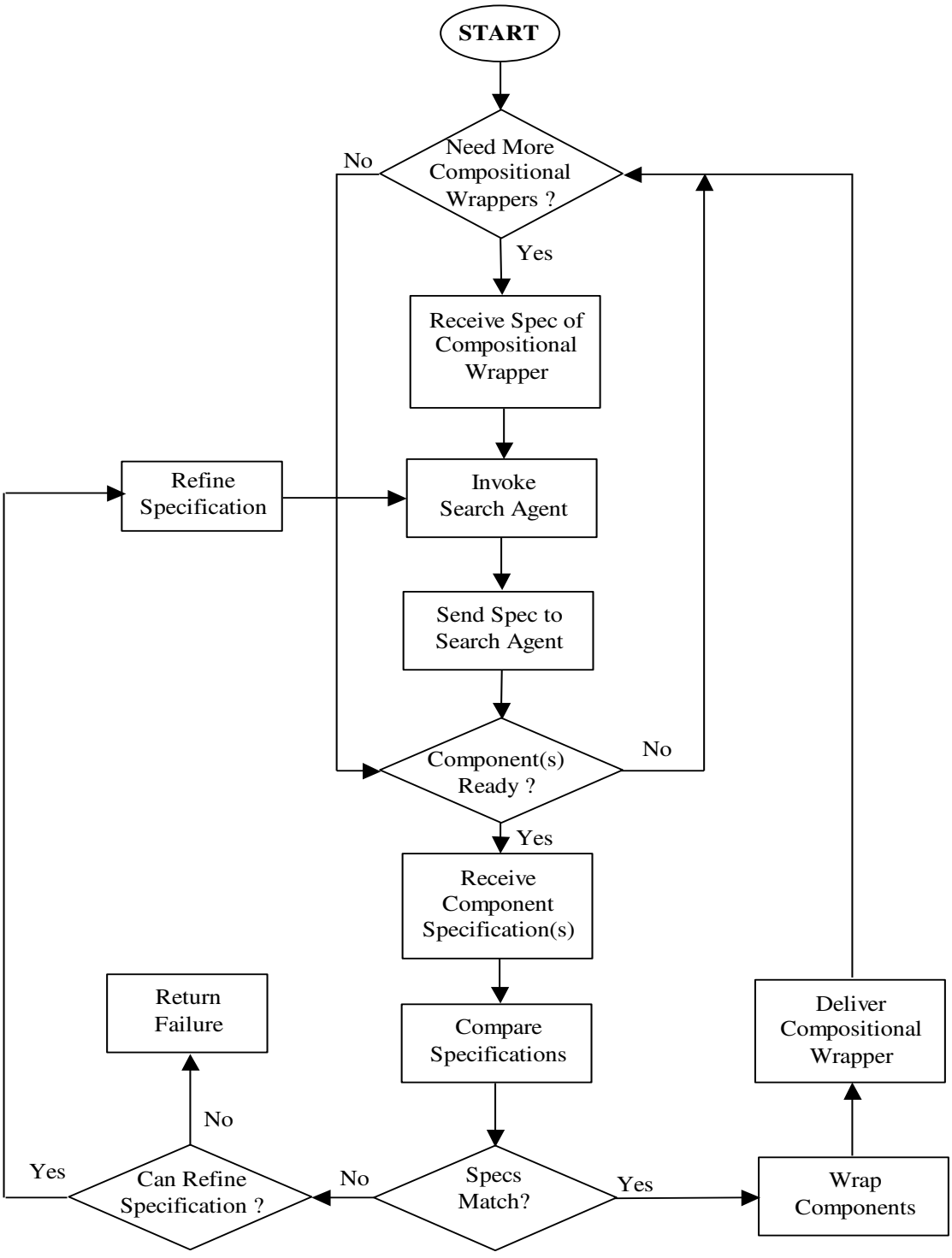

Figure 18: Asynchronous interactions between the wrapper generator and search agents in composing compositional wrappers

The advantages of this approach include:

- Better utilization of teams. Teams and team members can work in parallel to accomplish the tasks assigned to them. Teams of any size and structure will have some work to do. Teams can be organized hierarchically so that each sub team can be assigned one or more areas of the system (such as the identification and specification of compositional wrappers, interaction with the wrapper generator, evaluation and testing of potential components, and the fitting of the generated wrappers in the developed system.)
- Needed compositional wrappers and their interactions can be specified and designed before the composition phase starts.
- Shorter development time since components are dealt with almost in parallel.
- Better utilization of resources. The idle time of each resource will typically be much lower than that of the synchronous approach.
- Better utilization of the wrapper generator and search agents.

The disadvantages are:

- It is likely that component costs will not be proportional to component granularity. For example, the cost of a component that performs n tasks could be much lower than the cost of n small components each of which performs a single task. Therefore, the overall system cost using an asynchronous approach might be higher than

that of the synchronous approach (since the components will often be of finer granularity.)

- The final system might not be operational before most of the wrappers are in place (since wrappers can be plugged-in in an almost random fashion.)
- Higher risk of failure because of the larger number of suppliers to deal with and the logistics involved with each component, which will multiply as the number of components increases.
- Resource conflicts could occur when two or more components require the same resource during the testing and composition phases. Such conflicts can result in longer development time and higher overall costs due to the increase in development time and the higher reservation costs imposed by component suppliers.

9.4 Preparing for The Composition Process

The discussion of this section describes the preparation required before the first use of the wrapper generator. Before beginning to assemble and generate compositional wrappers, the system integrator must initialize the wrapper generator with some information that will be needed throughout the composition process. Note that most of the activities listed below will need to be entered only once in the first use of the wrapper

generator. However, some activities will need to be updated, or at least approved, before each iteration of the process.

- *Collect vendors' information*. At the first use of the wrapper generator, a global search of component vendors is conducted. The search may be repeated from time to time, as wrapper generator users survey the market for new or updated component vendors. Newly collected vendor information should be saved for future reference.

- *Enter preferred vendors*. The wrapper generator should enable its users to provide a list of preferred vendors to be consulted during component searching and acquisition process. The wrapper generator users will initially need to select at least one preferred vendor. The list of vendors can then be updated as the generator users start using the wrapper generator and as new or updated information about vendors become available. The following code snippet outlines some of the utilities needed to manage the master list of vendors that will be maintained by the wrapper generator.

```
GetVendor (ID) {
      p = Vendors;
      while (p <> nil && p.ID <> ID) {
            p = p.next;
      }
      return (p);
}

GetVendorName (ID) {
      p = GetVendor (ID);
      if (p <> nil) {
            return (p.name);
      }
```

```
      else {
            return (FAIL);
      }
}

GetVendorAgent (ID) {
      p = GetVendor (ID);
      if (p <> nil) {
            return (p.agent);
      }
      else {
            return (FAIL);
      }
}
```

The following code snippet implements a simple mechanism for a bi-directional communication with vendor agents.

```
InteractWithVendor (comp, vendor_ID, info) {
  p = GetVendorAgent (vendor_ID);
  if (p <> FAIL) {
    // Are we passing information to the vendor?
    if (info <> "") {
      result = p.Receive (comp.ID, info);
    }
    else {
      result = p.Send (comp.ID);
      // Make sure the returned information contains no
      // malicious or suspected code.
      Scan (result);
      // Make sure the returned information is  legitimate.
      Authenticate (result);
    }

    return (result);
  }
  else {
    return (FAIL);
  }
}
```

- *Build the master list of search agents*. During the first use of the wrapper generator, system integrators will need to survey the market for available search agents

and the pros and cons of each of them. (Integrators could use the Internet or published accounts to identify and select search agents.) For each found search agent, the generator adds the search agent's information to the master list of search agents that the wrapper generator maintains. The list will then be updated whenever new or updated information about search agents are available.

- *Enter a default searching scope*. The wrapper generator should enable its users to provide a default searching scope for their target components. This information can be entered only once by wrapper generator users and updated when necessary. Since the Internet is probably the main source now consulted by software engineers looking for reusable software outside their own development environment, it is a good candidate for the default search scope.

9.5 Specifying Needed Components

This section elaborates on the activities of developing the CWB system (activity 1.0 in Table 3.) As indicated in Table 4, no automation is assumed for the design activities, whether applying a wrapper generator (G) or without using a generator (WG). A high-level discussion of the manual effort to be performed is provided below.

Table 4: Level automation possible during system design

	Automation			
	Manual	**Low**	**Medium**	**High**
0.0 Develop and operate a compositional wrappers-based				

			Automation			
			Manual	**Low**	**Medium**	**High**
(CWB) system						
	1.0 Design the CWB system					
		1.1 Identify components	G, WG			
		1.2 Specify components	G, WG			
		1.3 Review system design	G, WG			
	2.0 Build the CWB system (Chapter 10 to Chapter 14)					
	3.0 Operate the CWB system (Chapter 15)					

This stage starts by identifying the next needed compositional wrapper. The developer would start by treating the whole system as one component. This helps the wrapper generator in building larger wrappers. This means that the initial wrapper specification should be general enough to cover the whole system or large modules of it. The developer can recursively send its wrapper specification to the wrapper generator, evaluate the received components, refine the wrapper specification, if necessary, and send the refined specification back to the generator for a new search. This process continues until the developer is satisfied with the characteristics of the found components.

CHAPTER 10: SEARCHING FOR COMPONENTS

Chapter 10 to Chapter 13 provide elaborations for the activities required to collect components. This chapter focuses on the process of searching for components. Table 5 gives a detailed activity list for the "Search for components," the "Wrap legacy system components," and the "Develop new components" activities. The reason for addressing the wrapping of legacy systems and the development of new components at this stage is to illustrate the logical behavior taken by the system integrator when the search process fails to find off-the-shelf components. Table 5 indicates possibilities for automation (at low, medium or high levels versus manual/no automation.) The footnotes to the table provide the justification for the automation levels provided. (As in Table 4, "G" indicates the automation level when a wrapper generator is used, and "WG" indicates the automation level without a generator.)

Table 5: Detailed breakdown of the activities involved in collecting components

	Automation			
	Manual	**Low**	**Medium**	**High**
0.0 Develop and operate a compositional wrappers-based (CWB) system				
1.0 Design the CWB system (Chapter 9 and 9.5)				
2.0 Build the CWB system (Chapter 10 to Chapter 14)				
2.1 Collect components (Chapter 10 to Chapter 13)				
2.1.1 Procure components (Chapter 10 to Chapter 13)				
2.1.1.1 Search for components				

	Automation			
	Manual	**Low**	**Medium**	**High**
2.1.1.1.1 Form the search query	WG	G[4]		
2.1.1.1.2 Invoke an available search agent				G, WG[5]
2.1.1.1.3 Submit the search query to the search agent				G, WG[5]
2.1.1.1.4 Receive search results from the search agent				G, WG[5]
2.1.1.1.5 Examine search results				
2.1.1.1.5.1 Compare the wrapper specification with the received component specifications	WG			G[6]
2.1.1.1.6 Refine the search criteria	WG		G[7]	
2.1.1.2 Evaluate components (Chapter 11 and Chapter 12)				
2.1.2 Wrap legacy systems				
2.1.2.1 Identify the existing legacy system to wrap	G, WG			
2.1.2.2 Analyze the functionality of the legacy system	G, WG			
2.1.2.3 Specify and design the needed wrapper around the legacy system	WG	G[8]		
2.1.2.4 Develop the wrapper around the legacy system	WG	G[9]		
2.1.2.5 Test the wrapped legacy system	WG	G[10]		
2.1.3 Develop new components				
2.1.3.1 Identify needed functionality in the new component	WG	G[11]		
2.1.3.3 Specify and design the needed component				
2.1.3.3.1 Design the required persistent storage model	G, WG			

[4] The generator can parse the wrapper specification and suggest some keywords to use in the search query.

[5] It is assumed that the user is submitting the search request through any search API, such as the familiar Google or Yahoo search engines.

[6] At this stage, the wrapper generator needs only to inspect the results to make sure that at least one component has been retrieved.

[7] The wrapper generator can do a list of steps to refine the specification as explained in Section 10.2.1.

[8] Though the generator cannot help in preparing the actual specification and design of the needed wrapper, it can list the missing interfaces and other functional and non-functional properties that the legacy system is supposed to provide.

[9] The generator can use the developed specification in developing the wrapper. Additionally, the generator can inject extra functionality, such as some of the smart features, to make it easier for the generator to handle the wrapped legacy system.

[10] The generator can help by preparing some test cases and by executing them. The user is responsible for capturing and analyzing the test results.

[11] The generator can list the missing interfaces and other functional and non-functional properties that the developed component is supposed to provide.

	Automation			
	Manual	**Low**	**Medium**	**High**
2.1.3.3.2 Design the user interface	G, WG			
2.1.3.4 Develop the needed component				
2.1.3.4.1 Implement the business logic necessary to support the behavior defined in the specification of the new component	WG	G[11]		
2.1.3.4.2 Implement the required persistent storage model	G, WG			
2.1.3.4.3 Implement the user interface	G, WG			
2.1.3.5 Test the new component	WG	G[10]		
2.2 Wrap components (Chapter 14)				
2.3 Deliver compositional wrapper (Section 14.11)				
2.4 Assemble components into the CWB system (Section 15.1)				
2.5 Test the CWB system (Section 15.2)				
3.0 Operate the CWB system (Chapter 15)				

10.1 Interpreting the User Requirements

The wrapper generator should provide a mechanism, such as an online form, to allow integrators to enter the specification of the required compositional wrapper. The entered information should conform to the wrapper specification language. The wrapper generator should also allow integrators to supply the wrapper specification from a file. If the wrapper generator is to read the wrapper specification from a file, then the generator would need to make sure that each construct in the wrapper specification conforms to the wrapper specification language and appears as many times as allowed by the wrapper specification language.

If the wrapper specification were entered interactively, no parsing of the specification would be needed. Otherwise, the algorithms of Section 12.5 could be used to parse the specification file and to identify required interfaces as well as other entities.

The generator's built-in structure (discussed in Section 4.1) would be used to hold such information until the composition process is complete. The generator will use the information stored in that structure to build the search query and to do any refinement necessary.

10.2 Searching For Components

There are at least three ways to search for components. The first is to pass the whole wrapper specification to a search agent and to let the agent do the matching between the wrapper specification and component specifications. However, this approach seems inefficient since each component specification will be parsed and analyzed twice: first by the search agent and second by the wrapper generator when it tries to find candidate components. The second approach is to let the wrapper generator send only search criteria (mentioned later) to search agents. The search agents will try to find components whose specifications have close match with the search criteria. The search agents will send back to the wrapper generator the specifications of candidate components. The third approach requires the wrapper generator to have access to a variety of component repositories. (Examples of existing component repositories or search engines include MS Repository [BROW00], Code Librarian in MS Office, MS Visual Component Manager [MSDN02], INSEAS [PARK99], Agora [SEAC98a, SEAC98b], and ICE [ERIK00]. Component Service Providers (CSP) [BUNT00] could also be a good source for components.) With this third approach, the wrapper generator will deal with all components as

if the components were local. The wrapper generator will be responsible for doing the matching between the wrapper specification and the component specifications. Automation by the wrapper generator of any of these approaches is possible using currently available search agent APIs from component providers or APIs implemented in the wrapper generator.

10.2.1 Formulating the Search Request

There is a plethora of information that can be passed to search agents about the compositional wrapper of interest. The generator is supposed to build the search query from the wrapper specification using the generator's internal structure discussed in Section 4.1. However, since the information in a wrapper specification would usually be not suitable for use as a search query because of its length and format, the generator might find it difficult to identify the keywords to use in the search. The generator can extract from the wrapper specification an initial list of keywords and the system integrator would need to approve or modify such list. Depending on the interfaces provided in the available search agents, the entered search criteria could also be complemented by a description of the interactions allowed among the components and additional constraints on both individual components and the components as a group.

Several challenges will have to be resolved before one can fully automate the interactions with search agents. For example, some of the search information that system integrators supply will be incomplete. Such incomplete information could result in either

no components or many components matching the few pieces of supplied information. When the search agent finds no matching component, it is, in most cases, because of the stringent constraints that the system integrator supplied. In other times, supplying few, very general information could result in many components. In all these situations, the system integrator will need to revise the compositional wrapper specification and repeat the search request process. In addition, the wrapper generator could attempt to refine the specification to make it less general (or more specific.) Section 10.4.1 gives details on how the wrapper generator can refine the wrapper specification.

Another challenge is the lack of standardization of information encoding and enumeration when dealing with commercially available search agents. This could force the wrapper generator to implement more than one encoding scheme of the passed specification to allow the agents to interpret the specification correctly.

The following code snippet will collect the information needed to form the search request. The approach shown in this code is very simple. The *FormSearchRequest* utility first enumerates most of the constructs in the wrapper specification and then passes such information back to the caller (i.e., the wrapper generator.)

```
FormSearchRequest (spec) {
  request = nil;
  new (request);
  request.provided_interfaces = spec.provided_interfaces;
  // <name> is used to refer to the cost name, for example,
  // Single.LicenseCost. This is to differentiate it from the
  // wrapper's or component's name.  For example, wrapper.name
  // refers to the wrapper's name while wrapper.<name> refers to
  // the wrapper's specific cost name.

  for prefix in ('single', 'developer', 'site')
  {
```

```
    for cost_name in ('License', 'Upgrade', 'Maintenance', 'Support',
           'Training', 'Installation')
    {
      // Form the cost name
      name = prefix + "." + cost_name + "Cost"

      request.<name> = spec.<name>
    }
  }

  request.suppliers = spec.preferred_suppliers
  request.technologies = spec. preferred_technologies
  request.operating_systems = spec.preferred_operating_systems
  request.languages = spec.preferred_languages
  request.formats = spec.preferred_formats
  request.software_constraints = spec.software_constraints
  request.RAM = spec. hardware_constraints.max_RAM
  request.storage = spec. hardware_constraints.max_storage
  request.network_constraints = spec.network_constraints
  request.supplier_types = spec.supplier_types

  return (request);
}
```

10.2.2 Specifying the Search Scope

This step is needed to tell the generator where to look for components. The generator should supply a place in its search request form to allow the system integrator to specify the search scope. Searching scopes could include the local computer that is currently running the wrapper generator, the integrator's Intranet, or the global Internet. The system integrator has already given the generator a default search scope to use when the integrator first started to use the generator (as explained in Section 9.4.)

10.2.3 How to Interact With Search Agents?

The generator should have the capability to utilize commercially available or newly developed search agents. Further, the system integrator must be able to manage the list of agents used. Management tasks could include selection of one or more search agents to be utilized in the search process, launching them in their own processes, interacting with them, monitoring their progress, and terminating them and their processes. The generator should have some means of protecting itself against intrusions by search agents or vendors' systems.

The selection of any of the models presented below depends on the amount of delegation that system integrators want to give to search agents. The client in these models will usually be the wrapper generator itself or separate threads dedicated to interact with individual search agents. Component sources are component repositories that are within the reach of search agents. Component sources can also refer to software agents residing in the servers of component suppliers. The symbols shown for each model are as follows:

C_1	client # 1
A_1, A_2	agent #1, agent # 2
A_i, A_j	one or more agents
S_j	one or more component sources (suppliers)
R_i, R_j	one or more search requests

Each of the following models is encoded as:

list of clients : request combinations <= list of agents => list of component sources : request combinations

- (C_1:R_i <= A_1 => S_j:R_i)

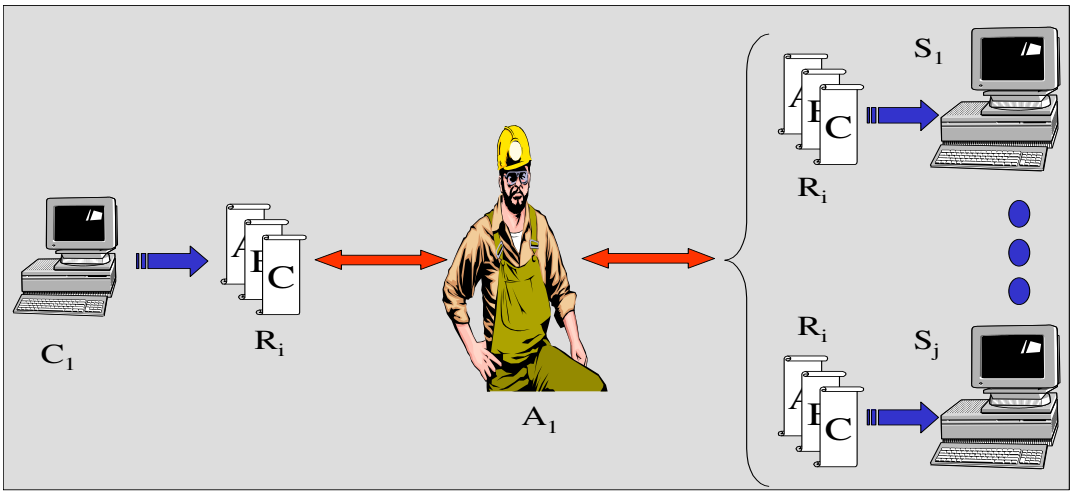

Figure 19: Interactions with search agents (Model 1)

In this model, one agent will be responsible for all interactions with the client and all component sources. This means that the following are true for this model:

- One client (C_1) will issue one or more requests (R_i).
- One agent (A_1) will service all search requests from the client and will handle all interactions with all component sources.
- One or more component sources (S_j), each of which can receive and service one or more search requests (R_i) from the search agent.

- (C_1:R_i <= A_1, A_2 => S_j:R_i)

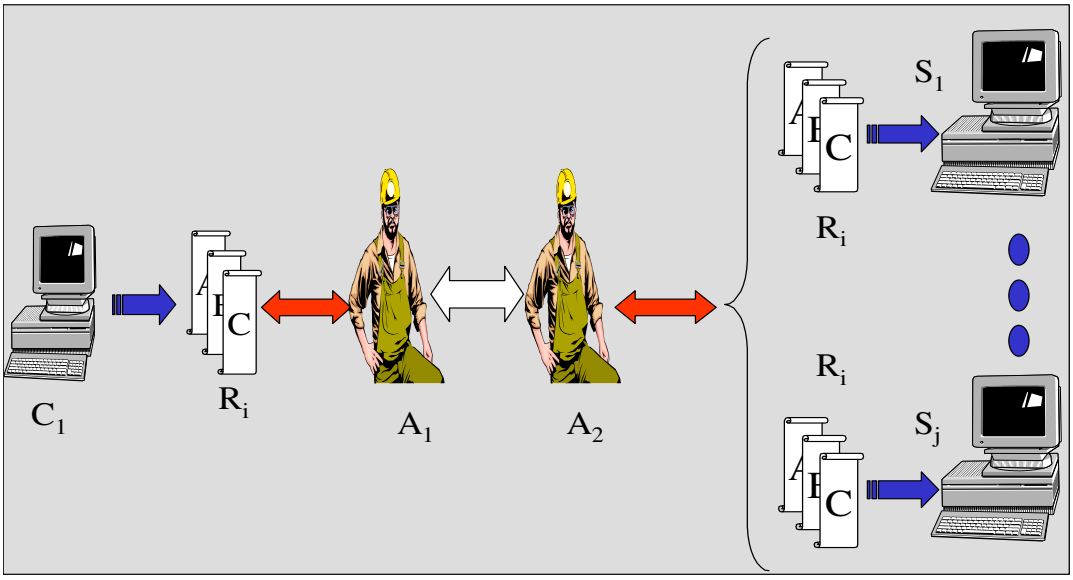

Figure 20: Interactions with search agents (Model 2)

In this model, one search agent is responsible for all the interactions with the client and another agent for all the interactions with all component sources. This model requires some mechanism for the two agents to pass information back and forth between each other. However, this is outside the scope of the work of this dissertation.

- (C_1:R_i <= A_1, A_j => S_j:R_i)

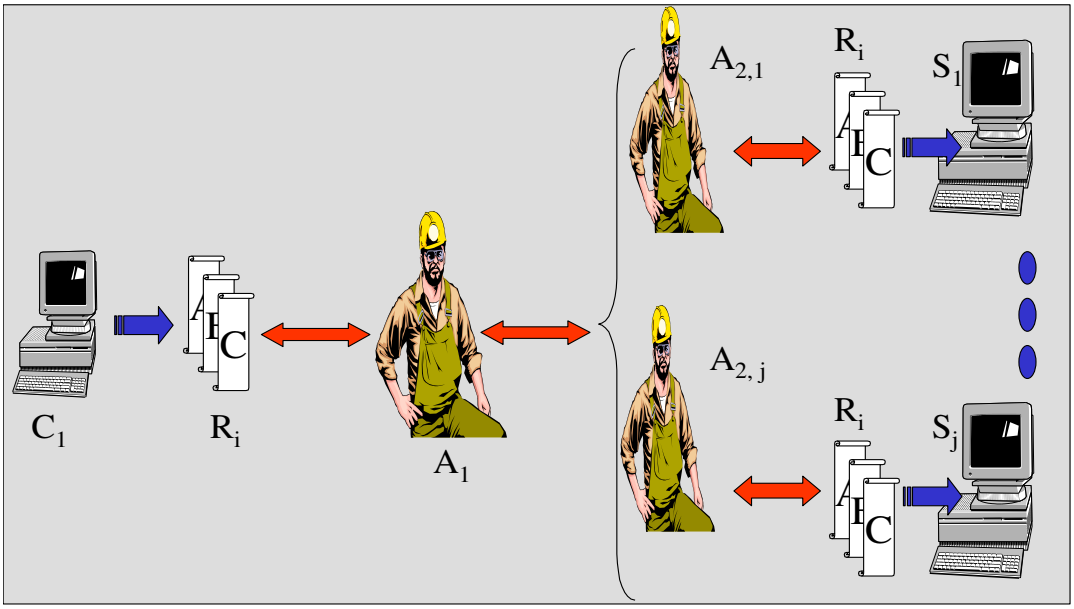

Figure 21: Interactions with search agents (Model 3)

In this model, one search agent will be responsible for all the interactions with the client. Additionally, for each component source, a separate search agent is responsible for all the interactions with that component source and for servicing any of the R_i requests. A_1 will also communicate with all the $A_{2,j}$ agents.

- (C_1:R_i <= A_i => S_j:R_i)

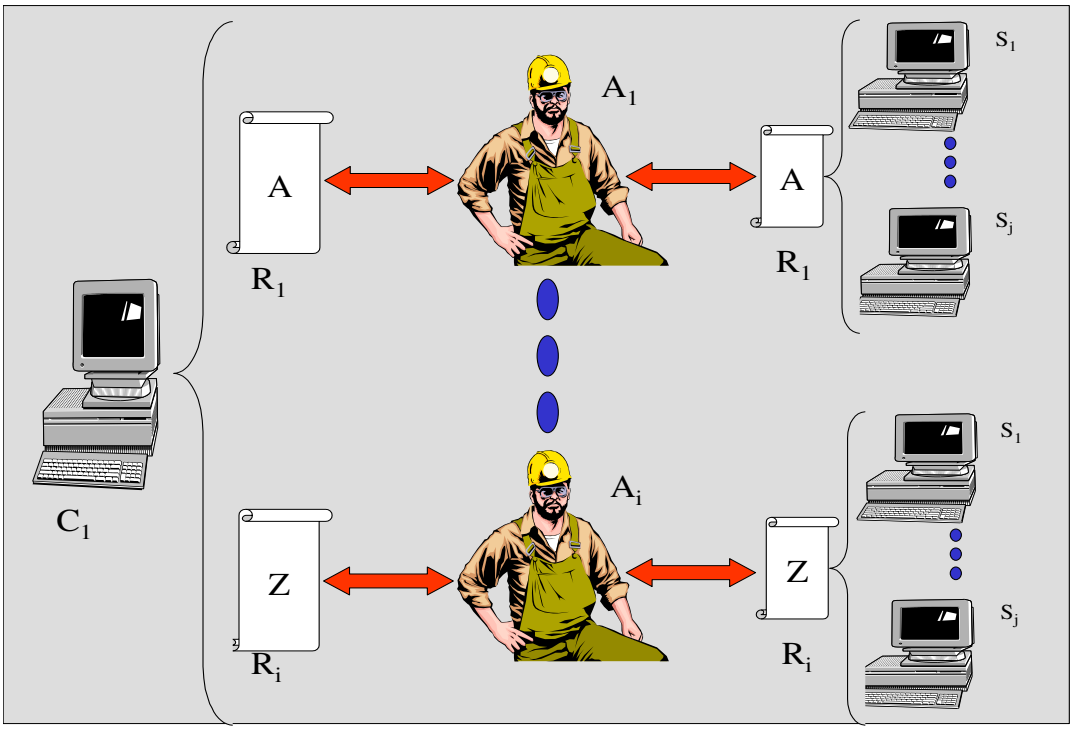

Figure 22: Interactions with search agents (Model 4)

This model employs one search agent for each client request. Each of these A_i agents is responsible for all interactions with the client and with all the component sources servicing the request that the agent is assigned with. No interaction among the various search agents is needed.

- (C_1:R_i <= $A_{1,i}$, $A_{2,i}$ => S_j:R_i)

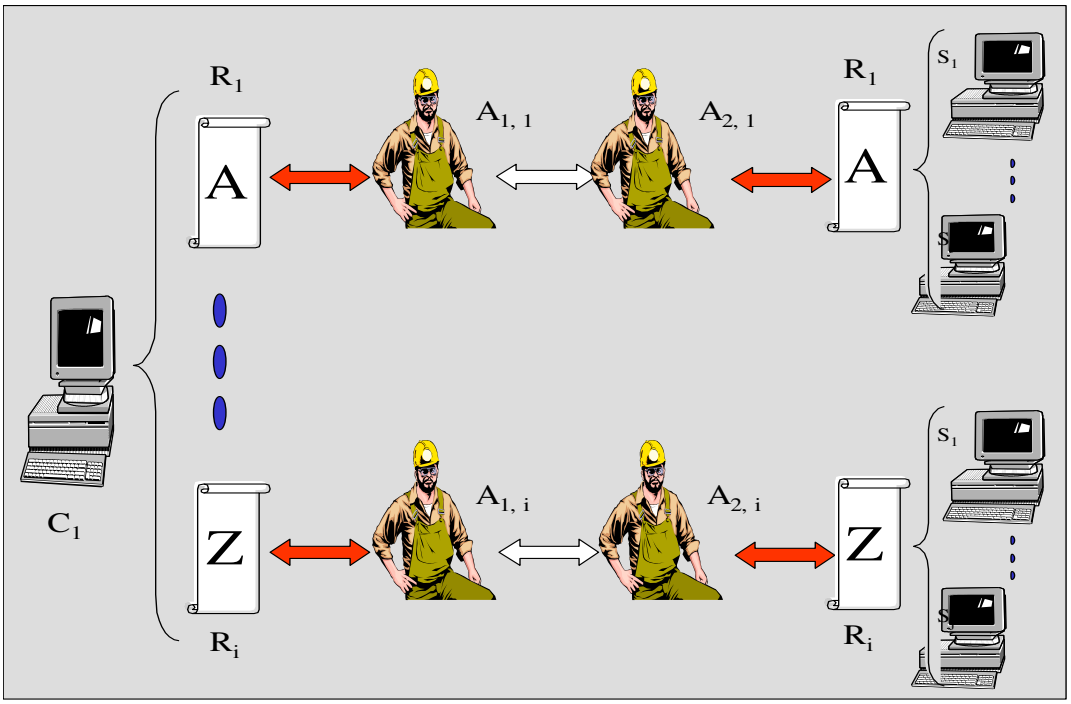

Figure 23: Interactions with search agents (Model 5)

For each client request, two search agents are initiated. One agent will be responsible for all the interactions with the client and the second agent with all the interactions with all component sources.

- (C_1:R_i <= A_i, $A_{i,j}$ => S_j:R_i)

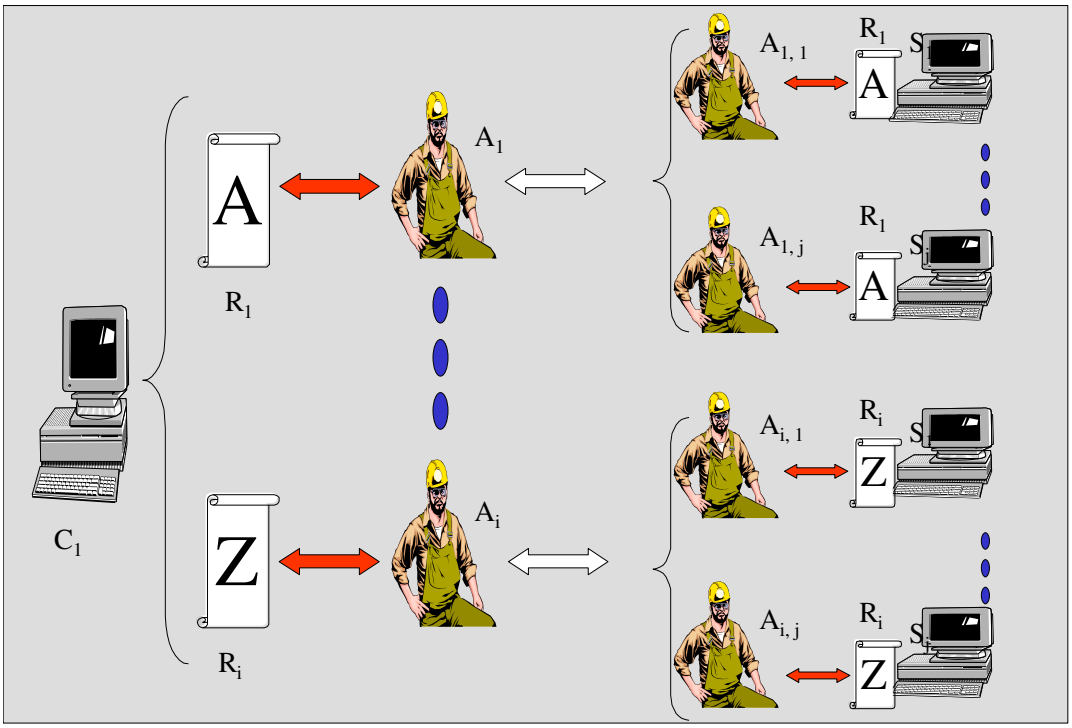

Figure 24: Interactions with search agents (Model 6)

In this last model, for each client request, one agent will be initiated to service all the interactions with the client. In addition, for each of the R_i client requests, S_j additional search agents will be initiated to interact with each of the available S_j component sources that can service one of the R_i client requests. This means that this model employs (j+1) agents to service each of the R_i clients requests.

The wrapper generator can initiate a separate thread to interact with each client agent. Once the client agent receives search results from and finalizes interactions with the search agent(s), the thread passes the results back to the wrapper generator and then it

terminates. The wrapper generator can also request from all or from specific threads to abort the search process and terminate. In this case, the thread requests all search agents that it is interacting with to terminate. After that, the thread would terminate without submitting collected results.

The results that each search agent is expected to handle to the thread it is interacting with should include the following:

- The list of servers (component suppliers) visited and the results so far collected from each server.
- The list of components collected so far. For each component, the agent should present at least the following set of information:
    - The component specification.
    - Information about the component supplier, including the supplier's name, the supplier's type (original vendor, reseller, government, public library, etc.), the supplier's URL, and the supplier's IP.
    - Reservation information, if one is made by the agent (see Section 11.2.)

The following utilities demonstrate the implementation of some of the wrapper generator's management and interaction activities that are related to search agents. (These utilities were discussed at the beginning of this section.) In these utilities, *Agents* holds a copy of the master list of search agents that the wrapper generator maintains. This *Agents* list is used only for the current search request. It is the generator's responsi-

bility to terminate all search agents in the *Agents* list upon the completion of the current search process. *delay* and *dispose* are built-in utilities in the target implementation language. *Alive* is a public interface of the search agent component.

```
ProbeSearchAgents () {

// This utility will probe each search agent at certain intervals
// and will terminate those that are not responding.

      p = Agents;
      while (p <> nil) {
            result = p.Alive ();
            delay (1000);     // Wait for one second
            if (result <> ALIVE) {
                  TerminateSearchAgent (p);
            }
            p = p.next;
      }
}

TerminateSearchAgent (p) {
      q = Agents;
      prev = nil;
      while (q <> nil && q <> p) {
            prev = q;
            q = q.next;
      }

      if (p == q) {     // found the agent
            if (prev == nil) {
                  // Terminated agent is in front of the list
                  Agents = Agents.next;
            }
            else if (q.next == nil) {
                  // Terminated agent is at the end of the list
                  prev.next = nil;
            }

            dispose (q);
            return (SUCCESS);
      }
      else {
            return (FAIL);
      }
}
```

The following utility returns all currently idle search agents.

```
SelectSearchAgents () {
      p = Agents;
      agents = nil;
      while (p <> nil) {
            if (p.Idle ()) {
                  new (q);
                  q = p;
                  q.next = agents;
                  agents = q;
            }
            p = p.next;
      }
      return (agents);
}
```

The following utility receives a wrapper specification, uses it to form a search criteria, passes the search criteria to all currently idle search agents, tracks the search progress of the search agents involved in this search activity, receives results from them, and passes the received results back to the wrapper generator. The *FormSearchRequest* utility was presented in 10.2.1. *Search* is a public interface of the search agent component.

```
SendSearchRequest (spec) {
      request = FormSearchRequest (spec);
      agents = SelectSearchAgents ();
      p = agents;
      while (p <> nil) {
            p.Search (request);
            p = p.next;
      }
      result = TrackSearchProcess ();
      return (result);
}
```

The following code implements a simple mechanism for tracking the progress of search agents. It probes each agent at certain intervals. If no response is received within

one second of sending the probing message, then it removes that agent from the list of search agents working on the current search request. If the search activity takes more than a specified time, then the generator terminates all remaining search agents. During the tracking process, the generator keeps the system integrator informed by updating the search progress on the integrator's screen. The *ResponseAvailable*, *GetResponse*, and *GetComponent* are public interfaces of the search agent component.

```
TrackSearchProcess () {
      comps = nil;
      T = MAX_WAITING_TIME;
      while (T > 0) {
            p = Agents;
            while (p <> nil) {
                  if (p.ResponseAvailable()) {
                        r = p.GetResponse();
                        if (r == SUCCESS) {
                              c = p.GetComponent();
                              c.next = comps;
                              comps = c;
                              UpdateDisplay ();
                        }
                  }
                  p = p.next;
            }
            delay (1000);           // wait one second
            T--;
      }
      CancelSearch (nil);     // Terminate remaining search agents
      return (comps);
}
```

The following utilities allow the system integrator to track the search progress and control the searching process. The integrator can stop, resume, or cancel the process at any time. The *Idle*, *Stop*, *Resume*, and *Cancel* are public interfaces of the search agent component.

```
CancelSearch (p, search_ID) {

// This utility removes one or all search agents from the current
// search task.  Search ID holds an ID for the current search
// activity

      if (p == nil) {   // cancel search for all agents
            p = Agents;
      }

      while (p <> nil) {
            if (not p.Idle () && p.search_ID == search_ID) {
                  // Request from the search agent to cancel the
                  // current search task
                  p.Cancel ();
                  TerminateSearchAgent (p);
            }
            p = p.next;
      }
}

StopSearch (p, search_ID) {

// Temporarily stops the search process for one or all agents.
// The search will resume when the ResumeSearch() utility is called.

      if (p == nil) {   // Stop search for all agents
            p = Agents;
      }

      while (p <> nil) {
            if (not p.Idle () && p.search_ID == search_ID) {
                  // Request from the search agent to pause the
                  // search activity
                  p.Stop ();
            }
            p = p.next;
      }
}

ResumeSearch (p, search_ID) {

// Resumes the search activity for one or all agents that were
// stopped using the StopSearch () utility.

      if (p == nil) {   // Resume search for all agents
            p = Agents;
      }

      while (p <> nil) {
            if (not p.Idle () && p.search_ID == search_ID) {
                  p.Resume ();
            }
```

```
            p = p.next;
      }
}
```

The following utility sends information to specific search agents or receives information from them depending on whether or not any information is passed to this utility. The *Send* and *Receive* are public interfaces of the search agent component. The *Scan* and *Authenticate* are wrapper generator utilities that are not discussed in this dissertation.

```
InteractWithAgent (p, info) {
      if (info <> "") {
            // Are we passing information to the search agent?
            result = p.Receive (comp.ID, info);
      }
      else {
            result = p.Send (comp.ID);
            // Make sure the returned information contains no
            // malicious or suspected code.
            Scan (result);

            // Make sure the returned information is  legitimate
            // (i.e., not modified in its way to the wrapper
            // generator.)
            Authenticate (result);
      }

      return (result);
}
```

10.2.4 How Does the Wrapper Generator Interact With Middleware Agents?

Middleware agents refer to those software agents that are provided and managed by other vendors or organizations and for which the wrapper generator and the system integrator have no direct responsibility or control over their existence, function, or management. The wrapper generator can employ several approaches in dealing with such

agents. Depending on the communication means used to interact with each middleware agent, the wrapper generator has to switch to the appropriate method accordingly.

Assume that the wrapper generator will employ its own set of agents (call them client agents) to interact with the middleware agents. The following interaction models can be used for many of the wrapper generator tasks, such as searching for components, negotiating prices with component vendors, sending payments to component suppliers, or receiving license keys from them.

- Method1: Direct communication between the client agents and the middleware agents.
    - The client agent sends its request to one of the available middleware agents.
    - The middleware agent broadcasts the received request to all server agents.
    - All participating server agents receive the message, decode it, and, optionally, respond to it.
    - The middleware agent can collect responses from the server agents and send them back to the client agent as a group or it can pass the responses back individually as it receives them from the server agents.
    - When the client agent receives a reply, it decodes the reply, checks if the reply satisfies its request, and possibly interacts directly with the server agent that sent the reply.

- If the received response seems to satisfy the client agent's needs, the client agent can optionally request from the middleware agent to inform other server agents that the current request is fulfilled and that they can withdraw such request from their consideration.
- The client agent passes back the received information to the wrapper generator and, then, terminates.

- Method 2: The mechanism of this method is similar to that of Method 1, except that there is a central location (a central server or a message board) that the middleware agents and the server agents use to interact with each other. (This approach is similar to the concept of blackboards.)
  - The client agent sends its request to an available middleware agent.
  - The middleware agent posts the request to a message board.
  - The server agents read the request and, optionally, post their reply to the message board.
  - The middleware agent monitors the message board for replies. If it finds one, it passes it back to the client agent.
  - The client agent checks the reply and, if the reply is appropriate to its need, it could start direct negotiations with the server agent that posted the reply.

- If the response is satisfactory, then the agent can inform the middleware agent to end the current request.
- The middleware agent informs the server agents that the request is fulfilled. It can do so by either removing the original message from the message board, by posting a new message indicating the completion of its previous task, by modifying the original message, or by combinations of these three approaches.

- Method 3: This approach employs multiple client agents to work in parallel to fulfill requested tasks. The process of locating components is used here to illustrate the steps involved in this method. As with Method 2, this method employs a central server for its interactions.
  - Each of the client agents (call them *stationary agents*) will employ their own *mobile agents*.
  - The stationary agent consults a central server for a list of component suppliers.
  - Then, the stationary agent invokes one or more mobile agents. Each of these mobile agents will be required to consult one or more component suppliers and to return results to the stationary agent.
  - Each mobile agent can perform its task synchronously or asynchronously. In the synchronous situation, the mobile agent moves to the supplier's server, interacts with it, and does not leave that server to another one until

it receives a response from the stationary agent that the result is either accepted or rejected. On the other hand, the asynchronous mode allows the mobile agent to perform only initial contact with the current supplier in its list before moving to another supplier or returning results to the stationary agent. Before leaving the current server, the mobile agent dispatches a small agent that is required to communicate with the host server and the stationary agent.

  - The mobile agent terminates once it visits all the servers in its assigned list and once it submits the results to the stationary agent.
  - The dispatched small agents terminate once they receive a response from the server that they are assigned to communicate with and once they return the results to the stationary agent. The dispatched small agents would also terminate when requested by the stationary agent to abort their task.

- Method 4: This model is based on processes that communicate using message queues (also called mailboxes) [TROY97]. In this model, communication between the client agents and server agents is asynchronous, and achieved by sending messages to mailboxes. Each server agent belongs to its *domai*n: the address of the machine (or network of machines) where it lives. The convention used in forming email addresses is used here to refer to server mailboxes. For example, if a server agent resides at domain *ucf.edu*, then that server can be referred to by *@ucf.edu*. This allows the easy integration of this model with the WWW and the usage of

some of its services, like name servers. Each mailbox will have a unique global address (*mb@domain*), given by its name *mb* and its domain name *@domain*. If the domain is omitted, then the local domain name is used. Omitting the mailbox name means referring to *all* the mailboxes currently available at that domain. This mechanism allows sending broadcast messages to whole domains. There are two basic communication primitives: *Send* and *Receive*. The first one sends a message to a mailbox address, and the second one reads a message from a mailbox. *Receive* is a blocking operation in case the mailbox is empty, while *Send* is non-blocking. The searching begins with one of the client agents sending a message to all the mailboxes of each supplier using *@supplier-domain*. The client agent will, periodically, use the *Receive* method to receive any acknowledged requests. The agent can interact with each mailbox that it receives acknowledgement from using the *mb@domain* construct.

10.3 Examining Search Results

The generator should provide its users with a mechanism to review the search results. During this stage, the interest is in whether the search criteria generated any results. Other minor analyses could be conducted by the generator to make sure the results refer to actual software components and not to other related entities, such as documentation or web pages. The results displayed to the system integrator should be accompanied by any analyses or conclusions that the wrapper generator has reached during the inter-

pretation of the results process. The system integrator can exclude any results from the list.

If, after filtering out other information, the search still resulted in a number of components, then the wrapper generator can move to the preliminary evaluation stage (discussed in the next chapter.) If, on the other hand, the search resulted in no candidate components, then the wrapper generator would give the system integrator the choice of either to keep looking for ready-made components or to consider other options. If system integrators opt to continue the search for ready-made components, then the search criteria must be modified (with possible refinement of the wrapper specification) and the search request is resubmitted. Otherwise, other means (discussed in the next section) of getting the target component can be considered (for example, developing needed components from scratch or wrapping legacy systems or existing components.)

10.4 How Does the Wrapper Generator React to Unsatisfied Search Requests?

After the wrapper generator passes its request to the search agent, it waits for pre-specified ***T*** time units. If no response is received within that period, the wrapper generator will try initially to perform some simple refinements to the wrapper specification, as will be explained in Section 10.4.1. The wrapper generator then passes the new specification to an available search agent to start a new search for components. The wrapper generator repeats the refinement process until the specification cannot go any further refinement. At this point, the generator ends the refinement process. The wrapper genera-

tor then informs the integrator of its failure to find a component or a set of components that would satisfy the system integrator's request. In this case, the system integrator can further refine the wrapper specification, wrap an existing system, or develop a new component that can realize the full specification or just the missing interfaces. The following subsections will discuss specification refinement, the wrapping of legacy systems, and the development of new components.

10.4.1 Specification Refinement

Specification refinement using wrapper generators does not always translate to removing or loosening constraints. In some cases, it only translates to redesigning the specification of the required interfaces. Several research areas must be matured before one can build a wrapper generator that can fully and accurately automate the specification refinement process. These areas include: accepted standards for specification refinements, identification and classification of all elements needed in a refinement process, what can and what cannot be refined, and effects of such refinements on various other issues, such as performance, quality, and meeting user goals.

Using the wrapper specification language, the wrapper generator can take one of the following two approaches in refining the wrapper specification:

First, the wrapper generator can "soften" some of the non-behavioral constructs. For example, for constructs that use numerical values to mean minimums or maximums, such as the various cost constructs and the hardware constraints' constructs, the wrapper

generator can either increases minimums or decreases maximums. To enable full automation of this process, these constructs would need to be specified as ranges of numbers and not as fixed numbers. With the existence of such construct ranges, the wrapper generator can "play" with the numbers as long as it binds itself to that specified range. For the other non-behavioral constructs that use lists to encode their options and with the ability of ordering such options according to their importance to the development organization, the wrapper generator can shrink the set of options by removing the least important options first. Doing so will eventually result in better chances of finding candidate components. This can be done repeatedly until the generator ends up with suitable number of components.

The generator can also work on the behavioral constructs. The best opportunity is in interfaces that require more than one output. Splitting such interfaces into two or more interfaces for which each of them have a subset of the original set of outputs would result in better chances of finding candidate components. The next refinement could be in input constructs. Using the user-specified tolerance criteria to required inputs (see tion 6.1.3.1), the generator can play with such inputs by removing them in order of their importance. Finally, if rearranging inputs and outputs did not work, then the generator can work on rearranging assertions in behavior clauses could give better chances of finding matched components.

10.4.2 Wrapping Legacy Systems

In some cases, especially if the allocated budget is tight, existing systems or third-party components owned by the user can provide much of the functionality defined in a wrapper specification. However, much of the available functionality in existing systems or components might not be in the correct form to include in a compositional wrapper (for example, the existing system could be executing in a different platform, written in another language, or packaged in an incompatible form.) In this case, such system or component must be first wrapped to make it usable for inclusion in the developed system. The role of this wrapper will be to hide such mismatches between existing and required functionality. Additionally, the existing legacy system or pre-built component might not have the exact functionality as that specified in the wrapper specification. In this case, the developer can either change the wrapper specification to match the characteristics of the existing system, or create a mapping between the existing and required functionality. The wrapper generator could have the capability to develop the needed wrapper. In this case, the system integrator's responsibility will only be to provide the generator with the new specification or the mapping between the existing system or component and the required functionality.

10.4.3 Developing New Components

If the user has the expertise to develop new components and is not on a tight schedule, then the next alternative is to implement the complete specification or just the missing interfaces. The user must understand that in a component-based approach, the

process of developing new components requires the translation of a component specification into a conforming component implementation. In preparation, the component developer must first transform the specification from its abstract view to a component that can plug into the developed system without inducing new problems or exacerbating existing ones. In this journey, the developer usually passes through various stages, including:

- *Designing the implementation of the component.* Usually, there will be significant effort required by the developer to refine and articulate the information included in the wrapper specification. Such efforts could include implementing new interfaces that cannot be found in ready-made components.

- *Implementing the business logic necessary to support the behavior defined in the specification.* The specification will usually contain semantic details on interface names, their signatures, and the changes of state that each must support. The developer must implement such logic according to the chosen technology and in conformance to standards or business rules required by the organization.

- *Designing the persistent storage model.* Though the specification might not contain details on the persistent storage required for the component, the developer has to design such mechanism in order for the component to operate independently in the environment in which it will be put.

- *Creating the user interface for the component.* In cases where the component requires a user interface to enable its users to interact with its services, the developer will be responsible for designing such interface. Other user interfaces, such as the

ones to help the development team in testing and diagnosing the component, might be necessary to implement. Design tools are necessary in this stage to help the developer enforce organization standards and employ good user interface practices.

CHAPTER 11: PRELIMINARY EVALUATION AND CANDIDATES SELECTION

Table 6 gives a detailed activity list for the first activity of "2.1.1.2 Evaluate components." In addition, the table indicates possibilities for automation (at low, medium, high level versus manual/no automation.) The footnotes to the table provide the justification for the automation provided (as in previous chapters, "G" indicates the automation level when a wrapper generator is used. "WG" indicates the automation level without a generator.)

Table 6: Activities breakdown of the preliminary evaluation and candidate selection stage

	Automation			
	Manual	**Low**	**Medium**	**High**
0.0 Develop and operate a compositional wrappers-based (CWB) system				
1.0 Design the CWB system (Chapter 9 and 9.5)				
2.0 Build the CWB system (Chapter 10 to Chapter 14)				
2.1 Collect components (Chapter 10 to Chapter 13)				
2.1.1 Procure components (Chapter 10 to Chapter 13)				
2.1.1.1 Search for components (Chapter 10)				
2.1.1.2 Evaluate components (Chapter 11 and Chapter 12)				
2.1.1.2.1 Conduct preliminary evaluation				

							Automation			
							Manual	**Low**	**Medium**	**High**
						2.1.1.2.1.1 Evaluate the component specification	WG	G[12]		
						2.1.1.2.1.2 Check the type of available documentation	WG	G[13]		
						2.1.1.2.1.3 Check supported technologies	WG			G[14]
						2.1.1.2.1.4 Check the underlying architecture	WG, G[15]			
						2.1.1.2.1.5 Check the number and type of provided interfaces	WG		G	
						2.1.1.2.1.6 Check the number and type of required interfaces	WG	G[16]		
						2.1.1.2.1.7 Check the installation and customization options	WG	G[17]		
						2.1.1.2.1.8 Check the price and licensing options	WG	G[18]		
					2.1.1.2.2 Select candidate components		WG	G[19]		
					2.1.1.2.3 Reserve candidate components					
						2.1.1.2.3.1 Consult the component vendor	WG			G[20]
						2.1.1.2.3.2 Submit personal information		WG[21]		G[20]
						2.1.1.2.3.3 Submit payment information		WG[21]		G[20]
						2.1.1.2.3.4 Obtain reservation information		WG[21]		G[20]

[12] The generator would need to make sure that the specification is written according to the component specification language, that all required constructs exist, and that no major violations to the user's requirements exist in the specification.

[13] The only automation possible is to use the *downloads* clause in the component specification to list all the supplied documentation.

[14] Can be done using the *technologies* clause in the component specification.

[15] Currently, the component specification language contains no construct to list the underlying architecture.

[16] The generator can collect such information from the *requires* clause.

[17] Though the component specification language provides constructs to list installation and customization requirements, the wrapper generator cannot make judgment on their ease compared to that of other components. However, the generator can check the existence of such constructs in the component specification and the existence of some of the clauses that can be included in these constructs and which are easy to automate by the wrapper generator.

[18] The only automation possible is to give the user the list of price and licensing options that are listed in the corresponding clauses in the component specification.

[19] Assuming that the user will enter a decision on the above items, the generator can select an initial list based on the user's decisions and on the initial parsing of some of the clauses in the component specification.

[20] Assuming the component supplier provides special API to do this activity.

[21] Assuming the component supplier provides special web interface to do this activity.

The *Evaluate Component* step begins with preliminary evaluation, including simple checks, such as checking and matching of some of the non-behavioral constructs. Section 12.5.5 gives an algorithm to perform the matching of non-behavioral constructs. The wrapper generator presents the system integrator with a list of matched components. Depending on the system integrator's decisions on what components to accept and what to reject during the preliminary evaluation, the wrapper generator could either move to the next step (candidates selection) or prompt the system integrator for new or modified search criteria.

11.1 Candidates Selection

With the availability of the Internet and the many vendors that are providing pre-built components, a single search for a component could result in an overwhelming number of choices, which would make component selection a difficult task.

Researchers have demonstrated the use of formal specification matching [PENI00, ZARE97] in retrieving components, where automated reasoning is used to compare component specifications. However, [PENI97a] stated that it can be computationally expensive to use such formal specification matching to test a large number of components. [PENI97b] tried to find more efficient ways for component retrieval by using a semantic classification scheme to reduce the number of specification-matching

proofs that are required to process a query. Fortunately, the two formal specification languages presented in Chapter 5 to Chapter 8 use the same grammar to describe the behavior of compositional wrappers and off-the-shelf components. The two specification languages also have corresponding non-behavioral constructs. Existence of these features in the two languages make the matching activity performed by the algorithms of Section 12.5 a simple one. Additionally, the `<DownloadsDeclaration>` construct (see Section 7.1.13) in the component specification language makes it easy to specify and retrieve the component and any of its associated artifacts.

During this selection phase, system integrators must consider a variety of issues specific to economic, quality, planning, or risk demands, depending on the integrators' organizational priority and constraints. The characteristics of each individual component will have various impacts on each of the selection factors adopted by the organization. Each factor will also have different influence on the final selection decision. When the system integrator reaches a decision on what components look promising, the wrapper generator can move to the detailed evaluations stage. Alternatively, the generator can first give the system integrator the option of reserving one or more of the selected components, as outlined in the next section.

11.2 Component Reservation

In global business models that employ software agents to acquire components from vendors, component reservation becomes an issue. The need for such reservations

is apparent when considering the demand and supply issues. In a general market, a supplier cannot produce unlimited quantities of a component (not necessarily software components.) At the same time, a supplier does not want to be behind with consumer demands.

A consumer seeking a certain component may find it necessary to reserve the component that matched its criteria until a final decision is reached (possibly by a human) on whether or not to proceed with the purchase of the component. If the supplier of the software component has no restrictions on duplicating the component, then a reservation might not be needed. This situation is applicable to suppliers who are themselves the producers of the components. Public component libraries also fall in this category. On the other hand, component reservations are necessary for third-party component suppliers. Normally, third-party component suppliers purchase a known number of copies from original component developers or other third-party suppliers. The copies will normally be in the form of a license to duplicate the component a limited number of times. In this situation, a consumer who finds a good deal from a specific third-party supplier would find it necessary to reserve the component before someone else reserves it or purchases it.

The issue of component reservation is presented at this stage in the composition process because the next stage (detailed evaluations) could take a prohibitively long time. By the time the development team reaches a decision on whether or not to acquire the component, the component might no longer be available. In addition, the detailed evaluation stage could require accessing some of the component-supplier's resources for which

suppliers might not wish to disclose without at least an initial commitment from the consumer to purchase or license the component.

The wrapper generator or middleware agents can automate component reservation if component suppliers provide special APIs for such purpose. In this case, the generator might need to send the development organization's business and/or payment information to finalize the reservation process. For each reservation made, the wrapper generator will need to store (in its internal compositional wrapper's structure) the following set of information:

- Reservation ID
- Component ID
- Vendor ID
- Reservation Date
- Expiration Date
- Paid Fees

CHAPTER 12: DETAILED EVALUATION AND SELECTION OF BEST-FIT COMPONENTS

Table 7 gives a detailed activity list for activities 2.1.1.2.4 to 2.1.1.15. In addition, the table indicates possibilities for automation of such activities (at low, medium, high level versus manual/no automation.) The footnotes to the table provide the justification for the automation provided (as in previous chapters, "G" indicates the automation level when a wrapper generator is used. "WG" indicates the automation level without a generator.)

Table 7: Detailed activities breakdown of all the activities related to the detailed evaluation process

	Automation			
	Manual	**Low**	**Medium**	**High**
0.0 Develop and operate a compositional wrappers-based (CWB) system				
1.0 Design the CWB system (Chapter 9 and 9.5)				
2.0 Build the CWB system (Chapter 10 to Chapter 14)				
2.1 Collect components (Chapter 10 to Chapter 13)				
2.1.1 Procure components (Chapter 10 to Chapter 13)				
2.1.1.1 Search for components (Chapter 10)				
2.1.1.2 Evaluate components (Chapter 11 and Chapter 12)				
2.1.1.2.1 Conduct preliminary evaluation (Chapter 11)				
2.1.1.2.2 Select candidate components (Section 11.1)				

									Automation			
									Manual	**Low**	**Medium**	**High**
					2.1.1.2.3 Reserve candidate components (Section 11.2)							
					2.1.1.2.4 Perform detailed evaluation on candidate components							
						2.1.1.2.4.1 Manage component documentation[22]						
							2.1.1.2.4.1.1 Locate documentation[23]			WG[24]		G[25]
							2.1.1.2.4.1.2 Download documentation			WG[24]		G[25]
							2.1.1.2.4.1.3 Review documentation		G, WG			
							2.1.1.2.4.1.4 Evaluate documentation		G, WG			
							2.1.1.2.4.1.5 Revise documentation[26]		G, WG			
							2.1.1.2.4.1.6 Print documentation		G, WG			
							2.1.1.2.4.1.7 Organize documentation[27]		G, WG			
							2.1.1.2.4.1.8 Distribute documentation to interested teams and individuals		G, WG			
						2.1.1.2.4.2 Test components						
							2.1.1.2.4.2.1 Develop a prototype to test the component					
								2.1.1.2.4.2.1.1 Specify what to prototype	WG	G[28]		
								2.1.1.2.4.2.1.2 Design prototype	G, WG			
								2.1.1.2.4.2.1.3 Develop a wrapper around the component[29]		G[30], WG[31]		

[22] Supplied documentation could include manuals, user guides, help files, and online information.

[23] The *downloads* construct in the component specification language should list all documentation supplied with the component.

[24] The assumption here is that the user will use the Internet to do the search for and the download of documentation.

[25] The generator will utilize the *downloads* clause to identify and download provided documentations. It is also assumed that all available documentation will be listed in the *downloads* clause in the component specification and that it is possible to download such documentations before making commitments on whether or not to purchase the component.

[26] Documentation revisions could include adding amendments to vendor-supplied documentation and developing in-house supplemental documentation.

[27] Would include binding, color-coding, etc. of the documentation materials.

[28] The wrapper generator can use the behavioral constructs in the component specification to build an initial specification for the prototype.

	Automation			
	Manual	**Low**	**Medium**	**High**
2.1.1.2.4.2.1.4 Develop the prototype code	WG	G[32]		
2.1.1.2.4.2.2 Prepare a set of test cases suitable for testing the prototype	WG		G[33]	
2.1.1.2.4.2.3 Prepare required environment settings, equipments, and tools necessary to properly operate and test the prototype[34]	G, WG			
2.1.1.2.4.2.4 Execute test cases and observe and record test results	WG	G[35]		
2.1.1.2.4.2.5 Analyze test results	G, WG			
2.1.1.2.4.3 Evaluate the component vendor[36]				
2.1.1.2.4.3.1 Check vendor's market credibility	G, WG			
2.1.1.2.4.3.2 Check vendor's stability (the number of years in business and the risks of going out of business)	G, WG			
2.1.1.2.4.3.3 Evaluate available licensing options	WG		G[37]	
2.1.1.2.4.3.4 Check availa-	WG	G[38]		

[29] The wrapper is used to protect the testing environment from any hidden functionality of the prototyped component.

[30] The wrapper generator can instantiate a pre-existing template or script and configures it with parameters specific to the current component. The system integrator will need to check the generated code.

[31] Parts of this code can be reused in subsequent components. Therefore, for subsequent components, this activity will be partially automated.

[32] Parts of this code can be auto-generated. The auto-generated code can include interface calls, checking the type and format of returned results, and providing a simple user interface to allow the user to pass data to the component and to look at the returned results.

[33] Test cases can be generated from the inputs, outputs, and assertions of each component interface.

[34] Though this activity can be done once and reused in subsequent evaluations, the process is still considered to be manual because it will always involve humans that will need to make sure that the correct environment is present.

[35] Though executing test cases can be easily automated, observing and recording test results might require the user involvement.

[36] Some of the vendor evaluations, such as market credibility and stability, can be done once and reused when needed. Most of the other activities listed here are specific to the component at hand.

[37] There is a specific clause in the component specification language to allow specifying this property. However, that list might not be enough for the user. The user might want to read the details of each provided option.

								Automation			
								Manual	Low	Medium	High
							ble maintenance and upgrade options				
							2.1.1.2.4.3.5 Check available support	WG		G[37]	
							2.1.1.2.4.3.6 Check available training	WG	G[39]		
							2.1.1.2.4.3.7 Check offered warranties and required agreements	G, WG			
							2.1.1.2.4.3.8 Check vendor's willingness to escrow source code	G, WG			
							2.1.1.2.4.3.9 Check vendor's willingness to make modifications to component	G, WG			
							2.1.1.2.4.3.10 Evaluate risks involved in dealing with this vendor[40]	G, WG			
							2.1.1.2.4.3.11 Examine external evaluations and reviews by others[41]	G, WG			
						2.1.1.2.4.4 Locate and read external component evaluations by others		G, WG			
						2.1.1.2.4.5 Evaluate non-functional attributes[42]		WG		G[43]	
						2.1.1.2.4.6 Evaluate conformance to standards, architectural styles, etc.		G, WG			
						2.1.1.2.4.7 Evaluate constraints/restrictions in operating		WG	G[45]		

[38] Maintenance and upgrade costs could be supplied in the component specification. Other properties in this activity would need to be checked manually.

[39] Training costs could be supplied in the component specification. Other properties would need to be checked manually.

[40] [MORI00] identified many risks in regards to dealing with COTS vendors. Such risks include slippage in project schedules due to delays in releasing the initial or subsequent releases of COTS, lack or unreliability of supplied documentation, functions promised but never implemented, ripple effect of modifications of new releases of COTS in other COTS or software, and lack of vendor's responsiveness to customer's questions and concerns.

[41] This requires searching for such evaluations in the vendor's website (in the form of press releases), consulting evaluation and accreditation agencies, or reading published literature in journals and conference proceedings. One might also want to consult some of the vendor's clients, specifically those that are using or have used this component.

[42] Non-functional attributes would include performance, quality, reliability, reusability, needed resources, security options, etc.

[43] Many parts of this activity can be automated using the AHP technique (see Section 12.3.5.2) and the algorithms provided in Section 12.5.5.

							Automation			
							Manual	**Low**	**Medium**	**High**
						the component[44]				
						2.1.1.2.4.8 Evaluate built-in smart features[46]	G, WG			
						2.1.1.2.4.9 Evaluate risks of obtaining the component[47]	G, WG			
						2.1.1.2.4.10 Evaluate effort needed to design and implement backup plans in case of failure or inadequacy of the component[48]	G, WG			
						2.1.1.2.4.11 Compare the component with other similar or compatible components[49]	WG		G[50]	
						2.1.1.2.4.12 Evaluate other means of having the same functionality supplied by this component[51]	G, WG			
					2.1.1.2.5 Negotiate price, licenses, payment options, training, support, etc.		WG	G[52]		
					2.1.1.2.6 Select best-fit components[53]		WG	G[54]		

[45] Though most of this step can be automated using information supplied in different clauses in the component specification, vendors can supply such information in narrative form for which the generator cannot automate.

[44] Components from different vendors (and sometimes different components from the same vendor) often require specific versions of software or other components in order to work correctly. Sometimes the required versions are documented; often they are not.

[46] One could weigh the effort needed in locating and evaluating such built-in smart features in the component with the effort needed in injecting such features in the generated wrapper's code. This latter effort is almost negligible due to its automation by the wrapper generator.

[47] These could include commitment to the component's technology and the vendor's updates' schedule and style of communication.

[48] A backup plan could consider other compatible components developed by the same or other vendors.

[49] This implies the need to have access to a repository of components that are more cost-effective to the user than acquiring the currently evaluated component.

[50] The wrapper generator can use the algorithms of Section 12.5 to compare the specification of this component with the specifications of other components. This requires the wrapper generator to have access to a repository of specifications for ready-made components.

[51] This would include new development, wrapping legacy systems, reengineering, reuse, etc.

[52] If the vendor is utilizing software agents that the wrapper generator can communicate with, then this activity can be partially automated.

[53] This could require meetings with technical and management groups. It might be necessary to re-consult supplied documentation and inspect evaluation results. A component could be rejected if found to violate some parts of the required functionality. If the selection process is done manually (i.e., without a wrapper generator,) a component could also be rejected when it does not deliver the number and type of needed interfaces. This is not a problem when using a wrapper generator since the generator can acquire additional components to satisfy the user's needs and then wrap the components together to form the compositional wrapper.

	Automation			
	Manual	**Low**	**Medium**	**High**
2.1.1.2.7 Evaluate other components or interfaces required by the selected best-fit components[55]	WG	G[56]		
2.1.1.2.8 Evaluate and select optional features, plug-ins, or add-ons identified in the selected best-fit components[57]	WG	G[56]		
2.1.1.2.9 Match required interfaces with provided ones	WG		G[58]	
2.1.1.2.10 Identify remaining required interfaces[59]	WG		G[59]	
2.1.1.2.11 Refine the wrapper specification	WG	G[7]		
2.1.1.2.12 Search for, evaluate, and select remaining required components[60]	WG	G[56]		
2.1.1.2.13 Perform group evaluation on best-fit components				
2.1.1.2.13.1 Cross-compare the number and complexity of provided and required interfaces	WG	G[61]		
2.1.1.2.13.2 Cross-compare non-functional features	WG		G[43]	
2.1.1.2.13.3 Cross-check with organization's strategies and standards[62]	G, WG			

[54] The wrapper generator can present a list of candidate components using the algorithm of Section 12.5.13. However, there are other tasks (as noted in footnote 53) that the user will need to do manually.

[55] This would require performing all the sub-activities in activity 2.1.1.2.

[56] Since selection of such components or features and evaluating them would require going through all the previous activities under the *Evaluate Components* (activity 2.1.1.2,) the expected automation level selected for this activity is about the average of automation level of all other activities.

[57] This would be needed in two cases. The first one is when the matching of required interfaces to provided ones reveal some missing interfaces. In this case, it would be advantageous to the user to acquire the optional features that provide some of the missing interfaces than to acquire new components that could come from different vendors. Having more than one component from the same vendor could save the user effort, time, and money in all the different stages of the composition process. The second case of selecting the optional features, plug-ins, or add-ons is when the user finds them helpful in extending and/or enhancing the characteristics of the selected components. This activity would require all the evaluation activities preceding it.

[58] The generator will utilize the algorithm of Section 12.7.1 to do the matching process.

[59] Refer to the *Calculating Additional Needed Components* section on page 357 for details.

[60] This could require repeating all of the previous activities.

[61] The wrapper generator can perform few cross-comparison checks between the interfaces of different components. For example, it can check the number and type of passed parameters and returned results.

[62] Although standards can, and do, improve our ability to integrate products, they will never, of themselves, make integration difficulties disappear [WALL98b]. Moreover, in the commercial marketplace, innova-

							Automation			
							Manual	**Low**	**Medium**	**High**
						2.1.1.2.13.4 Select the components that passed the group evaluation process	G, WG			
						2.1.1.2.14 Select the minimum set of components that fulfill the user's requirements[63]	WG		G[58]	
						2.1.1.2.15 Evaluate opportunities in rejected components[64]	G, WG			
		2.2 Wrap components (Chapter 14)								
		2.3 Deliver compositional wrapper (Section 14.11)								
		2.4 Assemble components into the CWB system (Section 15.1)								
		2.5 Test the CWB system (Section 15.2)								
	3.0 Operate the CWB system (Chapter 15)									

Assessment is the process by which components are selected for use in the larger system being developed [ADAM01]. Detailed assessments of candidate components are needed in order to differentiate among the many similar features that components come with. Detailed assessments include designing, developing, and testing a prototype that would include the assessed component; evaluating the vendor supplying the component; and evaluating the non-functional features of each candidate component. Assessment efforts vary significantly by software product classes (i.e., operating system, database management system, user interface, device driver, and so forth) [BASI01].

Table 7 above lists all the activities that form the detailed evaluation process. The wrapper generator can automate only some of the activities involved in detailed evalua-

---

tion, not standardization, is often what distinguishes products and provides competitive advantage to software vendors.

[63] The assumption here is that the group evaluation activity resulted in a set of components that contain at least the features and interfaces that the user needs. If this is not the case, then the wrapper generator will need to go back and search for more components.

[64] Since the wrapper generator will be dealing with specification files only, storing, and possibly classifying, specification files of rejected components for future use could have great advantage in future-built systems. Another advantage of analyzing rejected components is to discover features that could have been missed in the design or specification of required components.

tions; others cannot be at all. For example, consulting and reviewing supplied manuals and documentations will require manual handling unless the manuals and documentations are written using a standard formal method. Evaluating the component vendor requires decisions by the development team. On the other hand, the generator can automate parts of the testing activity, such as preparing a wrapper around the tested component and selecting and executing test cases. The development team would still need to do some parts of the component testing, such as preparing the required environment settings and examining the test results.

An open issue for future research is whether the information in a wrapper specification sufficient to support a decision about what components to use in the compositional wrapper. There is no doubt that the wrapper and component specification languages described in Chapter 5 to Chapter 8 contain many constructs that can help in forming an initial decision. Additionally, the wrapper generator can use the links provided in the *downloads* clause of the component specification to access component documentations. However, detailed evaluation involves other activities (such as meetings, reading external evaluations, and performing decision-maker preferences) that are difficult to express in the wrapper specification.

The demonstration of the use of the WSL and the CSL specification languages to aid in automating the testing of off-the-shelf components is another area for future research. This is because a good amount of understanding of the component depends on the vendor's claims for the product. Initial experimentation of the testing issues indicates that testing behavioral constructs against actual component behavior is doable, provided

the right testing tools are provided. Section 12.3 provides information to support testing of components. However, before beginning components testing, testing engineers must be aware of the type and impact of any associated risks in the components to be evaluated. Sections 12.1 and 12.2 address these issues.

12.1 Vendor Evaluation

In COTS-based development, component integrators (as wells as compositional wrapper users) are dependent on the vendor to be the ultimate decision maker on the component functionalities, schedule for releases, architecture, reliability level, documentation, and service level [MORI00]. The purchaser has little or no influence on these issues. Rather, COTS software vendors are driven by today's fast-paced market, characterized by highly volatile business strategies and market positions [ADAM01].

There may be significant informal exchanges between the qualified vendors and the acquirer before receiving a formal vendor response [MOSK00]. Supplier cooperation can never be assumed; the supplier may be just as willing to adapt the product for the next big sale, leaving previous customers behind. Nor is supplier cooperation always desirable. Too much cooperation begins to resemble customization, which eliminates the advantages of buying in the open marketplace.

The following are some of the information that the development team would need to gather about evaluated vendors [BEUS00]:

- Component upgrade policy (e.g., based on new features and/or bug fixes)
- References for component use (customer base)
- Long-term component support strategy
- User support record
- Vendor's software development practice (e.g., ISO 9000)
- Vendor's popularity in a particular application domain
- Vendors' stake in a particular domain (if sufficient, it provides a scope for introducing component features relevant to that domain)
- Contract practice (guarantees given and obligations that the vendor is prepared to take on)

12.2 Risks Evaluation

Components-based development is currently a high-risk activity, with effort and schedule overruns exceeding non-CBS software overruns [BASI01]. Even if system integrators have managed to find the right component that appears to match their requirements, at the expected price, and with acceptable quality levels, there are still great deals of risks involved. System integrators must evaluate the danger involved in choosing a component that has not been sufficiently tested, has been developed according to a poor (or no) methodology, or does not provide any help for the technical difficulties that could

arise during its operational life. The lack of visibility in off-the-shelf components, vendors' temptations to over-promise, and the difficulty of estimating glue-code size combine with over-optimism and unrealistic cost and schedule commitments. Additionally, when multiple organizations are involved, or when components are externally developed (whether commercially or otherwise), assessment of software development, processes, and personnel capabilities are still not effective enough to assure a healthy operational life of the CBS system [VECE01].

Failure could occur not only in individual components, but also in component interactions (for example, when components are incorrectly composed, e.g., *A(B)* instead of *B(A)*). The probability of occurrence of such risks increases with the increase in the number of integrated components and the interactions among the integrated components [HISS99]. Failure of individual components or component interactions could cause the entire system to fail. For example feeding wrong or corrupt inputs to individual components could cause failure in the entire system.

COTS components can suffer from other types of problems [ABTS00b]. For example, delays in obtaining licenses, up-front licensing fees, recurring maintenance fees, and the lack of control over the upgrade and maintenance schedules can all have great effect on the development and operation of CBS systems. Additionally, the reliability of COTS components is often unknown or inadequate. COTS components sometimes come with too many unneeded features that can compromise the usability and performance of such components.

When COTS components come from multiple vendors, integration of such components become difficult because of possible incompatibilities among vendors and of the difficulty in synchronizing component upgrades from the multiple vendors. [MORI00] identified many other risks in regards to dealing with COTS vendors. Such risks include slippage in project schedules due to delays in releasing the initial or subsequent releases of COTS, lack or unreliability of supplied documentation, functions promised but never implemented, ripple effect of modifications of new releases of COTS in other COTS or software, and lack of vendor's responsiveness to customer's questions and concerns.

Security risks [LIND98] are also a big issue in risk assessments at this stage. Security risks could have several forms. For example, the system, through human misuse, could experience loss of confidentiality, integrity, or availability for any of its resources. The system can also, through misuse or by accident, experience the introduction of security vulnerabilities. Security vulnerability is a flaw that could later be exploited to cause a loss of confidentiality, integrity, or availability.

Generally, during the evaluation stage, the following activities can help in mitigating the impact of risks inherent in off-the-shelf components:

12.2.1 Feasibility Analysis

Even if the description and the specification of the component matches the requirements of the development organization that are set forth in the wrapper's specification, component integrators should also examine other issues, such as the adaptation re-

quired to get the component up and running. At a minimum, integrators should check the compatibility of the functional and technical characteristics of the component with the new system. Integrators should also check if the evaluated components had been successfully adapted in other similar systems.

12.2.2 Development Team Capability Assessment

The knowledge and experience of the development team are important factors in evaluating and selecting the right components to include in the target compositional wrapper. The development team should collect various information about the evaluated components. For example, understanding the methodology used in constructing the component and knowing the type and quality of available documentation, support, training, and on-line help can all help in making the evaluation process smooth and effective.

12.3 Testing Components

In *black-box* testing, no knowledge about the internal implementation is assumed. On the other hand, *white-box* testing uses the component's implementation details in setting testing criteria. Black-box testing can be further classified as specification-based testing and interface-based testing [ZHU97]. *Specification-based* testing uses features present in the component specification to specify the testing requirements. *Interface-*

based testing focuses only on the domain and range of the component inputs without reference to any internal features of the component or of its specification.

In most situations, components might not be developed by the same organization that is using them, and source code is not provided. Therefore, white-box testing is generally not possible. In addition, white-box component testing is less effective when a component's configuration, environment, and dependencies can be changed at integration or deployment time. At the same time, though performing component-level black-box testing can increase confidence in the component, developing the associated test oracles can be an expensive proposition. System level testing can be effective, but it has a high associated cost, and cannot cover all potential behaviors for most software. Thus, additional confidence must be obtained through other means.

12.3.1 What to Test For?

In addition to the many activities outlined in Table 7 that are required in testing components, the following aspects are as important to gain the confidence that the component will actually deliver what it promises in its specification:

- The *human, hardware,* and *software requirements* for operating the component.
- The *response time* for generating expected outputs.
- The *output accuracy* in attaining the expected results.
- The *number* and *type* of *applications* that are currently using the component.

- The *development methodology* used in constructing the component. Good methodologies will eventually lead to a low probability of system failure.
- The *component maturity* attributes. These will include the component's first release date, the number of checks conducted, the errors detected, the number of repairs performed in the last year, and the date of the latest modification due to error.
- The *credibility* of the *developer organization*. A company with a good reputation in component development will give a better guarantee that exhaustive quality controls have been performed.
- The *license clauses* and the *guarantees* that such clauses hold in covering component failures.

12.3.2 Test Cases

Specifications can play two main roles in software testing [ZHU97]. The first is to provide the necessary information to check whether the output of the component is correct. The second is to provide information to select test cases and to measure test adequacy.

Test cases can be used to find discrepancies between the system and its requirements and to test the specifications of the tested systems. Test cases are normally generated for all classes of anticipated system usage (correct and incorrect) and for possible variations in the operating environment. Some test cases are targeted to test the system

features and the quality requirements. Other test cases could be used to test the load and stress effects on a system. In critical systems, security and performance are also important testing issues.

Many test-cases generation tools are currently available. For example, decision tables and state transition diagrams [MAYR90] are an easy mean to generate test cases just by walking through each of these structures and generating them. However, new systematic methods are needed to construct test drivers and stubs for diverse components and various customizations [GAO00a]. Component test stubs are needed in the construction of component frameworks. Each test stub simulates a black-box function and/or behavior of a component. The essential issue is how to generate reusable, configurable, and manageable test drivers and stubs for a component.

There are two general approaches to generating test stubs. The first approach is model-based, in which component stubs are developed to simulate a component, in a black box view, using a formal model, such as a finite state machine, a decision table, or a message request-and-response table. The second approach is operational script-based, in which test stubs are constructed to simulate a specific functional behavior of a component in a black-box view. This mechanism can enable access and exercise of the built-in tests in a consistent way.

12.3.3 Tests Adequacy

In addition to generating test cases, the adequacy of the test cases must be determined with respect to the tested component requirements and the organization needs. There are several approaches to measure test adequacy (see [ZHU97] for a survey.) Selection of one approach depends on the testing requirements and the testing budget and schedule constraints. For non-trivial software components, it is impossible to be certain that there are no residual defects in the code because formal proofs or 100% test coverage are not viable options in most practical cases [ATKI02]. Additionally, even if a component is fault free, the component may fail in a deployed environment when it has to rely on the services of other components or has to compete with other third-party components for resources such as memory, processor cycles, and peripherals.

12.3.4 Executing Tests

Testing software components is more complicated when using wrapper generators. On one hand, at this stage, the development organization has not made commitments on purchasing the needed components. (Organizations cannot make such decision without first testing the components for conformance to their specifications.) On the other hand, testing components requires having access to them. In most situations, component access would not be possible without purchasing or licensing the components. To work around the problem, this section proposes three approaches that can be used to test the components without actually downloading them or making commitments on buying them.

12.3.4.1 Let the Component Vendor Do the Testing

In this model, test engineers pass the wrapper specification to component vendors along with other testing materials. Component vendors would test their components and then pass back the results to the test engineers. Test engineers would need to pass the following to component vendors:

- The wrapper specification.
- Any test cases prepared by the test engineers.
- Any restrictions, other than those imposed by the wrapper specification. Such restrictions would include testing costs restrictions, performance criteria, the type and format of test result documentations expected from vendors, and the time frame for when the results are needed.

Once vendors receive such information, they would test their components using the passed information. Upon testing completion, test engineers would expect to receive the following from component vendors:

- The results collected from executing the supplied test cases.
- The list of components and interfaces that passed the tests along with the specifications of such components and interfaces. Component vendors could also include a

list of components that partially satisfy test engineers' requirements. This will give test engineers more options to consider.

- Description of the operating environment in which each component was tested.
- The vendor's conditions for releasing the tested components to the user. This would include components availability, types of support offered, and any promotions or discounts available.
- A statement to guarantee the authenticity of the testing results.

This approach has several advantages including:

- It protects component vendors from exposing their implementation details to outsiders. Of course, this is not an issue with off-the-shelf components.
- It relieves both vendors and users from finding ways to encode the semantics of tested components. Nevertheless, having such semantics at hand would help vendors in understanding what component users are looking for. Semantics' specification would be readily available through wrapper specifications being passed to vendors.
- It is safe to the user since all the testing is done on the vendor's side.
- It is a profit-generating business since, in most cases, vendors would impose some fees for doing the testing.

- It requires no commitment from the user to purchase the component until being assured of the component's compliance with its specifications.

Despite the many stated advantages, there are still some disadvantages in using this approach, including:

- This approach is costly to the user since most component suppliers might impose fees to do the testing on the user's behalf.
- It is a slow process. Testing time is only one reason for such slowness, especially if the testing is to be done manually by the vendor. Other factors that could slow the process would include the mechanisms used to send component specifications to vendors and to receive results from them.
- The model might not be applicable for fine-grained components, especially when considering the time it takes and the costs incurred to do the testing.

12.3.4.2 Vendor-Supplied Test Cases and Test Results

In this approach, vendors include sample test cases and test results within component specifications. Another possibility is to build the testing capability within the component itself and to provide, in the component's specification, some means of executing them [GAO00a]. After finding candidate components (using the methods discussed in the previous chapter), test engineers can use the test cases provided to verify the compo-

nent's compliance with its specification. However, this approach would still require the development organization to purchase, and possibly download, the component unless public testing interfaces (as suggested by the third approach below) are available. This approach would also require introducing new constructs in the CSL language to enable vendors to encode test cases and test results within component specifications.

12.3.4.3 Public Interface for Executing the Component

With this approach, vendors would supply a public testing interface as part of component specifications. Test engineers would then develop test suites that utilize the testing interfaces. Engineers would then perform required tests using available testing techniques, such as the plug-and-test technique [GAO00a] or the Interface Propagation Analysis (IPA) [VOAS98f]. IPA is a fault-injection based technique for injecting "garbage" into the interfaces between components and then observing how that garbage propagates through the system. IPA analyzes the behavior of a component by looking for specific outputs that the user wants the technique to be on the lookout for.

To make this approach in wide use, it is necessary to standardize test access interfaces. Some non-functional features, such as performance and security, might be difficult to assess using this approach because of possible degradation in network bandwidth. In addition, this approach would require some way to assure that the tested component (through the public interface) is the same as the one described in the acquired component specification.

12.3.5 Analyzing Test Results

After resolving other testing issues, such as deciding on the best approach to test components, and after performing the actual tests, the next problem that would need to be resolved is capturing and analyzing test results. The wrapper generator might be required to use some AI techniques or heuristics to compare component specifications with observed results. Such comparison techniques are only applicable for components that result in measurable and easy to capture outputs. For other types of outputs, test engineers would need to manually observe and interpret the results. Some types of outputs are difficult to capture and measure because of their nature (for example, animated or event-driven outputs), their length (for example, trace files), or of their sensitivity (for example, outputs that include classified information or modify live data).

Assuming test results can be captured completely and correctly, the next issue becomes on interpreting such results. Though automatic parsing and analysis of certain types of results might be possible, most of the times, it is the test engineers' responsibility to perform such analysis. Drawing conclusions or statistical inferences from test results is by itself a problem. The wrapper generator can help in the analysis by using the test results to compare the specifications of the tested components with that provided by the system integrator. This analysis process will result in one of the following three outcomes:

- The generator finds one or more components that exactly match the functionality, features, and constraints given by the system integrator.
- The generator might not be successful in finding a single component that matches all of the user's requirements. However, the generator could identify one or more groups of components for which the union of their functionality, features, and constraints matches what the system integrator wants.
- The generator could not identify a single component or a group of components that exactly match what the system integrator wants. In this case, the generator leaves the decision to the system integrator.

The following subsections discuss some of the issues that would need to be considered when analyzing the test results.

12.3.5.1 Measuring the Quality and Reliability of Tested Components

Many literature accounts (see for example, GOME00, SCHA98, and HANS99) discuss how to measure the software quality from various development and operativity viewpoints. However, there are no accepted standards for the quality of software components, largely because there is no theoretical foundation on which to base such standards [HAML00]. Most software component developers use subjective assessments of soft-

ware quality. Their decisions might lead to reengineering of the organization procedures to take maximum advantage of a components technology or product.

Reliability is defined as the "probability of fault-free operation of a computer program in a particular environment in a given time" [GOME00]. [EVER99] proposed a model for measuring the reliability of software components. His approach depends on the static and dynamic properties of components. While this technique gives a concise way of measuring component reliability, it is limited to components whose source code is readily available during testing. In addition, it requires measurement parameters, such as the compiler optimization factor, and values from previous releases of the component. Such information is usually only available to component developers and not component users.

12.3.5.2 Analyzing Non-Functional Attributes

The role of non-functional (or qualitative) requirements is very important because off-the-shelf software components have their functionality already built-in. Non-functional software requirements are difficult to elicit, express, quantify, and test [BEUS00]. Functional requirements need to be specified, but only to a level which enables efficient survey and evaluation of the available components on the market. Once a particular subset of components is identified, non-functional requirements are called in. External constraints on a component can help to eliminate a good proportion of component candidates. Non-functional requirements often correspond to strategic or business

objectives of end-user organization as a whole, and, therefore, are likely to have higher priority if conflicting with some of the functional requirements for the software component. Non-functional requirements frequently come from managers, who are ultimately responsible for taking decisions on acquiring components.

The Analytic Hierarchy Process (AHP) [LUNG00] is one technique that can be used to reach decisions on multiple non-functional attributes. AHP is used to make multiple criteria decisions (i.e. decisions that involve criteria of a complex and conflicting nature.) AHP was developed in the early 1970's by Dr. L. Saaty of the Wharton Business School. AHP allows both qualitative and quantitative criteria to be compared using informed judgments to derive weights and priorities. The following algorithm illustrates the application of the AHP method. The algorithm can be easily implemented in wrapper generators, as can be seen from the pseudo code listing that will be presented at the end of this section. To illustrate the use of the method and this algorithm, assume a component is selected based on three non-functional attributes: performance, fault-tolerance, and portability. AHP involves the following steps which lead to assigning suitability weights to the components at hand.

Step 1: The system integrator can use a scale of pair-wise comparisons to express the relative importance of one criterion over another. This is entirely at the discretion of the system integrator. A scale of importance might look something like:

1. Of equal importance

3. Of moderate importance

5. Of strong importance

7. Of very strong importance

9. Of extreme importance

Step 2: The second step is to rank each non-functional attribute using the above scale, as shown in the following table:

Performance	Fault-Tolerance	Portability	
1/1	1/2	3/1	Performance
2/1	1/1	4/1	Fault-Tolerance
1/3	1/4	1/1	Portability

The two numbers in each cell compares the attribute in the column to that in the row. For example, in the above table, the "4/1" cell says that *portability* has more than moderate importance to the system integrator than *fault-tolerance*.

Step 3: The next step is to turn the fractions in the above table into decimals (using a fixed number of decimals), as follows:

Performance	Fault-Tolerance	Portability	
1.0000	0.5000	3.0000	Performance
2.0000	1.0000	4.0000	Fault-Tolerance
0.3333	0.2500	1.0000	Portability

Step 4: The resulting matrix is, then, squared to yield:

Performance	Fault-Tolerance	Portability	
2.9999	1.7500	8.0000	Performance
5.3332	3.0000	14.0000	Fault-Tolerance
1.1666	0.6667	2.9999	Portability

Step 5: The rows of the new matrix are summed to produce the following matrix:

12.7499

22.3332
4.8332

Step 6: This matrix is, then, normalized to produce the eigenvector, as follows:

0.3194
0.5595
0.1211

Step 7: Steps 4 to 6 are repeated until the eigenvector does not significantly change from one iteration to the next (by subtracting the eigenvector in the new iteration from that of the previous iteration). Doing so, the following eigenvector is obtained:

0.3196	Performance
0.5584	Fault-Tolerance
0.1225	Portability

The weights indicated by the eigenvector reflect the non-functional attributes' importance to the system integrator. In the above example, fault-tolerance is the most important followed by performance and then portability.

Step 8: Now, the components at hand need to be similarly weighed using the above technique and individually for each of the non-functional attributes. For example, assume four components are to be compared: A, B, C, and D. Then, the importance of each of the three non-functional attributes to the four components is as follows:

Fault-Tolerance:

A	B	C	D	
1/1	1/4	4/1	1/6	*A*
4/1	1/1	4/1	1/4	*B*
1/3	1/4	1/1	1/5	*C*
6/1	4/1	5/1	1/1	*D*

After computing and normalizing the eigenvector, the following matrix is obtained:

0.1160	A
0.2470	B
0.0600	C
0.5770	D

Using the same technique for performance and portability, the following normalized eigenvectors are obtained:

Performance:

0.3010	A
0.2390	B
0.2120	C
0.2480	D

Portability:

0.3790	A
0.2900	B
0.0740	C
0.2570	D

Step 9: Finally, the matrices of the normalized eigenvectors of both the non-functional attributes and the components are multiplied to yield the following:

$$\begin{bmatrix} \textit{Fault Tolerance} & \textit{Portability} & \textit{Performance} & \\ \hline 0.1160 & 0.3790 & 0.3010 & A \\ 0.2470 & 0.2900 & 0.2390 & B \\ 0.0600 & 0.0740 & 0.2120 & C \\ 0.5770 & 0.2570 & 0.2480 & D \end{bmatrix} * \begin{bmatrix} 0.5584 & \textit{Fault Tolerance} \\ 0.1225 & \textit{Portability} \\ 0.3196 & \textit{Performance} \end{bmatrix} = \begin{bmatrix} 0.2072 & \textit{Ranking } 3 & A \\ 0.2497 & \textit{Ranking } 2 & B \\ 0.1103 & \textit{Ranking } 4 & C \\ 0.4390 & \textit{Ranking } 1 & D \end{bmatrix}$$

The resulting vector ranks component D as the best followed by B, A, and C, respectively.

The AHP technique is very simple to use and easy to automate using wrapper generators. Additionally, the technique requires little involvement from the system integrator's part. However, the technique is more suitable for selecting non-compatible components and less effective for compatible candidate ones, because non-functional attributes' importance for compatible candidates is of equal importance. Compatibility in this context is addressed with respect to selecting best-fit components in terms of their behavioral matching. Compatibility is not addressed here with respect to non-functional attributes. It will always be considered with respect to functional features.

The following code snippet shows how the above algorithm can be implemented in the wrapper generator.

```
RateNonfunctionalProperties (comps) {
      GetOverallNonfunctionalRatings ();
      rated_components = CalculateRatings (comps);
      return (rated_components);
}

CalculateRatings (comps) {

// Use the Analytic Hierarchy Process (AHP) for rating non-functional
// properties.

      // Step 1:  Enter the rating scale.  This is done globally in
      // the initial use of the wrapper generator and reviewed by
      // the user for every generated wrapper.

      // Step 2:  Let the user rank each non-functional property.
      // This is done in the call to GetOverallNonfunctionalRatings().
      // The ratings are stored in the N x N NonfunctionalRatings
      // matrix.

      // Step 3:  Convert ratings to fractions.  This is done
      // automatically in the GetOverallNonfunctionalRatings ().
```

```
      Step4 (NonfunctionalRatings, NonfunctionalCount);
      Step5 (NonfunctionalRatings, NonfunctionalCount);
      Step6 (eigenvector1);
      Step7 (NonFunctionalRatings, NonFunctionalCount, eigenvector1);

      // Rate non-functional properties with respect to each
      // individual component.

      count = ComponentsCount (comps);
      for (i=0; i < NonFunctionalCount; i++) {
            GetComponentNonfunctionalRatings (ComponentRatings[i],
                  count);
            Step4 (ComponentRatings, count);
            Step5 (ComponentRatings, count);
            Step6 (eigenvector2[i]);
            Step7 (ComponentRatings, count, eigenvector2[i]);
      }

      // Form matrix of normalized eigenvectors for all components.

      for (i=0; i < count; i++) {
            for (j=0; j < NonFunctionalCount; j++) {
                  eigenvectors [i, j] = eigenvector2 [j][i];
            }
      }

      // Multiply the matrix of normalized eigenvectors for all
      // components with that of the non-functional properties.

      ratings = Multiply (eigenvectors, eigenvector1, count,
                          NonFunctionalCount);

      // Order components according to their rankings

      p = comps;
      for (i=0; i < count; i++) {
            list[i].comp = p;
            list[i].rating = ratings[i];
      }

      list = SortRatings (list, count);
      rated_components = nil;
      for (i=0; i < count; i++) {
            AddComponent (rated_components, list[i].comp);
      }

      return (rated_components);
}

Step4 (Ratings, count) {
      // Square the Ratings matrix
```

```
      for (i=0; i < count; i++) {
            for (j=0; j < count; j++) {
                  Ratings [i, j] = Squared (Ratings, i, j);
            }
      }
}

Step5 (Ratings, count) {
      // Returns the sum of the rows in the Ratings matrix

      for (i=0; i < count; i++) {
            sum = 0;
            for (j=0; j < count; j++) {
                  sum += Ratings [i, j];
            }
            RowSum [i] = sum;
      }
}

Step6 (RowSum, count) {
      // Normalize the row sums to get the eigenvector.

      RowSum = Normalize (RowSum, count);
}

Step7 (Ratings, count, RowSum) {
      // Step4() to Step6() are repeated until the new eigenvector
      // does not significantly change from the old one.

      repeat = true;
      while (repeat) {
            old = RowSum;
            Step4 (Ratings, count);
            Step5 (Ratings, count);
            Step6 (RowSum);

            repeat = StillDifferent (old, RowSum);
      }
}

Squared (matrix, rows, cols) {
}

Multiply (matrix1, matrix2, rows, cols) {
}

Normalize (vector, count) {
}

StillDifferent (vector1, vector2, count) {
      for (i=0; i < count; i++) {
            if (abs (vector1[i]-vector2[i]) > THRESHOLD) {
                  return (true);
```

```
                }
            }
}
```

12.3.6 Displaying Analysis Results

The displayed results will include test cases or heuristics used, inputs injected, outputs observed, and environment settings used to test the component. The system integrator can approve the displayed results to allow the generator to move to the group evaluation step. Alternatively, the system integrator can reject results and require the generator to go back to the component selection phase to find new components.

Depending on the analysis results that the generator reached in the previous section and the conclusions (about the quality, reliability, and non-functional attributes) that the system integrator reached, the generator can display the results as follows:

- If the wrapper generator finds one or more components that individually match exactly what the system integrator wants, the generator would present such list of components to the system integrator. The system integrator can further investigate minute discrepancies among the components. The system integrator can also look at the available non-functional constraints, such as the vendor's market credibility, frequency of updates, terms of maintenance and support, and available press releases on the component. After all such investigations, the system integrator will identify one or more single components to be acquired. If the system integrator selects more than one component, then the wrapper generator will acquire all such

components and include them in the generated wrapper. The generated code will include any logic necessary to select among the multiple compatible components. Such code would be needed when one or more components cease to operate as expected or when doing component upgrades or performance re-balancing.

- In the case where one or more groups of components satisfy the integrator's requirements, the generator will present the list of groups to the system integrator and allow the system integrator to review the functional and non-functional specifications of each individual component in each group. The system integrator will need to investigate discrepancies among the components of each group. Such investigations will result in the system integrator selecting one group of components.
- If the generator cannot identify single components or groups of components that satisfy the system integrator's requirements, the generator will then present to the system integrator the identified groups of components along with their missing functionality and features or the constraints that they violate. The system integrator can review the groups and select one of them (as in the second option) or go back and revise the search criteria. If the system integrator selects a group from the list, then the generated wrapper could have fewer provided interfaces than required and, possibly, different or incompatible behavior.

The following code snippet gives a high-level picture of the wrapper generator's behavior in dealing with matching of individual components or groups of components:

```
SelectComponents (wrapper) {
      DisplayComponents (wrapper.matching_components);
      DisplayGroups (wrapper.candidate_groups);

      GetSelection ();
      selected = ParseSelection ();
      return (selected);
}
```

12.3.7 Dealing With Component Interdependencies

Interdependent components usually come as a unit. One solution to the interdependency problem is to pack the interdependent components into one module such as a DLL library [ERIK00]. In contrast, the wrapper specification language, provided by this research, offers the *optional* and the *requires* constructs where such interdependencies can be listed and, optionally, specified. However, while components listed in the *requires* clause must be acquired along with the main specified component, optional components usually add or enhance the functionality of the specified main component.

12.4 Optional Features Selection

Some components come with additional features (or plug-ins) that can be purchased (or at least configured) separately. The list of optional features could include the following:

- Additional services that the component can support

- Enhanced performance, security, or reliability options and measures
- Extended payments and/or licensing options
- Extended support options
- Sample customization scripts

The component specification language provides the *optional* construct to allow vendors to specify optional plug-ins or add-ons that can be acquired with the main component. As part of the detailed evaluation, the system integrator should decide on what, if any, of the optional features to acquire with (or instead of) the evaluated component. The wrapper generator will be responsible for retrieving the specifications of such optional features and for including such specifications as part of the list of components to be evaluated.

Depending on what features the system integrator selects, their specifications could be reached in three ways. First, the specifications of the optional features can be embedded in the specification file of the main component. Second, the specification of the main component can point to the location of the specification files of the optional components. Third, the optional components can be described in narrative terms within the specification of the main component.

For the first option, the wrapper generator would only need to add the embedded specifications to the pool of component specifications to be evaluated. For the second option, the generator would need to acquire the specification of the optional component

using the URL provided in the component specification. If the generator were successful in retrieving the specification of the optional component, then the generator would add that specification to the pool of component specifications to be evaluated. The third option requires the system integrator to find the location(s) of the optional component specifications and to point the generator to them. The generator would try to retrieve such specifications and if it were successful, then it would add the specifications to the pool of component specifications awaiting evaluation. (Note that the pool of component specifications awaiting evaluation could grow with each component evaluation.)

12.5 Selection of Best-Fit Components

The candidate components that resulted from analyzing the test results could still differ in many aspects (such as the number and quality of their provided interfaces.) The candidate components could also differ in the size, performance, cost, interface language, and provided documentation. One could also differentiate components by looking at the vendors providing them. Vendors could differ in their market credibility, support of their products, and frequency of their products' updates. Such enormous number of choices makes it a difficult task for the wrapper generator to select the "elite" components from among the masses.

Thus, the selection process is one of tradeoffs [LOON00]. For example, fitness of functionality may be in conflict with compatibility: a software component that provides the required functionality may be hard to integrate with the rest of the system. The

tradeoff between robustness and throughput is another example: a higher performance component may not necessarily be the most robust component. However, with the formal constructs that deal with interface behaviors (provided by the component specification language (CSL) and the wrapper specification language (WSL) described by this research) and with the aid of the matching algorithms outlined in Section 12.5, the wrapper generator can be used to automate the selection process. Section 12.5.13 outlines the algorithm to use in selecting best-fit components after doing all the matching steps.

The list of criteria that can be used during the selection process is a long one. Some component characteristics could make the selection process a difficult task. For example, components may not have the exact functionality required, they may not be extendable, or they may have come from untried sources. Clearly, it would be difficult to reconcile all of these criteria. For example, assuming a need for 50 components, if there were two other compatible components for every component that was selected, 150 component evaluations would be needed [WALL98b]. Within the tight cost and schedule constraints of most system-development efforts, such a large number of components cannot possibly be examined to the depth required to understand all of the design implications.

Component selection is further complicated due to the lack of clarity on what constitutes a component [KURU99]. In addition, incomplete understanding of the functional specification, performance attributes, and development constraints that need to be met have great effect on such difficulty. Furthermore, the same component from differ-

ent vendors will usually have different functionality. Besides, some requirements may be only discovered during component selection.

In this best-fit components selection, the wrapper would sort candidate components into groups based on their compatibility with each other. Then, the wrapper generator would select one or more compatible components from each group. An algorithm to perform compatibility checks is outlined in Section 12.5.11. The following subsections present some algorithms that discuss the details of matching the various parts of the wrapper specification with the corresponding parts in the component specifications. Each section builds on the information obtained from the section that precedes it. tion 12.5.13 will then put all such techniques together to show how best-fit components can be selected from the evaluated candidates.

12.5.1 How the Matching Process is Performed?

Associated with each component, C, is a signature, C_{sig}, and a specification of its behavior, C_{spec}. Given two components, C = (C_{sig}, C_{spec}) and C' = (C'_{sig}, C'_{spec}), [ZARE97] defined a generic component match predicate called *Match*:

$$Match\ (C, C') = match_{sig}\ (C_{sig}, C'_{sig}) \wedge match_{spec}\ (C_{spec}, C'_{spec})$$

Match returns true if both of the components' signatures and specifications match. Their approach differs from the approach of this dissertation in three ways. First, using

the WSL and CSL specification languages, compositional wrappers and components can be described using their behavioral and non-behavioral characteristics. Second, matching is not performed between components, but rather between the wrapper specification and component specifications. Third, there is no requirement for strict matching of behavioral and non-behavioral attributes. Instead, weighting schemes are used to select the components that best fit the development organization needs.

The following is a high-level algorithm that outlines the steps involved with the matching process. The subsequent sections will give details on how each matching activity is performed.

1. Looking only at the wrapper specification, use the ranking scheme of Section 8.1 to assign weights to all non-behavioral constructs that will be used in the evaluation process. Also, assign weights to the options of each construct. The following constructs will not be weighted: the wrapper's name, `name`, the wrapper's description, `description`, the wrapper's version number, `version`, the wrapper's owner, `owner`, the wrapper's list of developers, `developers`, all of the options in the software constraints, `software_constraints`, the `other` clause in the hardware constraints, `hardware_constraints`, the network constraints, `network_constraints`, the list of smart features, `smart_features`, the personal information, `personal_info`, the payment information, `payment_info`, the temporary directory, `temp_directory`, and the wrapping options, `wrapping_options`. (Section 12.5.2 describes the process of weight assignments to non-behavioral constructs in more details.)

2. Using the ranking scheme of Sections 6.1.3.1 and 6.1.3.2, assign weights to inputs and outputs in the wrapper specification. (Sections 12.5.3 and 12.5.4 give the details of this process.)

3. Create a list of all candidate components and assign each component a weight of 0. (Candidate components are components that passed the detailed evaluation.)

4. For each candidate component:

   a. Evaluate non-behavioral constructs. (Non-behavioral constructs are all constructs defined by the wrapper specification language except *input*, *output*, *behavior*, *stimuli*, and *architecture*.) Additionally, all non-weighted constructs (listed in step 1) are not considered for evaluation. The following algorithm outlines this evaluation process. (Section 12.5.5 gives a detailed algorithm of the process.)

```
for each non-behavioral construct
{
  if the construct exists in the component specification then
  {
    if the non-behavioral construct is singular
      (i.e., has no options) then
    {
      add the construct's assigned weight (from step 1)
       to the component's total weight
    }
    else // multi-options constructs
    {
      for each option in the current non-behavioral construct
      {
        if the construct's option is found in the
            corresponding component's construct then
        {
          add the weight assigned to the option
             (from step 1) to the component's total weight
        }
      }
    }
  }
```

```
}
```

b. Evaluate behavioral constructs (*inputs* (Section 12.5.6), *outputs* tion 12.5.7), *behaviors* (Section 12.5.12)).

5. Select the best-fit components according to the total weights accumulated during the matching process in step 4. (Section 12.5.13 gives the details of this process.)

12.5.2 An Algorithm to Assign Weights to Non-Behavioral Constructs and to Their Options

This algorithm uses the user-defined importance rankings of the general constructs that were discussed in Section 8.1.

1. First, take only the non-behavioral constructs that have importance rankings attached to them and order these constructs in descending order of importance rankings.

2. Assign weights to the ordered ranked non-behavioral constructs starting from 10 and in increment of 10. This means that the construct with the highest ranking (which means the less-important one) will be assigned a weight of 10, the one with the next highest rank will get a 20, and so on. This way, the most important ranked construct will get the highest weight and the least important ranked construct will get a 10.

3. For all other unranked non-behavioral constructs, assign them a weight of 0.

4. For each non-behavioral construct:

    a. Take only ranked options within the current non-behavioral construct and order them in decreasing order by assigned importance rankings.

    b. Assign weights to the ordered ranked options of the current non-behavioral construct starting from the weight assigned to the current construct to which the options belong + 1 and in increment of 1. For example, if the current non-behavioral construct is assigned a weight of 60, then the least important ranked option within this construct (which would have the highest rank) will get a weight of 61, the next least important ranked option will get a weight of 62, and so on.

    c. Give all other unranked options within the current construct the weight assigned to the construct to which the unranked options belong. For example, if a construct is assigned a weight of 60, then all unranked options within this construct will get a weight of 60.

12.5.3 An Algorithm to Assign Weights to Inputs

The algorithm to assign weights to inputs uses the ranking scheme of inputs that was discussed in Section 6.1.3.1.

1. Assign a -1 to all forbidden inputs. This is only a flag to tell the specification matcher to reject components that require such inputs.
2. Take only the "I rather not supply" kind of inputs and order them in descending order of criticality.
3. Assign negative weights to the ordered “I rather not supply” inputs starting from -10 and in increment of -10. This means that the least problematic input (i.e., the one with the highest criticality number) will be assigned a weight of -10, the next least problematic input will get a -20, and so on.
4. For all other tolerated (or “don’t care about” kind of) inputs, assign them a weight of 0.

12.5.4 An Algorithm to Assign Weights to Outputs

The algorithm to assign weights to outputs uses the ranking scheme of outputs that was discussed in Section 6.1.3.2.

1. Assign a -1 to all "must have" outputs. This is only a flag to tell the specification matcher to reject components that *do not generate* such outputs.
2. Assign a -1 to all forbidden outputs. This is only a flag to tell the specification matcher to reject components that *generate* such outputs.

3. Take only the important outputs (i.e., outputs marked with "+" and a ranking number) and order them in descending order of their importance.

4. Assign negative weights to the ordered important outputs starting from 100 and in increment of 100. This means that the least important output will be assigned a weight of 100, the next least important output will get a 200, and so on.

5. Next, take only the optional outputs and order them in descending order of their importance.

6. Assign positive weights to the ordered optional outputs starting from 10 and in increment of 10. This means that the least important optional output will be assigned a weight of 10, the next least important optional output will get a 20, and so on.

12.5.5 An Algorithm to Evaluate Non-Behavioral Constructs

The algorithm to evaluate non-behavioral constructs is an expansion to the one that was outlined in Section 12.5.1. The algorithm uses weights assigned to non-behavioral constructs (as discussed in Section 12.5.2.) The algorithm takes each weighted non-behavioral construct in the wrapper specification and tries to find a match for it in each component specification. If it finds a match, it adds the weight assigned to the wrapper's construct or to its option (if the wrapper has options) to the total accumulated weight of the candidate component. For multi-option constructs in both the wrapper

specification and the component specifications, the algorithm adds the weights of *all* options that match in the same construct. This gives components that offer more options an added advantage over those that offer fewer ones.

```
for each candidate component
{
  set the component's weight to 0


  // Evaluate suppliers

  for each option in wrapper.suppliers
  {
    if component.provider.name matches option then
    {
      add weight of option to the component's weight
      break;
    }
    else if component.provider.URL matches option then
    {
      add weight of option to the component's weight
      break;
    }
  }


  // Evaluate technology

  for each option in wrapper.technologies
  {
    if component.technology matches option then
    {
      add weight of option to the component's weight
      break;
    }
  }


  // Evaluate operating systems

  for each option1 in wrapper.operating_systems
  {
    for each option2 in component.operating_systems
    {
      if option1 matches option2 then
      {
        add weight of option1 to the component's weight
        break;
```

```
      }
    }
  }


  // Evaluate databases

  for each option1 in wrapper.databases
  {
    for each option2 in component.databases
    {
      if option1 matches option2 then
      {
        add weight of option1 to the component's weight
        break;
      }
    }
  }


  // Evaluate language

  for each option in wrapper.languages
  {
    if component.language matches option then
    {
      add weight of option to the component's weight
      break;
    }
  }


  // Evaluate formats

  for each option in wrapper.formats
  {
    for each option2 in component.formats
    {
      if option1 matches option2 then
      {
        add weight of option1 to the component's weight
        break;
      }
    }
  }


  // Evaluate license models

  for each option1 in wrapper.license_models
  {
    for each option2 in component.license_models
    {
```

```
      if option1 matches option2 then
      {
        add weight of option1 to the component's weight
        break;
      }
    }
  }


  // Evaluate license types

  for each option1 in wrapper.license_types
  {
    for each option2 in component.license_types
    {
      if option1 matches option2 then
      {
        add weight of option1 to the component's weight
        break;
      }
    }
  }


  // Evaluate costs

  // ToDollars will convert all costs from their original currencies
  // to US dollars to make costs comparable
  // Exists will return true if the given construct exists in the
  // given specification; otherwise, it returns false.
  // <name> is used to refer to the cost name.  (For example,
  // Single.LicenseCost.)  This is to differentiate it from the
  // wrapper's or component's name.  For example,
  // wrapper.name refers to the wrapper's name while wrapper.<name>
  // refers to the value assigned to the specific cost named <name>
  // that exists in the wrapper specification.

  for prefix in ('single', 'developer', 'site')
  {
    for cost_name in ('License', 'Upgrade', 'Maintenance', 'Support',
                      'Training', 'Installation')
    {

      // Form the cost name
      name = prefix + "." + cost_name + "Cost";

      if Exists(wrapper, <name>) and Exists(component, <name>) and
         ToDollars(component.<name>) <= ToDollars(wrapper.<name>) then
      {
        add weight of wrapper.<name> to the component's weight
      }
      else if Exists(wrapper, <name>) and
              NOT Exists(component, <name>) then
```

```
      {
        ignore
      }
      else if NOT Exists(wrapper, <name>) and
                  Exists(component, <name>) then
      {  // Don't know what to do with this cost
        inform system integrator
      }
    }
  }


  // Evaluate software constraints.

  // Since there is no way to check if such constraints are accepted
  // to system integrators, the algorithm reports such constraints
  // to system integrators and let them decide.

  if Exists(component, "software_constraints.program") then
  {
    inform system integrator
  }
  if Exists(component, "software_constraints.notpresent") then
  {
    inform system integrator
  }
  if Exists(component, "software_constraints.notloaded") then
  {
    inform system integrator
  }
  if Exists(component, "software_constraints.other") then
  {
    inform system integrator
  }


  // Evaluate hardware constraints

  if Exists(wrapper, "hardware_constraints.RAM") and
     Exists(component, "hardware_constraints.RAM") and
     component.hardware_constraints.RAM <=
       wrapper.hardware_constraints.RAM then
  {
    add weight of wrapper.hardware_constraints.RAM to the
       component's weight
  }
  else if Exists(wrapper, "hardware_constraints.RAM") and
          NOT Exists(component, "hardware_constraints.RAM") then
  {
    ignore
  }
  else if NOT Exists(wrapper, "hardware_constraints.RAM") and
          Exists(component, "hardware_constraints.RAM") then
```

```
{
  inform system integrator
}

if Exists(wrapper, "hardware_constraints.storage") and
   Exists(component, "hardware_constraints.storage") and
   component.hardware_constraints.storage <=
     wrapper.hardware_constraints.storage then
{
  add weight of wrapper.hardware_constraints.storage to the
     component's weight
}
else if Exists(wrapper, "hardware_constraints.storage") and
        NOT Exists(component, "hardware_constraints.storage") then
{
  ignore
}
else if NOT Exists(wrapper, "hardware_constraints.storage") and
            Exists(component, "hardware_constraints.storage") then
{
  inform system integrator
}

if Exists(component, "hardware_constraints.other") then
{
  inform system integrator
}


// Evaluate network constraints

if Exists(component, "network_constraints.other") then
{
  inform system integrator
}


// Evaluate support methods\

for each option1 in wrapper.support_methods
{
  for each option2 in component.support_methods
  {
    if option1 matches option2 then
    {
      add weight of option1 to the component's weight
      break;
    }
  }
}


// Evaluate payment methods
```

```
  for each option1 in wrapper.payment_methods
  {
    for each option2 in component.provider.payment_methods
    {
      if option1 matches option2 then
      {
        add weight of option1 to the component's weight
        break;
      }
    }
  }


  // Check existence of requires clauses

  if Exists(component, "requires") then
  {
    inform system integrator
  }
}
```

The following utility checks for violations that can be detected in non-behavioral constructs between the wrapper specification and the component specifications.

```
NonBehavioralConstraintsViolated (wrapper, comp) {
  violated = false;

  // <name> is used to refer to the cost name.  (For example,
  // Single.LicenseCost.)  This is to differentiate it from the
  // wrapper's or component's name.  For example,
  // wrapper.name refers to the wrapper's name while wrapper.<name>
  // refers to the value assigned to the specific cost named <name>
  // that exists in the wrapper specification.

  for prefix in ('single', 'developer', 'site')
  {
    for cost_name in ('License', 'Upgrade', 'Maintenance', 'Support',
          'Training', 'Installation')
    {
      // Form the cost name
      name = prefix + "." + cost_name + "Cost"

      if (comp.<name> > wrapper.<name>) then {
        violated = true;
      }
    }
  }
```

```
  if (comp.min_RAM > wrapper.max_RAM ||
      comp.min_storage > wrapper.max_storage) {
    violated = true;
  }

  if (not InList (GetVendorName (comp, vendor_ID),
                  wrapper.preferred_suppliers) ||
      not InList (GetVendorURL (comp, vendor_ID),
                  wrapper.preferred_suppliers) ||
      not InList (comp.technology, wrapper.preferred_technologies) ||
      not InList (comp.operating_systems,
                  wrapper.preferred_operating_systems) ||
      not InList (comp.language, wrapper.preferred_languages) ||
      not InList (comp.formats, wrapper.preferred_formats) ||
      not InList (GetVendorType (comp, vendor_ID),
                  wrapper.supplier_types) ||
      not InList (comp.license_methods,
                  wrapper.preferred_licensing_models) ||
      not InList (GetVendorPaymentMethod(comp, vendor_ID),
                  wrapper.preferred_payment_methods)) {
    violated = true;
  }

  return (violated);
}

InList (list1, list2) {
  for (i = 0; i < size (list1); i++) {
    for (j = 0; j < size (list2); j++) {
      if (list2 [j] == list1 [i]) {
      return (true);
    }
  }
  return (false);
}
```

12.5.6 An Algorithm to Evaluate Inputs

The following algorithm uses the weights assigned to inputs in Section 12.5.3 to find matching components from the candidate ones. The algorithm uses a ***Match*** list to store matches between the wrapper's input specifications and the components' input specifications. Additionally, a ***NoMatch*** list will hold the wrapper specification inputs

that have no matching inputs in the component specification. Components that violate the wrapper's input specifications are added to the ***Rejected*** list. This list can later be presented to the system integrator to make a deeper investigation on the rejected components and to have a final decision on whether or not to include them in the generated compositional wrapper. Compatibility check between two inputs is discussed in Section 12.5.8. Since all inputs, other than *forbidden* ones, are assigned negative values or zero for *don't care about* inputs, this means that component inputs will decrease component weights or, at best, does not add credit to them.

```
NONEFound = false;

for each wrapper.input
{
  if wrapper.input <> NONE then
  {
    for each candidate component
    {
      found = false;
      for each component.input
      {
        if component.input is compatible with wrapper.input then
        {
          found = true;
          AddElement (Match, "input", wrapper.input, component,
                      component.input);
          break;
        }
      }
      if found and weight of wrapper.input == -1 then
      {  // Forbidden input, so reject the component
        add component to the Rejected list
      }
      else if found then
      {  // all other inputs
        add weight of wrapper.input to component's weight
      }
      else
      {  // Not found
        AddElement (NoMatch, "input", wrapper.input, component);
      }
    }
```

```
   }
   else
   {  // wrapper.input == the keyword NONE
     NONEFound = true;
   }
}

// Check if components require extra inputs other than those stated
// in the wrapper's specification.  If they do and if the wrapper's
// specification states that no other input would be tolerated, then
// reject the component.

if NONEFound then
{
   for each candidate component
   {
     found = false;  // looking for incompatible inputs
     for each component.input
     {
       if component.input is in Match (input, -, component,
             component.input) then
       {
         found = true;
         break;
       }
     }
     if not found then
     {
       add component to the Rejected list
     }
   }
}
```

12.5.7 An Algorithm to Evaluate Outputs

Similar to evaluating inputs, this algorithm will reject components that have *forbidden* outputs, have missing *must have* outputs, or include extra outputs that do not match any of the ones specified in the wrapper's specification and the wrapper's specification states that no extra outputs are accepted. Absence of important outputs will result in subtracting large amounts from the components' total weights, which will greatly low-

er the component's chance of being selected. Section 12.5.9 explains how to check for compatibility of two outputs.

```
NONEFound = false;

for each wrapper.output
{
  if wrapper.output <> NONE then
  {
    for each candidate component
    {
      found = false;
      for each component.output
      {
        if component.output is compatible with wrapper.output then
        {
          found = true;
          AddElement (Match "output", wrapper.output, component,
                      component.output);
          break;
        }
      }
      if not found and weight of wrapper.output == 1 then
      {  // "Must have" output
        add component to the Rejected list
      }
      else if found and weight of wrapper.output == -1 then
      {  // Forbidden output
        add component to the Rejected list
      }
      else if not found and wrapper.output is important then
      {
        substract weight of wrapper.output from the component's
            total weight
        AddElement(NoMatch, "output", wrapper.output, component);
      }
      else
      {  // found optional or important output
        add weight of wrapper.output to component's weight
      }
    }
  }
  else
  {  // wrapper.output == the keyword NONE
    NONEFound = true;
  }
}

// Check if components generate extra outputs other than that stated
```

```
// in the wrapper's specification.  If they do and if the wrapper's
// specification stated that no other output would be tolerated,
// then reject the component

if NONEFound then
{
  for each candidate component
  {
    found = false;
    for each component.output
    {
      if component.output is in Match (output, -, component,
            component.output) then
      {
        found = true;
        break;
      }
    }
    if not found then
    {
      add component to the Rejected list
    }
  }
}
```

12.5.8 An Algorithm to Check the Compatibility of Two Inputs

This section presents a high-level approach to check for most of the input compatibilities. The algorithm is divided into several cases to ease discussing its various parts. The cases that require extensive logic to check for their compatibility are not included in the algorithm. The reason for leaving out such cases is to make the algorithm simple enough so readers can understand its logic and enhance it as needed. When implementing the algorithm, the pseudo code fragments in all cases should be taken as one unit.

1. The WSL and CSL specification languages define inputs to have the following format:

   ```
   input <InputName>( <Rank> ) = {
   ```

```
  <Comment>, <Type>, <Size>, <ValidData>, <Source>
};
```

2. Assuming two inputs, ***input1*** and ***input2***, are to be compared:

   Case 1: The two inputs match completely

```
if input1.<Type> == input2.<Type> and
   input1.<Size> == input2.<Size> and
   input1.<ValidData> == input2.<ValidData> and
   input1.<Source> == input2.<Source> then
{ // complete match
  compatible = true
}
```

   Case 2: In this case, the two inputs match in every part except that they come from different sources. The list of input sources that are supported by the WSL and CSL languages are defined in Section 6.1.3. The algorithm considers any two inputs that come from diffident sources as incompatible.

```
else if input1.<Type> == input2.<Type> and
        input1.<Size> == input2.<Size> and
        input1.<ValidData> == input2.<ValidData> and
        input1.<Source> <> input2.<Source> then
{
  compatible = false;
}
```

   Case 3: This case deals with inputs that match in all but the `<ValidData>` part. Since both of the component and the wrapper specification languages allow the use of the `any` keyword to mean no restriction on the input data, the presence of the `any` keyword in any of the two inputs is considered to be a match of the input data of one input to that of the other input.

```
else if input1.<Type> == input2.<Type> and
        input1.<Size> == input2.<Size> and
        input1.<Source> == input2.<Source> and
        (input1.<ValidData> == 'any' or
```

```
          input2.<ValidData> == 'any') then
{
  compatible = true;
}
```

Case 4: In this case, the two inputs match in every part except the <ValidData> part. In many programming languages, integers can also be taken as floats. Defining one input's valid data as float and the other input's valid data as integer implies compatibility of the two inputs.

```
else if input1.<Type> == input2.<Type> and
        input1.<Size> == input2.<Size> and
        input1.<Source> == input2.<Source> and
        ((IsInteger(input1.<ValidData>) and
          IsFloat(input2.<ValidData>)) or
         (IsFloat(input1.<ValidData>) and
          IsInteger(input2.<ValidData>))) then
{  // Integers and floats are compatible
  compatible = true;
}
```

Case 5: This case assumes matching of all parts of the two inputs except in their <ValidData> part. Since the valid data can be specified as a range, this case looks for one input whose set of valid data is enclosed within the set of valid data of the other input. LowerRangeValue returns the lower value of a range and, similarly, UpperRangeValue returns the upper value of a range. For valid data that is not in a range form, LowerRangeValue and UpperRangeValue return the valid data itself.

```
else if input1.<Type> == input2.<Type> and
        input1.<Size> == input2.<Size> and
        input1.<Source> == input2.<Source> and
        ((IsRange(input1.<ValidData>) and
          LowerRangeValue(input2.<ValidData>) >=
          LowerRangeValue(input1.<ValidData>) and
          UpperRangeValue(input2.<ValidData>) <=
```

```
           UpperRangeValue(input1.<ValidData>)) or
          (IsRange(input2.<ValidData> ) and
           LowerRangeValue(input1.<ValidData>) >=
           LowerRangeValue(input2.<ValidData>) and
           UpperRangeValue(input1.<ValidData>) <=
           UpperRangeValue(input2.<ValidData>))) then
{  // One range within the other
  compatible = true;
}
```

Case 6: In this case, the two inputs match in all parts except the input size. However, if the size of one input were tagged with the `any` keyword, then the two sizes would be considered compatible. Consequently, the two inputs would be considered compatible.

```
else if input1.<Type> == input2.<Type> and
        input1.<ValidData> == input2.<ValidData> and
        input1.<Source> == input2.<Source> and
        (input1.<Size> == 'any' or input2.<Size> == 'any') then
{
  compatible = true;
}
```

Case 7: This case is similar to Case 6. However, in this case the size of one input is smaller than that of the other input. If it were known that the smaller of the two inputs is a component input, then the two inputs would be considered compatible. However, the converse is not true. This is because component users will usually not object on supplying shorter inputs. However, since it is not known which of the two inputs is a component input, the inputs in this case are tagged as incompatible until their origin can be proven.

```
else if input1.<Type> == input2.<Type> and
        input1.<ValidData> == input2.<ValidData> and
        input1.<Source> == input2.<Source> and
        input1.<Size> <> input2.<Size> then
{
  compatible = false;
```

```
}
```

Case 8: This case considers two inputs that match in all parts except the input types. However, since both `Object` and `Reference` types will hold pointers to actual objects, inputs with these two types are considered compatible.

```
else if input1.<Size> == input2.<Size> and
        input1.<ValidData> == input2.<ValidData> and
        input1.<Source> == input2.<Source> and
        ((IsObject(input1.<Type>) and
            IsReference(input2.<Type>)) or
         (IsObject(input2.<Type>) and
            IsReference(input1.<Type>))) then
{
  compatible = true;
}
```

Case 9: This last case asserts that integer and float types are compatible. Therefore, any inputs that match in all other parts and have integers or floats for their types are considered compatible.

```
else if input1.<Size> == input2.<Size> and
        input1.<ValidData> == input2.<ValidData> and
        input1.<Source> == input2.<Source> and
        ((IsInteger(input1.<Type> ) and
            IsFloat(input2.<Type>)) or
         (IsFloat(input2.<Type> ) and
            IsInteger(input1.<Type>))) then
{
  compatible = true;
}
else
{ // For all other cases
  compatible = false;
}
```

12.5.9 An Algorithm to Check the Compatibility of Two Outputs

This section takes similar approach to that of Section 12.5.8. The presented cases follow similar logic for checking outputs compatibility. An explanation is provided only for Case 7. Any explanation needed for the other cases can be found in the corresponding case in Section 12.5.8.

1. Outputs in WSL and CSL specification languages are defined to have the following format:

```
output <OutputName>( <Rank> ) = {
  <Comment>, <Type>, <Size>, <ValidData>, <Destination> };
```

2. Assuming two outputs, ***output1*** and ***output2***, are to be compared:

```
if output1.<Type> == output2.<Type> and
   output1.<Size> == output2.<Size> and
   output1.<ValidData> == output2.<ValidData> and
   output1.<Destination> == output2.<Destination> then
{ // complete match
  compatible = true;
}
else if output1.<Type> == output2.<Type> and
        output1.<Size> == output2.<Size> and
        output1.<ValidData> == output2.<ValidData> and
        output1.<Destination> <> output2.<Destination> then
{
  compatible = false;
}
else if output1.<Type> == output2.<Type> and
        output1.<Size> == output2.<Size> and
        output1.<Destination> == output2.<Destination> and
        (output1.<ValidData> == 'any' or
            output2.<ValidData> == 'any') then
{
  compatible = true;
}
else if output1.<Type> == output2.<Type> and
        output1.<Size> == output2.<Size> and
        output1.<Destination> == output2.<Destination> and
        ((IsInteger(output1.<ValidData>) and
          IsFloat(output2.<ValidData>)) or
         (IsFloat(output1.<ValidData>) and
          IsInteger(output2.<ValidData>))) then
{  // Integers and floats are compatible
  compatible = true;
}
else if output1.<Type> == output2.<Type> and
```

```
        output1.<Size> == output2.<Size> and
        output1.<Destination> == output2.<Destination> and
        ((IsRange(output1.<ValidData>) and
          LowerRangeValue(output2.<ValidData>) >=
          LowerRangeValue(output1.<ValidData>) and
          UpperRangeValue(output2.<ValidData>) <=
          UpperRangeValue(output1.<ValidData>)) or
         (IsRange(output2.<ValidData> ) and
          LowerRangeValue(output1.<ValidData>) >=
          LowerRangeValue(output2.<ValidData>) and
          UpperRangeValue(output1.<ValidData>) <=
          UpperRangeValue(output2.<ValidData>))) then
{  // One range within the other
  compatible = true;
}
else if output1.<Type> == output2.<Type> and
        output1.<ValidData> == output2.<ValidData> and
        output1.<Destination> == output2.<Destination> and
        (output1.<Size> == 'any' or
            output2.<Size> == 'any') then
{
  compatible = true;
}
```

The following case (the first "else if" statement) is similar to Case 7 in Section 12.5.8. While wrapper users might not object if asked to supply a shorter stream of input than that specified in the wrapper's specification, it would not be the same case for outputs. In most situations, shorter or longer outputs are not desired. Therefore, unless one can know the exact nature of the outputs, it is not possible to assure compatibility of outputs that do not match in size.

```
else if output1.<Type> == output2.<Type> and
        output1.<ValidData> == output2.<ValidData> and
        output1.<Destination> == output2.<Destination> and
        output1.<Size> <> output2.<Size> then
{
  compatible = false;
}
else if output1.<Size> == output2.<Size> and
        output1.<ValidData> == output2.<ValidData> and
        output1.<Destination> == output2.<Destination> and
        ((IsObject(output1.<Type>) and
          IsReference(output2.<Type>)) or
         (IsObject(output2.<Type>) and
```

```
          IsReference(output1.<Type>))) then
{
  compatible = true;
}
else if output1.<Size> == output2.<Size> and
        output1.<ValidData> == output2.<ValidData> and
        output1.<Destination> == output2.<Destination> and
        ((IsInteger(output1.<Type>) and
            IsFloat(output2.<Type>)) or
         (IsFloat(output2.<Type>) and
            IsInteger(output1.<Type>))) then
{
  compatible = true;
}
else  // For all other cases
{
  compatible = false;
}
```

12.5.10 An Algorithm to Check the Compatibility of Two Assertions

This section presents an algorithm to check for compatibility of two assertions. Assertions, as defined in the WSL and CSL specification languages of this dissertation, can be of various forms. First, there are `return` assertions, which state that a returned output should be of a certain type defined by an `output` statement. Assertions can also be in the form of function calls. Here, only the case of *dummy* functions is considered since the assumption is that *inline* functions have already been converted to assertions by the algorithm of Section 12.5.11. Dummy functions are placeholders for operations performed by servers, agents, databases, etc. or performed on lists, stacks, queues, etc. The third kind of assertions deals with conditions. There are also assertions in the form of assignment statements, if-statements, and loops. These latter assertions are not consid-

ered in this section because of the extensive checks that would be needed to prove or disprove their compatibility.

Assume two assertions, ***assertion1*** and ***assertion2***, are to be checked for compatibility.

Case 1: This case considers `return` assertions that have compatible outputs as discussed in Section 12.5.9.

```
if assertion1.type == 'return' and
   assertion2.type == 'return' and
   assertion1.output_type is compatible with assertion2.output_type
then
{
  compatible = true;
}
```

Case 2: This case considers dummy function calls. Two dummy function calls are considered compatible if they are made to the same function and the arguments passed to them are compatible in terms of their order, type, and data sizes.

```
else if assertion1.type == 'function_call' and
        assertion2.type == 'function_call' and
        assertion1.function_name == assertion2.function_name and
        assertion1.arguments are compatible with assertion2.arguments
then
{
  compatible  = true;
}
```

Case 3: This case checks compatibility of two conditions. Conditions compatibility, though not discussed here, would consider the nature of the elements compared and the operators used in each condition.

```
else if assertion1.type == 'condition' and
        assertion2.type == 'condition' and
        assertion1.condition is compatible with assertion2.condition
then
```

```
{
  compatible  = true;
}
else  // For all other cases
{
  compatible  = false;
}
```

12.5.11 An Algorithm to Check the Compatibility of Two Interfaces

This section presents a general algorithm to check for behavior compatibility of two interfaces: ***interface1*** and ***interface2***. Section 12.5.10 provided an algorithm to check for compatibility of two assertions. That section also defined what assertions mean in the scope of the specification languages defined in this dissertation.

1. Convert inline functions (those that are defined within the specification of each interface) to assertions and replace calls to each inline function with the converted set of assertions for that function. This way, inline functions are taken as macros, similar to that supported by the C and C++ languages.
2. Build the list of assertions defined in the specification of ***interface1*** using the following algorithm (call this list ***List1***):

```
for each behavior section in interface1
{
  for each call statement in behavior
  {
    for each assertion in call
    {
      add assertion to List1
    }
  }
}
```

3. Build the list of assertions defined in the specification of ***interface2*** using the following algorithm (call this list ***List2***):

```
for each behavior section in interface2
{
  for each call statement in behavior
  {
    for each assertion in call
    {
      add assertion to List2
    }
  }
}
```

4. Use the collected lists of assertions of the two interfaces to check for compatibility between the two interfaces:

```
compatible = true;

for each assertion in List1 (call it assertion1)
{
  for each assertion in List2 (call it assertion2)
  {
    if assertion1 is not compatible with assertion2 then
    {
      compatible = false;
      break;
    }
  }
  if not compatible then
  {
    break;
  }
}
```

The following utility implements the above general algorithm and expands on it. The utility considers an interface to implement one or more services. Each service could have parameters.

```
MatchedInterfaces (p, q) {
  if (p <> nil && q <> nil &&
      ServicesCount (q) >= ServicesCount (p)) {
```

```
    s1 = p.services;
    matching = true;
    while (s1 <> nil && matching) {
      s2 = q.services;
      matching = false;
      while (s2 <> nil && not matching) {
        if (MatchedService (s1, s2)) {
          matching = true;
        }
        else {
          s2 = s2.next;
        }
      }
      if (matching) {
        s1 = s1.next;
      }
    }
    if (s1 == nil  && matching) {
      return (true);
    }
    else {
      return (false);
    }
  }
  else {
    return (false);
  }
}

MatchedService (s1, s2) {
  If (ParamsCount (s1) <> ParamsCount (s2) ||
       s1.return_type <> s2.return_type || s1.size > s2.size) {
    return (false);
  }
  else {
    a1 = s1.params;
    a2 = s2.params;
    while (a1 <> nil) {
      if (not MatchedParam (a1, a2)) {
        return (false);
      }
      a1 = a1.next;
      a2 = a2.next;
    }
    return (true);
  }
}

MatchedParam (a1, a2) {
  if (a1.type <> a2.type) {
    return (false);
  }
  if ((a1.inout == IN && a2.inout == OUT) ||
```

```
        (a1.inout == OUT && a2.inout == IN) ||
        (a1.inout == INOUT && a2.inout == IN) ||
        (a1.inout == INOUT && a2.inout == OUT)) {
    return (false);
  }
  if (a1.size > a2.size) {
    return (false);
  }
  return (true);
}

ServicesCount(p) {
  cnt = 0;
  s = p.services;
  while (s <> nil) {
    cnt++;
    p = p.next;
  }
  return (cnt);
}

ParamsCount(p) {
  cnt = 0;
  s = p.params;
  while (s <> nil) {
    cnt++;
    p = p.next;
  }
  return (cnt);
}
```

12.5.12 An Algorithm to Evaluate Behavioral Specifications

Section 12.5.10 presented an algorithm to check for compatibility of two assertions. Section 12.5.11 went a step further to check for compatibility of two interfaces. This section widens the scope to check for behavioral compatibility between the wrapper's specification and the specification of each candidate component.

```
for each wrapper.interface
{
  for each candidate component
```

```
  {
    found = false;
    for each component.interface
    {
      if component.interface is compatible with wrapper.interface then
      {
        found = true;
        AddElement (Match, "behavior", wrapper.interface.name,
                    component, component.interface.name);
        break;
      }
    }
    if not found then
    {
      AddElement (NoMatch "behavior", wrapper.interface.name,
                  component.name);
    }
  }
}
```

12.5.13 An Algorithm to Select Best-Fit Components

This section utilizes all the collected statistics and lists from the previous sections to select the best-fit components. The components will be added to the ***BestFit*** list in descending order of their closeness to the wrapper specification. This ***BestFit*** list will be used by the algorithm of Section 12.7.1 to match interfaces required by the wrapper generator and its components with the interfaces provided in the encased components.

1. Form a temporary list that contains all candidate components minus the ones in the ***Rejected*** list.
2. From the temporary list, select the components with no entries in the ***NoMatch*** list even if such components have lower weights. Add these components to the ***BestFit*** list. These components will be the best candidates because they define

only the required inputs and outputs and have compatible interface behaviors. (Most of the time, this step will yield no components due to its strict requirement.)

3. Next, select from the temporary list of candidate components the ones with no *output* or *behavior* entries in the ***NoMatch*** list even if such components have *input* entries in the ***NoMatch*** list. This says that supplying additional inputs can be tolerated more than having missing outputs or wrong behaviors. Add the selected components to the end of the ***BestFit*** list and in descending order of their weights. This way, components with higher weights will get better chance of being selected. Components in this category can be considered for inclusion in the wrapper. However, the system integrator must approve their inclusion.
4. Finally, from the remaining components in the temporary list, select the ones with no *behavior* entries in the ***NoMatch*** list. Add these components to the end of the ***BestFit*** list and in descending order of their weights. Components in this category might not work at all because they do not deliver all of the outputs that the user expects. Such components might require intensive review by the user. However, the user can decide if to include them or not.

The following snippet of code implements the best-fit selection algorithm outlined in this section:

```
BestFitSelection (wrapper) {
  p = wrapper.found_components;
  while (p <> nil) {
    check1 = MatchedFullFunctionality (wrapper, p);
    check2 = MatchedPartialFunctionality (wrapper, p);
```

```
    check3 = not NonBehavioralConstraintsViolated (wrapper, p);

    if (check1 && check3) {
      AddComponent (wrapper.matching_components, p);
    }
    else if (check2 && check3) {
      AddComponent (wrapper.candidate_components, p);
    }
  }
}

MatchedFullFunctionality (wrapper, p) {
  return (ImplementsAllInterfaces (wrapper, p));
}

ImplementsAllInterfaces (wrapper, p) {
  interface = wrapper.provided_interfaces;
  matching = true;
  while (interface <> nil && matching) {
    matching = false;
    q = p.provided_interfaces;
    while (q <> nil && not matching) {
      if (MatchedInterfaces (interface, q)) {
        matching = true;
      }
      else {
        q = q.next;
      }
    }
    if (matching) {
      interface = interface.next;
    }
    else {
      return (false);
    }
  }
  return (matching);
}

MatchedPartialFunctionality (wrapper, p) {
// Checks if the found component implements at least one interface

  interface = wrapper.provided_interfaces;
  matching = false;
  while (interface <> nil && not matching) {
    q = p.provided_interfaces;
    while (q <> nil && not matching) {
      if (MatchedInterfaces (interface, q)) {
        matching = true;
      }
      q = q.next;
    }
    interface = interface.next;
```

```
    }
    return (matching);
}
```

12.6 Groups Evaluation

Groups (also called *ensembles* [THOM00b]) are collections of components that act together to achieve some desired system capability. Group or ensemble evaluations are sometimes referred to as head-to-head tests [MOSK00]. In forming component groups, it may be discovered that less-than-adequate components must be added to a group to ensure the group's compatibility with the user's selection [SEAC01a].

The group evaluation step requires looking at some aspects of all the components collected so far, especially the non-functional properties. More simple decisions (e.g. to reject candidate components that are not compliant with simple requirements, such as cost) are made earlier in the process when there are more candidates. More complex decisions (e.g. to reject software candidates for noncompliance with complex customer requirements, such as system performance) are made in this step. Some parts of the process, such as evaluation of the non-functional properties can be automated using the Analytic Hierarchy Process (AHP) discussed in Section 12.3.5.2. This part of the group evaluation can be fully or partially automated. Other parts, such as crosschecking the component and vendor evaluation results with the organization's standards and strategies, can only be done by the development team.

The key point in a group evaluation is that the evaluation activity is focused on the group as a cohesive unit. This has important implications on the evaluation activity that are not addressed by current evaluation practice. There are at least two kinds of evaluations taking place, each involving collections of components rather than individual components. In the first kind of evaluation, component alternatives to implement different groups are evaluated. For this kind of evaluations, the wrapper generator can create "bins." The wrapper generator can either put in the same bin the components that have similar features (behavioral or non-behavioral) or it can put in the same bin the components whose total weight (as discussed in Section 12.5) falls within a certain range. In the second kind of evaluation, alternative groups are evaluated. [NCUB00] proposed using modeling techniques to evaluate and predict the reliability of different groups of components. Additionally, [NCUB00] suggested using statistical techniques, such as Bayes Theorem, to aid in drawing conclusions about the best groups of components that work best for the user needs. The wrapper generator will present each of these groups to the system integrator along with the missing features of each group (as was explained in Section 12.4.) The system integrator will have the final say on selecting the appropriate group.

If the system integrator is satisfied with any set of components evaluated in this step, the wrapper generator can continue to the component acquisition phase. Otherwise, the generator has to go back to previous steps or even to previous stages in the hope of finding better components. The system integrator's decision on which component or group of components are suited for the developed wrapper could be based on political,

organizational, or other accounts. Note that the selected components might not deliver all the functional requirements specified in the wrapper specification. However, this is the responsibility of the development team.

The following code snippet illustrates checking the complete group of candidate components that passed the detailed evaluations for violations of the constraints present in the wrapper generator's internal structure (see Section 4.1.)

```
GroupConstraintsViolated (wrapper) {
  if (ComponentsCount (wrapper.candidate_components) >
       wrapper.max_components) {
    return (true);
  }
  p = wrapper.candidate_components;
  violated = false;
  t1 = 0;
  t2 = 0;
  while (p <> nil && not violated) {
    cnt1 = InterfacesCount (p.provided_interfaces);
    cnt2 = InterfacesCount (p.required_interfaces);
    if (cnt1 < wrapper.min_provided_interfaces_per_component ||
        cnt2 > wrapper.max_required_interfaces_per_component) {
      return (true);
    }
    t1 += cnt1;
    t2 += cnt2;
    p = p.next;
  }
  if (t1 < wrapper.min_total_provided_interfaces ||
      t2 > wrapper.max_total_required_interfaces) {
    return (true);
  }
  else {
    return (false);
  }

  if (wrapper.candidate_components <> nil) {
    rated_components =
       RateNonfunctionalProperties (wrapper.candidate_components);
    p = rated_components;
    selected1 = p;
    continue = true;

    while (p <> nil && continue) {
```

```
      selected2 = selected1;
      p = p.next;
      q = p;
      violated = false;
      while (q <> nil && not violated) {
        AddComponent (selected2, q);
        if (not GroupConstraintsViolated (selected2)) {
          q = q.next;
        }
        else {
          violated = true;
        }
      }
      if (not violated) {
        AddComponent (selected1, p);
        if (ImplementsAllInterfaces (wrapper, selected1)) {
          AddGroup (wrapper.candidate_groups, selected1);
          continue = false;
        }
      }
    }
  }
}

InterfacesCount (list) {
  p = list;
  cnt = 0;
  while (p <> nil) {
    cnt++;
    p = p.next;
  }
  return (cnt);
}

ComponentsCount (list) {
  p = list;
  cnt = 0;
  while (p <> nil) {
    cnt++;
    p = p.next;
  }
  return (cnt);
}
```

12.7 Calculating Additional Needed Components

After selecting the set of best-fit components to be included in the compositional wrapper, there could still be some required interfaces in the wrapper specification that cannot be matched by any of the provided interfaces in the selected components. In this situation, the development team (or the wrapper generator) will be forced to locate and acquire additional components that have the missing interfaces as part of their provided ones. Alternatively, the development team could develop the code necessary to implement such missing required interfaces, as explained in Section 10.4. However, since the goal is to automate as much of the composition process as possible, this latter option of developing new components would not be considered unless the development team is unable to find an off-the-shelf component that can provide the missing interfaces.

It is necessary for the wrapper generator to estimate the number of remaining needed components in each iteration of the process. Knowing such a number would also be beneficial in estimating the remaining effort and cost of the developed system, as well as estimating other attributes, such as the complexity and quality of the to-be-generated wrapper or of the complete system. In order for the wrapper generator to be able to estimate such a number, it must first try to match the interfaces required by the wrapper specification and the selected components with any of the interfaces provided in the selected components. Section 12.7.1 explains how this interface matching is performed. Section 12.7.2 explains how to estimate the number of components needed to satisfy the remaining unmatched interfaces.

12.7.1 Matching Provided and Required Interfaces

A required interface *P* matches a provided interface *Q* if the following conditions hold [EGYE00]:

- *P* and *Q* have identical parameter counts and types.
- Either both *P* and *Q* have a return value or neither does.
- The types of *P*'s parameters are subtypes of the types of *Q*'s parameters.
- The type of *P*'s return value is a super-type of the type of *Q*'s return value.
- *P*'s preconditions imply *Q*'s preconditions.
- *Q*'s post-conditions imply *P*'s post-conditions.

The following algorithm discusses how to match required and provided interfaces. At this point, information about all the selected components is stored in the ***BestFit*** list. The algorithm also requires access to the wrapper specification and the specifications of all components in the ***BestFit*** list.

1. Add all required interfaces specified in the wrapper specification to the ***Required*** list.
2. For every component in the ***BestFit*** list (call the specification of this component ***main***)
    a. Add all provided interfaces of ***main*** to the ***Provided*** list.

b. For each component identifier in the *requires* clause of ***comp*** (call this component identifier ***required_comp***), the wrapper generator has the following options:

   i. If ***required_comp*** were specified internally (i.e., within the current specification document), then re-invoke this algorithm starting from step **2.a** passing the specification of ***required_comp*** as ***main***.

   ii. If ***required_comp*** were specified externally (i.e., outside the current specification document), then the wrapper generator would need to retrieve the specification document of ***required_comp***, wherever it is, and to re-invoke this algorithm starting from step **2.a** passing the retrieved specification of ***required_comp*** as ***main***.

   iii. If ***required_comp*** were described only in narrative terms, then the wrapper generator can do nothing about locating and retrieving its specification. The system integrator would need to search for the specification of ***required_comp***, acquire it, and inform the wrapper generator of its location. Once the location of the ***required_comp***'s specification is at the disposition of the wrapper generator, the wrapper generator would retrieve the specification document of ***required_comp*** and re-invokes this algorithm starting from step **2.a** passing the retrieved specification of ***required_comp*** as ***main***. If the system integrator decided not to acquire the specification of ***required_comp***, then ***required_comp*** will be added to the ***Required*** list.

c. For every interface identifier in the *requires* clause of ***comp*** (call this interface identifier ***required_int***), the wrapper generator can do the following:

   i. If ***required_int*** were already added to the ***Required*** list, then the wrapper generator would need to do nothing.

   ii. If ***required_int*** were specified internally (i.e., within the current specification document), as part of the specification of ***main*** or the specification of other components, then the wrapper generator would only need to add ***required_int*** to the ***Required*** list.

   iii. If ***required_int*** were specified externally (i.e., outside the current specification document), as part of the specification of another component, then the wrapper generator would need to retrieve the specification document of that component and re-invoke this algorithm starting at step **2.a** with the new component specification as ***main***.

   iv. If ***required_int*** were described only in narrative terms, then the wrapper generator can do nothing about locating and retrieving the specification document in which ***required_int*** is fully specified. The system integrator will need to search for that specification, acquire it, and inform the wrapper generator of its location. Once the specification that includes ***required_int*** is at the disposition of the wrapper generator, the wrapper generator would re-invoke this algorithm starting from step **2.a** with that specification as ***main***. If the system integrator decided not to acquire the

component that contains ***required_int***, then ***required_int*** will be added to the ***Required*** list.

3. Once the wrapper generator resolves all references to provided and required interfaces, it then starts the process of matching required interfaces with provided ones. In the following algorithm, the ***Final*** list holds a partial list of the ***BestFit*** list. The ***Final*** list is initially empty. Components that match any required interface listed in the ***Required*** list are added to the ***Final*** list. This algorithm selects only the first provided interface that matches the currently checked required interface. If one needs to include all compatible interfaces to use them in load-balancing and/or emergency recovery (as will be discussed in Section 14.1), then major changes have to be done to this algorithm.

```
for each interface in the Required list (interface1)
{
  found = false;

  // First, consult the component specifications that are in
  // the Final list

  for each specification in the Final list (comp)
  {
    for each provided interface in comp (interface2)
    {
      apply the algorithm discussed in Section 12.5.11 to check
         the compatibility of interface1 to interface2
      if interface1 is compatible with interface2 then
      {
        if comp is not yet in the Final list then
        {
          add comp to the Final list
        }
        found = true;
        break;
      }
    }
    if found then
```

```
    {
      break;
    }
  }

  // If we did not find a matching interface in the component
  // specifications that are currently in the Final list, then
  // we search in the BestFit list

  if not found then
  {
    for each component specification in the BestFit list (comp)
    {
      for each provided interface in comp (interface2)
      {
        apply the algorithm discussed in Section 12.5.11 to
         check the compatibility of interface1 to interface2
        if interface1 is compatible with interface2 then
        {
          if comp is not yet in the Final list then
          {
            add comp to the Final list
          }
          found = true;
          break;
        }
      }
      if found then
      {
        break;
      }
    }
    if not found then
    {
      export interface1 to be a required interface in the
            generated wrapper
    }
  }
}
```

12.7.2 Estimating Additional Needed Components

There are situations when it becomes necessary to know the number of components that are needed to satisfy a set of interfaces in the wrapper specification. For example, detailed evaluations could have resulted in interfaces in the wrapper specification

that cannot be matched with any interface provided by the so-far selected components. In this situation, the algorithm of this section can be consulted to estimate the number of additional needed components. Section 16.2 also uses the algorithm presented in this section to estimate the total number of components that are needed to satisfy a complete wrapper specification.

Table 8: Global variables and loop counters used to refer to the various activities in the wrapper generation process

N	The total number of compositional wrappers in the CWB system.
x	The total number of components retrieved in a search process.
n	The total number of candidate components after applying the preliminary evaluation to the *x* retrieved components.
m	The total number of best-fit components in a single iteration after applying the detailed evaluation to the *n* candidate components.
t	The accumulated sum of *m* best-fit components in all iterations.
w	The total number of best-fit components after applying the group evaluation to the *t* accumulated components.

Table 8 shows some of the global variables and loop counters that will be used in the calculation of additional needed components. *Iteration* is defined as a complete round of the composition process starting from the formulation of a search request to the selection of best-fit components. Table 9 gives the details of doing the calculations of the remaining components.

Table 9: Calculation of the number of additional components that need to be obtained and evaluated to satisfy all of the interfaces required by the wrapper specification or by any of the selected best-fit components

GPI	The number of interfaces to be provided in the compositional wrapper. This will initially be the list of interfaces specified in the wrapper specification. However, fine-tuning of this parameter is possible since some interfaces could be split into more than one interface each of which contains fewer functions.
GRI	The maximum number of interfaces that can be required by the generated wrapper. Currently, there is no clause provided in the wrapper specification language to supply this attribute. Therefore, the system integrator would need to supply it. Alternatively, the wrapper generator can use a default value for it.
CPI	The number of interfaces provided by the component. This can be obtained directly from the component specification by counting the number of interface specification sections that are listed in the component specification.
CRI	The number of interfaces required by the component. This can be obtained from the *requires* clause in the component specification. However, the *requires* clause can include references to complete component specifications as well as to narrative descriptions of components and interfaces. Therefore, obtaining this attribute might require parsing of other referenced specifications and/or estimation of other interfaces that are described using the narrative form.
$TCPI = \sum_{i=1}^{t} (CPI)_i$	The total number of interfaces provided by all of the *t* selected best-fit components.
$TCRI = \sum_{i=1}^{t} (CRI)_i$	The total number of interfaces required by all of the *t* selected best-fit components.
TRI = TCRI + GPI	The number of interfaces that need a match. These would include all interfaces required by all of the *t* selected best-fit components and the set of interfaces that must be provided by the generated wrapper.

MPRI	This list holds any of the *TRI* interfaces that can be matched with any of the *TCPI* interfaces. The algorithm presented in Section 12.5.11 can be used to check for interface compatibility.
$PRI = TRI - MPRI$	The remaining number of interfaces that need to be provided in the compositional wrapper or are required by one of the t selected best-fit components, but cannot be matched with any interface provided in the best-fit components collected to this point.
$RI = PRI - GRI$	The remaining number of interfaces after deducting the maximum number of interfaces that can be tolerated by the generated wrapper. The remaining *RI* interfaces can only be serviced by acquiring additional components. If each component would provide at least one interface, *RI* would be the *minimum* number of additional components that would be needed. However, [BASI01] hypothesized that integrating c components involves potentially $c(c\text{-}1)/2$ interfaces. This equation can be used as a rough estimate of the remaining number of components, as can be seen from the calculation of *RC*.
$RC = c = \left\lceil \frac{1+\sqrt{1+8RI}}{2} \right\rceil$	The estimated number of additional components that must be found during the next iteration. This equation was obtained by solving for c in the equation $RI = c(c\text{-}1)/2$, as suggested by [BASI01]. Of course, each of these additional components might have its set of required interfaces that cannot be serviced by any of the so-far selected components. This means that more components might be needed and the process might repeat.

The left column in Table 9 lists the parameters and equations involved in the estimation process. The result of each parameter or equation is used as a parameter in the equations that follow it. The right column gives sufficient details on the nature and derivation of each parameter or equation. The last equation in the table produces *RC*, which holds an estimate of the remaining components that would need to be considered in the next iteration of the process.

After obtaining *RC*, it is expected that a new and complete iteration of the composition process will commence. The process will start by using information from the remaining unmatched required interfaces as the search criteria. The process will proceed from searching to preliminary evaluation to detailed evaluations. When the process reaches the detailed evaluation stage, then the new components that pass the detailed evaluation will be merged with the other selected components that passed detailed evaluations in previous iterations. If the algorithm of Section 12.7.1 results in required interfaces that cannot be matched, then the algorithm of Table 9 can be consulted to estimate the components that are still needed. If the system integrator decided to consider acquiring the additional *RC* estimated components, then a new iteration in the composition process can restart.

The system integrator will continue these iterations until all required interfaces are matched or until the system integrator decides to stop. The system integrator may also choose to abort the process completely in which case the system integrator would need to refine, or sometimes rewrite, the wrapper specification and start the complete process all over. The system integrator may also decide to complete the remaining stages of the composition process with whatever components evaluated and selected up to this point. In this case, all remaining unmatched interfaces will be exported as required interfaces in the generated compositional wrapper. This means that the generated wrapper could have less (and possibly different) functionality from that specified in the wrapper specification.

CHAPTER 13: COMPONENTS ACQUISITION AND ADAPTATION

Table 10 gives a detailed activity list for the components' acquisition and adaptation process. In addition, the table indicates possibilities for automation (at low, medium, and high level versus manual/no automation.) The footnotes to the table provide the justification for the automation provided (as in previous chapters, "G" indicates the automation level when a wrapper generator is used. "WG" indicates the automation level without a generator.)

Table 10: A detailed breakdown of the activities involved in the components' acquisition and adaptation stage

	Automation			
	Manual	**Low**	**Medium**	**High**
0.0 Develop and operate a compositional wrappers-based (CWB) system				
1.0 Design the CWB system (Chapter 9 and 9.5)				
2.0 Build the CWB system (Chapter 10 to Chapter 14)				
2.1 Collect components (Chapter 10 to Chapter 13)				
2.1.1 Procure components (Chapter 10 to Chapter 13)				
2.1.1.1 Search for components (Chapter 10)				
2.1.1.2 Evaluate components (Chapter 11 and Chapter 12)				
2.1.1.3 Acquire selected components				
2.1.1.3.1 Send reservation information (if one is made) to the component vendor		WG[21]		G[20]
2.1.1.3.2 Conduct any necessary price and license negotiations with	WG	G[52]		

						Automation			
						Manual	**Low**	**Medium**	**High**
					the component vendor				
					2.1.1.3.3 Submit personal information		WG[21]	G[52]	
					2.1.1.3.4 Submit payment		WG[21]	G[52]	
					2.1.1.3.5 Finalize the deal with the component vendor[65]	G, WG			
					2.1.1.3.6 Download the component	WG		G[66]	
					2.1.1.3.7 Obtain the component licenses	WG		G[52]	
				2.1.1.4 Install components					
					2.1.1.4.1 Consult documentation	G, WG			
					2.1.1.4.2 Conduct necessary training on installing the component	G, WG			
					2.1.1.4.3 Install the component	WG	G[67]		
					2.1.1.4.4 Test the installed component	WG	G[68]		
				2.1.1.5 Adapt components					
					2.1.1.5.1 Consult documentation	G, WG			
					2.1.1.5.2 Review the organization's standards and business rules	G, WG			
					2.1.1.5.3 Review the developed system's specifications and designs	G, WG			
					2.1.1.5.4 Prepare customization and testing plans	WG	G[69]		
					2.1.1.5.5 Conduct necessary training on adapting the component	G, WG			
					2.1.1.5.6 Perform required configuration	WG	G[70]		
					2.1.1.5.7 Test the configured component	WG	G[68]		
		2.2 Wrap components (Chapter 14)							
		2.3 Deliver compositional wrapper (Section 14.11)							
		2.4 Assemble components into the CWB system (Section 15.1)							
		2.5 Test the CWB system (Section 15.2)							
	3.0 Operate the CWB system (Chapter 15)								

[65] This could include signing agreement forms (online forms or paper forms), filling surveys or questionnaires, etc.

[66] Most of this activity can be automated by utilizing the *downloads* clause in the component specification.

[67] Though most of this activity can be automated by executing the installation steps outlined in the *installation* clause of each component specification, there could be some other work left for users to do.

[68] The wrapper generator can help in the testing process by wrapping the tested component and inject some of the tracking and testing smart features in the wrapper code.

[69] The wrapper generator can help by displaying the configuration steps outlined in the *configuration* clause in the component specification.

[70] Though most of this activity can be automated by executing the configuration instructions outlined in the *configuration* clause of each component specification, there could be some other work left for users to do.

Component acquisition (as defined by this dissertation) is the process of interacting with component suppliers (or with their software agents) for the purpose of obtaining one of their components. The acquisition process can involve passing to the component supplier any required information (personal or organizational); submitting required payments (for commercial components); downloading the component and any associated artifacts; and obtaining license keys that are necessary to use the component. The wrapper generator can fully or partially automate all of these steps if component suppliers offer APIs that would permit doing such tasks. Otherwise, all the steps involved in the acquisition process will need to be carried out manually by the development team.

The acquisition process will include the following tasks:

1. Contact the component vendor. Various methods of contact can be used depending on their support by the wrapper generator and the systems of component suppliers. These techniques will include email, RPC calls, RMI calls, or direct system calls through the APIs provided by component suppliers.
2. Pass the component specification or reservation information (if reservation is made). The component attributes will come from the wrapper generator's internal structure (described in Section 4.1.)
3. Pass the organization's information. Depending on the communication method used, the development organization's information will be encoded and passed to component vendors.

4. Finalize the deal. Depending on vendor requirements, the development organization might be required to sign agreements or pass payment information. Agreement signups will be in the form of encrypted signatures using available encryption techniques or in the form of authentication certificates. Payment information could include credit card numbers or PIN codes.

The following code snippet contains some of the utilities necessary to manage the development organization's profile:

```
GetPersonalInfo () {
      return (Master.PersonalInfo);
}

GetNextCheckNumber () {
      n = Master.CheckNumber;
      Master.CheckNumber++;
      return (n);
}

GetCardInfo () {
      return (Master.CardInfo);
}
```

13.1 Conducting Negotiations

Before acquiring required components, organizations might prefer to negotiate components' pricing and licensing issues with component suppliers. Such negotiations can save organizations money and effort and establish good relations with component vendors. Sometimes, organizations can get special deals from vendors. For example, educational institutions can get academic discounts. Other organizations can get better

deals on site licenses. However, [SEAC99] pointed out that, while organizations can get better prices on components if negotiations are done on organization level, those types of negotiations can also have negative impact by leading projects to select components that may not be the best fit for the particular needs of the project. Once negotiations with vendors (or with vendor agents) are complete, vendors might require system integrators to pass their organization's information to include in the component procurement agreement.

The following utilities demonstrate the possibilities of automating negotiation activities. These utilities will be used in the component acquisition utilities that will be presented in the next section.

```
PriceNegoatiationSuccessful (wrapper, p) {
  // wrapper holds the wrapper specification
  // p is a reference to the vendor's agent

  result = NegotiatePrice (wrapper, p, LICENSE,
           p.suggested_license_price, wrapper.max_licenses_cost);
  if (result == OK) {
    result = NegotiatePrice (wrapper, p, UPGRADE,
             p.suggested_upgrade_price, wrapper.max_upgrade_cost);
    result = NegotiatePrice (wrapper, p, MAINTENANCE,
             p.suggested_maintenance_price,
             wrapper.max_maintenance_cost);
    result = NegotiatePrice (wrapper, p, SUPPORT,
             p.suggested_support_price, wrapper.max_support_cost);
    result = NegotiatePrice (wrapper, p, TRAINING,
             p.suggested_training_price,
             wrapper.max_training_cost);
    result = NegotiatePrice (wrapper, p, INSTALLATION,
             p.suggested_installation_price,
             wrapper.max_installation_cost);
  }

  return (result);
}

NegotiatePrice (wrapper, p, category, retail_price, max_price) {
  if (retail_price > max_price) {
```

```
    return (p.agent.PriceOK (p.ID, p.reservation_ID, category,
            max_price));
  }
  else {
    return (p.agent.PriceOK (p.ID, p.reservation_ID, category,
            retail_price));
  }
}

LicenseNegotiationSuccessful (wrapper, p) {
  // wrapper holds the wrapper specification
  // p is a reference to the vendor's agent

  q = wrapper.preferred_licensing_models;
  OK = false;
  while (q <> nil and not OK) {
    if (p.agent.LicenseModelSupported (p.ID, q)) {
      if (q == FREE || q == DEMO) {
        OK = true;
        p.license_method = q;
      }
      else if (q == PER_USE || q == PER_USER) {
        OK = p.agent.LicenseOK (p.ID, p.reservation_ID, q,
             wrapper.license_count);
        if (not OK) {
          p.suggested_license_count =
          p.agent.GetMaxLicenseCount (p.ID, p.reservation_ID, q);
          OK = p.suggested_license_count > 0;
        }
      }
      else {   // timed licenses
        OK = p.agent.LicenseOK (p.ID, p.reservation_ID, q,
             wrapper.license_duration);
        if (not OK) {
          p.suggested_license_duration =
                p.agent.GetMaxLicenseDuration (p.ID,
                p.reservation_ID, q);
          OK = p.suggested_license_duration > 0;
        }
      }
    }
    q = q.next;
  }
  return (OK);
}
```

13.2 Downloading Components

During the acquisition phase, the generator will contact the provider of each component (or its software agent) and supply it with the information present in the wrapper specification. Passed information could include personal or organizational information, payment information, license information, and suggested component costs. The development organization might be required to sign agreements or supply payments. If the component provider accepts such a deal and if a download option is provided, the generator will download the component to the system integrator's environment.

The component-download process involves many technical details, such as doing the actual download of the component and possibly unzipping acquired files. Vendors could have special requirements or restrictions on downloading optional features. For example, plug-ins are often downloaded through a special interface that comes with the main component. This means that one cannot download the plug-ins before the system that hosts the component is up and running. In such cases, the sequencing of this and the following steps (installation, configuration, and wrapping) can be different for plug-ins. The downloading step can be fully or partially automated depending on the nature of the downloaded files and the logistics provided by component suppliers in performing the downloads. Once the download process is complete, the generator will either point system integrators to where the component is downloaded or provide them with a link to where the component resides in the vendor's premise.

The following code snippet demonstrates the process involved in acquiring the selected set of individual components or group of components. The goal of these utilities is

to automate as much of the acquisition process as possible by utilizing any software agents available both in the system integrator's side and the vendor's side.

```
AcquireComponents (wrapper) {

  // First, try to automate as much of the process as possible

  p = wrapper.selected_components;
  while (p <> nil) {
    // Is there a software agent (in the vendor's side)
    // available to automate the acquisition process?

    if (p.agent <> nil) {
      retries1 = 0;
      while ((result = PriceNegotiationSuccessful (wrapper, p))
            != OK && (retries1 <= MAX_WAITING_TIME) {
        delay (1000);  // wait one second
        retries1++;
      }

      if (result == OK) {
        retries2 = 0;
        while ((result =
               LicenseNegotiationSuccessful (wrapper, p)) != OK
               && (retries2 <= MAX_ WAITING_TIME) {
          delay (1000);  // wait one second
          retries2++;
        }

        if (result == OK) {
          retries3 = 0;
          while ((result =
                 PaymentSubmissionSuccessful (wrapper, p)) != OK
                 && (retries3 <= MAX_ WAITING_TIME) {
            delay (1000);  // wait one second
            retries3++;
          }

          if (result == OK) {
            retries4 = 0;
            while ((result =
                   DownloadComponent (wrapper, p)) != OK
                   && (retries4 <= MAX_ WAITING_TIME) {
              delay (1000);  // wait one second
              retries4++;
            }

            if (result == OK) {
              retries5 = 0;
```

```
              while ((result =
                    ObtainLicenses (wrapper, p)) != OK
                    && (retries5 <= MAX_ WAITING_TIME) {
                delay (1000);  // wait one second
                retries5++;
              }

              if (result == OK) {
                InformUser ("Successfully downloaded component " +
                            p.name + " from vendor " +
                            GetVendorName (p.vendor_ID) + ".");
                p = p.next;
              }
              else {  // Obtain licenses manually
                AddComponent (NeedLicense, p);
                p = p.next;
              }
            }
            else {  // Download the component manually
              AddComponent (NeedDownload, p);
              p = p.next;
            }
          }
          else {  // Submit payment manually
            AddComponent (NeedPaymentSubmission, p);
            p = p.next;
          }
        }
        else {  // Perform license negotiations manually
          AddComponent (NeedLiceseNegotiation, p);
          p = p.next;
        }
      }
      else {  // Perform price negotiations manually
        AddComponent (NeedPriceNegotiation, p);
        p = p.next;
      }
    }
    else {  // do the complete acquisition process manually
      AddComponent (NeedPriceNegotiation, p);
      p = p.next;
    }
  }

  // Second, manually complete any remaining parts of the process

  if (NeedPriceNegotiation <> nil) {
    p = NeedPriceNegotiation;
    while (p <> nil) {
      NegotiatePriceManually (wrapper, p);
      NegotiateLicensesManually (wrapper, p);
      SubmitPaymentManually (wrapper, p);
      DownloadComponentManually (wrapper, p);
```

```
      ObtainLicensesManually (wrapper, p);
      p = p.next;
    }
  }
  else if (NeedLicenseNegotiation <> nil) {
    p = NeedLicenseNegotiation;
    while (p <> nil) {
      NegotiateLicensesManually (wrapper, p);
      SubmitPaymentManually (wrapper, p);
      DownloadComponentManually (wrapper, p);
      ObtainLicensesManually (wrapper, p);
      p = p.next;
    }
  }
  else if (NeedPaymentSubmission <> nil) {
    p = NeedPaymentSubmission;
    while (p <> nil) {
      SubmitPaymentManually (wrapper, p);
      DownloadComponentManually (wrapper, p);
      ObtainLicensesManually (wrapper, p);
      p = p.next;
    }
  }
  else if (NeedDownload <> nil) {
    p = NeedDownload;
    while (p <> nil) {
      DownloadComponentManually (wrapper, p);
      ObtainLicensesManually (wrapper, p);
      p = p.next;
    }
  }
  else if (NeedLicense <> nil) {
    p = NeedLicense;
    while (p <> nil) {
      ObtainLicensesManually (wrapper, p);
      p = p.next;
    }
  }
}

MakeCopy (p) {
  new (q);

  q.name = p.name;
  q.ID = p.ID;
  q.version = p.version;
  q.vendor_ID = p.vendor_ID;
  q.reservation_ID = p.reservation_ID;
  q.agent = p.agent;
  q.URL = p.URL;
  q.license_models = p.license_models;
  q.license_types = p.license_types;
  q.suggested_license_duration = p.suggested_license_duration;
```

```
  q.actual_license_duration = p.actual_license_duration;
  q.suggested_license_count = p.suggested_license_count;
  q.actual_license_count = p.actual_license_count;
  q.suggested_license_price = p.suggested_license_price;
  q.actual_license_price = p.actual_license_price;
  q.suggested_upgrade_price = p.suggested_upgrade_price;
  q.actual_upgrade_price = p.actual_upgrade_price;
  q.suggested_maintenance_price = p.suggested_maintenance_price;
  q.actual_maintenance_price = p.actual_maintenance_price;
  q.suggested_support_price = p.suggested_support_price;
  q.actual_support_price = p.actual_maintenance_price;
  q.suggested_training_price = p.suggested_training_price;
  q.actual_training_price = p.actual_training_price;
  q.suggested_installation_price = p.suggested_installation_price;
  q.actual_installation_price = p.actual_installation_price;
  q.license_keys = p.license_keys;
  q.payment_method = p.payment_method;
  q.download_method = p.download_method;
  q.technology = p.technology;
  q.operating_systems = p.operating_systems;
  q.databases = p.databases;
  q.language = p.language;
  q.formats = p.formats;
  q.support_methods = p.support_methods;
  q.new_updates = p.new_updates;
  q.downloads = p.downloads;
  q.installation_commands = p.installation_commands;
  q.configuration_options = p.configuration_options;
  q.provided_interfaces = p.provided_interfaces;
  q.required_interfaces = p.required_interfaces;
  q.software_constraints = p.software_constraints;
  q.hardware_constraints = p.hardware_constraints;
  q.network_constraints = p.network_constraints;
  q.optional = p.optional;
  q.specification = p.specification;
  q.execution_method = p.execution_method;

  q.next = nil;
  return (q);
}

AddComponent (list, p) {
  q = MakeCopy (p);
  q.next = list;
  list = q;
}

PaymentSubmissionSuccessful (wrapper, p) {
  q = wrapper.preferred_payment_methods;
  OK = false;
  while (q <> nil && not OK) {
    if (p.agent.PaymentMethodSupported (p.ID, q)) {
      if (q == CHECK) {
```

```
        OK = p.agent.AcceptPayment (p.ID, p.reservation_ID, q,
             Master.GetNextCheckNumber (),
             Master.GetPersonalInfo (), Master.GetBankInfo ());
        if (OK) {
          p.payment_method = q;
        }
      }
      else if (q == CREDIT_CARD) {
        OK = p.agent.AcceptPayment (p.ID, p.reservation_ID, q,
             Master.GetCardInfo (), Master.GetPersonalInfo ());
        if (OK) {
          p.payment_method = q;
        }
      }
    }
    q = q.next;
  }
  return (OK);
}

DownloadComponent (wrapper, p) {
  q = wrapper.preferred_download_methods;
  OK = false;
  while (q <> nil && not OK) {
    if (p.agent.DownloadMethodSupported (p.ID, q)) {
      if (q == DOWNLOAD_ANYTIME) {
        OK = StoreComponent (wrapper, p);
        p.download_method = q;
      }
      else {  // other download methods will be handled at
           appropriate time
        p.download_method = q;
        OK = true;
      }
    }
    else {
      p = p.next;
    }
  }
  return (OK);
}

ObtainLicense (wrapper, p) {
  if (p.license_method == FREE || p.license_method == DEMO) {
    return (true);
  }
  else if (p.license_method == PER_USE ||
           p.license_method == PER_USER) {
    if (p.suggested_license_count > wrapper.license_count) {
      p.license_keys = p.agent.GetLicenses (p.ID,
      p.reservation_ID, p.license_method, wrapper.license_count);
      return (true);
    }
```

```
    else {  // not successful since available license count is
            // less than that specified by the user
      return (false);
    }
  }
  else {  // TIMED licenses
    if (p.suggested_license_duration > wrapper.license_duration) {
      p.license_keys = p.agent.GetLicenses (p.ID,
          p.reservation_ID, p.license_method,
          wrapper.license_duration);
      return (true);
    }
    else {  // not successful since available license duration is
            // less than that specified by the user
      return (false);
    }
  }
}
```

13.3 Installing Downloaded Components and Optional Features

Usually, one has to follow a certain procedure to prepare the downloaded component artifacts to make them ready for inclusion in the compositional wrapper. This step uses the *installation* clause in the component specification to install the component as required by its developers. By just considering the options provided in the *installation* clause, this step can be easily automated by the wrapper generator. However, there could be other instructions provided in the component specification that require user's involvement (for example, the *other* option in the *installation* clause.)

13.4 Components Adaptation

Adaptation refers to those activities that would have to be performed to prepare a particular component for inclusion in the compositional wrapper. Since experience has shown that off-the-shelf components will rarely fit together without some sort of third-party work, the development team will eventually need to adapt the chosen components, which can be expensive, time-consuming, or both [CARN97]. Adaptation techniques are numerous and include modification of the component (also called tailoring), instantiation mechanism, parameterization, and specialization [ANIO02].

Tailoring can be done in three ways. The first form of tailoring is to modify the component in ways that are supported by the component developer. The second form of tailoring modifies the behavior of the complete component-based system. The third form of tailoring involves wrapping the component and injecting extra functionality through the wrapper or letting the wrapper augment or transform the output generated by the component. This dissertation avoided the first two forms of tailoring because of their high costs and risks. The adopted third approach involves building a compositional wrapper to correct some forms of undesired component behaviors, inject extra functionality through the use of smart features, and possibly transform the components to different forms through the wrapping options that are supplied to the wrapper generator.

Adaptation can be a complex activity that requires technical expertise at the detailed system and specific component levels [BROW98b]. Adaptation activities include initializing parameter values; specifying I/O screens or report formats; setting up security protocols; etc. Adaptation must take into account the interactions among custom components, COTS products, any non-developmental item components, any legacy code, and

the architecture, including infrastructure and middleware elements. Adaptation effort can range from being trivial to huge [GOME00]. [BROW00] reported that adaptation cost can rise to over 80% of the total cost of the component acquisition process. Much of the effort involved is in understanding the organization's current business practices, aligning those business practices with the processes supported by the component, and tailoring the component for maximum operational efficiency within the organizational context. [ABTS00d] reported that for the 20 projects in their COCOTS database, the total tailoring effort of each of 32 COTS components took less than half person-years. However, it took as much as 60 person-years to tailor some of the components.

Once the development team understands how the component operates, the development team can start adapting the component to the new system being built. This adaptation will require less effort if there is a correspondence between the system needs and the characteristics of the adaptable component. Therefore, in order to decide whether a component is to be adapted by the in-house development team or by an external organization, the following characteristics of the component should be taken into account:

- It is important to determine ***the percentage of the system functionality that is covered by the component*** in order to calculate the work that will be needed to meet operating requirements.
- It is important to understand whether ***the component's access modes*** coincide with that of the developed system. It may be necessary to develop a wide range of adaptation schemes to align the component to correctly interact with other parts of the system. Such adaptation schemes could include translation to other access

modes, adaptation to the execution environment, development of communications protocols, or conversion to different storage structures.

- ***Availability of adaptation and testing tools***. Some suppliers provide integrated tools to visualize, modify, and test their components. If no such tools exist, components would have to be adapted by experienced engineers.

It will take the development team some time to learn to use and adapt the acquired components. The development team will need to consider the following component characteristics that could affect the way such components are adapted:

- ***Construction methodology***: It is essential to understand the methodology that was used in the construction of the component. The interest here is in the structuring of interfaces and the methods used to pass parameters or arguments to such interfaces. Since the component specification language (CSL) has no constructs to provide information about the design and construction methodologies used to develop the component, the development team has to hunt for such information using other means, such as consulting supplied documentation or contacting the component's vendor.
- ***The quality of supplied documentation*** will greatly affect the time needed to understand and adapt the component.
- ***Availability of training courses and technical support*** on the adaptation and use of the component is essential to the success of the adaptation process.

- ***The component granularity*** could affect the time it would take to learn and adapt the component.
- ***The complexity*** of interfaces and the type and number of configuration options provided and the relationships that could exist between the different component interfaces are of great importance in the developers' ability to correctly and easily adapt the component.
- Understanding the list of configuration options available and ***the possibility of automating*** each of them is important in estimating the extent of adaptation effort required from the development team.

Using the composition process described by this research, the wrapper generator can perform component adaptation in the following manner:

- The generator reads configuration options from the *configuration* clause provided in each component specification.
- Then, the wrapper generator enumerates the configuration options in a way that is easily understood by integrators and easily dealt with by the generator.
- The generator performs whatever can be automated of the configuration options.
- After that, the wrapper generator presents the configuration options that cannot be automated to the system integrator (with any recommendations from the generator.)

- The system integrator takes appropriate actions regarding the remaining configuration options.

The generator would still need to validate and save the configuration actions performed by the system integrator. Validation actions could involve checking conformance of entered data or performed actions to that outlined in the wrapper specification. It can also involve testing the configured component. Configuration options can come either in the form of setting some parameters to be passed to the component, setting global or environment variables, calling specific interfaces in the component, or filling out a template that accompany the component (usually in a file).

Though CSL gives a start on tackling the configuration issue through the *configuration* clause, the issues addressed in this section are still need to be taken into account when performing the actual adaptation. Automating the adaptation process requires generating various kinds of inputs or events, which would require careful analysis from the wrapper generator's part.

The literature includes some hints on automating the component adaptation process. For example, [RUTT98] described a *property editor* that can be included in compositional wrappers to automate the customization of components. [ORFA96] included the property editor feature as part of the essential attributes of their "supercomponents." [BATO97] suggested encoding domain-specific intelligence into wrapper generators so that generated wrappers could perform automatic customization and optimization of themselves and their components. Despite the difficulty inherent in the process of encod-

ing component properties and features that can be used in the customization of components, building such self-customization features into compositional wrappers can provide greater flexibility to wrapper users. Interactive customization can be costly if implemented at the component level. However, building such capability into compositional wrappers would require domain-specific intelligence to be encoded into the composition process.

Once the configuration options are validated and accepted, the component must be tested. As stated in the detailed evaluations chapter (Chapter 12), some parts of the testing can be automated and other parts must be carried out manually by the development team.

The following code is a simple illustration of enumerating and displaying configuration options. (Most of the mentioned utilities are not yet coded.)

```
DisplayConfigurationOptions (wrapper) {
      DisplayProcessHeader ("Configuration Options");
      p = wrapper.selected_components;
      while (p <> nil) {
            DisplaySectionHeader ("Configuration Options For", p);
            q = p.configuration_options;
            while (q <> nil) {
                  DisplayConfigurationOption (p, q);
                  q = q.next;
            }
            p = p.next;
            DisplaySectionFooter (p);
      }
      DisplayProcessFooter ("Configuration Options");
}

DisplayConfigurationOption (p, q) {
      if (q.type == YESNO || q.type == TRUEFALSE) {
            DisplaySelectionItem (p, q, q.type);
      }
      else if (q.type == LIST) {
            DisplayList (p, q);
```

```
        }
        else if (q.type == RADIOGROUP) {
              DisplayRadioGroup (p, q);
        }
        else if (q.type == CHECK) {
              DisplayCheckItem (p, q);
        }
}
```

CHAPTER 14: WRAPPING COMPONENTS

Table 11 shows a detailed breakdown of all the activities involved in the process of wrapping components. In addition, the table indicates possibilities for automation (at low, medium, high level versus manual/no automation.) The footnotes to the table provide the justification for the automation provided (as in previous chapters, "G" indicates the automation level when a wrapper generator is used. "WG" indicates the automation level without a generator.)

Table 11: The detailed activities involved in the process of components' wrapping

	Automation			
	Manual	**Low**	**Medium**	**High**
0.0 Develop and operate a compositional wrappers-based (CWB) system				

	Automation			
	Manual	**Low**	**Medium**	**High**
1.0 Design the CWB system (Chapter 9 and 9.5)				
2.0 Build the CWB system (Chapter 10 to Chapter 14)				
2.1 Collect components (Chapter 10 to Chapter 13)				
2.2 Wrap components				
2.2.1 Write the necessary code to wire required interfaces with provided ones		WG[31]		G[32]
2.2.2 Write the necessary code to implement required smart features		WG[31]		G[32]
2.2.3 Write the necessary code to add any required additional functionality[71]	G, WG			
2.2.4 Write the necessary code to provide the component's essential services[72]		G[32], WG[31]		
2.2.5 Apply required wrapping options				
2.2.5.1 Write the necessary code to account for the *FastLoading* and *PreserveMemory* wrapping options		WG[31]		G[32]
2.2.5.2 Compress components		WG[73]		G[74]
2.2.5.3 Write the necessary code to decompress components		WG[31]		G[32]
2.1.3.4 Encrypt components		WG[73]		G[74]
2.2.5.5 Write the necessary code to decrypt components		WG[31]		G[32]
2.2.5.6 Write the necessary code to decrypt the compositional wrapper		WG[31]		G[32]
2.2.5.7 Write the necessary code to decrypt components		WG[31]		G[32]
2.2.6 Resolve any conflicts among the code segments	WG	G[58]		
2.2.7 Generate the compositional wrapper's glue code		WG[75]		G[76]
2.2.8 Compress the compositional wrapper		WG[73]		G[74]
2.2.9 Encrypt the compositional wrapper		WG[73]		G[74]
2.3 Deliver the compositional wrapper	WG		G[77]	
2.4 Assemble components into the CWB system (Section 15.1)				
2.5 Test the CWB system (Section 15.2)				

[71] This could account for any standards or business rules set forth by the developing organization.

[72] Essential services would include storage management, memory management, interfaces to the OS, communication facilities, etc.

[73] The user can utilize some available operating system utilities, such as *gzip* and *tar* in Unix, to accomplish the work required by this activity.

[74] The generator should have a built-in utility to perform this activity or have access to an external utility that can do the work of this activity.

[75] The user can utilize available integrated development environments to generate the wrapper.

[76] Since by this time the generator has all the required code fragments that compose the glue code of the generated compositional wrapper, this task is merely outputting such fragments in a logical manner.

[77] This activity only requires the wrapper generator to point the user to where the compositional wrapper resides. The generator can also provide a drag-and-drop facility.

		Automation			
		Manual	**Low**	**Medium**	**High**
	3.0 Operate the CWB system (Chapter 15)				

Components' wrapping involves allowing the system integrator to specify wrapping options, such as the encryption and/or compression of components before wrapping them and preparing the necessary code to implement the desired functionality and smart features into the generated wrapper. The generated code will also include the logic needed to internally wire the component interfaces, intercept events to/from the components, and present the component outputs in a consistent way. The generated code should also be responsible for allocating and managing needed resources. The final step for the wrapper generator is to output the code either in a source or binary format (depending on the target language supported by the wrapper generator) and to set up a separate folder that will hold the generated code and all the files (component files, documentation files, specification files, license key files, etc.) related to the generated compositional wrapper. The generator could optionally dump all such items into one file, similar to the **".jar"** files in Java. The wrapper generation process will include the following steps:

1. Collect all component binaries (or just the signatures or URLs, if late composition (see Section 14.2) is implemented) into a single location in the system integrator's environment. This could require the wrapper generator to resolve any naming conflicts either in component files or in component interfaces, as will be discussed in Section 14.9.

2. Looking at each included component individually and taking into account any generation options selected, the generator could transform the component into a different form (for example, encrypting and/or compressing the component.) The generator could also prepare certain snippets of code to be included in the generated wrapper and specifically designed for the individual component (such as the required code to decompress and/or decrypt the components.) Such code could account for monitoring the component licenses, tracking the component usage, updating the component files, and wrapping the component to meet certain standards or to support certain technologies.

3. The wrapper generator will then try to match the required interfaces of the compositional wrapper and all of its components with those provided by the encased components (as discussed in Section 12.7.1.) Any unsatisfied required interfaces will be exported as required interfaces in the generated wrapper.

4. The wrapper generator will collect any code generated in the previous steps and generate additional code as required. The final code should account for all functional and non-functional requirements of individual components as well as for other features, such as security, upgrade, and maintenance of the components.

The wrapper generation step mostly involves producing code snippets and then collecting them in a logical manner to form the compositional wrapper. While the code for these parts of the process (such as the code to implement the smart features) could be prepared in advance in the form of templates that the generator can instantiate at genera-

tion time, the code for other parts would need to be figured out at runtime. However, it is likely that most of the actions in this step can be automated despite the initial design and development work required from developers of the wrapper generator. Automation of the whole wrapper generation process is desirable. However, the costs incurred to the wrapper generator developers in preparing the code fragments needed during the wrapping step could be very high due to the goal of making such code fragments flexible enough to fit most of the targeted components (within the generator's domain.) It is expected that this high cost during the generator development stage would be offset by the ease and simplicity of the generation process.

Figure 25 illustrates an example of a compositional wrapper generated through the proposed wrapper generator. The wrapper encases three simpler (possibly composite) components. The wrapper's glue code, shown as the shaded box around the inner box, is responsible for interacting with each component and with the outside environment. All the information flowing into the components must first be checked by the wrapper control code and possibly transformed into another form or augmented with additional information. Similarly, the wrapper must first check all the information generated by components before passing such information to the host application. The wrapper can also filter the generated information or inject additional information into it. In the case where more than one component implements an interface, the wrapper's code would be responsible for selecting the best interface from the pool of compatible ones provided by any of the encased components. The wrapper's code would then be responsible for passing control to the selected interface whenever a call is initiated to the same named interface.

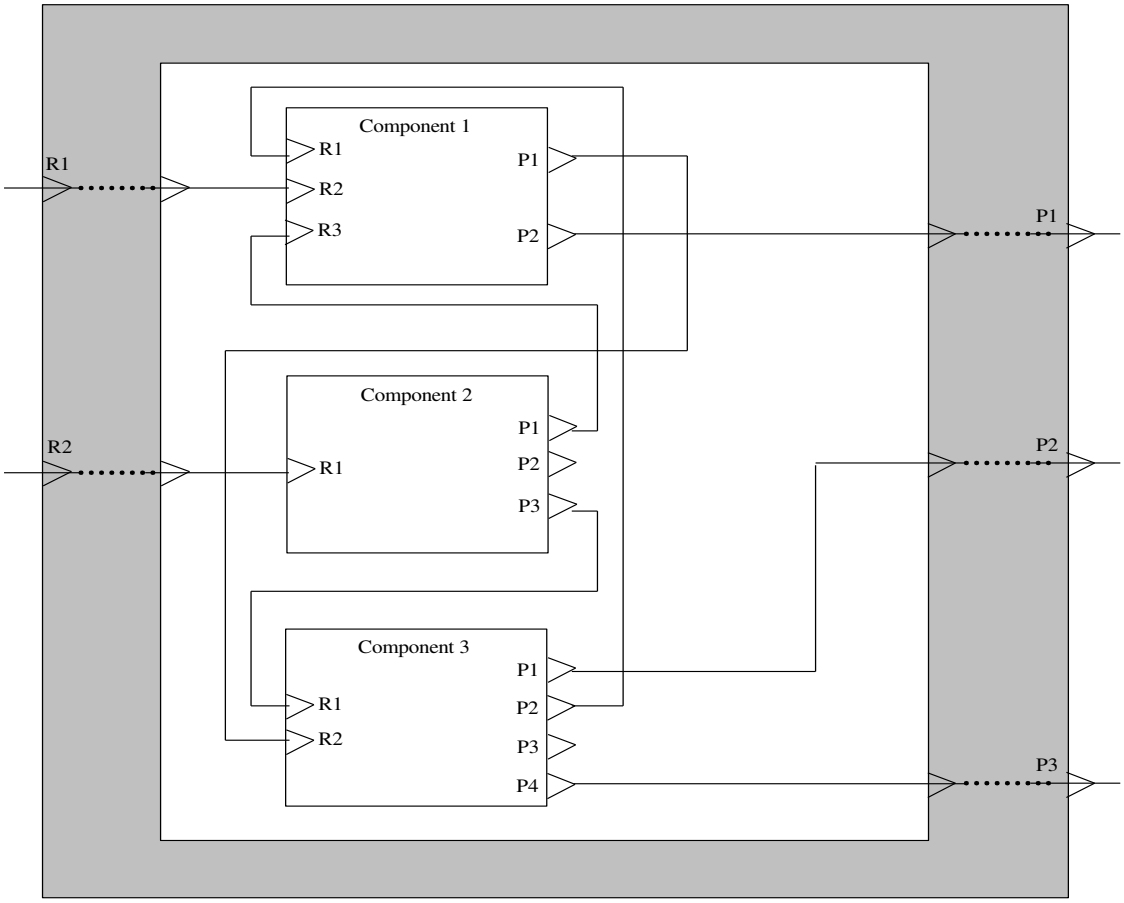

Figure 25: A compositional wrapper generated by the wrapper generator

As illustrated by Figure 25, the wrapper generator will try to satisfy all required interfaces of all encased components. To do so, the wrapper generator will try to match such interfaces with interfaces provided in any of the other encased components. If there are still interfaces left unsatisfied, then the wrapper generator exports them as required interfaces of the generated wrapper (for example, *R2* from Component 1 and *R2* from Component 2.) It would be the responsibility of the compositional wrapper's user to sat-

isfy such required interfaces or to delegate that responsibility to higher-order wrappers that encase this generated wrapper.

14.1 Load Balancing and Emergency Recovery

There will be times when the wrapper generator will provide more interfaces than its users wants. Examples of such interfaces in Figure 25 include *P2* in component 2 and *P3* in component 3. To resolve this situation, the wrapper generator can ignore such interfaces (i.e., not forwarding any event to the interfaces and ignoring any event generated by them.) Alternatively, the generated wrapper could use the unused interfaces for load-balancing or emergency recovery. Load-balancing is needed when the compositional wrapper determines (through some included logic) that a selected interface has received too many calls and therefore has degraded the component's performance. (This situation may occur, for example, when the component is running in a separate thread.) In this case, the wrapper may include the necessary logic to direct some calls to one or more (presumably idle) components that have a compatible interface as the loaded one. Emergency recovery works in a similar manner. However, in emergency recovery, the reason for directing the calls to the other compatible interfaces is because the interface that is supposed to receive such calls is "out of order." An interface becomes out of order because of several reasons (for example, when upgrading the component to which the interface belongs or when a bug, a performance problem, a data integrity problem, or any other problem of this sort is detected by the wrapper's control logic.)

14.2 Late Composition

The traditional way of building systems from components is by assembling their component binaries into a single executable. Late composition refers to the idea of assembling components at runtime. This concept is similar to late binding or dynamic binding where object references are mapped at runtime. In late composition, however, component binaries could be made available only at runtime. At assembly time, only component interface signatures or even URLs to their locations could replace the component binaries. Such options have their advantages as well as their disadvantages. Components could be executed on the vendor's server instead of on the wrapper user's system. This has the potential advantage of relieving the client's system from dedicating the resources required by the component. It also protects the client from some of the liabilities or consequences that could result from executing the component.

Late composition can take various forms. For example, the generated wrapper could include only interface signatures to components and not the components' binaries. The components are downloaded from their respective suppliers on their first usage. This approach could affect the system performance and is prone to network bottlenecks or server problems. However, it has the advantage of low upfront investment since wrapper users do not need to pay for components before actually using them. Another advantage is that the wrapper user is guaranteed to run the latest release of the component. Additionally, if the *downloads* clause in CSL were modified to include the required infor-

mation to implement late composition in compositional wrappers, then no additional information would be needed from the component vendor or from the system integrator.

However, before making this choice practical, many technical and logistic issues would need to be addressed. For example: should the client supply payment information and licensing options when assembling the components or should such information be entered on the first use of the components? Supplying such information at assembly time is more convenient to the client since the client and not the wrapper user is faced with the payment problem. On the other hand, paying on the first usage of the component is more appropriate to the component vendor since newer releases might cost more and the vendor does not need to commit itself to fixed prices.

A variation to this late composition model is to let the compositional wrapper encase only the component URLs to where the components reside, and not to encase their actual interface signatures. This approach is beneficial to vendors since vendors can change interface signatures of their components at any time. However, it will be difficult to design a general-purpose compositional wrapper that can capture many variations of such interfaces. The components must be bound within a certain context in order for the wrapper generator to be able to make decisions about acquiring them. Even if the wrapper generator were successful in implementing this approach, the approach would still suffer from the problem of passing information back and forth between the remote components and the client's system.

Late composition of components brings with it other problems, such as licensing and integration testing. Two licensing models can be used: *design-time licensing* and

runtime licensing [APPL98]. The *design-time* approach limits the use of the component in a development environment without a proper license code. Late composition, however, already implies existence of the design-time model since only component signatures or URLs are used. The *runtime licensing* approach uses a licensing service component to prevent the licensed component from executing unless the proper license code is installed in the system. The licensing service should support a variety of licensing models.

The other problem with late composition is integration testing. Though a "sample" of the component could be downloaded and tested during the composition process, it will be difficult to test the component at runtime. Thus, there would be no guarantee that the component will work as it did before with its tested "sample." On-the-fly testing of such components, though complicated, is doable. However, on-the-fly testing would require having new constructs in both specification languages to guide the testing process using new or enhanced wrapper-generation process activities.

14.3 Confinement of Components

Components should only be allowed to function within the boundary of their specifications. For example, a component that normally does not perform any network communication should not be allowed to connect to the network. Confining black-box components is highly desirable, but is difficult to do in practice [LIND98]. It might be difficult to correctly determine the minimum set of resources that black-box components require to function in all cases. If this is not correctly determined, the component may

not be able to operate in an unforeseen situation. On the other hand, the user may specify a constraint that is too lax to provide an appropriate level of security. The wrapper generator should only grant each encased component the requirements stated in their specifications. The generator's implementation could decide to wrap each component before including it in the composition wrapper. It must be assumed that careful analysis of requirements was conducted during the detailed evaluations stage. Failure to fully investigate the requirements stated in component specifications could result in leaving loopholes for such components to act in unexpected manners.

14.4 Code Size Estimation

The goal of this section is to present a technique for estimating the size of compositional wrappers' glue code. Knowing the glue code size before writing it or auto-generating it can help in determining the amount of effort and schedule needed to generate compositional wrappers. Other benefits for estimating the glue code size include support of estimation of the memory and execution speed requirements of the developed system. Since code size is difficult to define for pre-built components, the estimation technique presented in this section does not rely on such attribute. Rather, the component specification is used to infer the amount of functionality delivered by the compositional wrapper's components. Only the portion of each component's interfaces and functionality that is actually used in the compositional wrapper is accounted for. Section 14.4.1 presents some of the software sizing techniques that are used in industry or discussed in

the literature. Unfortunately, most of these techniques are targeted for complete systems. Additionally, some of these techniques, such as functional points, require extensive amounts of information that is rarely available at the wrapper generation stage. Section 14.4.2 will outline a recommended glue code sizing technique that can be used by wrapper generators to estimate the glue code size of compositional wrappers.

14.4.1 Available Sizing Techniques

Currently, there are several techniques available to estimate the size of a software system. These techniques differ in their estimation accuracy and the depth of knowledge required about the estimated system. Some of the techniques are general to any software system. However, only few of the techniques are suitable for use with components-based systems. Examples of the general software sizing techniques include quick sizing, function points, the Mark II method, feature points, and 3D function points [GARM01, JONE98a, FISC00]. These techniques are not suitable for estimating software size for components-based systems because such techniques are dependent upon information that is not available until later in the development life cycle. These techniques also require detailed knowledge about system processing that is not available early enough for accurate counting, such as states and transitions. Among the reported techniques that have been used to estimate the size of software components are COCOMO II and COCOTS [ABTS00b].

14.4.2 A Proposed Technique for Estimating the Glue Code Size of Compositional Wrappers

[VIGD98] defines glue code to include all the code written by the integrator for the purpose of gluing together the various software components. [CLAR00] clarifies this definition by defining glue code to be:

- Code needed to facilitate data or information exchange between the components and the system into which the components are integrated.
- Code needed to provide required functionality missing in the integrated components AND which depends on or must interact with other components.

The level of complexity of the glue code will vary depending on how well the various parts of the system fit together and the requirements imposed by the operating environment [LOON00]. Glue code typically takes longer and requires more effort to complete than assessment or tailoring [ABTS00d, BOEH02].

In the following pages, a model is presented for estimating the glue code size of compositional wrappers. The model tries to consider most of the factors that could have an effect on the size of generated or developed glue code. Wherever applicable, weights are assigned to each size factor in the form of parameterized functions. Examples of the relative ratios of these parameters are provided.

14.4.2.1 System-Wide Factors

Shown in Table 12 are the factors that either affect every component in the system or that affect the characteristics of the complete system. The list of options in the examples that are shown for some of the factors and the values assigned to each option in these examples are only for demonstration purposes. Such options and their assigned values are, however, selected to show the relative contribution of each option to the size factor to which such options belong. Decisions on the values assigned to each size factor will normally be taken at design time. The technical staff must consult with its management before making final decisions on these factors. Some of the factors will be extracted from the wrapper specification and others would need to be supplied to the wrapper generator by the system integrator in advance before embarking in the wrapper composition process.

System-wide factors (SWF) are a function of many factors as explained in Table 12. It is difficult to estimate the contribution of each of the factors that are shown in Table 12 on the final glue-code size estimate. However, it is interesting to note that all the factors that are shown in Table 12 will eventually increase the code size except for the number of previously generated wrappers (GW). This is because the more wrappers the organization generates, the more experience it accumulates and the more assets (in terms of owned components and established relationships with vendors) that the organization possesses, the less effort and schedule that will take the organization to generate new compositional wrappers.

Table 12: The system-wide factors that affect the glue-code size estimation

System-Wide Factors (SWF)	
ER	Is encryption required? This attribute will be supplied in the wrapper specification as part of the wrapping options. Example of initial assignments to this attribute are: ER(*no*)=1.0 and ER(*yes*)=1.2.
CR	Is compression required? This attribute will also be supplied in the wrapper specification as part of the wrapping options. Example of initial assignments to this attribute are: CR(*no*)=1.0 and CR(*yes*)=1.2.
GPI	The number of interfaces to be provided in the generated wrapper. This will initially be the list of interfaces specified in the wrapper specification. However, fine-tuning of this parameter is possible since some interfaces that are defined in the wrapper specification can be split into more than one interface each of which contains fewer functions.
GRI	The maximum number of interfaces that can be required by the generated wrapper. Since no clause is currently provided in the wrapper specification language to supply this attribute, then either the system integrator would need to supply it or the wrapper generator could use a default value for it.
EO	The component download and execution option. This indicates when and where to execute the acquired component. Section 14.2 discussed some of the possible download scenarios. Examples of the various options in this attribute and their initial assignments would be: EO(*download at design time*)=1.0, EO(*download at runtime*)=1.2, and EO(*execute at vendor's premise*)=1.3. Currently, the *downloads* clause in the component specification language assumes downloading at design time only.
SF	The number of security features required. Security features could include: ☒ Filtering inputs that are passed to each interface of each encased component. Enabling this feature will allow only those inputs that conform to the input specifications outlined in the component specification. ☒ Filtering out events generated by the encased components. With this option, only events that conform to the output specifications that are outlined in the component specification will be allowed to reach the user. ☒ Tracking the component activities by using the interfaces provided by some of the smart features.
AU	Is automatic updating of components needed? Information concerning this attribute is supplied by the *AutoUpdate* smart feature (see Section 8.2.9.6.)

	Example of initial assignments to this attribute are: AU(*no*)=1.0 and AU(*yes*)=1.5.
MC	The requirement to support multiple components for load-balancing or emergency recovery (see Section 14.1.) Examples of the various options in this attribute and their initial assignments could be: MC(*no requirement*)=1.0, MC(*use the same component*)=1.1, and MC(*use compatible components*)=1.5.
DT	Importance of system slowdown (or even shutdown) due to component updating. Examples of the various options in this attribute and their initial assignments would be: DT(*not important*)=1.0, DT(*little importance*)=1.1, DT(*important*)=1.2, DT(*very important*)=1.3, and DT(*critical*)=1.5. The component-updating utilities included in the wrapper's glue code will vary in their sophistication depending on the tolerance level accepted to the occurrence of this event.
GW	The number of previously generated wrappers. Such number has an effect on the amount and robustness of code that can be reused or that developers will produce. Optimized code will eventually be smaller in size.

14.4.2.2 Component-Specific Factors

Component-specific factors characterize each component selected for inclusion in the compositional wrapper. Though it would be difficult, if not impossible, to know the actual values of most of these factors before acquiring the target component, system integrators can estimate the value of each factor. Estimated values could be based on historical data from previous projects developed by the same organization that is using the component or from other organizations that developed similar projects. If such historical data is not readily available, an average- or worst-case estimates can be used.

The two factors of Table 13 can be used to estimate the number of updates that will be released throughout the expected life of the component. Various other parameters

could affect the calculation of LF and UF. However, for many cases, update counts can be estimated using the following equation:

$$UC = LF / UF$$

Table 13: Estimating the number of component updates throughout the expected life of a component

Update Counts (UC)	
LF	The expected life duration of the component (in months). This will usually be the same as the expected life duration of the host application. However, there are cases when the life duration of the component will be shorter than that of the application. For example, the component user could have planned to develop a better or more suitable version of the component in the near future. In this case, the component life will be from its first initiation in the application until it is replaced by the new component. Other factors could affect the life duration of the component. These could include the component vendor's maturity and the vendor's willingness of supporting and enhancing its component. Another factor is the worthiness of the host application to the component user's organization.
UF	The update frequency (in months). This is the average duration between updates. The estimate usually comes from the developing organization of the component. The estimate might be difficult to obtain, especially for first releases of the component. However, the organization can use estimates of update durations for other components of similar size and compatible functionality. If the organization is new to the component development business, then it can use published data from other organizations. For example, [BASI01] reported that the average COTS software product undergoes a new release every eight to nine months, with active vendor support for only its latest three releases. [BOEH02] reported that expected duration between releases of COTS components went up from 6.3 months in 1999 to 8.5 in 2000 to 8.75 in 2001.

Table 14 lists many of the component-specific factors that can affect the process of estimating the glue code size. It is intuitive to notice that all the factors that are shown in Table 14 will increase the overall glue code size except the component maturity (*Y*). Component maturity is expected to lower the glue code size by a factor proportional to the number of years since the component was first released. This is because as the component matures, users' trust and dependence in it and in its vendor will increase. Such trust will cause the removal of parts of the glue code that are usually included to account for unexpected component behavior.

Table 14: The component-specific factors that affect glue-code size estimation

Component-Specific Factors (CSF)	
CV	The number of suppliers providing the component. Component suppliers would include distributors of the same component and original vendors of components that are compatible to the current component. The wrapper generator can estimate this number by counting the number of retrieved components that are compatible to the current component. Availability of compatible components could require additional utilities in the glue code to search for and evaluate such compatible components during component updates or when there is a need to replace the installed component with a different one.
UC	The number of estimated component updates (as discussed at the beginning of this section.)
CC	Will other components with compatible functionality be included in the compositional wrapper? The answer to this question would come from the value of *CV* and by considering other means of supplying components with compatible functionality. For example, if $CV > 1$, then the answer to this question is yes. However, if $CV = 0$ (i.e., no other supplier has compatible components), then one needs to look at other means of supplying compatible components. Such means would include compatible components from the same component supplier, compatible components that are already owned or developed by the organization that is doing the estimate, and compatible newly developed components or wrapped legacy systems that the organization is willing to

	provide. Example of initial assignments to this attribute are: CC(*no*)=1.0 and CC(*yes*)=1.1. The extra effort will be needed in writing the code to monitor the performance of the compatible components and in selecting the best-performing component using some kind of voting schemes [COOK99].
CPI	The number of interfaces provided by the component. This can be obtained directly from the component specification by counting the number of interface specification sections that are listed in the component specification.
CRI	The number of interfaces required by the component. This can be obtained from the *requires* clause in the component specification. However, since the *requires* clause could include references to complete component specifications as well as to narrative descriptions of components and interfaces, it might require parsing of other referenced specifications and/or estimation of other interfaces that are described using the narrative form before being able to obtain a value for *CRI*.
LM	The license model for this component. The *licensing_models* clause in the component specification supplies such information. Examples of the various options in this attribute and their initial assignments would be: LM(*free*)=1.0, LM(*demo*)=1.1, LM(*per use*)=1.3, LM(*per user*)=1.3, and LM(*timed*)=1.2. Since the *licensing_models* clause in the component specification can include more than one model, the system integrator would need to find some way of assigning weights to the different combinations of the listed licensing models.
CO	The number of configuration options provided for this component. This can be obtained from the *configuration* clause in the component specification.
OF	The number of optional features, components, plug-ins, or add-ons that are offered with this component. The component specification lists such additional features or components in the *optional* clause. However, one needs only to count those features, plug-ins, or add-ons that are considered for inclusion in the compositional wrapper. Stand-alone components listed in the *optional* clause need not be counted here since the optional components that will be included in the compositional wrapper will be treated as any other component and will have their own set of factors that will contribute to the overall glue code size estimation.
CS	The component type. Examples of the various options in this attribute and their initial assignments would be: CS(*local*)=1.0, CS(*public*)=1.1, and CS(*commercial*)=1.2. *Local* components include self-developed components, reused software, reengineered software, and wrapped legacy systems. *Public* components include freeware and public libraries. *Commercial* components include shareware, COTS (Commercial-off-the-Shelf), MOTS (Military-off-

	the-Shelf), and GOTS (Government-off-the-Shelf). It might be possible for the wrapper generator to infer this attribute by checking the search scope that was used to locate the selected components.
BS	The component's binary size (in K bytes)
Y	The component maturity. This refers to the number of years since the first release of the component. Since there is no clause in the component specification to tell the component maturity, the generator can use a default value (for example, one year.)
CT	The component technology (CORBA, COM, JavaBeans, etc.) This attribute is included for two reasons. First, different technologies give varying levels of support to the component user. Forms of support from these technologies could be in the number of built-in interfaces, the types of available tools to configure the components, or the richness and availability of reference materials. The second reason is that some technologies require more effort from the integrator's part than others. The number of interfaces that the technology requires developers to make available in their components is an example of the extra effort that need to be considered. Since the impact of each technology on the developed system depends on the nature of the developed system and the homogeneity of the acquired components, system integrators would need to decide on the relative values of this factor to assign to each used technology.

14.4.2.3 The Glue Code Expansion Factor

Using the system-wide factors (SWF) that were discussed in Section 14.4.2.1 and the factors specific to each component (CSF) that were discussed in Section 14.4.2.2, one can calculate the Glue Code Expansion Factor (GCEF). GCEF refers to the factor by which the glue code size will expand from that of a basic wrapper (as discussed in Section 14.4.2.4 below).

The following equation captures the *SWF* and *CSF* factors along with *w* which refers to the number of components to be included in the compositional wrapper:

$$GCEF = f(w, SWF, CSF)$$

SWF will be a function of all the factors that are shown in Table 12. Similarly, *CSF* will be a function of all the factors that are shown in Table 14. However, the contribution of each factor will vary depending on the nature of each factor and the importance of each factor to the developing organization. For example, factors that could have discrete values (for example, yes or no) might be assigned fractions (as can be seen from the example assignments to some of the factors in the two tables.) However, for those factors that have numerical quantities, one might need to change them to fractions. For example, one might decide to divide security features (*SF*) by 10. Organizations might also need to use other forms to convert some of the factors from their quantitative form to fraction form. For example, instead of dividing component maturity (*Y*) by another number, one can decide to use exponential or logarithmic forms instead. Such decisions would usually be based on past experience or on existing similar models.

The following equation can give an initial guess on the resultant expansion factor. This equation is simply multiplying together all system-wide and component-specific factors. The C and C_i constants give more degrees of freedom to each organization to assign different weights of each *SWF* and *CSF* to reflect their importance in the final glue-code expansion factor.

$$GCEF = (C * SWF) + \sum_{i=1}^{w} \left[C_i * (CSF)_i \right]$$

14.4.2.4 The Basic Wrapper

The calculation of the final glue code size (as shown in Section 14.4.2.5) depends on the glue-code expansion factor (discussed in the previous section) and on the glue code size of a basic wrapper. This section defines what a basic wrapper means and how an organization can calculate (or estimate) the glue-code size of such basic wrapper. In the following discussion, the values assigned to the various factors are either quantities derived from the stated assumption of each factor or from the fractions that were listed in the example assignments in Table 12 and Table 14.

A basic wrapper is defined to be simply an encasing around a *single* COTS component (w=1). It is assumed that the COTS component has been in the market for exactly one year (Y=1), provides only one interface (CPI=1), and requires no additional interfaces (CRI=0). The component is obtained *free* of charge (LM=1.0) from the host computer (CS=1.0). The component's binary is 1K bytes in size (BS=1.0) and assumed to be stable with no plans for upgrading it (UF=0, AU=1.0, DT=1.0). There are no compatible components available (CC=1.0) and only one supplier is offering this component (CV=1). The component needs no configuration (CO=0) and offers no optional features (OF=0.) The component would be downloaded at design time (EO=1.0.) This basic wrapper is assumed to be the first wrapper generated by the organization (GW=0.) The wrapper would be in plain binary code (i.e., not encrypted (ER=1.0) and not compressed (CR=1.0).) The wrapper implements no security (SF=0) or load-balancing (MC=1.0) features. The basic wrapper's code would include the minimum necessary code to:

- Pass all events generated by the host application to the wrapped component (*GRI*=1).
- Intercept all events generated by the wrapped component and pass them to the host application (*GPI*=1).

The actual code size of the basic wrapper (*BWCS*) will depend on many factors, such as the development language, the developer's experience with the development language, and the coding standard used. Development organizations can estimate *BWCS* for their environment by developing the actual wrapper that was described in this section. Organizations can also use published statistics or historical data from past projects to estimate the code size of this basic wrapper.

14.4.2.5 The Estimated Compositional Wrapper Glue-Code Size

Estimating the glue code size of the compositional wrapper would merely be to put the estimated glue code size of the basic wrapper (*BWCS*) with the glue-code expansion factor (*GCEF*) that was discussed in Section 14.4.2.3 in some kind of mathematical form. For example, the following equation assumes that the glue code size of the basic wrapper will scale linearly with the glue-code expansion factor. The *C* in this equation is a constant multiplier that allows the organization to scale *BWCS* up or down depending on their confidence with the estimated or obtained value of *BWCS*.

$$WGS = GCEF * (C * BWCS)$$

Organizations can decide to use other forms of obtaining *WGS* from *BWCS* and *GCEF*. Such decisions would usually be based on historical data, published reports, or accepted similar models. Additionally, organizations would need to calibrate the value of *BWCS* as technologies change or as the organization matures.

14.4.2.6 An Example

Table 15 demonstrates the applicability of the glue-code estimation technique presented in the previous sections. Shown in this table are three hypothetical compositional wrappers. The values assigned to each factor are either quantities assumed by each case or fractions derived from the listed options in the example assignments of Table 12 and Table 14. The *Minimum-Complexity* compositional wrapper contains two components with the minimum set of constraints. The *Medium-Complexity* compositional wrapper uses the options in each factor that provide an average complexity. For this case, five components are encased, each with five provided interfaces and five required interfaces. In the *Maximum-Complexity* case, options that give the tightest constraints were selected. For this case, ten components are encased; each with ten provided interfaces and ten required ones. In all the three cases, it is assumed that the weight assigned to each code size factor (shown in Table 15) will be the same across all components encased in each

compositional wrapper. (In reality, the weights assigned to each factor could be different for each acquired component.) Note that for the M*inimum-Complexity* case, each factor is assigned the smallest possible value of the example assignments of Table 12 and Table 14. Such values would make the characteristics of the *Minimum-Complexity* compositional wrapper close to that of a basic wrapper. However, for the *High-Complexity* case, it is difficult to anticipate the characteristics of complex compositional wrappers since there could exist more complex configurations than those shown in Table 15. [JONE98a] emphasized this observation by showing that application complexities are more than a hundred folds in range depending on the application domains. Configuration complexities could also be influenced by organization needs or by the complexities of encased components.

For this example, *WGS* is obtained using the example equation that is shown in the previous section. *BWCS* is assumed to be 1. The following equations are used to calculate *SWF*, *CSF*, and *GCEF*.

$$SWF = \left[ER * CR * \left(1 + \frac{GPI}{10}\right) * \left(1 + \frac{GRI}{10}\right) * EO * \left(1 + \frac{SF}{10}\right) * AU * MC * DT * \left(1 + e^{-GW}\right) \right]$$

$$CSF = \left[\begin{array}{l} \left(1 + \frac{(CV-1)}{10}\right) * CC * \left[1 + \frac{CPI}{10}\right] * \left(1 + \frac{CRI}{10}\right) * LM * \left(1 + \frac{UF}{10}\right) * \\ \left(1 + \frac{CO}{100}\right) * \left(1 + \frac{OF}{10}\right) * CS * \left(1 + \frac{(BS-1)}{100}\right) * \left(1 - e^{-1} + e^{-Y}\right) * CT \end{array} \right]$$

e in the above equations refers to the exponential function.

$$GCEF = SWF + \sum_{i=1}^{w} (CSF)_i$$

Table 15: Estimating the glue-code size of three different compositional wrappers

System-Wide Factors						
	Minimum Complexity		***Medium Complexity***		***High Complexity***	
ER	No	1	No	1	Yes	1.2
CR	No	1	Yes	1.2	Yes	1.2
GPI		2		5		10
GRI		0		5		10
EO	Download at design time	1	Download at runtime	1.2	Execute at vendor's premise	1.3
SF		0		2		5
AU	No	1	Yes	1.5	Yes	1.5
DT	Not important	1	Important	1.2	Critical	1.5
MC	No	1	Same Compo-nent	1.1	Compatible Components	1.5
GW		0		10		100
SWF		2.4		7.7		37.9
Component-Specific Factors						
	Minimum Complexity		***Medium Complexity***		***High Complexity***	
CV		1		2		5
CC	No	1	Yes	1.1	Yes	1.1
CPI		1		5		10
CRI		1		5		10
LM	Free	1	Timed	1.2	Per User	1.3
UF		1		2		2
CO		0		5		10
OF		0		2		5
CS	Local	1	Public	1.1	Commercial	1.2
BS		1		10		100
Y		0		1		2
CSF		2.2		5.9		29.1
Glue-Code Expansion Factor						

w	2	5	10
$\sum_{i=1}^{w}(CSF)_i$	4.4	29.5	291
GCEF = SWF + $\sum_{i=1}^{w}(CSF)_i$	6.8	37.2	328.9
WBCS	1.0	1.0	1.0
WGS = GCEF * WBCS	6.8	37.2	328.9

The above example says that the final glue code size of the *Minimum-Complexity* compositional wrapper is expected to be around 6.8K bytes. Similarly, the expected glue code size for the *Medium-Complexity* and *High-Complexity* cases is 37.2K and 328.9K bytes, respectively.

14.5 Selecting the "Best" Interfaces

The wrapper generator (and component integrators) will need to identify and understand the provided, required, and optional interfaces of each acquired component. *Provided interfaces* are the set of interfaces that a component makes available to other components. *Required interfaces* are the set of essential interfaces without which the component cannot function properly. *Optional interfaces* are the set of interfaces that the component may make use of. Failure to fully understand the nature of each interface could have severe effects on the remaining activities of the wrapping process. For example, wrapper designers will face the problem of selecting the best-provided interfaces to include in the generated compositional wrapper. Such a problem arises when two or more of the encased components offer the same or compatible interfaces. If the wrong

interface is selected, system performance and total system cost could be severely affected because commitments to include an interface in a system result in commitments to certain vendors, technologies, and many other attributes. To work around this problem, [COOK99] suggested to use voting schemes to select the "best" component from a group of compatible ones. In this manner, the reliability of the compositional wrapper as displayed to the system is greater than the reliability of the individual components. The voting scheme recommended by [COOK99] works as follows:

- Formally, specify the input domain that the compatible interfaces are addressing.
- For each interface invocation by the system, call each individual interface that is compatible with the invoked interface.
- Select the computational result of each interface and match the results with the specified domain. Use gathered statistics (see next point) to select among the interfaces that yielded the same "correct" result. Note that deciding when results are the same is not always a trivial equality comparison. Some data types and problem domains need more expressive tests for deciding when two values are equal (e.g., floating point numbers.)
- Gather statistics by logging information about the correctness of each compatible interface. At specified intervals, the wrapper's control software would "inspect" the statistics for each compatible interface and may take any of two actions (besides doing nothing.) First, the compatible interface in one or more components might be marked as faulty, because it always yielded the wrong result. Such faulty

interfaces will not be invoked in future calls to this specific interface. However, such interfaces might be considered as compatible with other interfaces that match their domain constraints. Second, the domain constraints of such compatible interfaces might be modified, because it is failing in some part of its own domain. This could result in modifying compatibility associations among components (i.e., an interface that is currently compatible with one set of interfaces might be modified to be considered as compatible with a different set of interfaces.)

Complex logic would need to be included to implement voting scheme in the generated wrapper. In addition, the voting scheme could only solve performance problems. Other issues (such as safety and data integrity) still exist. The voting scheme bases its decision on what interfaces to use only by comparing the results generated by the compatible interfaces without regards to other parameters involved in the process, such as performance and resource usage. The conclusion here is that selecting the best interface is not a trivial task. Thus, it typically requires careful attention from the wrapper generator designers and developers and cannot be replaced by a voting scheme.

14.6 Wiring Required and Provided Interfaces

After selecting the best interfaces (or at least the best components) to include in the generated wrapper, the next task for the wrapper generator is to wire together the provided and required interfaces of the selected components. There are three kinds of possi-

ble interface connections [PLAS99]: The first type binds a required interface in one component directly to a provided interface in another component. The second type delegates the work of a provided interface of a composite component or wrapper to a provided interface in a nested or encased component. In other words, the interfaces specified in the wrapper specification are delegated to the provided interfaces in the encased components. The third type exports a subcomponent's required interface as a required interface in the enclosing composite component or wrapper. This happens when there are no available provided interfaces in the encased components to satisfy the exported ones. However, using the technique of Section 12.7, more components could be acquired to cover the missing interfaces. Additionally, the development team could decide to provide such missing interfaces using other means, such as wrapping legacy systems or developing new components (as explained in Section 10.4.)

14.7 Adding Smart Features to Generated Wrappers

The wrapper generator must include in the generated compositional wrapper the necessary code that implements all the smart features selected in the wrapper specification using the `<SmartFeaturesDeclaration>` construct (see Section 8.2.9.) Additionally, Section 8.2.9 presented the individual smart features relevant to each discussed smart feature category. In this section, only the interfaces that are needed to implement the various smart features are discussed.

The following two interfaces will be included in all generated wrappers to allow wrapper users to selectively enable or disable any of the smart features at runtime:

```
EnableSmartFeature ("<currentusercode>", <feature_name> )
DisableSmartFeature ("<currentusercode>", <feature_name> )
```

The <currentusercode> is a valid authentication code (see Section 8.2.9.1) for the current wrapper owner. The <feature_name> is one of the valid feature names that will be discussed in the following subsections. The feature name will consist of the smart feature's group name (for example, selfdoc or security) and the feature name in that group (for example, DisplayIcon in selfdoc and LimitCallers in security.) The two parts will be joined by a period (for example, SelfDoc.DisplayIcon or Security.LimitCallers.)

The following subsections discuss the smart features that are attractive to include in compositional wrappers, so to make them behave more intelligently. Though some of these features can be implemented in individual components, the features are more suitable to compositional wrappers because of their complexity and nature of their function. The features are broken into the same categories that were used in the specification of smart features in Section 8.2.9.

14.7.1 Managing Authentication Codes

Authentication codes are discussed in Section 8.2.9.1. The following interfaces will be provided in the compositional wrapper to help manage authentication codes:

PrintOwnerAuthenticationCode (“<currentownercode>”)
ChangeOwnerAuthenticationCode (“<currentownercode>”, “<newcode>”)
PrintLicensorAuthenticationCode (“<currentlicensorcode>”)
ChangeLicensorAuthenticationCode (“<currentlicensorcode>”, “<newcode>”)
PrintDeveloperAuthenticationCode (“<currentdevelopercode>”)
ChangeDeveloperAuthenticationCode (“<currentdevelopercode>”, “<newcode>”)
PrintUserAuthenticationCode (“<currentusercode>”)
ChangeUserAuthenticationCode (“<currentusercode>”, “<newcode>”)

All of the above interfaces require valid authentication codes. The <currentcode>s and <newcode> are the current code of each party (original owner, developer(s), licensor, or current owner) and the new authentication code, respectively.

14.7.2 Self-Documentation

The self-documentation smart feature is discussed in Section 8.2.9.2. Some or all of the following interfaces will be provided in the generated compositional wrapper to help access the features selected in the wrapper specification.

- For the `<SelfDocumentation_DisplayIcon>` construct:

  DisplayIcon
  ChangeIcon (“<filename>”)

- For the `<SelfDocumentation_PrintDescription>` construct:

  PrintDescription

- For the `<SelfDocumentation_PrintState>` construct:

  PrintState (“<currentusercode>”)

- For the `<SelfDocumentation_SendState>` construct:

SendState (“<currentcode>”, “<destination>”)

- For the <SelfDocumentation_PrintHistory> construct:

  PrintHistory (“<currentusercode>”)

- For the <SelfDocumentation_SendHistory> construct:

  SendHistory (“<currentcode>”, “<destination>”)

- For the <SelfDocumentation_ListComponentsInfo> construct:

  ListComponentsInfo

- For the <SelfDocumentation_ListInterfaces> construct:

  ListInterfaces

The <filename> is the full path to the new icon file. The <currentcode> is the current code of one of the four parties (original owner, developer(s), licensor, or current owner) that are allowed to access the SendState and SendHistory interfaces. The <currentusercode> is the current authentication code for the current wrapper’s owner. This means that only the current wrapper owner can access the PrintState and PrintHistory interfaces. The <destination> is either a valid URL to a server agent that can receive such information or a valid email address to which the information will be sent. The SendState and SendHistory interfaces can be accessed using remote method calls, such as Java’s RMI or Unix’ RPC. The ListComponentsInfo and ListInterfaces require no authentication codes, which means any application can call them.

14.7.3 Tracking

The tracking smart features are discussed in Section 8.2.9.3. The following is a list of interfaces that will be provided in the generated wrapper to help access the tracking features:

- The following interfaces will be included in all generated wrappers if *any* of the tracking smart features is selected:

  StartTrackingAll (“<currentcode>”)
  PauseTrackingAll (“<currentcode>”)
  ResumeTrackingAll (“<currentcode>”)
  StopTrackingAll (“<currentcode>”)

- For the `<Tracking_TrackCalls>` construct:

  StartTrackingCalls (“<currentcode>”)
  PauseTrackingCalls (“<currentcode>”)
  ResumeTrackingCalls (“<currentcode>”)
  StopTrackingCalls (“<currentcode>”)

- For the `<Tracking_TrackLicenses>` construct:

  StartTrackingLicenses (“<currentcode>”)
  PauseTrackingLicenses (“<currentcode>”)
  ResumeTrackingLicenses (“<currentcode>”)
  StopTrackingLicenses (“<currentcode>”)

- For the `<Tracking_TrackHistory>` construct:

  StartTrackingHistory (“<currentcode>”)
  PauseTrackingHistory (“<currentcode>”)
  ResumeTrackingHistory (“<currentcode>”)
  StopTrackingHistory (“<currentcode>”)

- For the `<Tracking_AdministrativeActivities>` construct:

  StartTrackingAdminActivities (“<currentcode>”)
  PauseTrackingAdminActivities (“<currentcode>”)
  ResumeTrackingAdminActivities (“<currentcode>”)
  StopTrackingAdminActivities (“<currentcode>”)

- For the <Tracking_PrintTrackingInfo> construct:

  PrintTrackingInfo (“<currentusercode>”)

- For the <Tracking_SendTrackingInfo> construct:

  SendTrackingInfo (“<currentcode>”, “<destination>”)

The <currentcode> is the current authentication code of one of the four parties (original owner, developer(s), licensor, or current owner) that are allowed to access the SendTrackingInfo interface. The <destination> is either a valid URL to a server agent that can receive such information or a valid email address to which the information will be sent.

For each tracked item, there will be four interfaces provided. A “start” interface will clear any previous tracking information collected and then restart the tracking process. The “pause” interface will temporarily stop the tracking process of such item until the “resume” interface is called. Any activity collected before the call to the “pause” interface will be saved and the tracked information collected after calling the “resume” interface will be amended to that old information. The “stop” interface will stop all tracking activities for that item. The collected tracking information will be left untouched until the “start” interface is re-called where the collected information will be cleared. Therefore, calls to the PrintTrackingInfo or the SendTrackingInfo interfaces will print or send whatever tracking information collected since the last call to the “start” interface. The SendTrackingInfo interface can be accessed using remote method calls, such as Java’s RMI or Unix’ RPC.

If the wrapper specification lists TrackOwners as one of the required smart features, then the wrapper generator must include the necessary code in the generated wrapper to maintain the list of owners of the generated wrapper. The list must be in chronological order of ownership time with most recent owners at the top of the list.

If the wrapper specification lists TrackLicenses as one of the required smart features, then the wrapper generator should include the necessary code in the generated wrapper to maintain licenses information. The following information must be kept for each license:

- License Type (*free*, *demo*, *timed*, *metered*, etc.)
- License Effective Date (for *demo* and *timed* licenses)
- License Expiration Date (for *demo* and *timed* licenses)
- License Credit (for *metered* licenses)
- License Key (the actual authorization key)

14.7.4 Licensing

The licensing smart feature options are discussed in Section 8.2.9.4. The following is a list of interfaces that will be provided in the generated wrapper to help access the licensing smart features:

- For the `<Licensing_PrintLicenseInfo>` construct:

  PrintLicenseInfo (“<currentusercode>”)

- For the `<Licensing_PrintCurrentOwner>` construct:

  PrintCurrentOwner

- For the <Licensing_PrintOwners> construct:

  PrintOwners (“<currentusercode>”)

- For the <Licensing_SendOwners> construct:

  SendOwners (“<currentcode>”, “<destination>”)

- For the <Licensing_Suspend> construct:

  Suspend (“<licensorcode>”, “<message>”)

- For the <Licensing_Resume> construct:

  Resume (“<licensorcode>”)

The <currentcode> is the current authentication code of one of the four parties (original owner, developer(s), licensor, or current owner.) The <currentusercode> is the current authentication code of the current wrapper owner. The <licensorcode> is the authentication code of the wrapper licensor (the previous wrapper owner.) The <message> is the message that the wrapper will return to its callers when it is in the inactive state (after a call to the Suspend interface.) The <destination> is either a valid URL to a server agent that can receive such information or a valid email address to which the information will be sent. The SendOwners, Suspend, and Resume interfaces can be accessed using remote method calls, such as Java’s RMI or Unix’ RPC.

The only public interface in this group is PrintCurrentOwner. Suspend and Resume can only be accessed by the licensor of the compositional wrapper. The current wrapper owner is the only one authorized to access the PrintLicenseInfo and PrintOwners interfaces.

14.7.5 State and Structure

The state and structure smart features are discussed in Section 8.2.9.5. The following is a list of interfaces that will be provided in the compositional wrapper to help access the state and structure features:

- For the `<State_Schedule>` construct:

  Schedule (“<currentusercode>”, “<starttime>”, “<endtime>”)
  ClearSchedule (“<currentusercode>”, “<starttime>”, “<endtime>”)

- For the `<State_PrintSchedule>` construct:

  PrintSchedule

- For the `<State_Pause>` construct:

  Pause

- For the `<State_Resume>` construct:

  Resume

The <currentusercode> is the authentication code for the current wrapper’s owner. The <starttime> and <endtime> are the start time and end time, respectively, of one period during which the wrapper is active (i.e., delivering all its functionality.) Once the activity schedule is set for the wrapper, the wrapper will monitor its schedule and will cease to function outside the scheduled periods. The wrapper can further be put in an inactive state using the Pause interface. The Resume interface will put the wrapper

back in operation. The ClearSchedule interface is used to clear scheduled periods within a certain timeframe. The Schedule and ClearSchedule interfaces require the authentication code of the current owner.

14.7.6 Auto-Updating

The auto-updating smart features are discussed in Section 8.2.9.6. The following is a list of interfaces that will be provided in the generated compositional wrapper to help access the auto-updating features:

- For the `<Update_SelfAutoUpdate>` construct:

  SetSelfUpdateURL (“<currentusercode>”, “<Url>”)
  SetSelfUpdateFrequency (“<currentusercode>”, “<update_frequency>”)
  SkipNextSelfUpdate (“<currentusercode>”)

- For the `<Update_ComponentsAutoUpdate>` construct:

  SetComponentsUpdateProfile (“<currentusercode>”, “<profile>”)
  SetComponentsUpdatePaymentInfo (“<currentusercode>”, “<payment_info>”)
  SetComponentsUpdateFrequency (“<currentusercode>”, “<update_frequency>”)
  SkipNextComponentsUpdate (“<currentusercode>”)

- For the `<Update_AutoUpdateOptions>` construct:

  SetAutoUpdateOptions (“<currentusercode>”, “category”, “<options_list>”)
  ClearAutoUpdateOptions (“<currentusercode>”, “category”, “<options_list>”)
  SetAutoUpdatePreReminder (“<currentusercode>”, “category”, “<when>”,
  “<reminder_format>”)
  SetAutoUpdatePostReminder (“<currentusercode>”, “category”, “<when>”,
  “<reminder_format>”)
  SetAutoUpdateInstallation (“<currentusercode>”, “category”, “<when>”,
  “<installation_format>”)

The <usercurrentcode> is the authentication code for the current wrapper's owner. The <Url> is the full path to a CSL specification file. The <update_frequency> is the list of update frequencies, as outlined in the <UpdateFrequency> construct. The <profile> is a valid personal profile specification as given by the <ProfileSpecification> construct. Similarly, the <payment_info> should be a valid payment specification as given by the <BankAccountSpecification> or the <CreditCardSpecification> sections. The <profile> and <payment_info> are not identifiers, but complete specifications as were defined by the <ProfileSpecification> section (see Section 8.4), the <CreditCardSpecification> section (see Section 8.6), or the <BankAccountSpecification> section (see Section 8.7). The <category> is used to refer to the group (i.e., 'SELF' for self auto-updating or 'COMPONENTS' for components auto-updating) for which the auto-updated information applies. The <options_list> is a list of auto update options as given by the <Update_AutoUpdateOptions> section. The <when> is a valid reminder time as given by the <when> construct. The <reminder_format> is one of the valid reminder formats listed in the <ReminderFormat> construct. The <installation_format> is one of the valid installation formats listed in the <InstallationFormat> construct.

The interfaces provided for this smart feature give the wrapper's owner full control on when and how the new updates of both the wrapper and its components are performed. There are also two interfaces provided to instruct the compositional wrapper to skip the next update. The SkipNextSelfUpdate tells the compositional wrapper to ignore the next new wrapper update completely. Similarly, the SkipNextCompo-

nentsUpdate requires the compositional wrapper to ignore the next new update of each component.

14.7.7 Testing

The testing smart feature is discussed in Section 8.2.9.7. The following are the two interfaces that will be provided in the generated wrapper:

StartPretending (“<currentusercode>”)
StopPretending (“<currentusercode>”)

Both of these interfaces require valid user authentication code. The <currentusercode> is a valid authentication code for the current wrapper user.

14.7.8 Security

The security feature is discussed in Section 8.2.9.8. The following is the only interface that will be provided for this smart feature:

LimitCallers (“<currentusercode>”, “<access_criteria>”)

The <currentusercode> is a valid authentication code for the current wrapper user. The <access_criteria> is a list of valid domain names or component IDs as given by the <Callers_Option> construct.

14.8 Specifying the Wrapping Options

During the wrapper generation phase, the generator will first present the system integrator with a list of wrapping options as specified in the `<WrappingOptionsDeclaration>` construct (see Section 8.2.8) of the wrapper specification. After giving system integrators the option to modify the list, the wrapper generator must include the necessary code to account for the selected options. The following guidelines should be observed both by the wrapper generator and by its users:

- The `EncryptWrapper` option will cause the wrapper generator to encrypt the generated wrapper's object or source code, depending on the implementation language. The wrapper generator would then save the encrypted code into a different file. Finally, it would generate a small shell (a wrapper) that will be responsible for doing the following:
    - Decrypt the encrypted wrapper's code.
    - Load the decrypted code (the actual wrapper) into a separate process or thread.
    - Pass control to that process or thread.
    - Terminate itself (i.e., the small shell.)

  To the wrapper user, the small shell will be the compositional wrapper. The shell's name will be the same as that of the compositional wrapper.

- The `EncryptComponents` option will encrypt all components at wrapper generation time. The compositional wrapper must have the necessary code to decrypt the components and to load them into their own processes or threads either during the wrapper instantiation or every time one of the wrapper components is called. When only encrypting the components, the small shell is not needed since the compositional wrapper will perform the tasks assigned to the small shell.

- The `EncryptAll` option will encrypt both the generated wrapper code and all of its components' files. A small shell (as discussed previously) will be needed to do the decryption and loading of the compositional wrapper and its components.

- Compression works similar to encryption. When the `CompressWrapper` or `CompressAll` is selected, the small shell, mentioned above, will also be responsible for doing the decompression of the wrapper and/or its components.

- The `FastLoading` option will use the *interfaces* table discussed in Section 14.9. The FastLoader routine will be responsible for copying the designated interface module into a separate process or thread and then passing control to it.

- The `PreserveMemory` option works similar to `FastLoading`. However, it even splits modules into smaller, but self-contained, chunks of code that will consume the least amount of memory.

- The wrapper generator must also include in the compositional wrapper the necessary code to enforce the selected wrapping options on every update of the compositional wrapper or of any of its components. For example, if the `EncryptAll` is

selected, then whenever the compositional wrapper or any of its components are updated, then the old compositional wrapper will first perform the update. The old wrapper will then encrypt the files for the new version, whether it is the compositional wrapper or one of its components. After that, the old compositional wrapper generates a new version of the small shell that takes the new update into account. The last step for the old compositional wrapper is to delete its own files (and possibly the old files of the updated components) and, then, to terminate itself.

14.9 Other Guidelines for Successful Generation of Compositional Wrappers

The following guidelines will need to be observed by the wrapper generator when generating the compositional wrappers:

- The wrapper generator will need to set the following three environment variables every time it loads a (or creates a new) wrapper specification:
    - `$GENERATOR_HOME` will contain the directory from which the generator is launched.
    - `$WRAPPER_TEMP` will contain the directory specified by the `<TempDirectoryDeclaration>` construct (see Section 8.2.7) in the wrapper specification. If such construct is missing, then `$WRAPPER_TEMP` will be set to equal "`$GENERATOR_HOME/$WRAPPER_NAME/temp`". The `$WRAPPER_NAME` environment variable is set to equal the compositional

wrapper's name when the generator parses the <NameDeclaration> construct in the wrapper specification.

- $WRAPPER_INSTALL will contain the directory specified by the <DestinationDirectory> clause (see Section 8.2.8) in the wrapper specification. If this clause is missing, then $WRAPPER_INSTALL will be set to equal "$GENERATOR_HOME/$WRAPPER_NAME/install".

- If the wrapper generator runs on a local machine, then it will not have problems saving its files (or files of acquired components and generated wrappers) on either local folders or remote ones. However, if the wrapper generator is accessed through the Internet, then it would be difficult to create local folders and save files into them. It still can save its stuff on remote locations. Therefore, the wrapper generator should make sure that any directory reference in the wrapper specification or the component specifications is accessible by the wrapper generator. The generator can perform such checks every time it parses any of these specifications.

- Any software component could assert its identity through the assignment of a unique identifier (conceptually similar to ISBN identifiers assigned to books). The unique identifier would accompany a component for its entire life and would provide a method for third parties to unambiguously refer to the component. Of course, the utility of a unique identifier would be seriously compromised if variations of a single component are allowed to carry the same identifier [MOOR94]. The wrapper generator must generate a unique wrapper ID for each generated

wrapper. This means that no two wrappers generated by ***any*** wrapper generator that conforms to the specification language discussed in Chapter 8 will have the same IDs. The <CallersOption> in the <Security_LimitCallers> smart feature (see Section 8.2.9.3) refers to this ID. This ID will be auto-generated when generating the wrapper and can be retrieved at later time by calling the GetWrapperID interface provided in the generated wrapper. The ID will consist of three parts put in the following format: "X-Y-Z." X is the wrapper generator's supplier number. It is a number similar to that used in UPC (Universal Product Code) codes or to the one used in published materials ISBNs. Y is a sequence number for the wrappers generated by this X supplier. Z is a timestamp designating when the wrapper was generated.

- To resolve naming conflicts that could occur among the selected components that will be included in the compositional wrapper or among their interfaces, this research proposes the following two solutions:
    - In the first solution, the wrapper generator creates the directory specified in the <DestinationDirectory> construct of the wrapper specification language (see Section 8.2.8.2.) The wrapper generator would then create a separate directory for each selected component. This will be a subdirectory with the component identifier as its name and descends from the <DestinationDirectory>. The generated compositional wrapper will contain an *interfaces* table to include all the interfaces of all encased components. The interfaces will need to be renamed in some way to

account for naming conflicts that may occur. One naming scheme could be to use uniform names, such as sequence numbers, to name each interface of each component. The first column in the *interfaces* table will include the new name of the interface as well as the interface signature. The second column will contain the original interface name. The third column will have the component's file name to which the interface belongs along with the folder's name in which the component was saved. The table contains two other columns that will be discussed in a later bullet. For any interface call, the compositional wrapper will first look up the called interface in the *interfaces* table to get the original interface name and the component's file name. After that, it restructures the call by replacing the called interface with its original name and prefixes it with the component's file name.

- The second solution is similar to the first one. The wrapper generator could use some naming scheme, such as the one described in the first solution, to rename all component files and then to put all the files in the directory pointed by the *destination_directory* clause. It can also use the same or similar naming scheme to rename interfaces. With this option, the generated wrapper will also need to use the *interfaces* table, discussed in the previous bullet. However, the third column of the *interfaces* table will contain only the new name of the component file.

The following snippet of code implements the name resolution issues:

```
ResolveNamingConflicts (wrapper) {
  component_names = [];
  names_count = 0;
  interface_names = [];
  interfaces_count = 0;
  file_names = [];
  files_count = 0;

  p = wrapper.selected_components;
  while (p <> nil) {
    // Resolve file names

    i = 0;
    cnt = 0;
    while (i < files_count) {
      if (file_names[i] == "COMP"+cnt) {
        cnt++;
        i = 0;
      }
      else {
        i++;
      }
    }
    if (i >= files_count) {
      files_count++;
      file_names[files_count] = "COMP" + cnt;
    }
    p.new_file = file_names[files_count];

    // Resolve component names

    // Check if the name of this component has been used
    // previously

    found = false;
    for (i=0; i<names_count; i++) {
      if (p.name == component_names[i]) {
        found = true;
      }
    }
    if (found) {
      i = 0;
      cnt = 0;
      while (i < names_count) {
        if (component_names[i] == p.name+cnt) {
          cnt++;
          i = 0;
        }
        else {
          i++;
        }
```

```
      }
      if (i >= names_count) {
        names_count++;
        component_names[files_count] = "COMP" + cnt;
      }
      component_names[i] = p.name + cnt;
      p.new_name = p.name + cnt;
    }
    else {
      component_names[i] = p.name;
      p.new_name = p.name;
    }

    // Resolve interface names

    q = p.provided_interfaces;
    found = false;
    for (i=0; i<interfaces_count; i++) {
      if (q.name == interface_names[i]) {
        found = true;
      }
    }
    if (found) {
      while (q <> nil) {
        i = 0;
        cnt = 1;
        while (i < interfaces_count) {
          if (interface_names[i] == q.name+cnt) {
            cnt++;
            i = 0;
          }
          else {
            i++;
          }
        }
        if (i > interfaces_count) {
          interfaces_count++;
          interface_names[interfaces_count].old = q.name;
          interface_names[interfaces_count].new = q.name+cnt;
        }
      }
    }
    else {
      interface_names[interfaces_count].old = q.name;
      interface_names[interfaces_count].new = q.name;
    }
  }
}
```

- The wrapper generator must include in the generated wrapper the necessary code to maintain the *interfaces* table. Each table entry should include the following:
    - The interface's new name along with the interface's original signature
    - The original interface name
    - The component's file name to which the interface belongs along with the folder name in which the component is saved
    - The offset (from the beginning of the component's file) where the interface's code resides. If the component's code is embedded within the generated wrapper's code, then this entry should hold the memory location at which the interface's code is loaded.
    - Performance rating of this interface, as described in the next bullet
- When the generated wrapper receives a call, it checks the *interfaces* table for interfaces that can service the call. If it finds more than one interface that can do the job, the wrapper invokes all of those interfaces. The generated wrapper would then pass to the caller the result it receives from the currently best-performing interface (or from the first interface that returns a result, if no performance statistics is yet collected.) For all interfaces that were invoked to service the current call, the compositional wrapper updates the performance-rating column of each interface (the last column in the *interfaces* table.)
- The generated wrapper must include a variable to hold its state (active or inactive.) This state variable will be managed by the Suspend/Resume interfaces in the *Li-*

censing smart feature and by the Pause/Resume interfaces in the *State* smart feature. The wrapper generator must also include (in the generated code) an IsActive interface that applications or components can use to query the current state of the generated wrapper.

- The wrapper generator must include the necessary code in the generated wrapper to allow the generated wrapper to check the hosting environment for consistency with the software, hardware, and network constraints that were imposed by the wrapper specification and by each component specification. The code should also be responsible for assuring that required interfaces or components are present when the component is instantiated. Similarly, the code should make sure that the properties described in the *not_present* and *not_loaded* software constraints are actually not violated. In case of violated constraints, the included code must provide the wrapper user with informative information regarding the violated constraints and could possibly suggest ways to overcome them (for example, listing the specification of the violated constraints.)

14.10 Generating the Compositional Wrapper

Once the system integrator enters the wrapping options, the generator collects the components and any associated information, wraps the components according to the selected generation options, and outputs the wrapper's source code or the wrapper's binary.

The generated code will include the data structure shown in Table 16. This data structure will hold information about the internal state of the compositional wrapper. The generated code will also contain the necessary utilities to: load this data structure into memory; update it with new or updated information and collected statistics; and save the structure back to permanent storage when the compositional wrapper becomes inactive. The smart feature interfaces (discussed in Section 14.7) use and update the information in this data structure.

Table 16: The compositional wrapper's internal data structure

ID The wrapper generator must generate this unique ID for each compositional wrapper. A discussion on the structure and the process of its generation is presented in Section 14.9.
Name A unique name (within the developed system) for the current compositional wrapper. The value of this field will come from the *name* clause in the wrapper specification.
Description A brief narrative description of the compositional wrapper and its role in the system. The *description* clause in the wrapper specification will supply the content of this field.
Owners_List The list of IDs of all of the compositional wrapper owners. The ones on the top of the list are the most recent owners. Several of the smart feature interfaces (discussed in Section 14.7) use this list to interact with the wrapper owners.

Developers_List The list of IDs of all original developers of the compositional wrapper. This list will be used to send reports to the compositional wrapper's developers.
Licenses A list to hold the licensing information of each encased component. For each component, the following information will be stored:

	Component_ID The name or ID (when a compositional wrapper is used as a component) of the component.
	Vendor_ID The vendor's name or unique ID.
	License_Type Holds the license type which can be any of: FREE, DEMO, TIMED, or METERED.
	License_Start_Date For DEMO and TIMED licenses.
	License_Expiration_Date For DEMO and TIMED licenses.
	Metered_Type Metered licenses can be of any of the following types: PER_USE, PER_USER, or PER_INSTANCE.
	License_Limit For METERED licenses. The maximum amount that a component's usage is supposed to reach. After this limit, the component will be inactive until the limit is reset by paying due amounts.

	License_Credit For METERED licenses. The amount of money paid to the vendor as available credit (positive amount) or owed by the client to the vendor (negative amount).
	License_Keys A list of license keys granted by the vendor to the client.
Statistics The compositional wrapper will include the necessary code to collect, store, and make use of the statistics about its operation.	
	Calls This list stores information about each call to the compositional wrapper or any of its components. Stored information would include the call's date and time, the caller's ID, the name of the called component and interface, the call's signature (i.e., interface name, passed parameters, etc.)
	Execution_Profile This list will keep a history of all calls to the compositional wrapper or to its components and the compositional wrapper's state at the time of the call. Each entry to *Execution_Profile* will contain the same information as that of *Calls* in addition to the current values of all of the attributes and state variables of the compositional wrapper and its components.
Active? This attribute is used if the suspend/resume smart feature (see Section 8.2.9.4) is selected for this compositional wrapper.	
Suspend_Message The message to be displayed when the wrapper is suspended.	
Resume_Message The message to be displayed when the wrapper resumes its operation.	

Interfaces

This structure represents the *Interfaces* table that is discussed in Section 14.9. The table will contain the following five columns:

Interface_Name

The interface's new name and its signature.

Original_Name

The interface's original name (i.e., the component vendor's assigned name.)

Component_File

The component's file name to which the interface belongs along with the folder name in which the component is saved.

Offset

The offset from the beginning of the component's file or its location in memory (if component's code is embedded within the generated wrapper.)

Performace_Rating

Collected performance statistics of this interface.

Required_Properties

The list of properties or conditions that must exist for the compositional wrapper to operate properly. The compositional wrapper will contain the necessary code to make sure that these properties exist when the wrapper is instantiated.

Forbidden_Properties

The list of properties or conditions that must *not* exist when the compositional wrapper is instantiated.

Schedule This list represents the periods when the compositional wrapper will be active. Outside these periods, the compositional wrapper will cease to deliver its expected functionality. See Section 8.2.9.5 for a discussion of the scheduling smart feature.	
	Active_Periods The list of periods during which the wrapper is fully operational. Each active period will contain the following information: ☒ *Start Time*. The time at which the compositional wrapper becomes active. ☒ *End Time*. The time at which the compositional wrapper becomes inactive. ☒ *Initialization Script* (to be executed at the beginning of each active period). The script will be responsible for restoring the system to its previous state, doing any necessary initialization of parameters or storage, and recording and updating licensing parameters. ☒ *Inactivation Script* (to be executed at the end of each active period). This script will be responsible for saving the wrapper's state, freeing any used memory or storage space, and terminating any running threads. ☒ *Inactive Script* allows the component or wrapper to respond to the outer world when the compositional wrapper is not active. During the compositional wrapper's inactive state, this script will perform any reporting or interaction mechanism with the wrapper's owner(s) and/or developer(s). The script will also perform checks for new updates, and download and install them when available.

The following snippet of code utilizes all the wrapper generation concepts that were presented in previous sections to finalize the wrapper generation process. The actual code for most of the included utilities is not yet implemented. The generator should keep the system integrator informed on the progress and results of the generation process. This should include the components being processed, the type of code generated, and the percentage of the process remaining.

```
GenerateWrapper (wrapper) {
  ResolveNamingConflicts (wrapper);
  CopyFiles (wrapper);

  // Generate code

  GenerateInitializationCode (wrapper);
  GenerateSmartFeaturesCode (wrapper);
  GeneratePerformanceMonitoringCode (wrapper);
  GenerateMaintenanceCode (wrapper);
  GenerateOtherCode (wrapper);
  WriteGeneratedCode (wrapper);
  CreateWrapperBinary (wrapper);
  EncryptWrapper (wrapper);
  CompressWrapper (wrapper);
  GenerateShellScript (wrapper);
}

CopyFiles (wrapper) {
  DIR = CreateDirectory (wrapper);
  p = wrapper.selected_components;
  while (p <> nil) {
    if (wrapper.encrypt_component) {
      Store (Encrypt (p.location + p.file), p.location + p.file);
    }
    if (wrapper.compress_component) {
      Store (Compress (p.location + p.file), p.location + p.file);
    }

    CopyFile (p.location+p.file, DIR+p.new_file);
    p = p.next;
  }
}
```

14.11 Handing the Compositional Wrapper to the System Integrator

Once the wrapper generator wraps the selected components, it outputs the wrapper, the component files, and all associated artifacts into the directory specified by the `$WRAPPER_INSTALL` environment variable. The wrapper generator could provide some means to help the system integrator in using the new wrapper. For example, the generator could provide a drag-and-drop or cut-and-paste capability, as provided in

OpenDoc [JACO97], to allow system integrators to plug the compositional wrapper in their system. The generator could also provide a link or a download capability so system integrators can access or move the generated wrapper to their development environment. The generator could, additionally, provide the right code to include the compositional wrapper in other applications.

14.12 Wrapper Testing

Component-based systems are known to suffer from a variety of problems in regards to their tracking and testing. Some of these problems are [GAO01]:

- ***Hard to understand component behavior in a system***. In system testing and maintenance, system testers usually have difficulty understanding and monitoring component behaviors in a system due to the following reasons:
    - Component developers use ad hoc mechanisms to track the behavior of their components, causing system testers to have problems understanding the behavior of a system due to inconsistent trace messages, formats, and diverse tracking methods of the encased components.
    - Usually, no or inappropriate built-in tracking mechanisms and functions exist in third-party components for monitoring their external behaviors. This could require testers to use other means of testing the component, such as fault-injection [VOAS98c] or mutation-testing [ZHU97].

- No configuration functions exist for component clients to control and configure the built-in tracking mechanisms.
- No systematic methods and technologies exist to control and monitor the external behaviors of the components.

- ***Difficulty in component error isolation, tracking, and debugging***. In a system, components developed by different teams may use different tracking mechanisms and trace formats. Thus, inconsistent tracking mechanisms and trace messages could cause difficulty in error detection and isolation of components.
- ***High costs in performance testing and tuning of components***. Performance testing for component-based programs is a major challenge in system testing because current component vendors usually do not provide users with any performance information. Thus, system testers and integration engineers must put a lot of effort into identifying performance problems and the components that cause these problems.
- ***Lack of system resource validation for components***. Since most components do not provide their system resource information, it is difficult for system testers to locate the system resource problems in system testing and maintenance.

Despite these difficulties, it will still be necessary for the system integrator to make sure that the generated compositional wrapper delivers the functionality stated in its specification. Of course, the system integrator would be the one to perform the actual

testing. However, the generator could help by generating a sample application that uses the generated compositional wrapper. The application would have interfaces that map directly to the wrapper's provided and required interfaces. The generator should be able to guess what kind of suitable interface (GUI, command-line, etc.) to provide to the system integrator by checking the environment in which the wrapper generator operates (for example, Unix versus Microsoft Windows.) The sample application that is generated here would be similar to the one that need to be generated when testing individual components during the detailed evaluation stage. However, in this case, the generator could include in the application code any testing mechanism that can help the system integrator in testing the generated wrapper. For example, the generator can include the *Pretend* testing smart feature and some of the other tracking smart features.

CHAPTER 15: BEYOND THE WRAPPER GENERATION PROCESS

Table 17 shows a detailed list of activities that concern compositional wrappers but occur after the end of the composition process. Since the activities shown in Table 17 are beyond the focus of this dissertation, no attempt was made to classify the possibility of automating the activities that are shown in this table.

Table 17: Detailed activities breakdown of some of the tasks related to compositional wrappers that take place after the composition process

0.0 Develop and operate a compositional wrappers-based (CWB) system
1.0 Design the CWB system (Chapter 9 and 9.5)
2.0 Build the CWB system (Chapter 10 to Chapter 14)
2.1 Collect components (Chapter 10 to Chapter 13)
2.2 Wrap components (Chapter 14)
2.3 Deliver compositional wrapper (Section 14.11)
2.4 Assemble components into the CWB system
2.4.1 Train the staff needed to perform the system assembly
2.4.2 Review the system design
2.4.3 Plug components into the CWB system
2.4.4 Develop the necessary code to glue the system components together
2.5 Test the CWB system
2.5.1 Train the staff needed to perform the system testing
2.5.2 Prepare test plans
2.5.3 Acquire, install, and test other components or software that are necessary to perform the system testing
2.5.4 Develop glue code or other software that is needed to perform the system testing
2.5.5 Perform regression system testing
2.5.6 Perform integration system testing
2.5.7 Perform acceptance system testing
3.0 Operate the CWB system
3.1 Maintain the CWB system

			3.1.1 Perform needed updates				
				3.1.1.1 Train the staff that will perform the new updates			
				3.1.1.2 Obtain the specifications of the new updates			
				3.1.1.3 Evaluate the new updates[78]			
				3.1.1.4 Evaluate the risks associated with assessing, acquiring, installing, and testing the new updates			
				3.1.1.5 Perform any required reassessments[79]			
				3.1.1.6 Perform any necessary system redesign and data structure rework due to eliminating backward compatibility with other versions or components			
				3.1.1.7 Acquire the new updates[80]			
				3.1.1.8 Develop any necessary reworked or new glue code			
				3.1.1.9 Install the new updates[81]			
				3.1.1.10 Configure the new updates[82]			
				3.1.1.11 Evaluate and perform any required cascaded upgrades[83]			
				3.1.1.12 Perform any necessary data migration or data conversion due to performing the updates			
				3.1.1.13 Test the new updates[84]			
			3.1.2 Perform any required system maintenance				
				3.1.2.1 Evaluate the required maintenance			
				3.1.2.2 Train the staff that will perform the system maintenance			
				3.1.2.3 Evaluate the risks associated with performing the required maintenance			
				3.1.2.4 Perform the required maintenance			
					3.1.2.4.1 Perform any necessary system redesign and data structure rework due to performing the required maintenance		
					3.1.2.4.2 Evaluate, acquire, install, and configure any required components to correctly perform the required maintenance		
					3.1.2.4.3 Evaluate and perform any cascaded upgrades that are necessary to perform the required maintenance		
					3.1.2.4.4 Develop any necessary reworked or new glue code		
					3.1.2.4.5 Perform any necessary data migration or data conversion due to performing the maintenance		
					3.1.2.4.6 Test the system after completing the required maintenance		
		3.2 Monitor and control the CWB system					
			3.2.1 Track, evaluate, and correct abnormal events				
				3.2.1.1 Conduct any necessary personnel and user training needed to track, evaluate, and recover from abnormal events			
				3.2.1.2 Evaluating the risks associated with the occurrence of the abnormal event			

[78] Evaluation activities would include the preliminary evaluation (activity 2.1.1.2.1), the detailed evaluation (activity 2.1.1.2.4), the needed negotiations (activity 2.1.1.2.5), and activities 2.1.1.2.7, 2.1.1.2.8, and 2.1.1.2.9.

[79] These could include reassessing the system's design, reassessing its components, or reassessing relationships with existing and emerging vendors and technologies.

[80] The acquisition activity will include all the sub activities of activity 2.1.1.3 except sub activity 2.1.1.3.1 (i.e., Sending Reservation Information.)

[81] This would include the same sub activities as in activity 2.1.1.4.

[82] This would include the same sub activities as in activity 2.1.1.5.

[83] This is the upgrading of other components or related software in order for the new component updates to function properly.

[84] This would include testing new features and bug fixes and performing full system integration testing.

				3.2.1.3 Perform necessary system redesign and data structure rework due to the occurrence of the abnormal event
				3.2.1.4 Locate, assess, acquire, install, configure, and test any needed replacements or upgrades to components affected by the abnormal event
				3.2.1.5 Develop any necessary reworked or new glue code
				3.2.1.6 Perform any necessary migration or conversion of data affected by the abnormal event
				3.2.1.7 Test the complete system
			3.2.2 Track, evaluate, and react to recurring events	
				3.2.2.1 Perform any necessary system enhancements
				3.2.2.2 Track and collect system data
				3.2.2.3 Perform configuration management activities
				3.2.2.4 Evaluate the complete system
				3.2.2.5 Evaluate component vendors
				3.2.2.6 Evaluate interactions with existing vendors
				3.2.2.7 Interact with related organizations
				3.2.2.8 Renew licenses
				3.2.2.9 Conduct market research for new technologies and techniques
				3.2.2.10 Conduct any relevant technology forecasting
				3.2.2.11 Conduct any necessary technology refreshments
				3.2.2.12 Conduct any necessary cultural changes or staff retraining

The previous chapters gave a complete presentation to the concept of smart compositional wrappers. Those chapters started by discussing the CSL and WSL specification languages that allow specification writers to formally specify the functionality of pre-built components and compositional wrappers. Then, a complete composition process was laid down with the aid of a smart wrapper generator. That journey ended with generating compositional wrappers.

However, most compositional wrappers are not intended for stand-alone use. Compositional wrappers will eventually be assembled into complete systems or higher-order compositional wrappers. Therefore, this chapter will briefly introduce the activities that are performed on such systems. These activities are beyond the scope of this dissertation. However, introducing them will help the reader to visualize the role that compositional wrappers could play in the life of CWB systems. Additionally, wrapper generators

can still help system integrators and maintainers in many aspects of these activities. During the discussion of each activity in the following sections, such opportunities of making use of the specification languages or the wrapper generator will be indicated.

15.1 System Assembly

Assembly of components is normally guided by using a component infrastructure that provides the binding to form complete systems from disparate components [MEHT00]. With wrapper generators, system assembly is no more than building another compositional wrapper (which is the complete system.) System integrators can specify the complete system using the wrapper specification language (WSL) similar to specifying single compositional wrappers. The building blocks of the complete system will be compositional wrappers that system integrators have generated using the wrapper generator, as well as other pre-built components. The composition process that will be used to assemble complete systems is similar to that of building compositional wrappers. However, assembling complete systems might require activities that are different from the ones of generating compositional wrappers.

15.2 System Testing

Testing a complete system, in many respects, is no different than testing compositional wrappers. Most of the issues presented and the techniques outlined in Section 12.3 to test compositional wrappers apply equally to testing complete systems. In contrast to testing components, complete systems are usually tested on actual users and live data. Additionally, there is no need to wrap the complete system during the testing process since the complete system forms a stand-alone product.

15.3 Maintaining CWB Systems

Despite all advances in software technologies and practices, software must often be changed to repair bugs and add new functions. Post development costs exceed the actual development costs [BASI01]. However, the promise of component-based software development is that larger, more complex systems can be built reasonably quickly and reliably from pre-fabricated ("off-the-shelf") building blocks. Such systems can be upgraded incrementally, simply by replacing individual components with their new versions.

Maintenance of components-based systems has changed in fundamental ways from traditional maintenance methodologies. It is no longer solely concerned with fixing existing behavior or incorporating new mission needs [BROW98b]. Vendors update their products on their own schedules and at differing intervals. In addition, a vendor may elect to eliminate, change, add, or combine features for a release. Updates to one product, such as new file formats or naming convention changes, can have unforeseen conse-

quences for other products in the system. Furthermore, COTS products require continual attention to license expirations and changes. Consumers must identify potential technologies and products, qualify candidates for fit within their system, and perform tradeoff analysis between competing technologies and products. An organization must continuously perform the entire process of monitoring, evaluating, qualifying, and analyzing the impact of technology and products given the constant changes within the commercial marketplace.

Despite its involvement and difficulty, component upgrades and maintenance can be made easy if one understands the following [BOEH99]:

- Consumers have no control over the evolution of their COTS components. The overall market needs and trends direct what new features to add and when.
- Tightly coupled components cause big problems to the products that are using them. Therefore, one should try to reduce the interconnections among such components.
- To offset costs and risks associated with long-term component evolution, it is necessary to stick to open commercial standards. Such standards make component evolution and maintenance more manageable.
- Before committing to specific vendors, one should try to evaluate their product evolution predictability record. Widely varying feature sets, too-frequent updates, and dramatic shifts in product capabilities can cause problems in the long term.

- It is always a good idea to establish a proactive system-release strategy by synchronizing performing component upgrades with new intended releases of the compositional wrappers or the complete system. Such synchronization will ensure that the same system continues to function harmoniously.

15.3.1 Component Updates

Off-the-shelf components are volatile by nature. *Volatility* [ABTS00d] in this context refers to the frequency with which new versions or updates of the components software being used in a larger system are released by vendors over the course of the system's development and subsequent deployment. Organizations or systems external to the organization that is operating the CWB system can require upgrades to components or software used in the CWB system. Examples of this include upgrades to commercial or government software incorporated into the CWB system and upgrades to legacy systems to which the CWB system must interface.

Practice has shown that, while it may improve certain aspects of an existing component, a new component version frequently introduces unforeseen problems [FRAK00, SEGA93, BASI01, JONE98b, CARN00]. Such problems would include:

- Vendors are market-driven in their upgrades, focusing upon additional features to attract new customers and fixes to problems encountered by their principal cus-

tomer base. Vendors may not be willing to fix problems experienced by only a few customers, even if the customer is willing to pay the vendor to fix the product.

- Because of time-to-market pressure, vendors perform limited testing on their software upgrades; especially regression testing of supposedly unchanged features.
- External evidence, such as change in version numbers or reports about fixed bugs in the new version, is not enough to assure users that the new upgrade is free of bugs. Fortunately, in compositional wrappers, the generated code to perform the automatic upgrading of components (if such smart feature was selected) will rely on the formal specification of the new upgrade and not on other external manifestations. The included code will be responsible for comparing the new update with the old version and for selecting the best version that matches the user's requirements. Additionally, if the load-balancing smart feature was enabled in the CWB system, then the two versions of the component could be running at the same time and the best-performing version will be selected by the included glue code. When multiple versions of the same component run at the same time, *configuration management* [CARN00, FRAK00] practices might need to be observed.
- In addition to introducing bugs in previously working capabilities, upgrades may decrease performance, increase computer resource utilization, and introduce incompatibilities with other software products.
- Upgrades may eliminate backward compatibility with previous versions, possibly necessitating design and data structure rework.

- Upgrading a COTS-based software system means that as new releases of the commercial components are made by the various vendors, the system may incorporate them [CARN97]. Thus, different pieces of the system will be upgraded at widely varying intervals; licenses will need to be revalidated for different parts of the system at random intervals. This means that a system with several commercial components has a very heavy dependence on various release cycles of the components vendors. Few solutions to this problem are emerging. For example, the SOFA (SOFtware Appliances) project [HOA98] aims at developing software packages that are composed of several components. Each package provides some computational functions and could be updated independently on other components. Compositional wrappers offer a better approach (using the auto-update smart options) by scheduling and synchronizing the compositional wrapper updates and the updates of its components at specific dates and/or times.

- In the same way as the initial software delivery is risky if performed via an insecure channel, software updates can be modified in transit or system owners can be fooled into installing fraudulent updates.

- Upgrades to one or more components that are not compatible with the rest of the system can introduce incompatibility problems. This could occur even though the components are still reliable when taken in isolation. Incompatibility occurs when the component vendor upgrades its components according to their biggest customers. Eventually, the fixes or changes in the new upgrade might not align properly with the rest of the system.

- The pressure to bring new features to market quickly may cause vendors to drop or slip other promised features, fixes, or upgrades for particular platforms.
- Sometimes, needed upgrades are delayed because of dependencies between software products (e.g., vendors waiting for the next operating system upgrade before issuing their next major upgrade).
- Because vendors support only a limited number of past releases, delaying implementation of upgrades can result in unsupported versions of software products in the operational system. When this happens, vendors will not provide fixes to bugs and will not provide consultation services. Another factor influencing the need to maintain currency with software upgrades is the limited life span of computer hardware platforms (e.g., workstations and servers). Most organizations plan on hardware upgrades every four to five years during sustainment. Maintaining currency with software releases is essential for upgrading to new hardware since it is not usually possible or desirable to execute old versions of the operating system and other software on new hardware platforms.

When performing an upgrade of an existing component, one can encounter three possible relations between the old and new versions of the component [RAKI00]:

(1) the new version provides the same functionality, but better performance, security, reliability, and so forth;

(2) the new version also provides additional functionality, but the basic approach for providing that functionality is unchanged; and

(3) the new version's functionality has some overlap with the old version's, but the implementation approach is completely different (e.g., a novel algorithm or more efficient data structure).

Every new software release requires a full application of most of the activities of the composition process to properly incorporate the new release into the developed system. For example, each new release can require performing the following activities [ADAM01]:

- Regression testing and prototyping to determine the behavioral characteristics of the new release as compared to the previous release and the compatibility of the new release with other software components that are currently running in the CWB system.
- Testing of new features and bug fixes. [JONE98b] reported that projects that use formal inspections plus formal testing average over 95% in defect removal efficiency while projects that only use formal testing are invariably below 90% in defect removal efficiency.
- Modifications to glue code and user interfaces. For example, if the CWB system was assembled using the wrapper generator described in this dissertation, then the CWB system would need to update its *interfaces* table (see Section 14.9) with every new release of the compositional wrapper or the encased components.
- Modifications to the structure and content of the components' database or files.

- Full software and system integration testing and requirements verification.
- Training for both the software developers and the system operators and users.

In order for the compositional wrapper to be able to perform automatic maintenance and updating of the components it encases, the wrapper's glue code should put several issues into consideration. New wrapper updates should NOT:

- Require any new hardware, operating systems, or network protocols.
- Degrade system performance or reliability.
- Break security measures.

On the other hand, each new update of the compositional wrapper MIGHT:

- Add new interfaces in the wrapper's glue code.
- Remove unnecessary interfaces.
- Rearrange required/provided interface connections.
- Allow more than one version of the same component to run concurrently.
- Replace multiple components with a single or fewer new ones.
- Inform other components within the compositional wrapper of new changes performed during the update.

15.3.2 Maintaining CWB Systems

Maintenance is expensive. Maintenance costs can easily exceed half of the total costs for a software project [FRAK00]. Using third-party components, immediate short-term gains are usually lost in the complicated long-term post-deployment maintenance. When dealing with multiple vendors, it is very likely that each vendor will suggest that the fault lies with some other vendor's component and not with their own component.

What makes isolating and correcting faults in components-based systems more difficult is that [HISS99]:

- Debugging no longer centers on finding erroneous code (i.e., 'bugs'), but focuses on observing the behavior of large, opaque components and making inferences from that behavior.
- The person who does this 'debugging' is not the person who created the component and knows its internals, but rather is the system integrator (or the maintenance or test engineers) who combined it with other components into the complex system.
- The outcome of the integrator's 'debugging' is not correcting source code, but instead is determining whether, and how, the component's unexpected behavior can be reconciled with the desired behavior of the whole system.
- Finally, even when the source of failure is isolated in a commercial component, getting the vendor to take notice and repair the fault may prove difficult or impossible. Many system maintainers have found themselves at an impasse because a

vendor is unable or unwilling to divert resources, change release schedules, or accommodate requests for corrections or improvements to the vendor's component.

Maintaining compositional wrappers is more complicated than maintaining individual components because of [VOAS98a]:

- Defective, unreliable components.
- Incorrect, incomplete, or ambiguous information in the specification of the component as to what it truly does.
- Incorrect, incomplete, or ambiguous information in the system requirements as to what each component is expected to do.
- Hidden component interfaces that the user does not know about.
- The differences between the competing standards (e.g., OMG's CORBA, Microsoft's OLE and COM) that each component is based on.
- Complex or defective wrapper's code.
- Legal restrictions, such as the Universal Commercial Code (UCC) that limit components' users ability to investigate and interact with all aspects of commercial software.

When a system that is composed of independently developed components fails, clients usually want to have a way to find out which vendor's software has failed. If a

component's identity (boundary) could not be clearly determined, it would become very difficult or even impossible to determine which vendor was the culprit. Fortunately, many existing tools and techniques can help in mitigating such problems. The choice of tool or technique is based on the kind of component and the type of suspected failure. For example, on-the-fly software replacement [HAUP96] can help in doing maintenance in a running system with minimal interruption of service. With this technique, repairing faulty software requires only, in most cases, replacing one or very few software units.

A method that has been in use for understanding and correcting system failure in software systems derives from the classic scientific method of observation, hypothesis, and prediction [HISS97]. This method defines a systematic approach for observing cause and effect (see Figure 26.) A hypothesis is formed based on one or more observations. To test that hypothesis, a prediction is made relative to some stimulus and its anticipated effect. An experiment is then designed to test the prediction. Results from the experiment will either support or contradict the hypothesis. If the hypothesis is not supported, new observations, refinement of the hypothesis, or subsequent experimentation are needed.

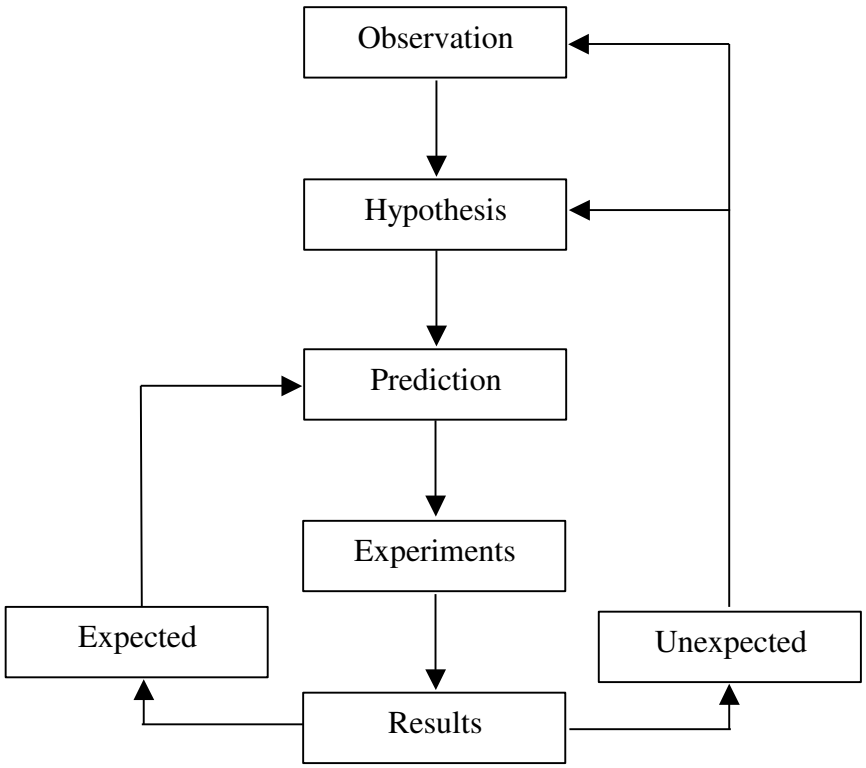

Figure 26: The scientific method

The assumption of the scientific method is that there is something to observe. However, in dealing with off-the-shelf components, their opaqueness means that while their external *behavior* can be observed, their inner workings cannot, and little or no insight into the components' underlying built-in assumptions or design constraints is available.

15.4 Monitoring and Controlling CWB Systems

During the operational life of CWB systems, organizations often require to perform activities that seem not directly related to the CWB system. However, in reality, such activities are essential for the longevity and maintenance of a healthy operational life of the CWB system. The following two sections present two categories of the activities that are usually carried out throughout the life of CWB systems. The occurrence of abnormal events (Section 15.4.1) forces organizations to react to such events in the correct manner. Recurring events (Section 15.4.2) necessitate performing certain activities on a regular basis. Such activities, though usually not pressing ones, can have great positive impact on the whole organization.

15.4.1 Abnormal Events

As with any component-based system, CWB systems are vulnerable to various forms of abnormal events. Abnormal events are those events that usually occur without the organization's prior knowledge and force the organization to react to them. Despite the fact that many of these events are usually external to CWB systems, abnormal events can affect CWB systems in various aspects. For example, changes in vendors' policies regarding the licensing and support of their components could force CWB system owners to redesign (or at least reevaluate) their operational CWB systems. Such reevaluation and redesign actions would usually result in interruptions of the CWB system users and in costs to the CWB system owners. The following is a list of some of the abnormal events that are relevant to CWB systems.

- *Product termination*. Products could be discontinued for various reasons. For example, vendors could go out of business or drop support to their old products. Clients in these situations are forced to discontinue using such products and to look for substitutes. Products could also be discontinued when new technologies or hardware emerge in which case clients are forced to change to such new technologies or hardware. Such changes could force clients to discontinue using some of their products due to their obsolescence or inadequacy.

- *Product replacement*. Components may need to be replaced or added at any time due to factors such as: elimination of vendor support, divergence from system needs, identification of unacceptable limitations or vulnerabilities, increased costs for licenses or support services, and new or modified user needs requiring changes in functionality or performance. Incorporating new components usually requires the latest version of the operating system and of other related commercial software to be in place. There are also situations when the vendor of one or more of the components in a CWB system decides to replace such components with different, and hopefully better, components. In such situation, the client might decide to replace the old components with their newer versions. Alternatively, the client could decide to replace the component with a completely different one and possibly from a different vendor. In all these cases, the client must go through the whole process of evaluating the new components, testing them, configuring them, and plugging them in the system. The client might also be forced to remove other unneeded (or

malfunctioning) components or software that can cause clients to lose license benefits due to their decisions of discontinuing to use such licensed products.

- *Dropped support for a product*. Vendors may go out of business, merge with, or be acquired by other companies. Vendors may also drop or de-emphasize products or hardware platforms, usually without warning. Vendors may also change the type and quality of their provided customer support. In particular, new vendors trying to gain a market position may initially be very responsive but may lose their responsiveness after establishing a larger customer base.
- *Marketplace and technology changes*. Organizations are sometimes forced to reevaluate or even redesign their running systems because of new changes in the marketplace or because of the introduction or change in technologies.
- *Changes to license agreements*. Fees and fee structures of licenses and services may change without warning potentially resulting in a large cost impact if changes occur after the developer's commitment to a particular software product [AD-AM01].

During the normal operation of CWB systems, maintainers of such systems need to carry various activities with respect to abnormal events. Such activities will help reduce the impact of such events on the operation of CWB systems. Some of the activities that organizations can perform would include:

- Training the staff needed to track, evaluate, and recover from abnormal events.

- Preparing recovery plans for the various types of events.
- Monitoring the operation of CWB systems and becoming alert of any signs of occurrence of any of the abnormal events.
- Continuous lookup for new vendors, technologies, and products that will help in reducing the number and effect of abnormal events that could occur in CWB systems.

When an abnormal event occurs, it requires various forms of actions to be taken by the owners and maintainers of CWB systems. Some of these actions include:

- Evaluating the abnormal event.
- Evaluating the risks associated with the occurrence of the abnormal event.
- Locating, assessing, and acquiring replacements or upgrades to components affected by the abnormal event.
- Obtaining licensing for the new replacements or updates.
- Configuring the new replacements or updates.
- Testing the new replacements or updates.
- Performing any needed rework to existing system glue code or developing additional new code to account for the new replacements or updates.
- Integrating the new replacements or updates in the CWB system.

- Performing any required cascaded upgrades, such as upgrading other components or related software that are affected by the new replacements or updates.
- Performing any necessary evaluation or redesign of data structures that are affected by the elimination of backward compatibility in the replaced or upgraded versions of components.
- Migrating and/or converting any data affected by the occurrence of the abnormal event or by the introduction of the new replacements or updates.
- Testing the complete system.

15.4.2 Recurring Events

Throughout the operational life of CWB systems, there are many activities that can take place and which are necessary to sustain the operation and longevity of such CWB systems. These activities are not triggered by events that act upon the CWB system, but rather continuous in nature. Most of these activities involve groups, departments, or even whole organizations. The following is a list of some of these activities:

- *System enhancement activities*. These include adding new features and enhancing the performance and/or the presentation of existing features. Such enhancements would require acquiring new components, upgrading existing components, or modifying the system's glue code. System enhancements would also include modify-

ing the system's glue code to account for new technologies, techniques, equipments, tools, and changes to standards or business rules.

- *Project tracking and data collection*. Activities in this area would include collecting and analyzing operational traces, tracking of performance and quality measures of the system components, and tracking of actual system operation and maintenance costs versus estimated ones and calibrating any cost estimation models used.

- *Configuration management activities*. Such activities could include *version control* mechanisms, *change control* procedures, and *build control* tracking to make sure that the right version of each component is upgraded with the correct version of other components or software products. Using third-party components, configuration management needs also to put under control the middleware in which the component will be running and all the tools (architecture and design, modeling, requirements management, testing, and most of the tasks of project management) that are related to component products [VIGD98].

- *System evaluation and assessment*. Organizations will not use their CWB system forever. There will come times when the current system (or parts of it) will need to be replaced by a better performing or a more cost-effective one. Decisions on such actions do not happen overnight, but rather because of continuous assessments.

- *Vendor evaluation and reassessment*. It is necessary that organizations keep good profiles with their vendors. At the same time, organization's loyalty to its vendors

is kept as long as the vendor is delivering the expectations set forth by the organization. Failure to meet such expectations could force the organization to hunt for better deals and better relationships with other vendors.

- *Evaluating interactions with existing vendors*. The teams that [MORI00] studied expected that vendor interactions would be simple and would end with the purchase of the component. However, the interaction continued throughout the life cycle of the development project and the flow of information was not merely one way. When a components-based system fails, the initial tendency is to rely on the components' vendors to provide a solution. In most cases, the wrapper user (or the wrapper owner) has the task of proving to one vendor that the at-fault component is the one provided by that vendor. Client-vendor interactions could include reporting bugs or dissatisfactions with the system or its components; relationship-building meetings, visits, and conversations; conducting surveys about the system or the vendor and which are completed by the client on behalf of the vendor; and presentations, meetings, or conferences to report new products or enhancements, or to solicit recommendations for new products or enhancements. [ABTS98] noted that small vendors often lack the capability to support their customers, while very large vendors have the capability, but not the motivation.

- *Interacting with related organizations*. Such organizations would include other teams using the same or similar components that are in the CWB system, independent evaluation teams, and other customers of affiliated vendors. Reasons for such interactions could include searching for and exchanging technical tips or

hacks about the used components; reporting, propagating, or hunting for good or bad experiences about the used components; and locating published evaluation materials for use with future system and vendor evaluations.

- *Renewal of owned licenses*. Activities in this area would include the tracking of licenses that are used in the system, contacting vendors for license renewals, filling and submitting any required forms, submitting payments, acquiring new license keys, and updating the renewed components with the new license keys.

- *Conducting market research for new technologies and techniques*. This type of research is sometimes referred to as product research or NDI/COTS research [KENN00]. The results of product research will usually result in introducing new technologies or techniques, or updating existing ones.

- *Conducting any relevant technology forecasting*. With the ever-changing technologies, it is important for organizations to keep pace with new and enhanced technologies. Lagging behind in conceiving and adopting such technological changes will only increase costs of running and maintaining the CWB systems. It will also give better chances to competitors to rise and advance. For proactive organizations, it is important to foresee scope and magnitude of such changes and to prepare plans and budgets for assessing and adopting them when the changes occur.

- *Conducting any necessary technology refreshments*. Technology refreshment can be defined as the periodic replacement of processors, displays, computer operating systems, and other software that are needed to assure continued supportability of

the host system through an indefinite service life. Conducting such technology refreshments could cause great interruptions to users and system operations and redesign of complete systems. Therefore, careful planning and risks analysis must be conducted before doing such refreshments.

- *Conducting any necessary culture changes or staff retraining*. A healthy organization is the one that not only maintain currency with technology, but also with attitudes and knowledge of their staff. Adopting any new or enhanced technology requires from organizations to first sell such technology to their people. It also requires changing the work culture to make it easy for the technology to penetrate to the hearts of the stakeholders. Since sharp transitions in culture changes are neither welcome nor beneficial both to the organization and to its stakeholders, it is very important to make such changes as smooth and unnoticeable as possible.

CHAPTER 16: RETURN ON INVESTMENT

The emphasis throughout this dissertation was on the automation of the activities involved in the process of assembling and generating compositional wrappers. Each of the chapters that discussed the composition process (Chapter 10 to Chapter 14) identified the activities in the respective stages of the process that can be automated and the expected level of automation. The goal of automating such activities was to reduce the cost, effort, and schedule of the composition process. This chapter demonstrates some aspects of possible savings (in cost, effort, or schedule) in return of automating the various activities of the composition process.

16.1 Potential Savings in Auto-Generating the Compositional Wrapper's Glue Code

Section 14.4 presented a technique for estimating the size of glue code in compositional wrappers. There are many benefits of having a handy estimate of the glue code size. For example, the following equation illustrates the savings in auto-generating the glue code of compositional wrappers. The savings are calculated by estimating the amount of money that will be paid to wrapper developers if the developers will write the glue code manually:

Glue Code Developers Cost (GCDC) = (WGS / AVDP) * AVDC

WGS is the estimated glue code size (see Section 14.4.2) of the compositional wrapper. AVDP is the average developer productivity of glue-code SLOC (Source Lines Of Code) in one person-month. One can estimate such a number from historical project data or from published studies. For example, [BASI01] reported that the median productivity rate for the 20-project COCOTS database is about 100 glue-code SLOC per person-month. On the other hand, the median applications productivity rate for the 161-project COCOMO II database is about 270 SLOC per person-month [BASI01]. Finally, AVDC is the average developer's cost (per month). This would include salary, incentives, training costs, etc.

One can illustrate the utility of the above equation by using the following published statistics and by assuming that the used wrapper generator will automate most of the activities involved in the wrapper code-generation stage (as discussed in Chapter 14). The study at the Flight Dynamics Division at the Goddard Space Flight Center of NASA that was reported by [MORI00] included 15 COTS projects utilizing 30 different COTS products representing a variety of domains. One of the projects used as many as 13 COTS products. The code developed (glue code and code for new components) ranged from 29 to 300 KSLOC. Out of the 20 projects whose historical data is available in the COCOTS database, [ABTS00d] reported that the glue-code size (in SLOC) of 10 COTS components was below 10K, between 20K-40K for six other COTS components, and 100K-500K for two more COTS components.

Using the above statistics and assuming that:

- the size of the glue code is *only* 10K SLOC;
- the developer's productivity is only 100 glue-code SLOC per person-months; and
- the cost of each developer is $1000 per month,

the cost to pay developers to write the glue code of one compositional wrapper can be estimated as follows:

WGS = 10K SLOC = 10 * 1024 = 10240 SLOC

AVDP = 100 glue-code SLOC per person-month

AVDC = $1000 per month

Glue Code Developers Cost = (10240 SLOC / 100 SLOC) * ($1,000) = $102,400

This means that, using conservative figures, organizations, such as those studied by [ABTS00d], will save around $100,000 dollars for each auto-generated compositional wrapper. Note that this saving is only for the developers' cost and using very conservative figures. Code generation involves other activities (such as preparing compilers and other tools, consulting manuals and documentation, collecting code fragments, and debugging code.) Automating the complete code generation process would eventually make the savings even greater. Additionally, the wrapper generation process is only one stage in the composition process. [BOEH02] reported that glue code development costs, on average, amounts to only about 31% of the total composition process costs. Since the wrapper generator automates many activities on all the other stages of the composition

process, using the wrapper generator that is proposed in this dissertation could save an organization around ($102,400 / 31%) > $300,000 for auto-generating compositional wrappers that have similar characteristics as those stated in this section. The following section presents a technique that can be used to estimate the costs (or savings if automated by the wrapper generator) of the other stages in the composition process.

16.2 The Cost of Assembling and Generating Compositional Wrappers

This section presents a technique that can be used to estimate the cost of assembling and generating compositional wrappers. The purpose of presenting this technique is to illustrate the applicability of the concepts, techniques, and algorithms that are outlined throughout this dissertation. Other useful techniques and procedures can be built in a manner similar to the one shown in this section.

This technique uses the various constructs in the CSL and WSL specification languages that can help in obtaining a rough estimate of the total costs for assembling and generating compositional wrappers. The technique also uses published reports on estimating the cost of software systems to augment the information that can be obtained from CSL and WSL. The estimation process is outlined in Table 18. Each row of this table represents one step in the estimation process. The left column in this table presents the parameters that are obtained in each step of the process. Subsequent steps build on values obtained in the steps that precede them.

Table 18: Calculation of the total costs of assembling and generating one compositional wrapper

Total Compositional Wrapper Cost (TCWC)	
RC	Start by estimating the number of required components to satisfy the compositional wrapper's specification. This estimation can be done using the algorithm of Section 12.7.2. For the purpose of this calculation, *CPI* and *CRI* in the algorithm of Section 12.7.2 must be set to 0.
LC	Calculate an estimate of the licensing cost of all the estimated *RC* components that will be encased in the compositional wrapper. The following equation can be used for such estimation: LC = RC * LicensingCost where RC is obtained in the previous step and LicensingCost is obtained directly from the wrapper's specification. If the licensing cost for single licenses is specified in the wrapper specification, then the value assigned to the *Single.LicenseCost* clause can be used as an estimate of LicensingCost in the above equation. If, on the other hand, the *Site.LicenseCost* or the *Developer.LicenseCost* clauses are specified in the wrapper specification, then the value assigned to such clauses can be used directly as an estimate of LC in the above equation. If the wrapper specification contains licensing costs for two or more of the licensing models (single, site, and developer), then *LC* can be calculated for each used model (as discussed above) and the calculation with the highest value is taken as the value for *LC*. This gives a worst-case estimate of the licensing cost. Note that other costs that have clauses in the wrapper specification (for example, maintenance costs, installation costs, and upgrade costs) can be added to the above equation.
WGS	Use the algorithm of Section 14.4.2 to estimate the glue code size of the compositional wrapper. It is assumed that the organization using the technique outlined in this section has already obtained estimates for *GCEF* and *BWCS* that are required to calculate *WGS*.
GCDC	Use the procedure of Section 16.1 to estimate the cost of developing the glue code. Though the glue code will be auto-generated, use this estimate of developers cost to estimate costs of other activities in the composition process (i.e., assessment, acquisition, adaptation, etc.).

TCWC	Use the glue code development cost (obtained in the previous step) and published data to estimate other costs that are incurred in assembling and generating compositional wrappers. For example, considering all the effort incurred in the process of integrating and operating one COTS component, [BOEH02] reported that the component assessment takes on average around 20% of the total effort of the complete process. Additionally, tailoring (i.e., adaptation) takes on average around 21% of the total effort and glue code development takes on average around 31% of the total effort. These statistics can be used to estimate the cost of the various stages of the wrapper composition process. Ignoring other costs (for example, costs of searching for, locating, and acquiring components), one can use the following equation to estimate the total cost of assembling and generating one compositional wrapper: Total compositional wrapper cost = Licensing costs + RC * [Assessment costs + Tailoring costs] + Wrapper generation costs Assuming that all the activities in the wrapper-generation stage will be automated, the wrapper generation costs can be ignored. In addition, the glue code developers cost (*GCDC*) is assumed to represent 31% of the total cost of locating, evaluating, acquiring, and wrapping one component. It is also assumed that *GCDC* is the average wrapping cost of every *RC* component. The following equation is based on these assumptions: Total compositional wrapper cost = LC + RC * [(GCDC * (20% / 31%)) + (GCDC * (21% / 31%))]

Looking at the current component marketplace, one can notice that component prices range from few dollars to hundreds of dollars. Prices for custom-built or complex components can be many folds higher. To illustrate the utility of the estimation technique that is outlined in Table 18, assume one wants to estimate the cost of assembling and generating one compositional wrapper that encases only two components. Assume also that the license cost of each of the two encased components is only $100 and that component vendors impose no other costs (i.e., installation, maintenance, etc.). Then,

using the estimated developers cost for one compositional wrapper of $102,400 (obtained in Section 16.1), the following equation estimates the total cost of the compositional wrapper:

Total compositional wrapper cost = LC + RC * [(Developers costs * (20% / 31%))
+ (Developers costs * (21% / 31%))]

Total compositional wrapper cost = (2 * $100) + 2 * [($102,400 * (20% / 31%))
+ ($102,400 * (21% / 31%))]

Total compositional wrapper cost = $200 + 2 * [$66,064.52 + $69,367.74]
= $271,064.52.

CHAPTER 17: SUMMARY AND CONCLUSIONS

The work presented in this dissertation integrated and summarized the current literature to provide a coherent and complete description of the activities involved in the development of CWB systems. The dissertation analyzed the concept of compositional wrappers and presented two new specification languages, a complete composition process, and a smart wrapper generator that can support the generation of compositional wrappers. When the literature provided details, such details were used to build the concepts and techniques presented in this dissertation. When gaps were identified or modifications were needed to support the goals of this research, new or augmented concepts and activities were developed.

The applicability of the two new specification languages, the composition process, and the wrapper generator were demonstrated by the implementation of a working wrapper generator prototype (discussed in the Appendix.) A return-on-investment (ROI) analysis was also performed and justified by basing such ROI analysis on actual projects as reported in the published software engineering literature.

The following subsection provides more details on the contributions of this dissertation. Section 17.2.3 will highlight some of the opportunities for evaluating the concepts and techniques that were presented in this research.

17.1 Highlights of Work Done and Lessons Learned

This dissertation has provided a foundation to support the development of systems using the next generation of software components: smart compositional wrappers. Compositional wrappers were discussed from various perspectives, including their advantages and shortcomings, their internal structures and contents, the way to specify their behavioral and non-behavioral characteristics, the smart features that can be built into them to allow them to exhibit intelligent behaviors, and the various ways to upgrade and maintain them during their operational life. A variety of algorithms and code snippets were presented throughout this dissertation to illustrate and explain the inter-working of compositional wrappers.

The work of this dissertation also involved the development and documentation of two new formal specification languages: The Wrapper Specification Language (WSL) and the Component Specification Language (CSL). These languages can support the development and automatic generation of compositional wrappers. In the discussion of the two new formal specification languages (WSL and CSL), not only the constructs that compose each language were highlighted, but also the various ways of using the constructs were discussed, especially those that deal with wrapper generators.

The two languages allow users to express the behavioral and non-behavioral aspects of components and compositional wrappers in formal, but easy to use and understand, constructs. Though the two languages have not been validated for commercial use,

the utility of the two languages was demonstrated by the code snippets and algorithms provided in the dissertation. In addition, utility of the two specification languages was demonstrated through the development of two complete WSL and CSL specifications for a bank example. The work of this dissertation also included a complete architectural specification for a gas station application and specifications for seven real-world cases. Additionally, the Appendix of this dissertation discusses a prototype consisting of 1373 lines of Perl code to demonstrate the parsing of WSL and CSL specifications. Section 17.2 discusses some areas of possible enhancements to these two new languages.

A complete composition process was also developed to guide the wrapper generator in the process of assembling and generating smart compositional wrappers. The composition process was developed to demonstrate the applicability of the concept of compositional wrappers and the two new specification languages. The complete process, from the identification and specification of compositional wrappers to the generation of the new compositional wrappers, was explained in detail. The discussion was extended to include some of the activities that can take place during the assembling and operation of CWB systems. The reason for discussing such activities is to show how such activities could benefit from the concept of compositional wrappers and from the available constructs in WSL and CSL.

A smart wrapper generator was presented as a mechanism to support the automatic generation of smart compositional wrappers. Chapter 4 introduced wrapper generators and laid out a framework that set the domain and goals of the wrapper generator that is discussed in this dissertation. Then, throughout the chapters that discussed the composi-

tion process, many algorithms and utilities were presented to make the implementation of wrapper generators simple and straightforward. The prototype presented in the Appendix demonstrates how such wrapper generators can be built.

Opportunities of automation by the wrapper generator of many of the activities in all the stages of the composition process were discussed. Automation of such activities were made possible by the variety and simplicity of the formal constructs of WSL and CSL; by the well-defined stages and activities of the composition process; and by the powerful built-in capabilities of the wrapper generator. Examples of the built-in capabilities of the wrapper generator include the utilization of available search and middleware agents, the support of user-defined code templates that the wrapper generator can instantiate to accomplish the various compositional wrapper's tasks, and the various options to control the wrapper generation process.

17.2 Future Research

This dissertation presented many of the new concepts, processes, and techniques that are ready for immediate implementation. However, many opportunities for building on and enhancing these ideas are still possible. The following are some areas that can be further explored:

17.2.1 Enhancing the Specification Languages

The following are some of the areas of possible enhancements to one or both of the specification languages that were presented in this dissertation. The discussion below presents cases where the two languages do not have specific constructs to address them. However, the function of such cases can, in most situations, be delivered using one or more of the other language constructs.

- The two languages currently do not address the sharing of resources among the encased components of a compositional wrapper. CSL contains constructs that list memory and storage requirements. WSL also has constructs that allow restricting the maximum memory and storage quotas that components can require. However, the two languages still lack constructs that can be used to manage the sharing of such resources.

- The languages do not provide adequate constructs to help wrapper users in testing components. The only construct provided by WSL that can aid in testing components is the "pretend" smart feature defined by the `<Testing_Pretend>` construct (see Section 8.2.9.7.) However, the purpose of this construct is only to report calls to and from components. The construct is not intended to give information on how components behave internally. Other specification languages, such as ADL [OBAY98, SANK94a, SANK94b], address this testing limitation. ADL provides many constructs that allow one to test the various aspects of component behaviors.

- Though the two languages do not provide explicit constructs to specify pre- and post-conditions, many existing constructs in the two languages actually do such job. For example, inputs and stimuli specifications act as pre-conditions to proper component operations. Additionally, the `<RequiresDeclaration>` construct allows specifying constraints on the components and interfaces that are required to coexist with the specified component. The constructs for specifying software, hardware, and network constraints can also act as pre-conditions to proper component operations. Output specifications are the only currently supported mean of specifying post-conditions.

- WSL currently does not have constructs to specify how compositional wrappers should react to unexpected behaviors from their encased components. ADL, for example, allows specifying both normal and abnormal behaviors.

- The `<CaseRulesDeclaration>` construct (see Section 8.3) gives a way to specify some forms of concurrency. With that construct, one can tell which of the component scenarios can happen at the same time or in sequence. However, complex concurrency scenarios, as supported by Rapide [LUCK96, KENN96], are not yet present in WSL.

- Though the two languages provide no explicit constructs to deal with aggregate inputs or outputs, input and output types can be used to induce the existence of such aggregations. For example, if an input in a wrapper specification is defined as a structure, then one can look for one or more inputs in a component specification

that, collectively, resemble the content of the structure. Aggregation can also be induced from output formats. However, the techniques just stated would require extensive logic to parse and reason about the various types, formats, data classes, and sizes of the various inputs and outputs.

- WSL currently contains no constructs to allow specifying the maximum number of interfaces that can be exported as required interfaces in generated compositional wrappers. WSL also lacks constructs to specify the minimum provided interfaces, the maximum provided interfaces, and the maximum required interfaces that each component that passed the evaluation stages can have. Additionally, there are no constructs to specify the maximum total provided and required interfaces of all matching components. These pieces of information are used by the various algorithms that were presented throughout this dissertation.

- Section 14.2 discussed several models of downloading and executing components. The `<DownloadsDeclaration>` construct currently supports downloading of components at design time only. Supporting other models of downloading components would require extending the `<DownloadsDeclaration>` construct or introducing new constructs.

- The issue of supporting multiple components for load-balancing or emergency recovery is discussed in Section 14.1. However, the `<SmartFeaturesDeclaration>` construct currently does not contain options to allow specifying and managing these features.

- The component specification language currently provides no constructs to allow entering vendor and component maturity. Vendor and component maturity are used during the evaluation stages. They are also used by the algorithms of tion 14.4 to estimate the glue code size of compositional wrappers.

- Section 10.4.1 discussed the various steps that the wrapper generator can follow to refine the wrapper specification when it cannot find enough or suitable components. However, WSL currently lacks the constructs that would guide the wrapper generator in performing such necessary refinement of the wrapper specification. Additionally, to enable the wrapper generator to automate the refinement of wrapper specifications, it must be possible to specify the clauses that use numerical values to mean minimums or maximums (such as the component costs and hardware constraints) as ranges of numbers instead of discrete values.

- To support automation of most of the evaluation and matching activities that the wrapper generator can perform, it is necessary to have constructs to guide the wrapper generator on how to deal with components that partially match the wrapper specification. The WSL language should also introduce constructs that act as rules on which the generator can base its selection decisions.

- The two specification languages offer a variety of constructs that allow specifying several types of component costs. However, there are situations when specification writers want to specify very high-priced components with low-priced ones.

WSL should support specifying categories of components. Existing constructs might require modifications.

- To allow the wrapper generator to automate component reservations (discussed in Section 11.2), WSL should provide constructs to guide the wrapper generator in performing this activity. For example, the constructs should put limits on the reservation fees that can be imposed by component vendors. In addition, the new WSL constructs should direct the wrapper generator on what personal and payment profiles to use when doing the reservations. CSL should also contain constructs that can explicitly specify reservation costs and the server or contact person that can handle component reservations.

- Currently, WSL allows specification writers to express their preferences on component suppliers using the *suppliers* and *supplier_types* clauses. However, there are situations when development organizations prefer to have components of certain characteristics from certain suppliers. This can be done by allowing specification writers to enter their supplier preferences with each interface specification.

- The ranking scheme of inputs (Section 6.1.3.1), outputs (Section 6.1.3.2), and non-behavioral constructs (Sections 8.1) is not yet implemented in CSL. Implementing such scheme would require modifying the syntax of all of WSL constructs that can be ranked.

17.2.2 Architecture Implementation

Section 8.3 presents a detailed discussion on how to specify compositional wrapper architectures. However, this dissertation does not include any discussion on how wrapper generators can deal with such architectures. In addition, No discussion was provided in the dissertation on the nature of the code to be generated in compositional wrappers to enforce architecture specifications. Implementing architectures in compositional wrappers could require: modifying some parts of the composition process, introducing new or modifying existing constructs in WSL (and possibly in CSL), and modifying existing or introducing new algorithms.

17.2.3 Behavioral Matching

Despite the many algorithms presented in Section 12.5 that can be used to check the behavioral compatibility of wrapper and candidate component specifications, there are still several unresolved problems. The following is a list of some of these problems. The solution to these problems can involve introducing new constructs or modifying existing ones in one or both of the specification languages. The solution can also involve adding or modifying activities in the composition process, and developing new or enhancing existing algorithms or techniques used by the wrapper generator.

- How to deal with missing or extra assertions in `call` statements? The `call` statement is used to list the sequence of behavioral assertions expected (in the case

of WSL) or performed by (in the case of CSL) components. However, specification writers might not list the complete set of assertions of each call. The wrapper generator should have some means of telling when such missing assertions exist and how to react to them.

- How to deal with assertions in a wrapper `call` that are compatible with but do not show up in the same order as assertions in a component `call`?

- How to check if a call to a dummy function is compatible with one or more assertions? Dummy functions (discussed in Section 6.1.4) are names for known sequences of common operations performed on data structures (for example, *push* and *pop* in a stack or *first* and *last* in a queue.) Sometimes, specification writers might prefer to list the sequence of actions in a dummy function as assertions instead of referring to the dummy functions by their names. The wrapper generator must be able to tell when such situations exist and how to deal with them.

- Are `stimuli` and `architecture` specifications important in deciding the compatibility of two specifications? Currently, `stimuli` and `architecture` specifications are not used in the algorithms of Section 12.5.12 that check for specification compatibility. Further research and analysis are required to tell if such specifications are needed in the matching process. If their inclusion is justified, then the algorithms of Section 12.5 would need to be modified.

APPENDIX: THE WRAPPER GENERATOR PROTOTYPE

This appendix describes the content of the enclosed CD. The CD contains information related to the wrapper generator prototype. Examples and cases as well as complete language listings of the WSL and CSL specification languages are provided in the CD. The content of the CD is divided into the following three categories:

- *The specification languages*. The folder named ***SPEC*** contains the following two files:
    - ***WSL.txt*** contains the complete language grammar for the wrapper specification language.
    - ***CSL.txt*** contains the complete language grammar for the component specification language.
- *Examples and cases*. The folder named ***EXAMPLES*** contains the following files:
    - ***Bank.wsl***

      This file contains a complete wrapper specification for a bank example. It is similar to the one presented in Figure 11.
    - ***Bank.csl***

      This file contains a sample component specification that conforms to CSL. The file's content is written as if it were to satisfy the wrapper specification of ***Bank.wsl***.
    - ***Case1.wsl*** to ***Case7.wsl***

These files present example cases to illustrate the use of the behavioral constructs in writing the behavioral specification part of a wrapper specification or a component specification. The cases increase in complexity from a very simple sequence generator presented in ***Case1.wsl*** to a complicated dinning philosophers' problem in ***Case7.wsl***.

- ***GasStation.wsl***

  This file contains the architectural specification of the *GasStation* problem that was presented in Figure 10.

- *The wrapper generator prototype*. The folder named ***PROTOTYPE*** contains sample code files written using the Perl language. To test the prototype, all the files in the ***PROTOTYPE*** folder must be uploaded to a website that contains a ***cgi-bin*** directory and supports executing Perl scripts. Restrictions on using CGI and Perl scripts must be observed. The following files exist in this folder:

  - ***Prototype.cgi***

    This file contains the main prototype driver. This is the file that needs to be executed to test the prototype. Other Perl files will be called from within this script.

  - ***Wrapper.cgi*** and ***wrappers.dat***

    The ***Wrapper.cgi*** file contains Perl scripts that implement the utilities that are specific to compositional wrappers. The scripts in ***Wrapper.cgi*** use

wrappers.dat as a repository to save information about each compositional wrapper.

- ***Component.cgi*** and ***components.dat***

  The ***Component.cgi*** file contains the utilities that are needed to manage collected components. Information for such components are stored in ***components.dat***.

- ***Profile.cgi*** and ***profiles.dat***

  The utilities in the ***Profile.cgi*** file are used to manage user profiles. The ***profiles.dat*** is used to store the user profiles' information.

- ***Vendor.cgi*** and ***vendors.dat***

  The ***Vendor.cgi*** file contains the utilities required to manage component vendors. Vendors' information are stored in ***vendors.dat***.

- ***SearchAgent.cgi*** and ***searchagents.dat***

  The ***SearchAgent.cgi*** file contains the utilities required to manage search agents. Information about available search agents are stored in ***searchagents.dat***.

- ***ParseDescent.pm***

  This Perl module implements a language parser. It is part of the Perl distribution kit. The latest version of this module can be obtained from

www.perl.org. The ***ParseDescent.pm*** file is located in the ***Parse*** subdirectory in the ***PROTOTYPE*** folder.

- ***Parser.pl***

  This sample Perl script uses the ***ParseDescent*** Perl module (***ParseDescent.pm***) to parse a subset of CSL's grammar. The ***Parser.pl*** script is written to demonstrate the use of Perl parsers to check the correctness of the CSL grammar. The script also illustrates how to embed Perl code within the language grammar to extract language-specific information from each language construct. The extracted information can be passed to other scripts (such as the prototype scripts discussed above.) To test a different subset of CSL or WSL grammars, one has to replace the subset of the CSL grammar that is currently embedded in ***Parser.pl*** with the new to-be-tested grammar. References to the old grammar would need to be replaced with the information that references the new grammar.

- ***DEMO***

  This folder contains fourteen HTML files for various screens from a live prototype. The same screens can be obtained when the complete prototype files (Perl scripts and repository files) are uploaded to a website (that supports CGI and Perl scripts) and invoked from that website.

LIST OF REFERENCES

[ABTS00a] C. Abts, "A Perspective on the Economic Life Span of COTS-Based Software Systems: the COTS-LIMO Model," University of Southern California, CA, Technical Report USC-CSE-2000-503, 2000.

[ABTS00b] C. Abts, B. W. Boehm, and E. B. Clark, "COCOTS: A COTS Software Integration Life Cycle Cost Model – Model Overview and Preliminary Data Collection Findings," University of Southern California, CA, Technical Report USC-CSE-2000-501, 2000.

[ABTS00c] C. Abts, "A Perspective on the Economic Life Span of COTS-Based Software Systems: the COTS-LIMO Model," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[ABTS00d] C. Abts, B. W. Boehm, and E. B. Clark, "Empirical Observations on COTS Software Integration Effort Based on the Initial COCOTS Calibration Database," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[ABTS98] C. Abts, "COTS Software Integration Cost Modeling Study," University of Southern California, CA, Technical Report USC-CSE-98-520, 1998.

[ACHE00] F. Achermann and O. Nierstrasz, "Applications = Components + Scripts — A Tour of Piccola," in *Software Architectures and Component Technology*, Mehmet Aksit (Ed.), Kluwer, 2000.

[ACHE99] F. Achermann, M. Lumpe, J. Schneider, and O. Nierstrasz, "Piccola – a Small Composition Language," in *Formal Methods for Distributed Processing, an Object Oriented Approach,* H. Bowman and J. Derrick, Ed. Cambridge University Press, 2001.

[ADAM01] R. J. Adams and S. Eslinger. (2001, June). Lessons Learned From Using COTS Software on Space Systems. *Cross Talk*. [Online].

Available: http://www.stsc.hill.af.mil/crosstalk/2001/jun/adams.asp

[AGHA98] R. P. Biuk-Aghai, "Customizable Software Engineering Environments for Flexible Distributed Software Teams," in *Proceedings of the1998 Asia Pacific Software Engineering Conference*, Macau University, Macau, December 2-4, 1998, pp. 228-235.

[ALEN92] A. Alencar and J. Goguen "OOZE," S. Stepney, R. Barden, and D. Cooper, ed. *Object Orientation in Z*, Workshops in Computing. Los Angeles, Ca.: Springer-Verlag, 1992.

[ALLE01] P. Allen, *Realizing e-Business With Components*, Addison-Wesley, 2001.

[ALLE94] R. Allen and D. Garlan, "Formal Connectors," Carnegie-Mellon University, PA, Technical Report CMU-CS-94-115, March 1994.

[ALLE92] R. Allen and D. Garlan, "A Formal Approach to Software Architectures," in *Proceedings of IFIP'92*, van Leeuwen, Ed., Elsevier Science Publishers B.V., September 1992.

[ALVA00] A. Alvarez, S. Bandinelli, A. Belén García, and E. Schaeidt, "Breaking Down Non-Technical Barriers to COTS Adoption," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[AMMA93] P. Ammann and S. Jajodia, "Distributed timestamp generation in planar lattice networks," *ACM Transactions on Computer Systems*, Vol. 11, No. 3, August 1993, pp. 205-225.

[ANDE98] R. J. Anderson and F. A. P. Petitcolas, "On the Limits of Steganography," *IEEE Journal on Selected Areas in Communication*, Vol. 16, No. 4, May 1998, pp. 474-481.

[ANDR00] S. Andriole. (2000). Why is outsourcing growing in popularity? [Online]. Available: http://www.cutter.com/consortium/freestuff/bitadvi0317.html

[ANIE98] A. Cimitile, U. De Carlini, and A. De Lucia, "Incremental Migration Strategies: Data Flow Analysis for Wrapping," in *Proceedings of the Fifth Working Conference on Reverse Engineering*, Sannio University, Benevento, Italy, October 12-14, 1998, pp. 59-68.

[ANIO02] P. Aniorte, "Component-Based Software Development: From Com-

ponent Model to Software Architecture," *ICSR7 2002 Workshop on Component-based Software Development Processes*, Austin, TX, April 16, 2002.

[APPL98] D. Appleman, *Developing COM/ActiveX Components with Visual Basic 6.0 -- A Guide to the Perplexed*, Sams Publishing, 1998.

[ASAI00] C. Asaipilai. (2000). The History of Distributed Object Oriented Technologies. [Online]. Available: http://www.metronet.co.uk/weegee/OOD.htm

[ATKI02] C. Atkinson and H. Groβ, "Built-in Contract Testing in Model-driven Component-Based Development," *ICSR7 2002 Workshop on Component-based Software Development Processes*, Austin, TX, April 16, 2002.

[ATKI00] C. Atkinson, J. Bayer, O. Laitenberger, and J. Zettel, "Component-Based Software Engineering: The KobrA Approach," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[BACH00] F. Bachman, L. Bass, C. Buhman, S. Camella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau, "Volume II: Technical Concepts of Component-Based Software Engineering," Software Engineering Institute, PA, Technical Report CMU/SEI-2001-TR-008, May 2000.

[BAGG97] V. Baggiolini and J. Harms, "Toward Automatic, Run-time Fault Management for Component-Based Applications," in *Proceedings of the 2nd Workshop on Component-Oriented Programming (WCOP'97)*, Jyväskylä, Finland, June 9, 1997, pp. 5-12.

[BAIL97] V. R. Baili, S. E. Condon, K. El Emam, R. B. Hendrick, and W. Melo, "Characterizing and modeling the cost of rework in a library of reusable software components," in *Proceedings of the 1997 IEEE 19th International Conference on Software Engineering*, Boston, MA, 1997, pp. 282-291.

[BAIL91] J. W. Bailey, and V. R. Basili. (1991). The software-cycle models for re-engineering and reuse. Presented at The conference on Ada: today's accomplishments; tomorrow's expectations. [Online]. Available: http://www.acm.org/pubs/citations/proceedings/ada/126551/p267-bailey/

[BASI01] V. R. Basili and B. Boehm, "COTS-Based Systems Top 10 List,"

Computer, Vol. 34, Issue 5, May 2001, pp. 91-95.

[BASS00] L. Bass, C. Buhman, S. Camella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau, "Volume I: Market Assessment of Component-Based Software Engineering," Software Engineering Institute, PA, Technical Note CMU/SEI-2001-TN-007, May 2000.

[BATO97] D. Batory, "Intelligent Components and Software Generators," in *Proceedings of the Software Quality Institute Symposium on Software Reliability,* Austin, Texas, April 1997.

[BATO96] D. Batory and B. Geraci, "Validating Component Compositions in Software System Generators," in *Proceedings of the Fourth International Conference on Software Reuse*, Austin, Texas, April 23-26, 1996, pp. 72-81.

[BATO94] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin, "The GenVoca model of software-system generators," *IEEE Software*, Vol. 11, Issue 5, September 1994, pp. 89-94.

[BATO92] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components," *ACM Transactions on Software Engineering*, Vol. 1, Issue 4, 1992, pp. 355-398.

[BEND96] W. Bender, D. Gruhl, N. Morimoto, and A. Lu, "Techniques for Data Hiding," *IBM Systems Journal*, Vol. 35, No. 3&4, 1996, pp. 313-336.

[BERG00] K. Bergner, A. Rausch, M. Sihling, and A. Vilbig, "Adaptation Strategies in Componentware," in *Proceedings of the 2000 Australian Software Engineering Conference*, Munchen, Germany, April 28-29, 2000, pp. 87-95.

[BERG99] J. Bergey, D. Smith, S. Tilley, N. Weiderman, and S. Woods, "Why Reengineering Projects Fail?," Software Engineering Institute, PA, Technical Report CMU/SEI-99-TR-010, April 1999.

[BERG97] S. Bergamaschi, A. Garuti, C. Sartori, and A. Venuta, "Object Wrapper: an Object-Oriented Interface for Relational Databases," in *Proceedings of the 23rd EUROMICRO Conference on New Frontiers of Information Technology*, Modena University, Italy, September 1-4, 1997, pp. 41-46.

[BEUS00] L. Beus-Dukic. (2000, June 4-5). Non-Functional Requirements for COTS Software Components. Presented at Workshop on Continuing

Collaborations for Successful COTS Development, Limerick, Ireland, June 4-5, 2000.

[BOEH02] B. Boehm. COTS Empirical Data: A Top-10 List and Its Implications. Presented at The 2002 Ground System Architectures Workshop. El Segundo, California, March 13-5, 2002. [Online]. Available: http://sunset.usc.edu/GSAW/gsaw2002/s2/boehm.pdf

[BOEH00] B. Boehm, E. Horowitz, R. Madachy, and C. Abts. (2000, April). Future Trends, Implications in Cost Estimation Models. *Cross Talk.* [Online]. Available: http://www.stsc.hill.af.mil/crosstalk/2000/apr/boehm.asp

[BOEH99] B. Boehm and C. Abts, "COTS Integration: Plug and Pray?," *IEEE Computer*, Vol. 32, Issue 1, January 1999, pp. 135-138.

[BOSC97] J. Bosch, W. Weck, and C. Szyperski, "Summary of 2nd Workshop on Component-Oriented Programming – (WCOP'97) ," in *Proceedings of the 2nd Workshop on Component-Oriented Programming (WCOP'97)*, Jyväskylä, Finland, June 9, 1997, pp. 1-4.

[BRAC92] G. Bracha, "The programming language Jigsaw: mixins, modularity, and multiple inheritance," Ph.D. thesis, Dept. of Computer Science, University of Utah, Utah, 1992.

[BRIA00] L. C. Briand, T. Langley, and I. Wieczorek, "A Replicated Assessment and Comparison of Common Software Cost Modeling Techniques," in *Proceedings of the 22nd International Conference on Software Engineering,* Limerick, Ireland, June 4-11, 2000, pp. 377-386.

[BROO87] F. P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, Vol. 20, Issue 4, April 1987, pp. 10-19.

[BROW00] A. W. Brown, *Large-Scale Component-Based Development*, Prentice Hall, 2000.

[BROW98a] A. W. Brown and K. C. Wallnau, "The current state of CBSE," *IEEE Software*, Vol. 15, Issue 5, September/October 1998, pp. 37-46.

[BROW98b] L. Brownsword, D. Carney, and T. Oberndorf. (1998, April). The Opportunities and Complexities of Applying Commercial-Off-the-Shelf Components. *Cross Talk*. [Online]. Available: http://www.stsc.hill.af.mil/CrossTalk/1998/apr/applying.asp

[BUNT00] R. Bunting, "Component Service Providers: An Evolution in Component Management," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[CANA97] C. Canal, E. Pimentel, and J. M. Troya, "On the Composition and Extension of Software Components," in *Proceedings of the Foundations of Component-Based Systems Workshop,* Zurich, Switzerland, September 26, 1997.

[CARN00] D. Carney, S. A. Hissam, and D. Plakosh, "Complex COTS-Based Software Systems: Practical Steps for Their Maintenance," *Journal of Software Maintenance: Research and Practice*, Vol. 12, Issue 6, December 21, 2000, pp. 357-376.

[CARN97] D. Carney. (1997, June). Assembling Large Systems from COTS Components: Opportunities, Cautions, and Complexities. [Online]. Available: http://www.sei.cmu.edu/cbs/papers/monographs/assembling-systems/assembling.systems.pdf

[CHAN99a] J. Chang, "Automating Specification-based Test Coverage Measurement and Test Selection," Dissertation thesis, University of California, Irvine, 1999.

[CHAN99b] J. Chang and D. J. Richardson, "Structural specification-based testing: automated support and experimental evaluation," in *Proceedings of the 7th European Engineering Conference held jointly with the 7th ACM SIGSOFT symposium on Foundations of Software Engineering*, 1999, pp. 285 - 302.

[CHAV98] A. Chavez, C. Tornabene, and G. Wiederhold, "Software Component Licensing: A Primer," *IEEE Software*, Vol. 15, No. 5, September/October 1998.

[CHEN00] N. Cheng, V. Berzins, Luqi, and S. Bhattacharya, "Interoperability With Distributed Objects Through Java Wrapper," in *Proceedings of the 24th Annual International Computer Software and Applications Conference*, Monterey, CA, October 25-27, 2000, pp. 479-485.

[CHEU96] S. Cheung and J. Kramer, "Checking Subsystem Safety Properties in Compositional Reachability Analysis," in *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, March 25-30, 1996, pp. 144 -154.

[CLAR00] B. Clark, B. Boehm, and C. Abts, "COCOTS: A Cost Model for COTS-Intensive Software Systems," *Presentation at the 15th Annual Software Engineering Symposium*, Software Engineering Institute, PA, September 19, 2000.

[CLEA96] C. J. Cleaveland, J. A. Fertig, and G. W. Newsome, "Dividing the software pie," *AT&T Technical Journal*, Vol. 75, No. 2, March-April 1996, pp. 8-19.

[COLL99] C. Collberg and C. Thomborsen, "Software Watermarking: Models and Dynamic Embeddings," in *Proceedings of the 26th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Antonio, Texas, January 20-22, 1999, pp. 311-324.

[COOK99] J. E. Cook and J. A. Dage, "Highly Upgrading of Components," in *Proceedings of the 1999 International Conference on Software Engineering*, Las Cruces, NM, May 16-22, 1999, pp. 203-212.

[COX95] B. Cox. (1995, November). No Silver Bullet Revisited. *American Programmer Journal*. [Online]. Available: http://www.virtualschool.edu/mon

[CRNK00] I. Crnkovic, M. Larsson, and F. Lüders, "State of the Practice: Component-Based Software Engineering Course," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[DESA98] M. Desai and L. Élan. (1998). UL 1998 - Software in Programmable Components. [Online]. Available: http://www.ul.com/pscs/UL_1998_FINAL.pdf

[DINC00] E. Dincel, "ADL Independent Architectural Representation in XML," University of Southern California, CA, Technical Report USC-CSE-2000-519, 2000.

[DING99] L. Ding, X. Zhang, and E. A. Rundensteiner, "The MRE Wrapper Approach: Enabling Incremental View Maintenance of Data Warehouses Defined on Multi-Relation Information Sources," in *Proceedings of the ACM second international workshop on Data warehousing and OLAP*, Kansas City, MO, November 2-6, 1999, pp. 30-35.

[DOMI02] F. Domínguez, R. Capilla, and J. M. Cueva, "OVBUS: A Scalable Platform for Combining Heterogeneous Components," in *Proceedings*

of the ICSR7 2002 Workshop on Component-based Software Development Processes, Austin, TX, April 16, 2002.

[DSOU96] D. D'Souza and A. Wills, "OOA/D and CORBA/IDL: A Common Base," ICON Computing Inc., Technical Report, 1996.

[DUKE96] R. Duke, G. Rose, and G. Smith, "Object-Z: A Specification Language Advocated for the Description of Standards," *Computer Standards and Interfaces*, 1996, pp. 17:511–533.

[DURR94] E. H. Dürr and N. Plat, (editors), *VDM++ Language Reference Manual.* Utrecht, The Netherlands: Cap Volmac, Mar. 1994.

[EDWA98] S. H. Edwards, B. W. Weide, and J. Hollingsworth, "A Framework for Detecting Interface Violations in Component-Based Software," in *Proceedings of the Fifth International Conference on Software Reuse*, West Virginia University, Morgantown, WV, June 2-5, 1998, pp. 46-55.

[EDWA74] N. P. Edwards and H. Tellier, "A Look at Characterizing the Design of Information Systems," in *Proceedings of the ACM National Conference*, November 1974, pp. 612-621.

[EGYE00] A. Egyed, N. Medvidovic, and C. Gacek, "A Component-Based Perspective on Software Mismatch Detection and Resolution," University of Southern California, CA, Technical Report *USC-CSE-2000-520*, November 2000.

[ELLI95] T. Ellis, "COTS Integration in Software Solutions—A Cost Model," *Software Engineering in the Global Marketplace*, NCOS International Symposium, St. Louis, MO, July 24-26, 1995.

[ELSI01] Elsinore Technologies. (2001). Visual Intercept. [Online]. Available: http://www.elsitech.com/

[ERDO00] H. Erdogmus, C. A. Sledge, M. Looney, and S. Allison, "Costing of COTS-Based Systems: An Initial Framework," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[ERIK00] N. Eriksson and T. Vaalundi, "Applying CBSE Theory on Corporate Resources," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[EVER99] W. W. Everett, "Software Component Reliability Analysis," in *Proceedings of the 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology*, Richardson, TX, March 24-27, 1999, pp. 204 - 211.

[FICK97] S. Fickas, "Software on demand: Issue for RE," in *Proceedings of the 1997 3rd International Symposium on Requirements Engineering*, Annapolis, MD, 1997, pp. 222-223.

[FISC00] L. Fischman and K. McRitchie. (2000, January). Off-the-Shelf Software: Practical Evaluation. *Cross Talk*. [Online]. Available: http://www.stsc.hill.af.mil/crosstalk/2000/jan/fischman.asp

[FITZ98] J. Fitzgerald and P. G. Larsen, *Modeling Systems: Practical Tools and Techniques in Software Development*, Cambridge University Press, 1998.

[FLAT99] M. Flatt, "Programming Languages for Reusable Software Components," Dissertation thesis, Rice University, Houston, Texas, 1999.

[FLYN98] J. Flynt and L. Élan. (1998). Software Conformity Assessment. [Online]. Available: http://www.ul.com/pscs/SoftwareConformityAssessment.pdf

[FRAK00] W. B. Frakes, "A Case Study of a Reusable Component Collection," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[FRAS99] T. Fraser, L. Badger, and M. Fredman, "Hardening COTS Software with Generic Software Wrappers," in *Proceedings of the DARPA Information Survivability Conference and Exposition*, Glenwood, MD, January 25-27, 2000, pp. 323-337 vol. 2.

[GACE98a] C. Gacek and B. Boehm, "Composing Components: How Does One Detect Potential Architectural Mismatches? ," in *Proceedings of the 1998 Workshop on Compositional Software Architectures*, Monterey, CA, January 1998.

[GACE98b] C. Gacek, "Detecting Architectural Mismatches During Systems Composition," Dissertation thesis, University of Southern California, 1998.

[GAO01] J. Gao, S. Shim, and E. Zhu. (2001, October). Tracking Software Components. *Journal of Object-Oriented Programming*. [Online].

Available:
http://www.adtmag.com/joop/article.asp?id=5121&mon=10&yr=2001

[GAO00a] J. Gao, "Component Testability and Component Testing Challenges," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[GAO00b] J. Gao, "Tracking Component-Based Software," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[GARL95a] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural Mismatch or Why it is hard to build systems out of existing parts," *IEEE Software*, Vol. 12, Issue 6, November 1995, pp. 17-26.

[GARL95b] D. Garlan, "Research Directions in Software Architecture," *ACM Computing Surveys*, Vol. 27, No. 2, June 1995, pp. 257-261.

[GARM01] D. Garmus and D. Herron. (2001, July 23). Estimating Software Earlier and More Accurately. *Object-Z.* [Online]. Available: http://objectz.com/columnists/2davids/07232001.asp

[GHOS99] A. K. Ghos and M. Schmid, "An Approach to Testing COTS Software for Robustness to Operating System Exceptions and Errors," in *Proceedings of the 10th International Symposium on Software Reliability Engineering*, Dulles, VA, November 1-4, 1999, pp. 166-174.

[GIBB94] W. Wayt Gibbs, "Software's Chronic Crisis," *Scientific American*, Sept 1994.

[GOED97] M. Goedicke and T. Meyer, "Design and Evaluation of Distributed Component-Oriented Soft-ware Systems," in *Proceedings of the 2nd Workshop on Component-Oriented Programming (WCOP'97)*, Jyväskylä, Finland, June 9, 1997, pp. 33-42.

[GOME00] A. Gómez-Perez and A. Lozano, "Impact of Software Components Characteristics Above Decision-Making Factors," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[GRAH96a] T. C. N. Graham, "Viewpoints Supporting the Development of Interactive Software," in *Proceedings of the SIGSOFT 96 Workshop*, San Francisco, CA, 1996.

[GRAH96b] T. C. N. Graham, C. A. Morton, and T. Urnes, "Clockworks: Visual Programming of Component-Based Software Architectures. " *Journal of Visual Languages and Computing*, July 1996.

[GRIS98] M. L. Griss, "Architecting for large-scale systematic component reuse," HP Laboratories, Technical Report 98-132, July 1998, pp. 1-7.

[GUNT00] M. S. Guntersdorfer, D. G. Kay, and D. S. Rosenblum. (2000, June 4-5). Using Software Patents to Support The Business Model of Software Components. Presented at Workshop on Continuing Collaborations for Successful COTS Development, Limerick, Ireland.

[GUTT93] J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing, *Larch: Languages and Tools for Formal Specification*, Springer-Verlag Texts and Monographs in Computer Science, 1993.

[HALL00] S. Hallsteinsen, O. Dragsten, M. Johnsen, and J. O. Ofstad, "A Component Oriented Domain Architecture for Fish Farming," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[HAML00] D. Hamlet, D. Mason, and D. Woit, "Theory of System Reliability Based on Components," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[HANS99] W. J. Hansen, "A Generic Process and Terminology for Evaluating COTS Software," *Technology of Object-Oriented Languages and Systems (TOOLS)*, pp. 30:547-551.

[HAUP96] S. Hauptmann and J. Wasel, "On-Line Maintenance with On-the-fly Software Replacement," in *Proceedings of the Third International Conference on Configurable Distributed Systems*, Aachen, Germany, May 6-8, 1996, pp. 70-80.

[HEIN01] G. Heineman and W. Counceill, *Component-Based Software Engineering*, Addison-Wisely, 2001.

[HELM85] D. Helmbold and D. Luckham, "Debugging Ada Tasking Programs," *IEEE Software*, Vol. 2, No. 2, pp. 47-57, March 1985.

[HENN97] S. Henninger, "An evolutionary approach to constructing effective software reuse repositories," *ACM Transactions on Software Engineering and Methodology*, Vol. 6, No. 2, April 1997, pp. 111-140.

[HISS99] S. A. Hissam and D. Carney, “Isolating Faults in Complex COTS-Based Systems,” *Journal of Software Maintenance: Research and Practice*, Vol. 11, Issue 3, June 30, 1999, pp. 183-199.

[HISS97] S. Hissam. (1997, September). Case Study: Correcting System Failure in a COTS Information System. *SEI Monographs on the Use of Commercial Software in Government Systems*. [Online]. Available: http://www.sei.cmu.edu/cbs/papers/monographs/case-study-correcting/case.study.correcting.pdf

[HOA98] N. D. Hoa, “Component Definition Language for Dynamic Component Updating,” in *Proceedings of the 7th Annual Conference of Doctoral Students - WDS'98*, Prague, June 9-12, 1998.

[HOEK98] A. Hoek, D. Heimbigner, and A. L.Wolf, “Versioned Software Architecture,” in *Proceedings of the Third International Workshop on Software Architecture*, Orlando, FL, 1998, pp. 73–76.

[INVE95] P. Inverardi and A. L. Wolf, “Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model,” *IEEE Transaction on Software Engineering*, Vol. 21, No. 4, April 1995, pp. 373-386.

[JACO97] B. Jacob and H. Jubin, *Part Development for OpenDoc*, Prentice Hall PTR, New Jersey, 1997.

[JARZ95] S. Jarzabek, “From reuse library experiences to application generation architectures,” in *Proceedings of the 17th international conference on software engineering on Symposium on software reusability*, 1995, pp. 114-122.

[JONE98a] C. Jones, "Sizing Up Software," *Scientific American*, December 1998, pp. 104-109.

[JONE98b] C. Jones. (1998). Online Discussions on the Estimating Software Cost book by Capers Jones. [Online]. Available: http://www.highq.be/quality/estimat.htm

[KALA97] J. Kalaoja, “Feature modeling of component-based embedded software,” in *Proceedings of the 8th International Workshop on Software Technology and Engineering Practice*, Los Alamitos, CA, September 1997, pp. 444-451.

[KENN00] R. Kennedy, “Technology Refreshment – A Management / Acquisition Perspective,” in *Proceedings of the Price US Symposium 2000*, Orlando, FL, March 21-23, 2000.

[KENN96] J. J. Kenny, “Executable Formal Models of Distributed Transaction Systems based on Event Processing,” Ph.D. Thesis, Stanford University, June 1996.

[KERR95] R. Kerr, M. Markley, M. Sonntag, and T. Trower, “Reno: A Component-Based User Interface,” in *Proceedings of ACM CHI’95 Conference on Human Factors in Computing Systems*, Vol. 2, 1995, pp. 19-20.

[KHAN00] K. M. Khan, J. Han, and Y. Zheng, “Security Characterisation of Software Components and Their Composition,” in *Proceedings of the 36th International Conference on Technology of Object-Oriented Languages and Systems*, Kingswood, NSW, Australia, October 30 - November 4, 2000, pp. 240-249.

[KLOU00] C. Kloukinas and V. Issarny, “Automating the Composition of Middleware Configurations,” in *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering*, Le Chesnay, France, September 11-15, 2000, pp. 241-244.

[KOPP01] N. V. Koppalkar, S. Uppala, and M. Madugundu, “Testing of Component-Based Software Systems,” in *Proceedings of the 3rd Annual International Software Testing 2001*, Bangalore, India, November 23, 2001.

[KOTO96] G. Kotonya and I. Sommerville, "Requirements engineering with viewpoints," *Software Engineering Journal*, Vol. 11, No. 1, Jan 1996, pp. 5-18.

[KOZA00] W. V. Kozaczynski, "Categorization of Business Systems Components,” *unpublished internal work*, Andersen Consulting, Center for Strategic Technology Research.

[KROP98] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek, “Automated Robustness Testing of Off-the-Shelf Software Components,” in *Proceedings of the Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, Carnegie Mellon University, Pittsburgh, PA, June 23-25, 1998, pp. 230-239.

[KURU99] I. Kuruganti, “A Component Selection Methodology with Application

to the Internet Telephony Domain," in *Proceedings of the 1999 Workshop on Component-Based Software Engineering*, Santa Barbara, CA, August 1999.

[LEMO01] R. de Lemos, "Describing Evolving Dependable Systems Using Cooperative Software Architectures," in *Proceedings. IEEE International Conference on Software Maintenance*, Florence, Italy, November 7-9, 2001, pp. 320 - 329.

[LIAO98] S. Y. Liao, Development process for the creation and reuse of object-oriented generic applications and components, *JOOP – Journal of Object-Oriented Programming*, Vol. 11, No. 3, June 1998, pp. 24-31.

[LIND98] U. Lindqvist and E. Jonsson, "A Map of Security Risks Associated with Using COTS," *Computer*, Vol. 31, Issue 6, June 1998, pp. 60-66.

[LIU99] L. Liu, W. Han, D. Buttler, C. Pu and W. Tang, "An XML-Based Wrapper Generator for Web Information Extraction," in *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, Philadelphia, PA May 31 - June 3, 1999, pp. 540-543.

[LOON00] M. Looney, H. Erdogmus, P. Oberndorf, J. Dean, C. A. Sledge, S. Allison, and G. Allan, "COTS Process Issues in Military Applications," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[LUCK97] D. C. Luckham. (1997, July). The Gas Station in Rapide. [Online]. Available: http://pavg.stanford.edu/rapide/examples/teaching/gas_station/

[LUCK96] D. C. Luckham, "Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events," *DIMACS Partial Order Methods Workshop IV,* Princeton University, July 1996.

[LUNG00] S. Lunga and M. Galoria, "Using COTS Components: Their Location, Qualification, and Selection," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[MAGE95] J. Magee, N. Dulay, S. Eisenbach and J. Kramer, "Specifying Distributed Software Architectures," in *Proceedings of the 5th European Software Engineering Conference*, Sitges, Spain, September 25-28, 1995, pp. 137-153.

[MAIE98] M. W. Maier. (1998). Architecting Principles for Systems-of-Systems. *InfoEd*. [Online]. Available: http://www.infoed.com/Open/PAPERS/systems.htm

[MANN98a] M. Mannion and B. Keepence, "Software jigsaws," *Computer Bulletin*, Vol. 10, No. 5, September 1998, pp. 24-27.

[MANN98b] M. Mannion, B. Keepence, and D. Harper, "Using viewpoints to define domain requirements," *IEEE Software*, Vol. 15, Issue 1, January/February 1998, pp. 95-102.

[MANR01] J. Manring and T. Restivo. (2001, July). Adapting Traditional Software Cost Estimating Techniques to New Technology – A PowerBuilder Case Study. [Online]. Available: http://www.pricesystems.com/downloads/pdf/estimating%20new%20software%20tech.pdf

[MART02] L. Martin, "Visual Component Integration and Regression Test," *ICSR7 2002 Workshop on Component-based Software Development Processes*, Austin, TX, April 16, 2002.

[MATT99] D. Mattox, Len Seligman, and Ken Smith, "Rapper: A Wrapper Generator with Linguistic Knowledge," in *Proceedings of the second international workshop on Web information and data management*, Kansas City, MO, November 2-6, 1999, pp. 6-11.

[MAYB96] M. J. Maybee, D. M. Heimbigner, and L. J. Osterweil, "Multilanguage interoperability in distributed systems," in *Proceedings of the 18th international conference on Software Engineering*, 1996, pp. 451-463.

[MAYR90] A. V. Mayrhauser, *Software Engineering Methods and Management*, New York: Academic Press Inc., 1990.

[MCGR99] G. McGraw. (1999, May). Why COTS Software Increases Security Risks. Presented at ICSE Workshop on Testing Distributed Component-Based Systems. [Online]. Available: http://www.cigital.com/papers/download/ses.ps

[MCIL69] M. D. McIlroy. (1969). Mass-Produced Software Components. [Online]. Available: http://cm.bell-labs.com/cm/cs/who/doug/components.txt

[MEHT00] A. Mehta and G. Heineman, "A Framework for COTS Integration and Extension," in *Proceedings of the 2000 Workshop on Component-*

Based Software Engineering, Limerick, Ireland, June 5-6, 2000.

[MEHT99] N. R. Mehta, N. Medvidovic, and S. Phadke, "Towards a Taxonomy of Software Connectors," in *Proceedings of the 21st International Conference on Software Engineering*, University of Southern California, CA, June 4-11, 2000, pp. 178-187.

[META98] META Group Consulting. (1998, March 20). CORBA vs. DCOM: Solutions for the Enterprise. [Online]. Available: http://www.sun.com/swdevelopment/news/CORBA.shtml

[MEYE00] B. Meyer, Preface to A. W. Brown's book: *Large-Scale Component Development*, pages xii-xiv.

[MEYE85] B. Meyer, "On Formalism in Specification," *IEEE Software,* Vol. 2, Issue 1, January 1985, p. 6-26.

[MICH00] B. Michiels and B. Wydaeghe, "Component Composition," in *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland, June 4-11, 2000, pp. 771.

[MILI00] A. Mili, S. F. Chmiel, R. Gottumkkala, and L. Zhang, "An Integrated Cost Model for Software Reuse," in *Proceedings of the 22nd International Conference on Software Engineering,* Limerick, Ireland, June 4-11, 2000, pp. 157-166.

[MINK01] A. F. Minkiewicz. (2001). The Real Costs of COTS. [Online]. Available: http://www.pricesystems.com/downloads/pdf/COTS.pdf

[MOOR94] J. W. Moore, "The Use of Encryption to Ensure the Integrity of Reusable Software components," in *Proceedings of the Third International Conference on Software Reuse*, Rio de Janeiro, June 4-11, 1994, pp. 118-125.

[MORI00] M. Morisio, C. B. Seaman, A. T. Parra, and V. R. Basili. (2000, June 4-5). Issues in COTS-based Software Development. Presented at Workshop on Continuing Collaborations for Successful COTS Development, Limerick, Ireland.

[MORT94] C. Morton, "Tool Support for Component-Based Programming." Department of Computer Science, York University, Technical Report CS-94-02, May 1994.

[MOSE98] L. Moser, P. Melliar-Smith, P. Narasimhan, V. Kalogeraki, and L.

Tewksbury. (1998, January 6-8). The Eternal System. Presented at The 1998 Workshop on Compositional Software Architectures, Monterey, CA, January 6-8, 1998. [Online]. Available: http://www.objs.com/workshops/ws9801/papers/paper034.ps

[MOSK00] M. Mosko, H. Jiang, A. Samanta, L. Werner. (2000, June 4-5). COTS Software Acquisition Meta-Model. Presented at Workshop on Continuing Collaborations for Successful COTS Development, Limerick, Ireland.

[MSDN02] Microsoft Corporation. (2002, October). The Component Object Model Specification. [Online]. Available: http://msdn.ms.com/library

[MUSL99] I. Muslea, S. Minton, and C. Knoblock, "A Hierarchical Approach to Wrapper Induction," in *Proceedings of the third annual conference on Autonomous Agents*, Seattle, WA, May 1-5, 1999, pp. 190-197.

[NCUB00] C. Ncube and N. Maiden, "COTS Software Selection: The Need to Make Tradeoffs Between System Requirements, Architectures, and COTS Components," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[NEIG80] J. M. Neighbors, "Software Construction Using Components," Dissertation thesis, University of California, Irvine, 1980.

[NIER00] O. Nierstrasz and F. Achermann, "Supporting Compositional Styles for Software Evolution," in *Proceedings of the International Symposium on Principles of Software Evolution*, IAM, Universitat Bern, November 1-2, 2000, pp. 11-19.

[NIER95a] O. Nierstrasz and T. D. Meijler, "Research Directions in Software Composition," *ACM Computing Surveys*, Vol. 27, No. 2, June 1995, pp. 262-264.

[NIER95b] O. Nierstrasz and T. D. Meijler, "Requirements for a Composition Language," in *Proceedings of the ECOOP94 Workshop on Models and Languages for Coordination of Parallelism and Distribution*, Springer Verlag, 1995, pp. 147-146.

[NING00] J. Q. Ning, "A Component-Based Development Case Study," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[OBAY98] M. Obayashi, H. Kubota, S. P. McCarron, and L. Mallet. (1998). The Assertion Based Testing Tool for OOP: ADL2. [Online]. Available: http://adl.opengroup.org/exgr/icse/icse98.htm

[OBER02a] Oberon Microsystems. (2002, October). BlackBox Component Builder. [Online]. Available: http://www.oberon.ch/prod/BlackBox

[OBER02b] Oberon Microsystems. (2002, October). Jbed. [Online]. Available: http://www.esmertec.com/index.html

[OBER00] T. Oberndorf, L. Brownsword, and C. A. Sledge, "An Activity Framework for COTS-Based Systems," Software Engineering Institute, PA, Technical Report CMU/SEI-2000-TR-010, October 2000.

[OLAW00] D. Olawsky, C. Payne, T. Sundquist, D. Apostal, and T. Fine, "Using Composition to Design Secure, Fault-Tolerant Systems," in *Proceedings of the DARPA Information Survivability Conference and Exposition*, Roseville, MN, January 25-27, 2000, pp. 380-390, vol. 2.

[OREI98] P. Oreizy, N. Medvidovic, R. N. Taylor, and D. Rosenblum. (1998, January 6-8). Software Architecture and Component Technologies: Bridging the Gap. Presented at The 1998 Workshop on Compositional Software Architectures, Monterey, CA, January 6-8, 1998. [Online]. Available: http://www.objs.com/workshops/ws9801/papers/paper007.pdf

[ORFA96] R. Orfali, D. Harkey and J. Edwards, *The Essential Distributed Objects Survival Guide*, John Wiley & Sons, New York, 1996.

[PARI96] I. Parissis and F. Ouabdesselam, "Specification-based testing of synchronous software," in *Proceedings of the fourth ACM SIGSOFT symposium on Foundations of software engineering*, 1996, pp. 127 - 134.

[PARK99] S. Park and C. Wu. (1999, December 7-10). Intelligent Search Agent for Software Component. Presented at The Sixth Asian Pacific Software Engineering Conference, Takamatsu, Japan, December 7-10, 1999. [Online]. Available: http://dlib.computer.org/conferen/apsec/0509/pdf/05090154.pdf

[PAUL99] J. T. Paul, "COTS Based Open Systems for Military Avionics," *IEEE Aerospace and Electronics Systems Magazine*, Vol. 14, Issue 9, September 1999, pp. 37-42.

[PENI00] J. Penix, B. Fischer, J. Whittle, G. Pour and J. van Baalen, “Automating Component Integration for Web-Based Data Analysis,” in *Proceedings of the IEEE Aerospace Conference*, Big Sky, Montana, March 2000.

[PENI97a] J. Penix and P. Alexander, “Toward Automated Component Adaptation,” in *Proceedings of the Ninth International Conference on Software Engineering and Knowledge Engineering*, Madrid, Spain, June 18-20, 1997.

[PENI97b] J. Penix and P. Alexander, “Efficient Specification-Based Component Retrieval,” *Automated Software Engineering*, July 1997.

[PERE00] A. Gómez-Perez and A. Lozano, “Impact of Software Components Characteristics Above Decision-Making Factors,” in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[PLAK99] D. Plakosh, D. Smith, and C. C. Wallnau, "Builder’s Guide for WaterBeans Components," Software Engineering Institute, PA, Technical Report CMU/SEI-99-TR-024, December 1999.

[PLAS99] F. Plasil, S. Visnovsky, and M. Besta, “Applying Behavior Protocols to Components,” in *Proceedings of the MSPLS Fall '99 Workshop*, University of Illinois, IL, October 1999.

[POUR98] G. Pour, “Towards component-based Software Engineering,” in *Proceedings of the 1998 IEEE 22nd Annual International Computer Software & Applications Conference*, Vienna, Austria, 1998, pp. 599.

[PRIN98] R. Prins, A. Blokdijk, and N. E. van Oosterom. (1998). Family Traits in Business Objects and Their Applications. *IBM Systems Journal*. Vo. 36, No. 1, 1998. [Online]. Available: http://www.research.ibm.com/journal/sj/361/prins.html

[RAKI00] M. Rakie and N. Medvidovic, “Increasing the Confidence in Off-the-Shelf Components: A Software Connector-Based Approach,” University of Southern California, CA, Technical Report USC-CSE-2000-218, November 2000.

[RAKO00] A. Rakotonirainy, A. Bond, J. Indulska, and D. Leonard, “A Simple Component Architecture Framework,” in *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages*, Queensland Univ., Qld., Australia, June 5-8, 2000, pp. 359-

370.

[RAMI00] J. F. Ramil and M. M. Lehman, "Cost Estimation and Evolvability Monitoring for Software Evolution Processes," in *Proceedings of the WESS 2000 Workshop on Empirical Studies of Software Maintenance*, San Jose, CA, October 14, 2000.

[RATI01] Rational Software Corporation. (2002, October). Rational ClearCase. [Online]. Available: http://www.rational.com/products/clearcase/index.jsp

[REID00] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide, "Knit: Component Composition for Systems Software," in *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 23-25, 2000, pp. 347-360.

[RICH92] D. J. Richardson, S. L. Aha, and T. O. O'Malley, "Specification-based Test Oracles for Reactive Systems," in *Proceedings of the 14th international conference on Software engineering*, 1992, pp. 105 – 118.

[RIO00] A. C. del Río, J. E. L. Gayo, and J. M. C. Lovelle, "Itacio: A Component Model for Verifying Software at Construction Time," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[ROSS98] G. van Rossum. (1998, January 6-8). Glue it All Together With Python. Presented at The 1998 Workshop on Compositional Software Architectures, Monterey, CA, January 6-8, 1998. [Online]. Available: http://www.objs.com/workshops/ws9801/papers/paper070.html

[RUTT98] P. Ruttan. (1998, January 6-8). Components Should Automate Their Own Usage. Presented at The 1998 Workshop on Compositional Software Architectures, Monterey, CA, January 6-8, 1998. [Online]. Available: http://www.objs.com/workshops/ws9801/index.html

[SADA02] S. Sadaoui, "Component Generalization with Formal Methods," in *Proceedings of the ICSR7 2002 Workshop on Component-based Software Development Processes*, Austin, TX, April 16, 2002.

[SALL99] F. Salles, M. Rodriguez, J. Fabre, and J. Arlat, "MetaKernels and Fault Containment Wrappers," in *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, Toulouse, France, June 15-18, 1999, pp. 22-29.

[SANK94a] S. Sankar and R. Hayes, "ADL --- An Interface Definition Language for Specifying and Testing Software," Carnegie-Mellon University, PA, Technical Report CMU-CS-94-WIDL-1, January 1994.

[SANK94b] S. Sankar and R. Hayes, "ADL - An Interface Definition Language for Specifying and Testing Software," *ACM SIGPLAN Notices*, 29(8), August 1994.

[SCHA98] R. Schantz, D. Bakken, D. Karr, J. Loyall, and J. Zinky. (1998, January 6-8). Distributed Objects with Quality of Service: An Organizing Architecture for Integrated System Properties. Presented at The 1998 Workshop on Compositional Software Architectures, Monterey, CA, January 6-8, 1998. [Online]. Available: http://www.objs.com/workshops/ws9801/papers/paper099.doc

[SCHN99] J. Schneider and O. Nierstrasz, "Components, Scripts and Glue" in *Software Architectures — Advances and Applications*, L. Barroca, J. Hall, and P. Hall (Eds.), pp. 13-25, Springer, 1999.

[SCHU98] D. Schuschel and C. Hsu, "A Weight Analysis-Based Wrapper Approach to Neural Nets Feature Subset Selection," in *Proceedings of the Tenth IEEE International Conference on Tools with Artificial Intelligence*, Mesa, AZ, November 10-12, 1998, pp. 89-96.

[SEAC01a] R. C. Seacord. (2001). Design and Search. *SEI Interactive*. Vol. 4, No. 3, 2001. [Online]. Available: http://interactive.sei.cmu.edu/news@sei/columns/the_cots_spot/2001/3q01/cots-3q01.htm

[SEAC01b] R. C. Seacord. (2001). Automating Design Search. *SEI Interactive*. Vol. 4, No. 4, 2001. [Online]. Available: http://interactive.sei.cmu.edu/news@sei/columns/the_cots_spot/cots-spot.htm

[SEAC00a] R. C. Seacord, "Case Study: Global Combat Support System – Air Force," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[SEAC00b] R. C. Seacord, S. Hissam, and K. Wallnau, "COTS Security Services for Certificate Management," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[SEAC99] R. C. Seacord and K. C. Nwosu, “Life Cycle Activity Areas for Component-Based Software Engineering Process,” in *Proceedings of the 1999 Workshop on Component-Based Software Engineering*, Santa Barbara, California, August 1999.

[SEAC98a] R. C. Seacord, S. A. Hissam, K. C. Wallnau, “Agora: A search engine for software components,” *IEEE Internet Computing*, Vol. 2, Issue 6, November/December 1998, pp. 62.

[SEAC98b] R. Seacord, S. Hissam, and K. Wallnau, "Agora: A Search Engine for Software Components," Software Engineering Institute, PA, Technical Report CMU/SEI-98-TR-011, May 2000.

[SEGA93] M. E. Segal and O. Frieder, “On-The-Fly Program Modification: Systems for Dynamic Updating,” *IEEE Software*, Vol. 10, Issue 2, March 1993, pp. 53-65.

[SEWE00] P. Sewell and J. Vitek, “Secure Composition of Untrusted Code: Wrappers and Causality Types,” in *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, Cambridge Univ., UK, July 3-5, 2000, pp. 269-284.

[SEWE99] P. Sewell and J. Vitek, “Secure Composition of Insecure Components,” in *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, Cambridge Univ., UK, June 28-30, 1999, pp. 136-150.

[SHAW96] M. Shaw, “Truth vs. Knowledge: The Difference Between What a Component Does and What We Know it Does,” in *Proceedings of the 8th International Workshop on Software Specification and Design*, Carnegie Mellon Univ., Pittsburgh, PA, March 22-23, 1996, pp. 181-185.

[SILV00] G. E. da Silveira and S. L. Meira, “CoDelivery: An Environment for Distribution of Reusable Components,” in *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages*, Natal, Brazil, June 5-8, 2000, pp. 371-382.

[SIMM93] S. S. Simmel and I. Godard. (1993). Metering and Licensing of Resources: Kala's General Purpose Approach. Presented at The Conference on Technological Strategies for Protecting Intellectual Property in the Networked Multimedia Environment. [Online]. Available: http://www.cni.org/docs/ima.ip-workshop/Simmel.Godard.html

[SMAR98] Y. Smaragdakis and D. Batory, "Implementing Reusable Object-Oriented Components," in *Proceedings of the Fifth International Conference on Software Reuse*, Austin, TX, June 2-5, 1998, pp. 36-45.

[SPIV92] J. M. Spivy, *The Z notation: a reference manual.* Prentice Hall, 1992.

[STER96] D. Sterns. (1996, April). Migrating Existing Information Systems to Component Architecture. [Online]. Available: http://msdn.microsoft.com/LIBRARY/BACKGRND/HTML/MSDN_MIGRATING.HTM

[STUK00] R. D. Stukze. (2000, April). Software Estimation: Challenges and Research. *Cross Talk*. [Online]. Available: http://www.stsc.hill.af.mil/crosstalk/2000/apr/stutzke.asp

[SUCC01] G. Succi, W. Pedrycz, E. Liu, and J. Yip, "Package-Oriented Software Engineering: A Generic Architecture," *IT Professional*, Vol. 3, Issue 2, March/April 2001, pp. 29-36.

[SUCC00] G. Succi, R. Wong, E. Liu, and M. Smith, "Supporting Dynamic Composition of Components," in *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland, June 4-11, 2000, pp. 787.

[SULL97] K. J. Sullivan, J. Cockrell, S. Zhang, and D. Coppit, "Package-Oriented Programming of Engineering Tools," in *Proceedings of the International Conference on Software Engineering*, University of Virginia, May 17-23, 1997, pp. 616-617.

[SUNT98] SunTest. (1998). ADL 2.0 for IDL Language Reference Manual, v1.2. *The Open Group Research Institute*. [Online]. Available: http://adl.opengroup.org/documents/ADLIDLRef.pdf

[SZYP00a] C. Szyperski. (2000). Components versus Objects. *Objective View*. Issue 5, 2000, pp. 8-16. [Online]. Available: http://www.ratio.co.uk

[SZYP00b] C. Szyperski, "Component Software and the Way Ahead" in *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, Ed, Cambridge University Press, 2000, pp. 1-20.

[SZYP98] C. Szyperski, *Component Software*, Addison-Wesley, 1998.

[TENE98] J. M. Tenenbaum. (1998, January 6-8). The Economy of Component Software. Presented at OMG-MCC-DARPA Workshop on Composi-

tional Software Architectures, Monterey, CA. [Online]. Available: http://www-db.stanford.edu/CHAIMS/Doc/Papers/commerce.html

[THOM00a] S. Thomason, P. Brereton, and S. Linkman, "CLARiFi: An Architecture for Component Classification and Brokerage," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[THOM00b] B. Thomas. (2000, Spring). Component-Based Systems. *SEI Interactive*. Vol. 3, No. 2. [Online]. Available: http://interactive.sei.cmu.edu/news@sei/features/2000/spring/feature_2/feature_2.htm

[TROY97] J. M. Troya and A. Vallecillo, "On the Addition of Properties to Components," in *Proceedings of the 2nd Workshop on Component-Oriented Programming (WCOP'97)*, Jyväskylä, Finland, June 9, 1997, pp. 95-104.

[UL98] Underwriter Laboratories. (1998). UL 1998 Standard for Software in Programmable Components. [Online]. Available: http://www.ul.com/pscs/ansi.html

[VECE01] G. Vecellio and W. M. Thomas, "Issues in the Assurance of Component-Based Software," *The Edge Perspectives*, Vol. 2, Issue 1, March 2001, pp. 32-37.

[VECE00] G. Vecellio and W. M. Thomas, "Issues in the Assurance of Component-Based Software," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[VIGD98] M. Vidger, "Inspecting COTS Based Software Systems," National Research Council Canada, Technical Report NRC 41604, November 1998.

[VOAS00a] J. M. Voas, "Developing a Usage-Based Software Certification Process," *Computer*, Vol. 33, Issue 8, August 2000, pp. 32-37.

[VOAS00b] J. M. Voas, "Limited Software Warranties," in *Proceedings of the Seventh IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, April 3-7, 2000, pp. 56-61.

[VOAS00c] J. M. Voas. (2000, June 4-5). Deriving Accurate Operational Profiles for Mass-Marketed Software. Presented at Workshop on Continuing

Collaborations for Successful COTS Development, Limerick, Ireland.

[VOAS98a] J. M. Voas, "Maintaining COTS-Based Systems: Is it Possible?," in *Proceedings of the International Conference on Software Maintenance*, November 16-20, 1998, pp. 220-221.

[VOAS98b] J. M. Voas and J. Payne, "COTS Software Failures: Can Anything be Done? ," in *Proceedings of the 1998 IEEE Workshop on Application-Specific Software Engineering Technology*, Sterling, VA, March 26-28, 1998, pp. 140-144.

[VOAS98c] J. M. Voas, "Certifying Off-the-Shelf Software Components," *Computer*, Vol. 31, Issue 6, June 1998, pp. 53-59.

[VOAS98d] J. M. Voas, "Maintaining Component-Based Systems," *IEEE Software*, Vol. 15, Issue 4, July/August 1998, pp. 22-27.

[VOAS98f] J. M. Voas. (1998, January 6-8). An Application-Specific Approach for Assessing the Impact of COTS Components. Presented at The 1998 Workshop on Compositional Software Architectures. Monterey, CA. [Online]. Available: http://www.objs.com/workshops/ws9801/papers/paper002.ps

[VOAS97] J. M. Voas and K. Miller, "Interface Robustness for COTS-Based Systems," *IEEE Colloquium on COTS and Safety Critical Systems*, Sterling, VA, January 28, 1997, pp. 7/1-7/9.

[WALL00] K. C. Wallnau and D. Plakosh, "WaterBeans: A Custom Component Model and Framework," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[WALL98a] K. C. Wallnau, D. Carney, and B. Pollak. (1998, June). How COTS Software Affects the Design of COTS-Intensive Systems. *SEI Interactive*. Vol. 1, No. 1. [Online]. Available: http://interactive.sei.cmu.edu/Features/1998/June/COTS_Software/Cots_Software.pdf

[WALL98b] K. C. Wallnau, D. Carney, and B. Pollak. (1998, June). COTS Software Evaluation. *SEI Interactive*. Vol. 1, No. 1. [Online]. Available: http://interactive.sei.cmu.edu/Features/1998/June/COTS_Software/Cots_Evaluation.pdf

[WILE98] J. C. Wileden and A. Kaplan. (1998, January 6-8). Middleware as

Underwear: Toward a More Mature Approach to Compositional Software Development. Presented at The 1998 Workshop on Compositional Software Architectures. Monterey, CA. [Online]. Available: http://www.objs.com/workshops/ws9801/papers/paper061.html

[WINT02] M. Winter, C. Zeidler, and C. Stich, "The PECOS Software Process," in *Proceedings of the ICSR7 2002 Workshop on Component-based Software Development Processes*, Austin, TX, April 16, 2002.

[YACO00] S. Yacoub, A. Mili, C. Kaveri, and M. Dehlin, "A Model for Certifying COTS Components for Product Lines," in *Proceedings of the 2000 Workshop on Component-Based Software Engineering*, Limerick, Ireland, June 5-6, 2000.

[YAU00] S. S. Yau and N. Dong, "Integration in Component-Based Software Development Using Design Patterns," in *Proceedings of the 24th Annual International Computer Software and Applications Conference*, Tempe, AZ, October 25-27, 2000, pp. 369-374.

[ZARE97] A. M. Zaremski and J. M. Wing, "Specification Matching of Software Components," *ACM Transactions on Software Engineering and Methodology*, Vol. 6, No. 4, October 1997, pp. 333–369.

[ZHU97] H. Zhu, P. A. Hall, and J. H. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys*, Vol. 29, No. 4, December 1997, pp. 366-427.

INDEX