INFORMATION TO USERS

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

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

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

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

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

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

UMI[®]

**RUNTIME COMPOSITION OF SOFTWARE
COMPONENTS USING AN XML ARCHITECTURE
DEFINITION METALANGUAGE**

**By
Ian Nunn**

**SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF COMPUTER SCIENCE
AT
CARLETON UNIVERSITY
OTTAWA, ONTARIO
NOVEMBER 2001**

© Copyright by Ian Nunn, 2001

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**385 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**385, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

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

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

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

0-612-66875-4

Canada

CARLETON UNIVERSITY
DEPARTMENT OF
COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled "**Runtime Composition of Software Components Using An XML Architecture Definition Metalanguage**" by **Ian Nunn** in partial fulfillment of the requirements for the degree of **Master of Computer Science**.

Dated: November 2001

Supervisor:

Professor Dwight Deugo

Director of School:

Professor Frank Dehne

CARLETON UNIVERSITY

Date: November 2001

Author: Ian Nunn

Title: Runtime Composition of Software Components
Using An XML Architecture Definition
Metalanguage

Department: Computer Science

Degree: M.C.S. Convocation: June Year: 2002

Permission is herewith granted to Carleton University to circulate and to have copied for non-commercial purposes, at its discretion, the above title upon the request of individuals or institutions.

Signature of Author

THE AUTHOR RESERVES OTHER PUBLICATION RIGHTS, AND NEITHER THE THESIS NOR EXTENSIVE EXTRACTS FROM IT MAY BE PRINTED OR OTHERWISE REPRODUCED WITHOUT THE AUTHOR'S WRITTEN PERMISSION.

THE AUTHOR ATTESTS THAT PERMISSION HAS BEEN OBTAINED FOR THE USE OF ANY COPYRIGHTED MATERIAL APPEARING IN THIS THESIS (OTHER THAN BRIEF EXCERPTS REQUIRING ONLY PROPER ACKNOWLEDGEMENT IN SCHOLARLY WRITING) AND THAT ALL SUCH USE IS CLEARLY ACKNOWLEDGED.

Abstract

Many approaches to reducing the cost, complexity and development time of software applications have been explored within the discipline of Software Engineering. Several language-based techniques, most recently the Object Oriented paradigm, have been developed to facilitate the reuse of code. Even when these techniques employ the notion of components, they do so only in a limited way within the language's environment at development time.

Recently, the study of Software Architecture has emerged to provide language support called an Architecture Definition Language (ADL), for building configurations of components into runnable systems.

In this thesis we develop a formal representation of software component architecture concepts where none currently exists. We use this representation to derive a specification for a unique, XML-based ADL. Finally, we specify and develop a prototype engine that uses this ADL for the automated assembly of software applications.

Acknowledgements

I would like to thank Dr. Dwight Deugo, my supervisor, for his thoughtful guidance, many suggestions and constant support during this research. The regular coffee meetings over this period will be a fond memory. I have become a better writer under his tutelage, and this I appreciate. The iterative refinement he has overseen has helped to make this thesis a better document. I would like to thank Dr. Timothy Lethbridge for his careful review of and comment on the document. I would like to express my appreciation to Dr. Tony White and Dr. Lethbridge for those constructive recommendations given during this document's defence that make it a more accurate and complete work. Finally, I would like to thank Texar Inc. and the Ontario Graduate Scholarship Program for financial support.

I am especially grateful to my wife Mary for her constant support and encouragement, patience and *love*. She has endured many years of partial attention and cancelled holidays in order that I might reach this point. Indeed this work is very much "ours".

Ottawa, Ontario
November, 2001

Ian Nunn

Table of Contents

Abstract	iv
Acknowledgements	v
Table of Contents	vi
List of Tables	ix
List of Figures	x
1 Introduction	1
1.1 The Problem	1
1.2 Motivation	2
1.3 Context: The Major Initiatives Behind	
Component-Based Computing	4
1.3.1 Software Architecture and Software Engineering	5
1.3.2 Process Algebras and Associated Calculi	5
1.3.3 Distributed Computing	6
1.3.4 Component-Based Software Engineering	6
1.3.5 Active Networks	7
1.4 Goals and Expected Contributions	8
1.5 Organization of This Thesis	10
2 Software Architecture	11
2.1 Background	11
2.2 Architectural Models	14
2.2.1 Components	16
2.2.2 Connectors	17
2.2.3 Configurations	20

2.2.4	Non-Functional Properties	23
2.3	Architectural Style	24
2.3.1	Style-based refinement	26
2.4	Architecture Definition Languages	27
2.4.1	Common ADLs	29
2.4.2	Inadequacies of Programming Languages	30
2.5	XML as an ADL Metalanguage	31
2.6	Summary	33
2.6.1	Architectural Domain	33
2.6.2	Architectural Style	36
2.6.3	Architecture Definition Language	36
3	An XML Architecture Definition Metalanguage	38
3.1	Formal Representation	39
3.1.1	Notational Conventions	41
3.1.2	Architectural Domain	42
3.1.3	Architectural Style	45
3.1.4	Architectural Design	47
3.2	An XML Metalanguage ADL Specification	47
3.2.1	The Component Schema	50
3.2.2	The Composition Schema	56
3.2.3	The Architectural Style Schema	61
3.3	An Engine Design for the Architecture	63
3.3.1	Choice of Operating Environment	64
3.3.2	XML Parsing and Interpretation	65
3.3.3	Component Discovery	66
3.3.4	Component Assembly	67
3.3.5	XML Component Wrapper Construction	69
3.3.6	Component Binding and Execution	70
3.4	The XML Metalanguage Specification as an ADL	70
4	Prototype Implementation and Discussion of Results	73
4.1	A Scheme Implementation of a Composition Engine	73
4.1.1	A Connector-Based Model Engine	74
4.1.2	A Connector-less Model Engine	75
4.1.3	A Component-Based Model Engine	79
4.2	Discussion of Results	81
4.2.1	Treatment of Data and Data Repositories	82
4.2.2	Suitability of the XML Specification	82

4.2.3	Success of the Engine Prototype	84
5	Conclusion	86
5.1	Evaluation of Goals	87
5.2	Evaluation of Contribution	89
5.3	Further Work	90
5.3.1	Component Location and Discovery	90
5.3.2	Component Identification and Interface Semantics	91
5.3.3	System Calls and Error Handling	93
5.3.4	Security	94
5.3.5	Dynamic or Runtime Adaptation (Hot Swapping)	95
5.3.6	Constraint Checking	97
5.3.7	Other Domains of Relevance	98
5.3.8	Tools	99
A	List of Acronyms	101
	Bibliography	104

List of Tables

3.1	ADL Component Characteristics.	71
3.2	ADL Configuration Characteristics.	72

List of Figures

3.1	XML component code specifying an and-gate.	53
3.2	Engine generated XML code specifying a half-adder circuit component.	54
3.3	XML configuration code to construct a half adder circuit.	59
4.1	Logic gate diagram of a half-adder circuit.	74
4.2	XML component code specifying a connector-based half-adder.	76
4.3	XML composition code for the connector-based half-adder.	77
4.4	XML composition code for creating a half-adder test-bed.	78
4.5	Component architecture of the third generation engine.	79
4.6	XML component code specifying a connector-based half-adder.	80

Chapter 1

Introduction

As software application markets mature, products generally increase in functionality, complexity, size and cost. Choice shrinks as smaller vendors are forced out of the market and it becomes virtually impossible for new vendors to enter. The major surviving wordprocessor, Microsoft Word, is an example of such a product, so complex that most users employ only a small fraction of its capabilities.

For new application concepts, the time and cost investment to bring most to market is very large requiring in the least, many person years of development time.

1.1 The Problem

This state of affairs creates a number of problems for users. They have little choice over the functional content of the products they pay for. They are locked into the vendor's marketing cycle of bug-prone upgrades. They may be tied into specific operating systems and operating environments.

For the developer there are likewise a number of major problems. Developing a new product from scratch involves a large investment in building generic functionality both in dollars and time. Maintenance of large monolithic systems is similarly costly.

The root of the problem lies with the traditional models of software development - those that use current programming languages and development environments. Even though most languages now support some degree of component modularity encouraging reuse, the macroscopic structure of systems is buried within the programming models, intermixed with support for algorithmic computation. This results in systems that are still costly, complex entities whose structures are difficult to discern at best.

1.2 Motivation

Instead of the traditional time-consuming method of building and distributing large monolithic applications we need a new paradigm that would allow users to assemble applications where and when they need them and containing exactly the functionality that meets their needs and specifications. Such a paradigm would require the following:

- standard components, each with a well-defined interface and behavior offering choice of vendor and low cost,
- a simple means to specify how the components should be associated or connected together,
- dynamic assembly of the standard components guided by the specification, noted in the previous bullet, to create a composite application on the user's computer system that is immediately runnable,
- dynamic upgrading of the composite application by unplugging components and plugging in new versions on demand even while the composite is running.

The benefits from such a component-based architecture include [WRMT95] [NR99] [Aoy98]:

- decreased system development time or rapid time to market,
- increased productivity of development personnel,
- decreased system development cost from low component cost due to amortization of component development costs over many users,
- a wide range of both general purpose and domain-specific components from multiple vendors,
- scalability of systems in terms of size, complexity, transaction rates, number of users, and any other relevant parameters,
- robustness of products.

These benefits are sufficiently important to motivate the development of proper languages, tool sets and development environments to allow the creation of application software by component assembly.

To understand the evolution of component technology, component-based engineering and component software architecture, we examine the primary sources of development of these notions.

1.3 Context: The Major Initiatives Behind Component-Based Computing

Explicit reference to and discussion of software components can be traced back at least as far as a paper by McIlroy [McI76]. His position is stated as:

My thesis is that the software industry is weakly founded, and that one aspect of this weakness is the absence of a software components sub-industry.

He then discusses some of the properties he envisions for a component-based industry. It would “offer families of routines” so users could choose ones with only the functionality they need. Components would be seen as “black boxes” and chosen from catalogues. He anticipates abstraction by noting the need for parameterized components. Finally, he notes seven aspects of components that will require explicit specification: precision of data, robustness of performance, generality or parameterization, time-space behavior, interface specification, access method and data structure.

McIlroy focuses a topic that had been under investigation in various forms by many researchers of the period. He offers a statement of direction for most subsequent work on components and software architecture. This subsequent work is found in five broad areas of research: Software Architecture and Software Engineering, Process Algebras and Associated Calculi, Distributed Computing, Component-Based Software Engineering and Active Networks. In the following subsections, each of the areas is briefly reviewed along with its relationship to this thesis.

1.3.1 Software Architecture and Software Engineering

The discipline of software engineering has developed an extensive body of research on components and component architectures under the topic of software architecture. Because of the background importance of this subject, Chapter 2 of this thesis is devoted to a detailed examination of the subject.

1.3.2 Process Algebras and Associated Calculi

At the time McIlroy was writing, early theoretical work was being applied to the composite behavior of concurrent asynchronous processes. The seminal books by Hoare [Hoa85] and Milner [Mil80] extended process algebra to develop variants of a process calculus as means to formally describe the behavior of such systems. Milner in particular talks of agents (components) and their composites under communication. In his book, Milner states his objective as that of developing a calculus of concurrent systems, the analogue of the λ -calculus for sequential algorithms.

Hoare's work, similar to and in parallel with that of Milner, provides the formal basis for an Architecture Definition Language (ADL) such as Wright [SG96a].

Milner's later work moves significantly towards his goal with the π -calculus [Mil93] [MPW89]. Others such as Lumpke [Lum00], have extended this work to allow the formal inclusion of name-based parameterization (such as that used with the eXtensible Markup Language (XML)) in addition to parameters in a tuple format as provided by Milner. Lumpke's work deals directly with the notions of components and component architectures.

The contribution of this area of research is primarily in providing means of formal specification of component behavior allowing for formal analysis of correctness of

composite systems. Consideration of specific detail is noted where applicable but not explored in this thesis.

1.3.3 Distributed Computing

The paradigm of distributed computing extends naturally to descriptions based on components and component architectures. The distributed elements of computation each have a natural embodiment as separately compiled code with at least implicit interface, semantics and behavior. While process algebra focuses on behavioral issues, this area focuses more on software development and system operation issues.

When issues related to the distributed structure of systems are examined, discussion moves into the domain of architecture, architectural style and component specification (interfaces) [Cra96] [MDEK95][PD99].

The influence of distributed computing on component architecture has been important even if it is not usually separately addressed within component domains. Its interests will be encompassed in the general review of software architecture in Chapter 2.

1.3.4 Component-Based Software Engineering

The Component-Based Software Engineering (CBSE) initiative seems to have evolved from work with object-based component middleware frameworks, notably CORBA, DCOM and Enterprise JavaBeans. Its goal [Aoy98]

is composing applications with plug & play software components on frameworks.

What distinguishes CBSE from the broad area of software component architecture is its focus on the association with existing object component middleware. The resulting interdependency is exemplified by the statement of Brown and Wallnau [BW98] that,

our understanding of what makes a component a component is inextricably linked to our understanding of the architectural constraints imposed on components by a component framework-cum-object model.

In the articles cited in this section, the lack of citations referencing any of the work on the topics of software architecture and ADLs, the central topics of Chapter 2 in this thesis, is another indication of the restricted focus of CBSE. Further, although there is mention of tool requirements to support CBSE [Bro98], there is no reference to existing tools suggesting development in the CBSE area is very early.

Concepts identified as desirable [Bro98], including “static structure of the elements of interest” (configuration), dynamic behavior of elements, multiple views, reuse, completeness and consistency checking (analysis) and interface issues are extensively discussed in the broader domain of software architecture. We will therefore not consider CBSE further in this thesis since our interest is not in the referenced middleware.

1.3.5 Active Networks

The Defense Advanced Research Projects Agency (DARPA) in the United States has sponsored a major area of network research concerned with the creation of a network that is flexible and extensible at runtime with respect to services offered [Age] [MP00].

Network architecture is based on a set of components representing network services that may be dynamically composed to process “smart packets” on active network nodes [Act99]. Architectures are primarily based on the pipe and filter style [AN 98].

Since the concepts contributed to the theory of component architecture are contained within the larger scope of software architecture in general, this field of research is not given further direct consideration in this thesis.

1.4 Goals and Expected Contributions

The primary goal of this thesis is to develop a component assembly model that:

- allows for the specification of a standard well-defined component format,
- provides a simple to use mechanism for specifying component assemblies or compositions,
- provides dynamic or runtime assembly which can be performed on most computers under most operating systems,
- allows for such assembly to proceed in an automated fashion with the composite being automatically run on completion of assembly,
- allows for the dynamic modification of a composite assembly.

A second goal is to implement such a model and demonstrate the successful assembly and execution of test examples.

In achieving these goals, we expect to make the following contributions to the discipline of software engineering and component assembly in particular:

- A formal definition of terms used in the discipline of software engineering for describing software components and component configurations or architectures.
- An XML representation of an ADL that meets all the commonly accepted requirements of an ideal ADL. This representation consists of two major parts:

A Component Schema: An XML-based set of instructions that describe a component sufficiently that an engine may properly select, locate and connect the component to other components.

A Composition Schema: An XML-based set of instructions that describe how a set of components are to be connected or “wired” together.

- an XML schema to model architectural style and a proposal of how the style information may be used by an engine to produce correct component assemblies.
- A technique for the automated assembly of components into persistent composite structures, and their execution. The assembly may be performed on any specified computer running a suitable engine. Components may exist locally or remotely and their composite may be local or distributed.
- A prototype of an engine that demonstrates the feasibility of the above concepts.

We claim that based on a survey of the literature, the application of XML that we demonstrate is unique. The formal representation is also unique and although there are many proposed composition systems we will demonstrate a simple working prototype of a composition engine.

1.5 Organization of This Thesis

As noted, Chapter 2 examines the field of software engineering with respect to components and component architectures in some detail. The notion of architectural models is reviewed followed by a review of the notion of architectural styles. Next, the characteristics of ADLs are reviewed. Then, the use of XML to specify ADLs is examined. Lastly, the important characteristics identified in these sections are summarized.

In Chapter 3 a formal representation of the concepts explored in Chapter 2 is developed. This provides the basis for creating the specification for our use of XML as an ADL. These are given for both components and configurations. Finally, our XML specification is compared to the characteristics summarized in Section 2.6.

In Chapter 4 we outline the development of our engine prototype and its testing, and discuss the results of our work. Section 4.1 reviews the evolution of the design of our engine prototype and details of some of the testing. Section 4.2 presents a discussion of the results of this research.

The final chapter in this thesis evaluates our claims and contributions to the solution of the problem identified in Section 1.1. This is followed by a discussion of some of the open areas of work that are identified in the literature and that this research has uncovered.

Chapter 2

Software Architecture

To understand the context for the central ideas in this thesis, it is necessary to examine the evolution of the discipline of software engineering, particularly with respect to developments in software architecture. This chapter begins with the identification of some of the early concepts related to software architecture discussed in terms of architectural models and styles. It then examines current work done with architectural definition languages that are the key tools with which specific architectures are built and systems are implemented. This chapter concludes with a summary of key concepts related to software architecture and architecture definition languages.

2.1 Background

In the 1990's, the field of software engineering evolved from the study of software design to the study of the structure of software systems [Sha01]: what has become known as software architecture. In particular, the need to find methods for describing the architecture of distributed systems was a strong force motivating this

research [Cat95].

In their seminal article on software architecture, Perry and Wolf [PW92] identify four benefits of an architectural approach to software development as:

- a means of satisfying requirements,
- a basis for design, cost estimation and process management,
- a basis for reuse of elements,
- a basis for dependency and consistency checking among elements.

They define software architecture as being comprised of three parts:

- **Software Elements** consisting of data elements which carry the data of the system, processing elements that transform data and connection elements that provide “glue” for the other elements.
- **Form** consisting of weighted properties and relationships intended to convey constraints among elements in the architecture.
- **Rationale** a means of specifying non functional aspects of the architecture such as economics, performance and reliability.

Shaw and Garlan [SG96b] give the characterization,

software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on those patterns.

This expression marks a shift in emphasis from algorithms and data structures to the topology of the system being designed and the interactions among components.

Cattaneo [Cat95] defines four viewpoints from which the concept of software architecture may be described:

- **Conceptual:** A view used for requirement definition and high-level design showing the domain-specific elements of a system and their relationships.
- **Execution:** A view that describes elements of the runtime environment including objects, threads, communication mechanisms, system resources and element location and migration.
- **Static:** A construction-time view that shows the relationships, dependencies and constraints of system elements and their organization.
- **Development Environment:** A view that captures system elements from the developer's perspective including the files and libraries used by editors, linkers, compilers, interpreters and integrated development environment (IDE) tools.

This classification demonstrates the variety of differing viewpoints that may be adopted by the various parties involved in developing, implementing and operating software systems. The research described in this thesis concerns issues related to the assembly of components both from a static and dynamic perspective and represents a fusion of static and execution views.

2.2 Architectural Models

From the early days of system design, the structure of software systems has been informally modelled by box and arrow diagrams as a means of capturing interaction mechanisms among system elements at a high level of abstraction [Cat95]. The informal nature of the representation has meant different practitioners have used the same graphical elements to represent different domains of discourse, different concepts within a domain and different levels of abstraction. Even within the domain of a single practice, meaning across system structure diagrams may be inconsistent.

The utility of such informal methods inferred from their widespread usage suggests we would want to model system architectures with some ontology corresponding to boxes and lines, or components and connectors. Since the problems alluded to in the above paragraph arise from the lack of an effective means of modelling semantic content of boxes and particularly lines, a first step to architectural modelling would be some classification of perspectives from which a domain and its associated semantic content may be described. To this end Shaw and Garlan [SG96a] identify five classes of architectural models:

- **Structural:** These models are based on components, connectors and semantic elements that capture configuration, rationale, constraints, style, analyses and requirements.
- **Framework:** Models that emphasize the structure of the whole over the details of the parts and tend to be specific to a problem domain.
- **Dynamic:** Models that capture behavior of systems and system elements in a manner that allows configuration change on the fly.

- **Process:** Models that focus on a constructive or procedural description of a system contained in a process script.
- **Functional:** Views of architecture as functional components that provide layers of service composed upwards.

These classes of models demonstrate the variety of differing approaches to modelling architectures for developing and implementing software systems. However, the most widely adopted models for discussing software architecture are structural variants consisting of computational components and structural connectors [SG96b] [AG94], both represented as first class objects, and a set of possible configurations. As Shaw and Garlan observe about the pipe and filter style of component and connector architecture,

an important property of this model is that pipes are self-contained entities and can be reasoned about independently of filters.

Recently, a box and line graphical notation has been formally specified with both graphic and text representations in the Unified Modelling Language (UML) [Obj01]. In UML, notions of components, constraints and connections (associations) have explicit definitions. To date, however, UML has not been used in association with existing ADLs.

As noted earlier in this section, the literature uses the concepts of components, connectors and configurations extensively. The following subsections capture the salient points regarding each concept.

2.2.1 Components

The notion of a component as a discrete, named, body of code has existed from the earliest stages of software engineering including code modules, executable binaries and other representable software objects. The role of the component in software architecture today is that of a locus of computation and state. Examples include programs, objects, processes, filters, databases, clients and servers [SG96a].

The common model of a component includes an interface defined by a set of ports of explicit type, and a specification of its function. A port defines a logical point of interaction between the component and its environment [AG94]. Associated with each port should be a specification of what facilities the component provides to or requires of other components. Function should include a description of the component's behavior in a form to facilitate analysis [LV95] and operation [CR99].

Medvidovic and Taylor [MT00] give the desired characteristics of a model of a component as specifying:

- **Interface:** The set of interaction points (ports) and associated services a component offers to or requires of its environment.
- **Type:** The abstract representation of the encapsulated functionality of the component that is instantiated in a specific architecture.
- **Semantics:** An explicit specification of a component's behavior.
- **Constraints:** A set of properties or assertions that if violated will result in improper functioning of the component.
- **Evolution:** The ability to accommodate change in a component's behavior,

functionality or interface over time.

- **Nonfunctional Properties:** Those properties that cannot be directly associated with or derived from a component's behavior.

These are the characteristics of a component we would want to be able to express in any language used to describe architectural composition.

A notion we will classify under components is a technique for handling legacy code. This involves creating a “wrapper” for the code that provides a component interface for it [Kni99].

2.2.2 Connectors

Traditionally, architectural representation in the form of box and line diagrams has provided much richer expressive capability for the semantics of boxes than for the semantics of the connections (lines) that often do not carry code and lack the discrimination among the different lines in a diagram to capture semantics of different types of interaction [SG96a].

In current software architecture the notion of the connection between two components is made explicit in the form of a connector. This moves much of the semantics and behavior of interaction, specifically communication and coordination [GMW97], from individual components into generic elements ranging in complexity from simple procedure calls to a range of complex protocols including multicast channels, event handlers, client-server protocols, database-accessing protocols, shared data and blackboards, and event dispatchers (implicit invocation) [AG94] [SG96b] [SG96a]. Connectors describe the relations among components.

Shaw and Garlan give six reasons why connectors deserve first class status [SG96b]:

- **Degree of sophistication:** A connector may implement a complex specification such as a communication protocol.
- **Localization:** The interaction between two or more components may be complex, deserving of its own localized description.
- **Abstraction:** By abstracting out the parameters of an interaction, its applicability may be extended to a class of architectures rather than just a single instance.
- **Distributed implementation:** An interaction may be distributed over several system elements in a distributed (physical) architecture.
- **Independence:** Independence of connectors from components may facilitate dynamic change at runtime. It may also facilitate static change such as the case where a communication protocol changes. In this case, a large variety of components may be unaffected, with changes localized to a small number of generic connectors.
- **Reusability:** Connectors implementing stylistic constraints become reusable within the style.

The common model of a connector includes an interface defined by a set of roles. A role has explicit type and a glue specification describing the interaction between the components that the connector connects. The role describes the expected local behavior of each of the components to which it connects [AG94] [SG96b]. The glue provides any data adaptation needed [MS77] [CWD00] and manages control issues.

In actual implementation, connectors may be refined to a set of local or remote procedure calls.

Medvidovic and Taylor [MT00] give the desired characteristics of a model of a connector as specifying:

- **Interface:** The set of interaction points and services associated with the components it connects. These services are offered by the connector not directly but in the sense of a proxy for those services offered by components to which it is connected. So a client-server connector offers the service provided by a server to a client. It does not itself create the service or modify it but only mediates its delivery.
- **Type:** The abstract representation of the encapsulated communication functionality of the component that is instantiated in a specific architecture.
- **Semantics:** An explicit specification of the communications protocol that specifies the component interaction that the connector mediates.
- **Constraints:** A set of properties or assertions that if violated will result in improper interaction among components the connector is attached to.
- **Evolution:** The ability to accommodate change in a connector's behavior, protocol or interface over time.
- **Nonfunctional Properties:** Those properties that cannot be directly associated with or derived from a connector's behavior.

As with components, these are the characteristics of connectors we would want to be able to express in any language used to describe architectural composition.

Some ADLs like Darwin lack connectors as a basic element. When complex communications protocols require representation this is performed by representing them as components.

2.2.3 Configurations

The final architectural element is configuration. A configuration must address structural composition issues, primarily control issues and data issues since these comprise the semantic content of the connectivity.

Control issues include [SC97]:

- **Topology:** The geometric form the wiring must take to characterize all of the control requirements of the composite entity.
- **Timing:** Dependencies among components in terms of the effects of their processes.
- **Binding:** At what stage are the control relations among components and connectors bound? At composition-time or runtime?

Similarly, data issues are:

- **Topology:** The geometric form the wiring must take to characterize the data flows of the composite entity.
- **Continuity:** Is data stream-oriented, batch-oriented or transaction-oriented? The answer affects the style (and hence configuration) chosen to represent the system.

- **Mode:** Is data passed as a discrete element, requested by implicit invocation, shared as in blackboard systems, or transmitted by some other means?
- **Binding:** At what stage are the data relations among components and connectors bound? At composition time or runtime?

Whereas the architectural elements of components and connectors in Sections 2.2.1 and 2.2.2 are described at the level of type or class, a configuration specifies the relations among instances of types or classes. So a configuration could include instances $C_1, C_2, \dots, C_k$ of some component C .

Medvidovic and Taylor [MT00] give the following classification of features of configuration specification:

- qualities of the configuration description:
 - *Compositionality:* The ability to specify hierarchies of components and composites, allowing for specification and development of systems through successive levels of abstraction.
 - *Refinement and Traceability:* Support for the correct, consistent and traceable refinement of an architectural specification into an operational system via a programming language implementation.
 - *Heterogeneity:* The ability to support a heterogeneous mix of system elements measured by topological location, granularity, style, implementation language, operating system support or communications protocol support.
- qualities of the described system:
 - Heterogeneity.

- *Scalability*: The ability to support the development of large-scale systems through means such as hierarchical composition.
 - *Evolvability*: Those features that allow systems to be changed or to evolve by the removal, replacement or new addition of system elements in an off-line manner.
 - *Dynamism*: Those characteristics that allow for runtime modification or the dynamic adaptivity of a system architecture.
- properties of the described system:
 - Dynamism.
 - *Constraints*: These are properties or assertions of a global nature that apply to the system as a whole. They may also exist for compositions of components and connectors capturing properties that cannot be expressed for individual elements alone.
 - *Nonfunctional Properties*: Those properties that capture development-time and runtime system-level issues.

For robust systems, configuration design must accommodate both anticipated and unanticipated dynamic adaptation. Particular issues of concern are component discovery, version management, interface changes including type, ports and behavior, and individual instance replacement (binding) [Kni99] [MS77].

In a model where algorithmic or computational complexity is hidden inside components and complex communications protocols are hidden inside connectors, the system developer is faced only with structural complexity. However, the issues that we have just listed that affect structural design, especially the successive levels of

abstraction possible, make the system designer's task potentially more complex and difficult than that of the designer of individual components.

2.2.4 Non-Functional Properties

There are a large number of system properties that are not captured by the descriptive properties of the architectural elements. These include [SG96a]:

- timing issues such as time requirements, timing variability, real-time response and latency,
- resource issues such as bandwidth, space requirements, space variability, minimum hardware configuration and access to specialized hardware,
- security,
- performance measures such as throughput, robustness, precision and accuracy, service capacity, ease of use, reliability and adaptability,
- conformance to interface standards and implementation standards.

Some of these are qualitative rather than quantitative and do not lend themselves easily to formal specification. How to incorporate them if indeed this is possible, into an architectural specification is an open problem. However, in Chapter 3 we offer a formal characterization of architectural concepts capable of capturing much of this information.

2.3 Architectural Style

Experience has shown that certain software structures or patterns of architecture appear over and over again such as the client-server pattern [BMR⁺96]. It is therefore useful to consider what constitutes an architectural style, hereafter referred to simply as a style, and to identify common examples.

One definition of an architectural style is a tuple consisting of a set of design elements or components, a set of well-formedness constraints on how they may be used and a set of connectors with semantic definitions particular to the style [MQR95].

Another characterization [SG96b] [GAO94] [Cat95] includes a vocabulary of design elements of components and connectors, a pattern of structural organization determined by a set of constraints and configuration rules, a set of underlying semantic models, and a description of analyses that can be performed on resulting systems.

Shaw and Clement [SC97] offer a classification of style based on control and data issues including:

- the kinds of components and connectors used,
- control and how it is manifest across a configuration,
- data and how it is communicated across a configuration,
- the interaction between control and data,
- the type of analysis (reasoning) supported by the style.

Others have observed that the features of an architectural style are that it

encapsulates important decisions about the architectural elements and emphasizes important constraints on the elements and their relationships [PW92].

It provides a mechanism for categorizing architectures and for defining their common characteristics [NR99].

In addition, styles can capture architectural assumptions, particularly those created by the use of specific middleware platforms. They provide a vocabulary of design elements, namely those components and connectors employed by the style.

Widely cited examples of architectural styles are listed below but not discussed in the context of this thesis. They include [SG96b] [MQR95] [Sha01] [GAO94]:

- Call-and-return systems: main-program and subroutine, OO designs and hierarchical layers.
- Data-centered systems or repositories: databases, hypertext systems and black-board shells.
- Independent components: communicating or cooperating processes and event systems or implicit invocation.
- Dataflow systems: batch sequential and pipe and filter.
- Virtual machines: interpreters and rule-based systems.
- Distributed computation systems: agents and peer-to-peer systems.
- Process control systems.

- Functional programming systems.
- Client-server systems.

The representation of components and connectors, common among these styles includes a well-defined interface [AG94]. For a component it consists of a set of ports that define the logical points of interaction between a component and its environment, and a specification that defines the behavior of the component. For a connector it includes a set of roles that define the participants of interaction and a specification of the glue or operations binding the components it connects. In some architectures, connectors like components, may be decomposed into subarchitectures. They also carry semantic description. When such subarchitectures are created, it is important to ensure bindings at the architectural level composite are promoted to the correct subcomponent [Kni99] [Men00].

2.3.1 Style-based refinement

The notion of refinement of styles is important for promoting flexibility in architectural design. In Chapter 3, we offer a formal representation that encompasses the concept. A brief review of the topic as found in discussion on software engineering follows.

Refinement in the classical sense involves the notion of behavioral substitutability [Gar96]. This is also a principal notion in process algebra found in Hoare's Communicating Sequential Processes (CSP) [Hoa85], the formal basis for Wright, and in Milner's π -calculus [MPW89], the formal basis for Piccola [ALSN00]. A second notion of refinement is provided by Moriconi et al [MQR95] as a mapping from one architectural schema to another.

So architectural refinement is the mapping or transformation of one architectural style onto another where we wish to preserve not only current but future behavioral equivalence and substitutability. A problem arises, however if one style includes a criterion the other style forbids (by exclusion in its specification). For example if a style representing an acyclic data flow as in the case of a linear pipe-and-filter architecture is mapped onto a style allowing configurations of cyclic graphs (e.g. feedback loops), adding a feedback connection at a later time could be done in the architecture of one style but not the other.

Moriconi et al. [MQR95] argue that a stronger correctness criterion is needed when determining substitutability, namely a specification of not only the permissible connections but those that are not permissible.

This same issue applies to refinements among substyles where there is an intent to map one substyle onto another substyle of a common parent style.

2.4 Architecture Definition Languages

An architecture definition language is a language for describing a system architecture. ADLs differ from ordinary programming languages in their ability to explicitly represent configuration or structure. An ADL takes as its basic elements, components and connectors and includes a syntax and semantics for expressing configurations of basic elements [NR99] [MT00].

An ADL can facilitate the formal description of the structure of a software system. It can also provide the basis for analyzing composite structures.

Attributes associated with an ideal ADL include [MT00] [Cat95] [NR99] [SG96b]:

- ability to model components in terms of property assertions, and interface including type and specification (function or behavior),
- ability to model connectors whether simple or complex in terms of property assertions, and interface including type and specification (glue or protocol),
- ability to specify configurations including hierarchical composition through topological and other constraints, both statically at composition-time and dynamically at runtime,
- support for dynamic identification and creation of components and connectors,
- ability to specify and compose behavior implying granularity of structure,
- ability to define and refine styles and discriminate views,
- support for refinement of composite elements in terms of more basic elements,
- support for abstraction and encapsulation, both of algorithmic computation and communication, and the parameterization of such,
- support for reusability of independent components and connectors,
- acceptance of heterogeneous elements both in terms of style and implementation (running environment),
- support for analysis.

Of these properties, the first three are most critical. The ADL ACME [GMW97] that was designed to provide an interchange format for architectural development

tools and environments takes as its canonical elements or types, components, connectors and systems or configurations, demonstrating this importance. In ACME, the remaining properties from the above list have other syntactical expression.

2.4.1 Common ADLs

Medvidovic and Taylor [MT00] provide a comparison of the most common ADLs' abilities to model components, connectors and configurations based on the authors' characterizations detailed in Sections 2.2.1, 2.2.2 and 2.2.3. The ADLs they discuss are ACME [GMW97], Aesop [GAO94], Chiron-2 or C2 [WRMT95], Darwin [MDEK95] [Dep97], MetaH, Rapide [LV95], SADL, Unicon, Weave and Wright.

Cattaneo [Cat95] classifies the above ADL characteristics as a set of 15 requirements that he uses to compare Aesop, Darwin, Rapide, UniCon, and Wright. we do not reproduce his classification here.

Other languages and frameworks that support component models or have some of the properties of ADLs include:

- Less common or older ADLs such as Conic [MKS89] and Yasmin [Der97] [Der].
- Languages or language extensions that support the component models DCOM [Mic96] and COM+ [Edd99], CORBA, and Enterprise Java Beans including JWire [PD99], ICARIS [Men00], and Bean Markup Language [CWD00].
- Specifications that are associated with research on networks such as Netscript [dSFY98], Service Location Protocol (SLP) [Gut99] and Distributed feature Composition [JZ98].
- Mobile agents used to model components [Lim96] [BM98].

We will not discuss these languages in this thesis as there are a number of excellent articles describing, analyzing and comparing them in detail [MT00] [NR99] [Sha01] [SG96a] [Cat95] [Men00]. However, the classification schemes we will use in evaluating the ADL developed in this thesis.

2.4.2 Inadequacies of Programming Languages

The question of why an existing programming language is not used as an ADL is discussed briefly in this section. No existing programming language is fully satisfactory as an ADL as each fails to meet the requirements of an ADL in one or more respects. The most common deficiencies or problems are [SG94]:

- **Module Import/Export:** When one module or package is imported into another, the exact names of functions within the imported module must be known and used by the importing module. This causes the problem of having to “look inside” the imported module to discover such names. Furthermore, changes to names within the imported module force modifications in the importing module. The second problem this causes is that the system interconnection structure becomes embedded in the importing module leading to a third problem, that of the intermingling of algorithmic and architectural content within a module.
- **Abstraction:** Programming languages restrict, often severely, the number of interaction mechanisms available to the designer limiting the degree of abstraction possible and the complexity of interaction that can be explicitly represented. The worst case is a language that provides only procedure calls and shared data.

- **Reuse:** Explicit import statements in modules prevent the substitution of alternative implementations and provide no support for reuse of patterns of composition.
- **Configuration:** Because the pattern of import and export statements is distributed across and embedded in programming modules, the overall configuration of the system is not readily apparent. Dynamic reconfiguration is not a capability supplied by these languages.
- **Heterogeneity:** In most cases, modules written in different languages cannot be composed easily if at all. Nor is there language support for different levels of abstraction.
- **Analysis:** Because the overall configuration is not visible, it is not possible to perform any analysis of it to detect such conditions as the possibility of deadlock.

In summary, programming languages were designed primarily for algorithmic expression and have little support for the structural expression necessary to capture the higher level requirements and abstractions as well as the non-functional characteristics of software architectures.

2.5 XML as an ADL Metalanguage

There have been few attempts to use XML as an ADL. Rather, XML has been used as an Interface Definition Language (IDL) in the manner that middleware platforms

like CORBA and DCM employ. In this section we review research that uses XML in any way related to components and software architecture.

Mckee and Marshall [MM99] describe the use of XML Document Type Definition (DTD) schemas for specifying the structure of the metadata specification of a component (i.e. its interface, semantics, constraints and other properties). Such schemas are hierarchical and can be composed in the sense that any element in the schema may itself be a DTD for some subset of the metadata associated with the component. Constraints in the form of policies, preconditions and postconditions are easily defined as XML elements or hierarchical structures of XML elements.

The first level in Mckee and Marshall's component DTD consists of four elements which we list without discussion, <specification>, <commerce>, <environment> and <usage>. These they decompose several levels further to capture all characteristics needed for the complete specification of a component.

Mennie also uses an XML DTD to capture the metadata specification of a service component in his ICARIS component framework [Men00]. The first level in his hierarchy consists of eight elements, <name>, <service provider>, <version>, <priority>, <dependencies>, <composable methods>, <info> and <functionality>. Some of the elements decompose another level.

Issues similar to those relating to components such as discovery, version management, change management, location and storage exist for DTDs used in the above contexts.

Chen et al. [CHDG99] model dynamic workflow applications as plug and play multi-agent systems. They use XML not as an IDL for components but as the language of communication among agents. Thus if agents are taken as components,

communication constitutes the interaction mediated by connectors. This in a sense, uses XML to describe connectors as opposed to the above examples that use XML to describe components.

Pruitt et al. [PSSC98] discuss the merits of XML as an ADL metalanguage and hint at work in developing schemas but offer no examples.

In summary, very limited use of XML has been made in providing the functionality of an IDL to support JAVA components. No examples or descriptions of XML used as a full ADL metalanguage have been found. In particular, none have used XML to describe configurations explicitly.

2.6 Summary

In this section we summarize the important characteristics from the research, of component architectures and ADLs. There is an extensive literature on the subject and this summary is intended to capture those features that are central to the discipline that are identified in the articles surveyed in the earlier part of this chapter. We shall summarize the information under the headings architectural domain, architectural style and ADLs. This summary will be used in the evaluation of the results of our own research described in Chapter 3.

2.6.1 Architectural Domain

As seen in Sections 2.1 and 2.2, there are a number of perspectives from which to discuss and represent architectural issues. These we will refer to as an architectural domain of discourse or architectural domain. The architectural domain for computer

software captures most of the conceptual content of the ADLs referenced in Section 2.4 when the execution and static viewpoints of Cattaneo are combined in a model of the structural and dynamic properties identified by Shaw and Garlan.

The basic elements of this domain are:

- **Components.** A component is a locus of algorithmic computation having the following properties:
 - **Interface.** A set of logical interaction points called ports, between the component and connectors. At each port a component either requires or provides a service.
 - **Type.** The component's named abstract representation.
 - **Semantics.** A specification of the component's functionality and behavior.
 - **Constraints.** A set of assertions about the component that must be satisfied for correct operation.
 - **Evolvability.** The ability to support change over time.
 - **Nonfunctional Properties.** A set of assertions about the component that may be used to guide its selection for composition.
- **Connectors.** A connector is a locus of interaction among components and other connectors and has the following properties:
 - **Interface.** A set of logical interaction points called roles, between the connector and components and other connectors. Each role is a proxy for a service that the connector is either receiving from or providing to an attached component.

- **Type.** The connector's named abstract representation.
 - **Semantics.** A specification of the connector's protocol of interaction otherwise known as glue.
 - **Constraints.** A set of assertions about the connector that must be satisfied for correct operation.
 - **Evolvability.** The ability to support change over time.
 - **Nonfunctional Properties.** A set of assertions about the connector that may be used to guide its selection for composition.
- **Configurations.** A configuration specifies the relations among (instances of) components and connectors and has the following properties:
 - **Topology.** The graph of components and connectors whose edges explicitly represent the paths of flow of control and data. Composition is hierarchical (recursive).
 - **Timing.** A set of control assertions about the temporal dependencies among components and connectors.
 - **Data representation.** A set of assertions about the representation of data that would affect the choice of style and elements for composition.
 - **Binding.** A specification for each interaction as to whether the binding is to be static and/or dynamic.
 - **Nonfunctional Properties.** Support for a number of properties of the resulting system including analysis for correctness and completeness, correct refinement into an implementation, heterogeneity, scalability, and static

and dynamic adaptation (evolution), security, responsiveness and performance, resource requirements, and conformance to standards.

2.6.2 Architectural Style

An architectural style is a pattern of composition of elements in the domain. As such, it specifies the global constraints on component and connector semantics, and configuration, above and beyond any constraints these elements assert as part of their basic specification. These constraints introduced by the semantics of the style are assertions of the following nature:

- **Topology.** Restrictions on the permitted topologies.
- **Timing.** Any timing dependencies or restrictions on flow of control or data.
- **Data representation.** Any restrictions on data representation or data flow.
- **Nonfunctional Properties.** Any other global properties the style designer wishes to impose.

Particular attention must be made to the correctness and completeness of the process of refinement between different styles and when configurations are composed within a style.

2.6.3 Architecture Definition Language

We take as a summary of desirable characteristics, the list developed in Section 2.4. These characteristics include:

- the ability to model or specify the components, connectors and configurations described in Section 2.6.1,
- the ability to support varying degrees of granularity by hierarchical or recursive composition of structures,
- the ability to specify the composition of and compose architectural elements into composite structures both statically and dynamically,
- the ability to change or adapt composite structures both statically and dynamically,
- the ability to define and refine styles,
- support for reusability of architectural elements,
- support for analysis of configurations including correctness within a style and completeness.

With a few exceptions noted in Section 3.1 these characteristics are neither formally defined nor expressed in any detail in the literature. The resulting generality allows us to accommodate a wide range of architectural expression.

Chapter 3

An XML Architecture Definition Metalanguage

A major contribution of this thesis lies with the specification of the XML schemas described in Section 3.2. Before we consider XML, however, we offer another contribution, that of a formal characterization of software architecture concepts. We do so:

- to provide a precise vocabulary of terms for discussing our approach where such is lacking in the literature,
- as a means of providing a representation for as many as possible of the the concepts and issues presented in a wide array of literature sources,
- to guide the development of our ADL and the specification of the functionality of the engine,
- to provide a means for comparing different ADLs including our own.

Then we introduce the XML specification showing how it derives from the formal concepts. We follow with a discussion of a schema interpretation and execution engine prototype. We finish this chapter with a comparison of our XML ADL characterization with the characteristics summarized in Section 2.6.

3.1 Formal Representation

The literature on software engineering is generally void of precise representations of the concepts and terms it uses. Shaw and Garlan [SG96b] show how to model the pipe and filter style of architecture using the Z specification language. Certain ADLs such as Wright [SG96b], Darwin [MDEK95] and Piccola [ALSN00] [Lum00] use variants of process algebra to model component interaction [AG94]. Anlauff et al. describe how to model component behavior using Montages-based abstract state machines [AKPS00]. However, terms relating to software architecture such as “architecture” and “architectural style” have escaped explicit definition although Moriconi et al. [MQR95] discuss these concepts in terms of logical theories. The architectural elements consisting of components and connectors have a wide variety of characterization and no formal definition which captures it all.

The review of software architecture research in Chapter 2 reveals that much of the information that directs architectural specification is semantic in nature. As such, it is outside the scope of expression and manipulation by traditional computer languages and language-based tools such as parsers and compilers. In this section we provide definitions and show how they may be used in architectural specification and analysis and particularly how they can capture semantic content. We will use these definitions for deriving our ADL in Section 3.2 and for discussing future extensions to our model

in Section 5.3.

Architecture is a concept relevant to a domain of discourse that informally consists of configurations of components and connectors, both of which are elements in the domain. Element membership in the set of components or the set of connectors is determined by the element's semantic content in a manner that is outside the scope of this thesis. For example, if the domain of discourse is computer software, components are sorting algorithms and databases rather than dogs and cats, and connectors are message passing facilities and database query protocols. We do not describe any methodology for, given an object, deciding what domain it belongs to. We assume an a priori knowledge of domain membership.

We use a naive notion of a primitive or atomic component such as a sorting algorithm as one that for our purposes has no internal structure. We also have the notion of a composite component such as an address book that is a configuration of a simple graphical-user-interface component, a simple database component and a sorting component connected via a set of message interactions and database SQL interactions. The address-book composite may in turn be a component in some larger application.

Typically, we then talk of more complex structures based on configurations of both atomic and composite components arranged in a hierarchical fashion. In such discussions, we often treat composite components as primitive. So we may describe a personal assistant application as being composed of an address book, a scheduler, a note pad and a calculator, treating each as an atomic component in the description.

The degree of abstraction represented in this formalism raises it above the level of computer hardware or software components and indeed to allow domains that do not

involve computers at all. We provide precise definitions of these architectural terms after introducing some notational conventions.

3.1.1 Notational Conventions

Discussion of Software Architecture occurs on two levels in the literature. One level is the level of implementation referred to as “concrete architecture” or “architectural instance”. The other is a level of abstraction that is the level of models and styles. Where the distinction becomes critical is with the specification of domain elements. The first level corresponds to the idea of an instance in programming languages. The second level corresponds to the idea of a class or type. The difference will be made explicit when we discuss the topology of architectural configurations or structures in Section 3.1.2.

We will adopt the following the following conventions:

- Architectural domains and the sets of elements comprising them are represented by the uppercase calligraphic type style letters $\mathcal{A}, \mathcal{C}, \mathcal{N}, \mathcal{G}$.
- Architectural styles and the sets of constraints comprising them are represented by the uppercase calligraphic type style letters $\mathcal{S}, \mathcal{P}$. We will find it convenient to use a subscript to infer an imprecise concept of level of stylistic restriction, $\mathcal{S}_i$.
- Domain elements, such as components, connectors and configurations and other singular objects associated with them such as names, attributes and constraints are represented by lowercase roman italic type style letters e, c, n, g, a, p . We will use a subscript to indicate the i th instance of an element e as e_i .

- Terms that are part of domain element descriptions, such as interfaces, sets of constraints, implementations, sets of port names, sets of subelement names, sets of instances of elements and sets of connections are represented by uppercase roman italic type style letters I, C, M, P, N, V, E .

These conventions will be used throughout the remainder of this chapter.

3.1.2 Architectural Domain

An *architectural domain* (hereafter simply called a domain) is the tuple $\mathcal{A} \stackrel{def}{=} (\mathcal{C}, \mathcal{N}, \mathcal{G})$ consisting of $\mathcal{C}$, a possibly infinite set of elements in the domain called components, $\mathcal{N}$, a possibly infinite set of elements in the domain called connectors and $\mathcal{G}$, a set of configurations. The concepts ‘component’, ‘connector’ and ‘configuration’ are defined below.

Component

An *interface* of an element e is the tuple

$$I \stackrel{def}{=} (P, N) \tag{3.1.1}$$

where $P \stackrel{def}{=} \{x(e) | x(e) \text{ is a name for a port of } e\}$ is defined as a finite set of named ports and $N \stackrel{def}{=} \{a(e) | a(e) \text{ is a name for a subelement that } e \text{ is composed of}\}$ is defined as a finite set of named subelements. A port is not given a formal characterization other than a name. Semantically, it represents a logical point of interaction between a component and its environment. As such we infer a connection between the port and an internal method whose operation it represents and which we designate as the “port-handler” of the port. Otherwise, we are unconcerned about the

internal operation of a component, except perhaps for an abstract characterization of such as behavior.

A *constraint* $p(e)$ on an element e of the domain is a predicate or assertion that expresses a true statement about it. The semantics of the statement express a precondition, a postcondition, a dependency or some other restriction on the function or behavior of the element. Alternately, it may represent a rule.

An *implementation* M of an element in the domain is the programming language specification or code of the algorithmic function or interaction protocol that the element performs.

A *component* $c \in \mathcal{C}' \subseteq \mathcal{C}$ is an element or object in the domain defined by the tuple,

$$c =^{def} (a, t, I, C, M) \quad (3.1.2)$$

where a is the name of the component, t is its type, I is its interface, $C =^{def} \{p(c) | p(c) \text{ is a constraint on } c\}$ is a set of constraints on it and M is its implementation.

A component with an interface $I = (P, N)$ has one of two forms:

- **Atomic:** having a primitive semantic representation in the domain and characterized by the property $N = \emptyset$.
- **Composite:** characterized by the property $N \neq \emptyset$. It has the semantics of being the composition of two or more components and one or more connectors.

In the domain of software architecture, a component's semantic content is that of a computational function or algorithm.

Connector

A *connector* $n \in \mathcal{N}' \subseteq \mathcal{N}$ is an element in the domain defined by the tuple,

$$n =^{def} (a, t, I, C, M) \quad (3.1.3)$$

where a is the name of the connector, t is its type, I is its interface, $C =^{def} \{p(n)|p(n)$ is a constraint on $n\}$ is a set of constraints on it and M is its implementation.

A connector may be atomic or composite in the same sense that a component may be. Its semantic content is that of an interaction including a communications protocol. It may be as simple as a procedure call or as complex as a client-server protocol. Its programming language specification may be an explicit object or an implicit function associated with the components whose interaction it mediates such as a procedure call of one component on another.

The similarity in the expressions defining the concepts component, 3.1.2, and connector, 3.1.3, suggests a distinction is artificial. We give separate explicit formal representation to facilitate abstract consideration and to accommodate the use of the two concepts in the literature. Their similarity, however, motivates the consolidation of the terms in this thesis.

Configuration

A *configuration* $g \in \mathcal{G}$ is an element or object in the domain of discourse defined by the tuple,

$$g =^{def} (a, C, T) \quad (3.1.4)$$

where a is the name the composite will be given, $C =^{def} \{p(g)|p(g)$ is a constraint on $g\}$ is a set of constraints on it and T is a topology that represents its structure. In the

case of a configuration, a constraint may include the semantics of timing issues, data properties and other system properties that are not expressed by the algorithmic and interaction properties of components and connectors.

A *topology* T is a structure represented by the graph $T =_{def} (V, E)$ where V is the set of all instances of elements in the configuration of which T is a topology, and E is the set of edges signifying the connections between elements. More precisely, for some sub-architecture $\mathcal{A}' = (\mathcal{C}', \mathcal{N}', \mathcal{G}')$ where $\mathcal{C}' \subseteq \mathcal{C}$, $\mathcal{N}' \subseteq \mathcal{N}$, and $\mathcal{G}' \subseteq \mathcal{G}$, if $g \in \mathcal{G}'$, then $\forall e_i \in V, \exists e \in \mathcal{C}' \cup \mathcal{N}'$ where e_i is an instance of e and $i \in \{1, 2, 3, \dots\}$. We allow an abuse of notation to express our notion of sub-architecture as $\mathcal{A}' \subseteq \mathcal{A}$.

3.1.3 Architectural Style

An architectural style is a restriction on an architectural domain. Effectively, it is a sub-architecture as described in the last paragraph. In the domain of human habitation, Greek Corinthian-style architecture employed stone columns of a particular design. If we consider a column as a primitive component, a Corinthian column would be included in the set of components of the style whereas a Doric column would not be. Similarly, the sets of connectors and configurations of a style may each be a restriction of all the connectors and configurations possible in the domain.

In the literature, style is used loosely to refer both to the set of specific characteristics - primarily constraints and dependencies - and the resulting architectural structures. We choose to give 'style' the following explicit meaning.

First we define a *restriction* $\mathcal{P}_{\mathcal{E}}$ on a set of elements $\mathcal{E} \in \{\mathcal{C}, \mathcal{N}, \mathcal{G}\}$ in a domain to be a set of constraints applied to the elements of the set. Formally, $\mathcal{P}_{\mathcal{E}} = \{p(e) | \forall e \in \mathcal{E}, p(e) \Rightarrow \text{true iff } p \text{ is some assertion about } e\}$. Assertions could take the form of sets

of either predicates or rules, or the union of two such sets.

A restriction $\mathcal{P}$ on a style then is defined as $\mathcal{P} =^{def} (\mathcal{P}_{C'}, \mathcal{P}_{N'}, \mathcal{P}_{G'})$ for some $\mathcal{A}'$ as defined in the last section.

An *architectural style* $\mathcal{S}$ (hereafter simply called a style) is the tuple $\mathcal{S} =^{def} (\mathcal{A}', \mathcal{P})$. We extend this definition to be $\mathcal{S} =^{def} (\mathcal{C}'', \mathcal{N}'', \mathcal{G}'') \subseteq \mathcal{A}' \subseteq \mathcal{A}$ where if $\mathcal{E}' \in \{\mathcal{C}', \mathcal{N}', \mathcal{G}'\}$ and $\mathcal{E}'' \in \{\mathcal{C}'', \mathcal{N}'', \mathcal{G}''\}$ then $\mathcal{E}'' = \{e | \forall e \in \mathcal{E}' \text{ and } \exists p(e) \in \mathcal{P}_{\mathcal{E}'} \Rightarrow \text{true} \text{ then } \neg \exists p(e) \in \mathcal{P}_{\mathcal{E}''} \Rightarrow \text{false}\}$.

We may define a sequence of refinements of styles in the following manner:

- $\mathcal{S}_0 =^{def} (\mathcal{A}, \mathcal{P}_0) = \mathcal{A}$ where $\mathcal{P}_0 = (\emptyset, \emptyset, \emptyset)$,
- $\mathcal{S}_1 =^{def} (\mathcal{S}_0, \mathcal{P}_1)$ where $\mathcal{P}_1 \neq (\emptyset, \emptyset, \emptyset)$,
- $\mathcal{S}_i =^{def} (\mathcal{S}_{i-1}, \mathcal{P}_i)$ where $\mathcal{P}_i \neq (\emptyset, \emptyset, \emptyset)$,

for $i \in \{1, 2, 3, \dots\}$. If we allow a slight abuse of notation, namely that $\mathcal{P}_i \cup \mathcal{P}_j =^{def} (\mathcal{P}_{Ci} \cup \mathcal{P}_{Cj}, \mathcal{P}_{Ni} \cup \mathcal{P}_{Nj}, \mathcal{P}_{Gi} \cup \mathcal{P}_{Gj})$, where $\mathcal{P}_i =^{def} (\mathcal{P}_{Ci}, \mathcal{P}_{Ni}, \mathcal{P}_{Gi})$ and $\mathcal{P}_j =^{def} (\mathcal{P}_{Cj}, \mathcal{P}_{Nj}, \mathcal{P}_{Gj})$, then it is easy to show

$$\mathcal{P}_i = \bigcup_{j=0}^{i-1} \mathcal{P}_j \quad (3.1.5)$$

From the viewpoint of semantics however, it is important to ensure that $\mathcal{P}_i$ is consistent. That is, it is not the case that both p and $\neg p$ are members of some element of $\mathcal{P}_i$.

This notion of style allows for the transitive composition of styles or hierarchies of styles. Much the same as methods are inherited in OO programming languages, constraints are inherited in a hierarchy of styles. In our human habitation architectural example, a region's building code may be written as a style that would then

be composed with specific styles such as “Cape Cod”. It also allows us to capture constraints on the domain that include such things as performance criteria.

3.1.4 Architectural Design

In order to formalize the concept of a specific instance of an architecture, we define an *architectural design* (hereafter called a design) as the tuple $\mathcal{D} =^{def} (\mathcal{C}', \mathcal{N}', \mathcal{G}')$ where some configuration $g \in \mathcal{G}'$ specifies the structure of the design and $\forall f \in \mathcal{G}' \Rightarrow f = g$ and $V =^{inst} \mathcal{C}' \cup \mathcal{N}'$ in the topology of g . Here, the symbol $=^{inst}$ means equality up to and including multiple instances of elements of $\mathcal{C}'$ and $\mathcal{N}'$.

A design has the meaning of a set of components and a set of connectors with a single configuration whose topology includes at least one instance of every component and connector in their sets. In building architecture, this roughly corresponds to the blueprint or plan for a building.

The term architecture itself will be given no formal definition. Instead, it will be used as an imprecise equivalent to the term architectural style or architectural domain. As such it may refer to any style and will depend on context for explicit meaning.

3.2 An XML Metalanguage ADL Specification

XML is a document specification metalanguage initially created to allow content for the World-Wide Web to encompass all forms of information, not only natural language text documents. An object with an XML markup is still called a document but in fact could contain the the binary executable of a software program. The choice of

XML [W3C98] as a metalanguage for developing an ADL is motivated by several advantages [MM99] [PSSC98]:

- **it is an open standard, particularly for development done in the context of the World-Wide Web. This openness facilitates interchange between different XML ADL implementations,**
- **there is a large selection of support tools available both commercially and freely,**
- **it is platform independent (in terms of both operating system and programming language),**
- **it has a simple fixed syntax but unlimited vocabulary,**
- **its extensibility as a metalanguage allows for easy changes to derived languages,**
- **requirements documents and support manuals can have direct bi-directional links with architectural elements due to XML's nature as a document markup language,**
- **it allows the construction of a DTD or schema to specify validity constraints in addition to correctness of syntax,**
- **its composite expressions of elements are name-based rather than position-based as in tuple-like structures,**
- **XML support tools including graphics presentation packages, parsers and browsers can be easily integrated into ADL frameworks.**

Its disadvantages are:

- it does not easily allow recursive expressions,
- it is verbose.

If XML is to describe an architectural design it must be able to fully describe both components and connectors, and configurations. In the following two subsections we show how this can be done. In these subsections the term “element” refers to the XML concept rather than the concept of architectural element used throughout the rest of the paper.

Our ADL [ND02] consists of two schemas, one that describes components and one that describes configurations. We have chosen to treat components and connectors the same, namely as components. Some ADLs such as Darwin do not distinguish between them noting that any connector functionality that deserves first class treatment may be given such as a component. While the literature recognizes the importance of being able to abstract and encapsulate the complexity of interaction in a first class element, it provides no real justification for the elaboration necessary to distinguish component functionality from connector functionality in terms of treatment as a separate class of element.

We discuss our architecture in the context of an example, that of assembling and testing a half adder circuit from a set of components consisting of and-gates, or-gates and inverters. The choice of a software correlate to a hardware design problem, is appropriate for several reasons:

- it is a simple intuitive model of component assembly,
- it demonstrates how our formalism can present a software-domain model of a problem existing in a non-software domain,

- our formalism abstracts above component semantics. In this sense a “hardware” semantic is of no more consequence than any other. The only semantic reflected in our example is that of a software domain: units of code with clearly defined methods of interaction.

We now develop our XML schemas from this formalism.

3.2.1 The Component Schema

The component schema is an XML specification that specifies a component’s name, optionally its type, any qualifying or constraining attributes, its interface and its implementation or code. We will discuss this schema referring to the notational convention of the formal definition 3.1.2 of a component: $c \stackrel{def}{=} (a, t, I, C, M)$.

The root element in this schema is specified in XML as:

```
<component name = "... " version = "... " type = "... "
  start-method = "... " start-component = "... ">
  ...
</component>
```

Here we capture the name attribute a of a component c . Since the intent of a name is to act as a unique identifier, version may be considered an extension of the identifier and we include this possibility in our XML. We include an optional “type” attribute t to allow for consistency checking. The “start-method” and “start-component” attributes are used only if the component is runnable, such as may be the case with a user interface component. The definition we have given to the term constraint in Section 3.1.2 is that of an assertion about some property of c . We have

chosen to express assertions about runability as XML element attributes rather than distinct XML elements.

The root element contains three child elements, “interface”, “constraints” and “implementation”, corresponding to the notational terms I , C and M respectively. The constraints element represents the set of constraints C and its child elements represent individual members of C . We allow for constraints identified by a “class” attribute as either a “rule” or a “predicate”. The implementation element representing M contains an unparsed character data element containing the actual programming code of the component. The XML code for these elements is:

```
<interface>
    ...
</interface>
<constraints>
    <constraint class = "rule | predicate"> ... </constraint>
    ...
</constraints>
<implementation>
    <![CDATA[
        ...
    ]]>
</implementation>
```

The interface element in turn contains 0 or more port elements followed by 0 or more subcomponent elements. Since ports are the points of direct coupling with other software components, a component with 0 ports and 0 subcomponents would be a

monolithic stand-alone program that is runnable. Such a component may still have interaction points with the external world such as communications links or human interfaces. These elements correspond notationally to members of the sets P of ports and N of subcomponents in definition 3.1.1.

A port element is specified as:

```
<port direction = "input | output" name = "..." method = "..." />
```

The purpose of the port element is to identify the kind of connection the component requires and its name. The name of the internal method or port-handler that handles the connection activity is specified also. An optional type attribute may be used for more elaborate checking. A port is not named independently of its associated port-handler. Ports are designated as having a direction “input” or “output” inducing a directed edge in the configuration graph.

The subcomponent element is specified as

```
<subcomponent name = "..." version = "..." filename = "..." />
```

and is used to inform the component user what if any subcomponents are required as identified by the “name” and optional “version” attributes and also optionally, where they may be found.

An example of XML configuration code for specifying an atomic component is shown in Figure 3.1 and for specifying a composite component containing subcomponent declarations, in Figure 3.2. We note that the specifications contain no constraint elements as none were required for this example.

```

<?xml version="1.0" encoding="UTF-8" ?>\\
<component name="and-gate" version="1.0">
  <interface>
    <port direction="input" name = "input1" method="input1-changed" />
    <port direction="input" name = "input2" method="input2-changed" />
    <port direction="output" name = "output" method="add-new-port!" />
  </interface>
  <implementation>
    <![CDATA[
      (define (and-gate)
        ;; a portList is a list of component/method pairs that tell
        ;; this component which components to inform of changes.
        (let ((input1 -1) (input2 -1) (output -1) (portList '()))
          (define (logical-and b1 b2)
            (if (and (= b1 1) (= b2 1)) 1 0))
          (define (inform-each-component portList)
            (if (not (null? portList))
                (begin ((caar portList) (cdar portList)) output)
                    (inform-each-component (cdr portList)))))
          (define (add-new-port! portName)
            (set! portList (cons portName portList)))
          (define (calculate-output)
            (if (not (or (= input1 -1) (= input2 -1)))
                (begin (set! output (logical-and input1 input2))
                        (inform-each-component portList))))
          (define (input1-changed value)
            (cond ((not(or (= value 0) (= value 1)))
                  (error "Invalid and-gate message." value))
                  (else (set! input1 value)(calculate-output))))
          (define (input2-changed value)
            (cond ((not(or (= value 0) (= value 1)))
                  (error "Invalid and-gate message." value))
                  (else (set! input2 value)(calculate-output))))
          (define (dispatch procedure)
            (cond ((eq? procedure 'input1-changed) input1-changed)
                  ((eq? procedure 'input2-changed) input2-changed)
                  ((eq? procedure 'add-new-port!) add-new-port!)
                  (else (error "Invalid procedure." procedure))))
          dispatch))
    ]]>
  </implementation>
</component>

```

Figure 3.1: XML component code specifying an and-gate.

```

<?xml version="1.0" encoding="UTF-8" ?>\\
<component name="half-adder" version="1.0">
  <interface>
    <port direction="input" name = "input1" method="input1-changed" />
    <port direction="input" name = "input2" method="input2-changed" />
    <port direction="output" name = "sum" method="add-new-port1" />
    <port direction="output" name = "carry" method="add-new-port2" />
    <subcomponent name="or-gate1" filename="d:/or-gate.xml" />
    <subcomponent name="and-gate1" filename="d:/and-gate.xml" />
    <subcomponent name="inverter1" filename="d:/inverter.xml" />
    <subcomponent name="and-gate2" filename="d:/and-gate.xml" />
  </interface>
  <implementation>
    <![CDATA[
      (define (half-adder)
        (let* ((or-gate1 (or-gate))
              (and-gate2 (and-gate))
              (and-gate1 (and-gate))
              (inverter1 (inverter)))
          (define (input1-changed value)
            ((or-gate1 'input1-changed) value)
            ((and-gate1 'input1-changed) value))
          (define (input2-changed value)
            ((or-gate1 'input2-changed) value)
            ((and-gate1 'input2-changed) value))
          (define (add-new-port1! portName)
            ((and-gate2 'add-new-port!) portName))
          (define (add-new-port2! portName)
            ((and-gate1 'add-new-port!) portName))
          (define (dispatch procedure)
            (cond ((equal? procedure 'add-new-port2!) add-new-port2!)
                  ((equal? procedure 'add-new-port1!) add-new-port1!)
                  ((equal? procedure 'input2-changed) input2-changed)
                  ((equal? procedure 'input1-changed) input1-changed)
                  (else (error "invalid procedure." procedure))))
          ((and-gate1 'add-new-port!) (cons inverter1 'input-changed))
          ((or-gate1 'add-new-port!) (cons and-gate2 'input1-changed))
          ((inverter1 'add-new-port!) (cons and-gate2 'input2-changed))
          dispatch))
        ]>
    </implementation>
  </component>

```

Figure 3.2: Engine generated XML code specifying a half-adder circuit component.

The Component Schema DTD

The following is the DTD for the component schema. We use DTDs to concisely capture our schema information. DTDs are being replaced by a formally proposed XML Schema representation that overcomes certain limitations of DTDs. However, these limitations do not affect our particular use, so DTDs suffice.

```
<!DOCTYPE COMPONENT SYSTEM [
  <!ELEMENT COMPONENT (INTERFACE, CONSTRAINTS?, IMPLEMENTATION)>
  <!ATTLIST COMPONENT
    Name CDATA #REQUIRED
    Version CDATA #IMPLIED
    Type CDATA #IMPLIED
    Start-method CDATA #IMPLIED
    Start-component CDATA #IMPLIED>
  <!ELEMENT INTERFACE (PORT+, SUBCOMPONENT*)>
  <!ELEMENT PORT EMPTY>
  <!ATTLIST PORT
    Direction (input | output) #REQUIRED
    Name CDATA #REQUIRED
    Type CDATA #IMPLIED
    Method CDATA #REQUIRED>
  <!ELEMENT SUBCOMPONENT EMPTY>
  <!ATTLIST SUBCOMPONENT
    Name CDATA #REQUIRED
    Filename CDATA #IMPLIED>
```

```

<!ELEMENT CONSTRAINTS (CONSTRAINT+)>
<!ELEMENT CONSTRAINT CDATA>
<!ATTLIST CONSTRAINT
    Class (rule | predicate) #REQUIRED>
<!ELEMENT IMPLEMENTATION (#CDATA)>
]>

```

This DTD enforces the requirement that the component has at least one port in its interface.

3.2.2 The Composition Schema

The composition schema is a syntactic and semantic XML specification that defines the composition code (CC) that tells a composition engine how to assemble or compose a composite from individual components. We will discuss this schema referring to the notational convention of the formal definition 3.1.4 of a configuration: $g =^{def} (a, C, T)$.

The root element in this schema is specified in XML as:

```

<compositioncode name = "..." version = "..." location = "..."
    start-method = "..." start-component = "..." style = " ... "
    style-location = " ... ">
    ...
</compositioncode>

```

It has attributes that assert the composite's name, and optionally version and Uniform Resource Identifier (URI) of the location the composite is to be saved to. If the

composite is to be runnable, then the “start-component” attribute tells the engine which subcomponent is runnable and the “start-method” attribute tells the engine what the start-method’s name or port is. The engine also uses this name in building the composite’s interface. Finally, it has an attribute to identify any style and an associated location where to find it if such is to be used.

The root element contains two child elements, “constraints” as in the component schema and “topology”, corresponding to the notational terms C and T . The constraints element represents the set of constraints C and its child elements represent individual members of C . The XML code for these elements is:

```
<constraints>
  <constraint class = "rule | predicate"> ... </constraint>
  ...
</constraints>
<topology>
  ...
</topology>
```

The topology element contains one or more connection elements corresponding to members of the set E in the definition of T followed by two or more component elements corresponding to elements of the set V , this being a minimally constituted composite.

The purpose of the connection element is to inform the engine what types of connections it will have to build and what their names will be. In addition, “input” and “output” connection names for the composite will be part of the new composite component’s interface as discussed in Section 3.3.

A connection element is specified as:

```
<connection name = "..." type = "input | output | internal"/>
```

The purpose of the component element is to inform the engine what named components it will have to find and instantiate and what connections will be associated with each component. We have used an optional “filename” attribute to specify the location of the component in our examples. Other optional location attributes include “URL” for internet locations and “broker” where a component brokerage is to be searched. Another option is “search” for custom search algorithms.

The component element contains one (sub)connection element for each input and output connection the component requires. For the specific component instance, the “type” attribute informs the engine whether the connection is input or output and the “method” attribute identifies what method within the component handles the connection activity. The details of these port-handlers are discussed in Section 3.3.

A component element is specified as:

```
<component name = "..." filename = "...">
  <subconnection name = "..." type = "input | output" method = "..."/>
  ...
</component>
```

The important aspect in using this schema is to preserve naming consistency of connections across components.

The XML code for specifying the composition of a half adder is shown in Figure 3.3. This composite is not in itself runnable.

```

<?xml version="1.0" encoding="UTF-8" ?>\\
<compositioncode name="half-adder" version="1.0">
  <topology>
    <connection name="input1" type="input" />
    <connection name="input2" type="input" />
    <connection name="sum" type="output" />
    <connection name="carry" type="output" />
    <connection name="outputOr" type="internal" />
    <connection name="outputInv" type="internal" />
    <component name="or-gate" filename="d:/or-gate.xml">
      <subconnection name="input1" type="input" method="input1-changed" />
      <subconnection name="input2" type="input" method="input2-changed" />
      <subconnection name="outputOr" type="output" method="add-new-port!" />
    </component>
    <component name="and-gate" filename="d:/and-gate.xml">
      <subconnection name="input1" type="input" method="input1-changed" />
      <subconnection name="input2" type="input" method="input2-changed" />
      <subconnection name="carry" type="output" method="add-new-port!" />
    </component>
    <component name="inverter" filename="d:/inverter.xml">
      <subconnection name="carry" type="input" method="input-changed" />
      <subconnection name="outputInv" type="output" method="add-new-port!" />
    </component>
    <component name="and-gate" filename="d:/and-gate.xml">
      <subconnection name="outputOr" type="input" method="input1-changed" />
      <subconnection name="outputInv" type="input" method="input2-changed" />
      <subconnection name="sum" type="output" method="add-new-port!" />
    </component>
  </topology>
</compositioncode>

```

Figure 3.3: XML configuration code to construct a half adder circuit.

The Composition Schema DTD

The following is the DTD for the composition schema.

```
<!DOCTYPE COMPOSITIONCODE SYSTEM [  
  <!ELEMENT COMPOSITIONCODE (CONSTRAINTS?, TOPOLOGY)>  
  <!ATTLIST COMPOSITIONCODE  
    Name CDATA #REQUIRED  
    Version CDATA #IMPLIED  
    Location CDATA #IMPLIED  
    Start-method CDATA #IMPLIED  
    Start-method CDATA #IMPLIED  
    Style CDATA #IMPLIED  
    Style-location CDATA #IMPLIED>  
  <!ELEMENT CONSTRAINTS (CONSTRAINT+)>  
  <!ELEMENT CONSTRAINT CDATA>  
  <!ATTLIST CONSTRAINT  
    Class (rule | predicate) #REQUIRED>  
  <!ELEMENT TOPOLOGY (CONNECTION+, COMPONENT, COMPONENT+)>  
  <!ELEMENT CONNECTION EMPTY>  
  <!ATTLIST CONNECTION  
    Name CDATA #REQUIRED  
    Type (input | output) #REQUIRED>  
  <!ELEMENT COMPONENT (SUBCONNECTION+)>  
  <!ATTLIST COMPONENT  
    Name CDATA #REQUIRED
```

```

    Filename CDATA #IMPLIED>
<!ELEMENT SUBCONNECTION EMPTY>
<!--ATTLIST SUBCONNECTION
    Name CDATA #REQUIRED
    Type (input | output) #REQUIRED>
    Method CDATA #REQUIRED>
]>

```

This DTD enforces the requirement that the composition has at least two components in its topology.

3.2.3 The Architectural Style Schema

This schema performs the function of expressing stylistic restriction or constraint on sets of domain elements and configurations of such. For the sets of components and connectors, the principal restriction is on set membership. That is, a style may specify that certain types of an element are in the style and certain types are not. For configurations, the principal restriction is on the allowable topologies. We use a schema that expresses constraints in a manner similar to that used by the component and configuration schemas developed earlier in this chapter.

A composition engine can transitively compose the sets of constraints for an architectural style with the constraints specified for a specific component to ensure the composition specification is correct. That is, it is logically consistent with respect to constraints.

The XML style schema is:

```

<style name = " ... " version = " ... ">
  <cmpConstraints>
    <constraint class = "rule | predicate"> ... </constraint>
    ...
    <type action = "include | exclude"> ... </type>
    ...
  </cmpConstraints>
  <cnrConstraints>
    <constraint class = "rule | predicate"> ... </constraint>
    ...
    <type action = "include | exclude"> ... </type>
    ...
  </cnrConstraints>
  <cfgConstraints>
    <constraint class = "rule | predicate"> ... </constraint>
    ...
    <topology> ... </topology>
  </cfgConstraints>
</style>

```

We include name and version attributes to aid identification of the correct style to be used. A Document Type Definition (DTD) corresponding to this schema follows:

```

<!DOCTYPE STYLE SYSTEM [ <!ELEMENT STYLE (CMPCONSTRAINTS*
CNRCONSTRAINTS* CFGCONSTRAINTS*)>
  <!ATTLIST STYLE

```

```

    Name CDATA #REQUIRED
    Version CDATA #IMPLIED>
<!ELEMENT CMPCONSTRAINTS (CONSTRAINT* TYPE*)>
    <!ELEMENT CONSTRAINT CDATA>
        <!ATTLIST CONSTRAINT
            Class (rule | predicate) #REQUIRED>
    <!ELEMENT TYPE CDATA>
        <!ATTLIST TYPE
            Action (include| exclude) #REQUIRED>
<!ELEMENT CNRCONSTRAINTS* (CONSTRAINT* TYPE*)>
<!ELEMENT CFGCONSTRAINTS* (CONSTRAINT* TOPOLOGY*)>
    <!ELEMENT TOPOLOGY CDATA>
]>

```

By means of this third schema, we can incorporate architectural style information into our model. We discuss implications on engine design in Section 5.3.6.

3.3 An Engine Design for the Architecture

The role of the engine in component composition is to take a set of components in their XML configuration and assemble them into a composite according to an XML composition specification. The resulting composite must have all the properties of a component including a fully specified interface and an operable implementation. By operable we mean that it in itself is runnable or when composed with another element that is runnable, the composite is runnable.

The functions of the engine are:

- read and interpret an XML composition code,
- assemble the composite component and build its port structure,
- build an XML component wrapper for the composite and store the result if it is to be persistent,
- find the required components,
- instantiate the components and run the composite if it is runnable and is to be run.

To create this functionality, a number of design issues must be resolved. These are discussed in the following subsections. We do not attempt to incorporate engine functionality that can interpret and utilize the information in style schemas. Such an effort represents significant work and is outside the scope of this thesis.

3.3.1 Choice of Operating Environment

A major decision in engine design is the choice of the running environment for both the engine and composite assembly. Compiler-based implementations offer the possibility of easy heterogeneity since binaries can be compiled from most programming languages. However, the engines are potentially more complex having to provide the assistance of compilers and linkers for creating new composite binaries and their runtime execution. In addition, dynamic adaptation does not appear to be easy.

Interpreter-based environments offer an easier means for rapid prototyping and

building composite interfaces. They are better suited for dynamic or runtime binding and adaptation of components as will be discussed in Section 5.3.5. Language heterogeneity depends on what native language support the particular interpreter provides.

For development of our engine prototype we have chosen an interpretive environment, namely Scheme.

3.3.2 XML Parsing and Interpretation

XML parsers come in two flavours, validating and non-validating. Non-validating parsers check for syntactical correctness or well-formed expressions only. Validating parsers also check for validity and completeness of the document content and structure by checking against a DTD specification. One must choose which is to be used by the engine. The only issue this raises for the actual engine design is error handling for documents that fail validity checks.

The second issue in choosing a parser concerns the parsed output. XML data may be represented by a specification known as an XML Infoset. XML parsers use either of two common Infoset models. The first is the Document Object Model (DOM) format [W3C01a]. The second is the Simple API for XML (SAX) format [Meg00] that creates an output following the SXML specification [Kis01].

In the case of DOM, instead of specifying how objects may be represented in XML, DOM specifies how XML documents are represented as objects, so that they may be used in object oriented programs. A DOM parser converts the XML document abstract syntax tree that it stores in its entirety in memory into a hierarchy of classes and interfaces. The output may be represented graphically as a hierarchical structure

but the parent/child relationships depicted are logical relationships defined by the programming interfaces, not representations of any particular internal data structures.

SAX on the other hand converts the abstract syntax tree into a set of S-expressions. It processes the XML as a series of events based on the tag structure of the document. As a result, it is faster and has a small memory footprint. The S-expression output is particularly suited for processing by list-processing functional languages like Scheme.

We have chosen a non-validating SAX parser, partly based on availability in our Scheme environment [Sch] and partly based on the compatibility of S-expressions with Scheme.

3.3.3 Component Discovery

There are three general methods an engine may use to find the required components. One is a blind search on the Internet using some search algorithm or search engine. A second is to obtain a component from a component brokerage service. The third method is to be told explicitly from where a component may be retrieved.

The first method would rely on the component's name, version and type to provide the criteria to perform a search of the World-Wide Web. Either a search engine would have to be integrated into our engine or an interface to a commercial engine such as Google would have to be created. If needed, the component XML could be extended with additional attributes to facilitate such a search. We might want to include a vendor's name or a developer's name.

For the second method, our engine would need an interface such as a client to some component brokerage server. The issue of component identification is similar to the first case above. A protocol, possibly another simple XML schema, would

have to be developed to facilitate the interaction. A component brokerage might be associated with a proprietary component distribution and assembly environment or a standard mature component industry.

The third method involves an explicit reference to the component's location in the form of a URI representing a local filename, an FTP protocol based identifier, a URL to some Web-based site or some other explicit form of location reference. The location reference may be sufficient to uniquely identify the component. If it isn't, the attributes mentioned earlier in this section may be used. To accommodate this type of search, appropriate mechanisms must be included in the engine. In our examples and testing we have used explicit filenames as a convenience involving the minimal amount of engine code to support the testing of our prototype. We may extend the named attributes in our XML specifications to include additional location reference types beyond "filename".

The challenge of component discovery does not lie with the methodology of location. And the XML is sufficiently extensible that all three methods can be accommodated in their location information requirements. Rather, any difficulty is likely to emerge around the issue of making the assessment that a component is the right one for the task at hand from the viewpoint of algorithmic or functional content. A degree of intelligent direction may be required to make a correct selection either from the composition engine or the composition code writer.

3.3.4 Component Assembly

One of the first things the engine must do to create a composite is to create a set of unique names for the component instances to be assembled. It then must set the

bindings for these instances that will be established at runtime.

Since the primary characteristic of a component is its interface, the engine must build a correct interface for the composite implementation it assembles. This interface consists of a set of ports and associated port-handlers whose single function is to pass the port operation to the correct port on the appropriate subcomponent. Figure 3.2 shows the interface constructed for our half-adder example. Here, a set of four methods, two input and two output, have been constructed in the language of the components being used for assembly. One other port may be constructed by the engine. If the composite is runnable, a start port is constructed that calls the start port on the appropriate subcomponent.

The port-handlers for the two input ports on the composite in turn engage two subcomponent ports implementing a one-to-many relationship. The port-handlers for the two output ports on the composite implement a one-to-one relationship between each subcomponent port and a port on the composite. Such a one-to-one relationship is semantically correct.

Finally, to complete the encapsulation of the composite, we have chosen to add a dispatch method through which all activity requests to the composite must pass.

The resulting composite code contains no new algorithmic or interaction functionality. It functions strictly as a wrapper for the algorithmic functionality of its subcomponents. If the software architect wishes to include additional algorithmic computation beyond what the components offer, he must write such as a new component and “wire” it in with the other components.

3.3.5 XML Component Wrapper Construction

Figure 3.2 provides an example of a component wrapper constructed for our half-adder composite. The implementation content we discussed in the previous subsection. To build the interface part of the XML, the engine must create a set of port elements for the ports it constructs for the composite, and a set of subcomponent elements for the subcomponent instances it creates for the composite.

We noted earlier that we consider the port name to be the name of the port-handler. A port element, then, is created by first identifying a port requirement as a connection in the CC of type input or type output. Since connection names are unique (a property the engine must enforce), the connection name is used to generate the name of the corresponding port-handler. A port element in the wrapper is then created from this name and the associated port direction specified in the CC.

Each component element reference in the CC generates a subcomponent element in the composite wrapper. Since the CC only specifies a requirement for a component type such as an and-gate, and not a unique instance name such as “and-gate1”, the engine must create such a unique name. In the process, it uses this information along with the component location information which is an attribute of the component element in the CC, to build a subcomponent element in the XML.

The problem of location discovery beyond the explicit reference used above, is discussed in Section 5.3.1.

Finally, the engine must be told what to do with the XML for the finished component.

3.3.6 Component Binding and Execution

Component binding is a runtime activity the engine performs if the composite component is to be run. The engine must first locate the components required for composition and place them in memory. This is a recursive search down through the composition hierarchy.

Next the engine must interpret the component code, create and bind subcomponent instances and lastly, run the component in the interpretive environment.

In Section 4.1 we present the actual details of our engine prototype showing the results of sample runs. The implementation described is sufficient to create and run composite assemblies. Some of the desirable features of software architecture specification and ADLs in particular have not been included in our engine as such were unnecessary for basic operation. We discuss future extensions of the engine in Section 5.3.

3.4 The XML Metalanguage Specification as an ADL

In Section 2.6 we summarized the characteristics that an ideal ADL would manifest. In this section we compare our XML specification with these characteristics. This comparison allows us an easy assessment of how closely our ADL approaches the ideal. We note where a characteristic is not satisfied and provide a brief indication of how each characteristic is satisfied otherwise.

As noted in Subsection 2.6.1, the principle elements of an architectural domain are components, connectors and configurations. In Table 3.1 we indicate how our

Features	XML Specification
Interface	Services required or provided by a component are specified by port elements. The roles of a connector represented by a component are similarly specified.
Type	Type may be fully specified by name, version and optional type attributes.
Semantics	We have not provided a capability for expressing behavior. We discuss an XML extension for this in Section 5.3.2.
Constraints	We have provided the capability for expressing constraints on both components and configurations. The XML does not place any limitations on the form constraint specification may take. We discuss this issue further in Section 5.3.6.
Evolvability	We have not provided a capability in the XML for dynamic adaptation. Static change can be accomplished by reassembly. We discuss this issue further in Section 5.3.5.
Nonfunctional Properties	Assertions about such properties may be expressed as constraints for both components and configurations.
Reusability	The XML packages a composite in a form in which it may be stored and reused. It is the engine's function to create persistence. In our case we implement persistence, hence reusability.

Table 3.1: ADL Component Characteristics.

XML component schema compares with these characteristics as well as component-related characteristics listed in Subsection 2.6.3. Since we specify connectors as a kind of component, the comparison applies equally to connectors. In Table 3.2 we indicate how our XML configuration schema compares with the corresponding domain characteristics. The characteristics associated with architectural style, notably topology, timing, data representation and nonfunctional properties as noted in Subsection 2.6.2 are similar to the characteristics of configuration listed in Table 3.2. We include as well, configuration-related characteristics listed in Subsection 2.6.3. Issues of refinement of styles we discuss in Section 5.3.6.

Features	XML Specification
Topology	The topology element within the composition schema fully specifies the connection paths among the components. Since a composite is formed as a complete component also, composition is hierarchical creating varying granularity.
Timing	Assertions about timing properties may be expressed as constraints for both components and configurations. An implementation of component behavior carries the possibility of further expressing timing constraints.
Data Representation	Our model imposes no constraints on the representation of data other than those specifically asserted as constraint characteristics.
Binding	The XML specification of binding is static. The issue of providing dynamic binding we discuss in Section 5.3.5. The issue of how to update XML code is discussed as a separate issue in this section.
Nonfunctional Properties	Assertions about such properties may be expressed as constraints for both components and configurations.
Support for Style	We define a style as a tuple of constraints on architectural elements. The XML expresses constraints for both components and configurations but specifies nothing about their use. We discuss the issue of composing constraints in Section 5.3.6.
Support for Analysis	An XML DTD allows us to verify that a configuration specification is valid. We have not provided any specification for proving the correctness of a configuration. We discuss the issue in Section 5.3.6.

Table 3.2: ADL Configuration Characteristics.

Chapter 4

Prototype Implementation and Discussion of Results

In this chapter we discuss the results of the development of our prototype engine and the suitability of the XML schemas as an ADL. In fulfilling our objectives we uncovered a number of issues requiring further consideration. These are identified and briefly discussed in Section 5.3.

4.1 A Scheme Implementation of a Composition Engine

This chapter reviews the development and testing of the assembly engine prototype. The prototype evolved through three versions, each exploring different features and potentials of the model. As noted in Section 3.3.1, we have used the Scheme language for development of our prototypes. Testing of the prototypes involved initially the

Figure 4.1: Logic gate diagram of a half-adder circuit.

construction of a half-adder circuit shown in Figure 4.1 as mentioned in Section 3.2. It was extended by composing a test-bed for the half-adder as detailed in Section 4.1.2. Finally, to demonstrate the ability of our assembly model to extend to more significant applications, the engine was broken into components that were reassembled by the second generation engine to create a third generation engine detailed in Section 4.1.3.

4.1.1 A Connector-Based Model Engine

The first engine prototype implemented a connector-based model reviewed in Section 2.2.2. In this case, connectors were explicitly modelled within the engine itself. Since every composite would use at least one connector, a design decision was made to include the connector code within the engine to eliminate redundancy of this common code across components. Composites such as the half-adder shown in Figure 4.2 assembled from the CC shown in Figure 4.3 when runnable and when started by the engine would use instantiations of the connector or "connection" element loaded with the engine code.

Another feature of the initial design was to include the code of the subcomponents in the body of code of the composite. A positive aspect of this approach is to make the composite independent of the need to locate and load subcomponents. Because connectors are modelled within the engine, a runnable composite still requires the presence of the engine and is not truly stand-alone. The major disadvantage of this approach is that subcomponents cannot be replaced dynamically. Change of a single subcomponent requires the re-assembly of the composite.

4.1.2 A Connector-less Model Engine

The second engine prototype implemented a connectorless model based on the design decision we discuss in Section 3.2. In this case, we have dropped the feature of including subcomponent code in the composite structure. This could be reintroduced as an optional feature in future engine design resulting in completely independent runnable composites.

A second feature implemented in this version is the inclusion of subcomponent retrieval and loading by the engine for runnable components. This version also automatically runs any runnable composite. The CC in Figure 4.3 using this version of the engine results in the assembly of a half-adder whose XML component specification is similar to that shown in Figure 3.2.

To further test this version of the engine, we created two user interface components, one to request input information and one to display test results. These were wired together using the CC shown in Figure 4.4. The resulting composite provided a runnable test-bed for the half-adder circuit which worked perfectly when started by the engine.


```

<?xml version = "1.0" encoding = "UTF-8"?> <component name =
"half-adder"> <connection name = "input1"/> <connection name =
"input2"/> <connection name = "sum"/> <connection name = "carry"/>
<component name = "inverter"/> <component name = "and-gate"/>
<component name = "or-gate"/> <![CDATA[
  (define (half-adder input1 input2 sum carry)
    (define (or-gate input1 input2 output)
      (define (logical-or b1 b2) (if (or (= b1 1) (= b2 1)) 1 0))
      (define (or-gate-function value)
        (let ((in1 (get-message input1))
              (in2 (get-message input2)))
          (cond ((not(or (= in1 0) (= in1 1)))
                 (error "Invalid or-gate message." in1))
                ((not(or (= in2 0) (= in2 1)))
                 (error "Invalid or-gate message." in2))
                (else (set-message! output (logical-or in1 in2))))))
      (add-new-port! input1 or-gate-function)
      (add-new-port! input2 or-gate-function))
    (define (and-gate input1 input2 output)
      (define (logical-and b1 b2) (if (and (= b1 1) (= b2 1)) 1 0))
      (define (and-gate-function value)
        (let ((in1 (get-message input1))
              (in2 (get-message input2)))
          (cond ((not(or (= in1 0) (= in1 1)))
                 (error "Invalid and-gate message." in1))
                ((not(or (= in2 0) (= in2 1)))
                 (error "Invalid and-gate message." in2))
                (else (set-message! output (logical-and in1 in2))))))
      (add-new-port! input1 and-gate-function)
      (add-new-port! input2 and-gate-function))
    (define (inverter input output)
      (define (logical-invert b1) (if (= b1 1) 0 1))
      (define (inverter-function value)
        (let ((in (get-message input)))
          (cond ((not(or (= in 0) (= in 1)))
                 (error "Invalid inverter message." in))
                (else (set-message! output (logical-invert in))))))
      (add-new-port! input inverter-function))
    (let* ((output0r (connection))(output1n (connection)))
      (or-gate input1 input2 output0r)
      (and-gate input2 input1 carry)
      (inverter carry output1n)
      (and-gate output0r output1n sum)))
]]> </component>

```

Figure 4.2: XML component code specifying a connector-based half-adder.

```

<?xml version = "1.0" encoding = "UTF-8"?> <compositioncode name =
"half-adder">
  <connection name = "input1" type = "input"/>
  <connection name = "input2" type = "input"/>
  <connection name = "sum" type = "output"/>
  <connection name = "carry" type = "output"/>
  <connection name = "outputOr" type = "internal"/>
  <connection name = "outputInv" type = "internal"/>
  <component name = "or-gate" filename = "d:/or-gate.xml">
    <connection name = "input1"/>
    <connection name = "input2"/>
    <connection name = "outputOr"/>
  </component>
  <component name = "and-gate" filename = "d:/and-gate.xml">
    <connection name = "input2"/>
    <connection name = "input1"/>
    <connection name = "carry"/>
  </component>
  <component name = "inverter" filename = "d:/inverter.xml">
    <connection name = "carry"/>
    <connection name = "outputInv"/>
  </component>
  <component name = "and-gate" filename = "d:/and-gate.xml">
    <connection name = "outputOr"/>
    <connection name = "outputInv"/>
    <connection name = "sum"/>
  </component>
</compositioncode> ]]> </component>

```

Figure 4.3: XML composition code for the connector-based half-adder.

```

<?xml version = "1.0" encoding = "UTF-8"?> <compositioncode name =
"half-adder-test" version = "2.3" type =
  "start" start-component = "user-interface-in" start-method =
  "start">
  <connection type = "internal" name = "uIIn2"/>
  <connection type = "internal" name = "sum"/>
  <connection type = "internal" name = "carry"/>
  <component name = "user-interface-in" filename = "d:/user-interface-in.xml">
    <connection name = "uIIn1" type = "internal-out"
      output-method = "add-new-port1!"/>
    <connection name = "uIIn2" type = "internal-out"
      output-method = "add-new-port2!"/>
  </component>
  <component name = "half-adder" filename = "d:/half-adder.xml">
    <connection name = "uIIn1" type = "internal-in"
      input-method = "input1-changed"/>
    <connection name = "uIIn2" type = "internal-in"
      input-method = "input2-changed"/>
    <connection name = "sum" type = "internal-out"
      output-method = "add-new-port1!"/>
    <connection name = "carry" type = "internal-out"
      output-method = "add-new-port2!"/>
  </component>
  <component name = "user-interface-out" filename = "d:/user-interface-out.xml">
    <connection name = "sum" type = "internal-in"
      input-method = "input1-changed"/>
    <connection name = "carry" type = "internal-in"
      input-method = "input2-changed"/>
  </component>
</compositioncode> ]]> </component>

```

Figure 4.4: XML composition code for creating a half-adder test-bed.

There are subtle differences between the attribute values used in the XML CCs for the early connector-based tests and the later connector-less engine tests. These differences reflect the evolution of our XML model as we gained experience through development.

4.1.3 A Component-Based Model Engine

The third engine prototype implemented the connector-less model of the second with minor modifications to the XML component and composition specifications in order to be consistent with the DTDs specified in Sections 3.2.1 and 3.2.2. During the writing of this thesis, further thought on certain points resulted in extensions of the early XML specifications used in developing the first two prototypes. The major

Figure 4.5: Component architecture of the third generation engine.

change however involved breaking the engine into components. The structure for the composite is shown in Figure 4.5. These components were then assembled by a second generation engine into a third generation composite engine and run. This second concurrently running (third generation) engine was tested first by assembling

the half-adder whose resulting XML specification is shown in Figure 3.2. Then, this third generation engine was fed its own CC to create a composite copy of itself shown in Figure 4.6, that it then proceeded to run.

```
<?xml version = "1.0" encoding = "UTF-8"?> <component name =
  "ADLEngine" version = "3.0" type = "start" start-method = "start">
  <interface>
    <subcomponent name = "engine1" filename = "d:/engine.xml"/>
    <subcomponent name = "XMLparser1" filename = "d:/XMLparser.xml"/>
    <subcomponent name = "XMLwrapper1" filename = "d:/XMLwrapper.xml"/>
  </interface>
  <implementation>
    <![CDATA[
      (define (ADLEngine)
        (let* ((engine1 (engine))
              (XMLparser1 (XMLparser))
              (XMLwrapper1 (XMLwrapper)))
          (define (start) ((engine1 'start)))
          (define (dispatch procedure)
            (cond ((equal? procedure 'start) start)
                  (else (error "invalid procedure." procedure))))
          ((XMLparser1 'add-new-port!) (cons engine1 'process-Sexpression))
          ((XMLwrapper1 'add-new-port!) (cons engine1 'process-XML-Component))
          ((engine1 'add-new-port1!) (cons XMLparser1 'parse-XML))
          ((engine1 'add-new-port2!) (cons XMLwrapper1 'wrap-code))dispatch)))
    ]>
  </implementation>
</component>
```

Figure 4.6: XML component code specifying a connector-based half-adder.

It should be obvious from the XML component code in Figure 4.6 that this third prototype cannot run outside of the second running prototype which it depends on for providing its subcomponent code. To create a stand-alone version, we would have to include all the subcomponent code inside its implementation as noted earlier.

4.2 Discussion of Results

An important observation about this component model is that in the creation of a composite, no new computational functionality is added outside of subcomponents. The composite's code wrapper provides purely connective functionality avoiding the problem of "self" discussed in [BPM⁺98], which is the problem of wrapper methods overriding methods in subcomponents that they are connected to. Our engine constructs all of the required connective code for the wrapped subcomponents as may be seen in the XML component created for the engine itself shown in Figure 4.6.

If a designer wishes to add functionality in addition to what available components provide, he must create a new set of components embodying this new functionality and wire it in accordingly. This means the designer need not understand any code associated with the components being used treating the component as a black box. Additional code development may therefore be relatively modest. It also means that if components provide complete functionality, then the assembly process may be initiated by persons with no programming language knowledge. The half-adder example illustrates this latter claim. The XML for the half-adder CC in Figure 3.3 may be written directly from the circuit diagram shown in Figure 4.1.

The process of modifying or correcting component code requires only the replacement of the affected components. The possibility discussed in Section 5.3.5 is for both static or assembly-time, and dynamic or runtime modification. Our prototype currently only implements static reassembly. However, as noted in Section 4.1, an early prototype was modified to facilitate the inclusion of dynamic adaptation.

Practically, the engine should be able to assemble multiple composites (sequentially) and as a runtime framework, execute multiple composites in separate threads

while loading and maintaining a list of components serving the multiple composites. Our prototype maintains such a list of components but only implements single-threaded composite execution.

4.2.1 Treatment of Data and Data Repositories

Data may have a local nature or a global nature. The former is represented by local state internal to a component and has no exposure to the outside world except for special purposes such as dynamic replacement of components. Global data represents data as a system resource. As such it is operated on by components and may seem to deserve representation as a first class object in an architectural design.

The solution is to consider significant stores of data as special elements which we designate as repositories. Technically, they are a form of connector. They may implement a data management protocol such as a SQL server. However they do not operate on or transform data beyond possible formatting and presentation requirements.

Examples of architectural style in Section 2.3 offer insight into how this characterization can be used. For example, a client/server style would restrict the set of components to those of type “client” and the set of connectors to those of type “server”. Such repository elements may encapsulate significant computational complexity but none that would transform the data administered.

4.2.2 Suitability of the XML Specification

Part of the desired outcome of using an ADL depends on engine design as much as schema design. For example, we may provide the capability in the schema to specify a constraint such as a timing dependency, as an assertion of a property. The means of

validating the assertion depends on the reasoning algorithm employed by the engine. Such is outside the scope of an ADL.

Similarly, many issues of performance have little if anything to do with the XML other than the provision of the ability to express such issues in the form of constraints. Most such issues are in the domain of the specific implementation of the engine which processes the XML. The major area of performance concern lies in the performance cost of the method pass-through which occurs down a subcomponent hierarchy. This is very much a property of our ADL specification. In a distributed software application, such cost would likely be masked by latency of remote procedure calls. There is a requirement in future work, to measure this cost for a strictly local component application. The results may have some impact on scalability.

Our engine design implements an object model using a functional programming language. So functional languages are feasible for implementations and we expect an OO language like Java will prove equally suitable. This opens an interesting question, however. How well would an implementation in a logic programming language perform? The advantage of such an implementation may be in terms of incorporating reasoning components for constraint processing. This remains a topic for future research.

In summary, our XML metalanguage when compared against the characteristics of an ideal ADL listed in Tables 3.1 and 3.2, has all of the properties of an ideal ADL with two reservations. The first is, in terms of component (and connector) specification, it may be limited in its ability to express the semantics of behavior. The second is its support for dynamic modification may be inadequate. Both issues we explore briefly in Chapter 5. The issue of expression and use of non-functional

requirement information is related to the expression of information on style. The third schema developed in Section 3.2.3 accommodates such expression. However, we do not incorporate any style processing capability in our engine prototype. Experience with larger systems will be needed to affirm the adequacy of our ADL.

4.2.3 Success of the Engine Prototype

We present a detailed exposition on the evolution of engine design features and testing in Section 4.1. The results of this development are summarized in this section.

The example chosen for component assembly involved the construction of a half-adder circuit from individual logic gates expressed as atomic components. The initial engine prototype successfully created the composite circuit with explicitly modelled connectors in conformance with most current ADLs.

A second prototype was built treating connectivity as an implicit property of components, a design decision defended in Chapter 3. This engine also successfully composed the half-adder circuit. The next step in testing involved constructing interface components so that the half-adder composite could be composed into a test-bed which would allow a user to test it.

We then ran a two-pass test. The first pass composed a half-adder component from a set of three subcomponents, and stored the result locally. The second pass composed this composite with input and output interface components to create an interactive test-bed for the half adder. It then successfully ran the resulting complete application allowing the exercise of the half-adder circuit.

As development proceeded, it became necessary to modify the engine to accommodate extensions to the XML. In addition, there was a desire to test the model on

a somewhat more elaborate application. To accommodate both requirements, we decomposed the engine into three components and wrote a CC for its reassembly which the second prototype performed. The resulting third-generation prototype when run by the second one was able to not only assemble the half-adder correctly but also was able to assemble a copy of itself which it then in turn started.

In summary, the current (third-generation) prototype is able to assemble a set of components into a composite component, create a well-formed XML wrapper for the composite, store it, and run it in the case where the composite has a start method. The level of sophistication of the application assembled has been successfully tested up to a copy of itself.

Chapter 5

Conclusion

In the Introduction to this thesis we posited that problems associated with current methods of software development and marketing include:

- lack of meaningful user choice over functional content,
- lack of user control over development schedule,
- excessive complexity of products from the perspectives of both user and developer,
- requirement for developers to redevelop generic functionality with each new project,
- expense associated with large products both in terms of purchase costs for users and development costs for developers.

We proposed, as many others have done, a solution in the form of a software component paradigm consisting of:

- standard software components,
- a simple composition schema,
- a method of dynamic assembly to create an immediately runnable composite,
- a method of dynamically upgrading a composite.

Such a component approach has the potential for creating a large competitive market for components, producing wide choice in functionality both generic and specialized, at low cost. Such a market would provide the solution to the flexibility, choice and cost elements of the problem. Simplicity is provided by the underlying ADL and may be greatly enhanced by the development of support tools. Both application users and developers stand to benefit from this approach.

In the process of developing our component-based solution to these problems we have met the goals of this thesis and provided certain contributions which set our approach apart from that of others reviewed herein. In the following two sections, we evaluate how closely we have met our goals and claims of contribution.

5.1 Evaluation of Goals

In the Introduction, we stated one of our goals was to develop a component assembly model that under five subgoals would:

- allow for the specification of a standard well-defined component format,
- provide a simple to use mechanism for specifying component assemblies or compositions,

- provide dynamic or runtime assembly which can be performed on most computers under most operating systems,
- allow for such assembly to proceed in an automated fashion with the composite being automatically run on completion of assembly,
- allow for the dynamic modification of a composite assembly.

The XML component schema developed in Section 3.2.1 satisfies the first subgoal by providing the basis for such a well-defined standard format.

The XML composition schema developed in Section 3.2.2 satisfies the second subgoal by providing such a mechanism which we have argued and demonstrated to be simple to use.

These schemas are a metalinguistic specification based on XML which is a widely adopted standard implemented for most computer and operating systems. Any engine implementation using the engine design of Section 3.3 will therefore allow runtime assembly which can be performed on most computers under most operating systems. Hence subgoal three is satisfied.

We have automated the assembly process in our engine design to require operator intervention only to supply a composition code. Even this requirement could be eliminated using an agent-based format where the engine waits for agents to arrive and present such requests. Our fourth subgoal has therefore been satisfied.

We describe in Section 5.3.5 how dynamic adaptation may be implemented within the confines of our ADL and employing specified engine extensions. At this time, the adequacy and feasibility of dynamic adaptation remains unproven, due to the considerable effort anticipated to implement such. Thus we believe subgoal five is

feasible but cannot support the claim at this time with demonstrated results.

We have demonstrated an engine prototype which has successfully composed and run where appropriate, a set of test examples. This meets our second goal.

In summary, we have met the objectives of our goals stated in Section 1.4.

5.2 Evaluation of Contribution

In the Introduction, our claim of contribution consisted of:

- a formal definition of terms,
- a unique XML representation of an ADL meeting the requirements of an ideal ADL and consisting of a component schema and a composition schema,
- an XML schema to model architectural style and a proposal of how it maybe integrated with other constraint information by an engine,
- a technique for the automated assembly of components into persistent composite structures and their execution on any specified computer. Components may exist locally or remotely and their composite may be local or distributed,
- a working prototype of a composition engine.

In this thesis we have provided exactly the set of formal definitions we had proposed. Such definitions in turn proved to be a valuable guide to the development of the XML schemas we present for the specification of standard components and their composition. Our analysis indicates the resulting ADL has all the characteristics of an ideal ADL with two minor reservations.

We have demonstrated a technique and a working prototype of an engine which performs the local automated assembly, storage and execution of composites. We have indicated how this technique may be extend to created distributed applications.

We claim that based on a survey of the literature, the application of XML that we demonstrate is unique. The formal representation is also unique, both in its style and its extent.

5.3 Further Work

Much was learned as this project progressed. Many issues that surfaced could not be addressed within the scope and time constraints of the project and are briefly outlined below. Areas where simplifications were made are discussed also.

5.3.1 Component Location and Discovery

The engine has been constructed to search for components on the local system only. We specified the location of a component by explicit inclusion of such in the XML composition specification. By an extension of both the XML composition schema and the engine, it would be possible to find components on other systems using general URI references.

Since the composite wrapper can be thought of as a wiring or communication network specification, there is no requirement to have all or any subcomponents executing locally allowing a truly distributed component computing model. Some care would have to be taken to ensure the engine, in constructing the wrapper, used remote instead of local procedure call mechanisms where required. The internal list of

components the engine maintains may have to be modified to indicate remote versus local components.

More generally, widespread use of components will require component brokerages such as the knowledge component brokerage that the European IBROW3 project [BPM⁺98] has developed. These will be repositories to which component developers submit their components and which composition engines search for candidates. Open searches of the World-Wide Web, even with specially tailored search engines, are a less efficient alternative.

To include search capability, the engine will have to be extended. If brokerages are used, there will be an entire protocol the engine will have to incorporate and execute. If search algorithms are to be used these will have to be incorporated either directly in the engine design or incorporated using a plug-in model. The XML schema does not need to be extended. Failure to specify a location or failure of the location to provide the component could cause a default to a search mode. One extension that may be useful however, is a “recommended-brokerage” attribute or element which the component designer could include with his component design facilitating the discovery process.

5.3.2 Component Identification and Interface Semantics

The issue of what component to select in the first place is considerably more problematic.

It is useful to separate the roles of designer and assembler. The composition designer must know a great deal about a component in terms of its interface, behavior, constraints and properties in order to select the exact component needed for the

application and to specify properly a composition that uses it. Its name and version are secondary information that the designer passes on to the assembler via a CC. The assembler on the other hand is mainly interested in locating the correct component and not on its semantics unless the latter will help in making a correct selection.

In the case of our half-adder, we know from experience (and having built the component) exactly what an “and-gate” does. We know its behavior in terms of the transformations that it performs on inputs. We know its input and output data types and the domain and range of input and output values. As a result, we didn’t need any further specification to make the selection. In a commercial environment however, the component designer may wish to specify some of this information as constraints. For example, “input1 value is 1 or 0”. This aids selection decisions by designers who only have a black-box knowledge of the component.

A next step would be to introduce input and output semantic type attributes governed by a common or standard specification. By this we mean a commonly agreed upon meaning for type names. Since the range of types is unbounded, this may not be an effective approach. We have included a provision for type specification in the XML but did not use it in our testing. If necessary, constraints may be represented by attributes that specify such characteristics as value ranges or boundary conditions for the operation of the component.

Semantic information must be detailed enough that the users can assess whether a particular component meets their requirements. Users may be human designers or intelligent composition engines.

XML has an advantage over standard programming languages in that the ontologies defined by the tags in a document are represented by natural language text that

can be parsed and potentially understood by both humans and computers. Eventually, composition engines will have to have a strong AI component to assess the suitability of a candidate component.

Finally, we may want to include a behavior attribute that in some way identifies how inputs are modified or used to create well-defined outputs. Explicit behavior of concurrent processes have a formal representation in process algebra [Lum00] allowing for the development of tools for automated checking of the correctness of composite behavior. The issue of how to integrate such within XML is an open problem although it should not be difficult due to the extensible nature of XML. The issue of what kind of facility should be built into the engine to process the constraints is more difficult. There are some tools available that process expressions in process calculus which may be useful in this context.

5.3.3 System Calls and Error Handling

A component model benefits from a pure object concept allowing the construction of an interface whereby all access to functionality is through public and only public methods. Any necessary access to state is provided through public get and set methods. Unfortunately, programmers sometimes require and often use direct access to system resources without going through this interface. Examples are calls to a specific output port or a device such as a communication channel or printer.

An important case is that of how a component traps and reports errors. If a component developer reports an error to a system error device, the engine may not gain access to the condition. Worse, the system the component runs on may not offer such a service causing unpredictable results.

In a specific implementation such as an engine written in JAVA and running in a JVM these problems may not arise. However, a formal specification abstracted above specific platforms will require that components declare the semantics of *all* their requirements in their interfaces.

In the same sense that component interaction has been abstracted out into a connector element, we may wish to create a class of utility components that serve as abstractions for all input and output functions in an application.

These extensions are beyond the scope of this project.

5.3.4 Security

Security issues have been deemed to be out of scope of this project but cover three areas:

- security of the host system and the composition engine,
- security of the assembly process including any constraints and policies specified in the CC,
- security implemented in the composite through inclusion of security subcomponents such as cryptographic algorithms [Men00].

Our engine could easily integrate a security component into a composite using a pipe-and-filler style, thus satisfying the third point. Any commercial application will have to include security coverage of the first two points in engine design. It does not seem that any immediate extension to the XML is necessary to cover any of these security issues.

Research efforts that may relate to future development of security aspects, particularly associated with our XML schemas are XML Key Management Specification (XKMS) [W3C01b], a set of protocols for distributing and registering public keys, and Security Assertion Markup Language (SAML) [(SS)], an XML-based security standard for exchanging authentication and authorization information.

5.3.5 Dynamic or Runtime Adaptation (Hot Swapping)

A component may or may not encapsulate state. In our example of a half-adder, the and-gate component stores the value of its inputs and only produces an output if an input changes. To dynamically replace this component, its replacement must be composed with the same state values if the composite is to function without fault. Since the component design is fully encapsulated, it is up to the component designer to incorporate methods to report and change state values. This argues for a special class of methods since we potentially want to restrict access to these methods to the engine and not the component user.

In a full framework that manages a composite's execution, runtime adaptation or replacement of components is feasible. As an example, suppose in a composite, component C receives inputs from A and B and sends output to D . C is to be replaced with C^* . The engine locks C and asks it to report its state and its output notification lists. It then instantiates C^* and sets it with C 's state and C 's notification lists. It next informs A and B to remove C from their notification lists and insert C^* in its place. If A and B are not locked then the order in which these actions are performed may be important.

Since the engine maintains control of threads of execution, the above scenario with

component locking should be feasible.

Solutions to timing issues are not considered here but may be important, particularly in distributed systems. *A* and *B* may also have to be locked (and possibly their inputs and so on) to prevent loss of information while the swap is taking place. If there is any kind of synchronous processing among subcomponents, there is the risk of computational error. Defining the swap as an atomic activity as one source has proposed is not a realistic solution. Process algebra was designed to describe the behavior of concurrent processes in a formal manner. Since distributed systems are essentially systems of concurrent processes, modelling their behavior as described in Section 5.3.2 may offer means for modelling timing issues. This remains an important open area of research for component models.

The XML composition schema is a compact representation of an application and may be used as an effective small-footprint template for storing and transferring applications. An important issue then is the simultaneous adaptation or changing of XML CCs and the composites they represent. XML support of this nature may be built into the engine or performed with a separate tool.

The schema is a specification of a top level structure. When hierarchical compositions exist, the schema only identifies the top level occurrence of a component. Other occurrences may very well exist down the composition hierarchy. So if the engine is to make a replacement automatically, it must recursively search for and replace all occurrences. Should only certain occurrences need to be replaced - say the situation exists where different versions of the same component must be used in a configuration - then the replacement will have to be guided by an external specification. This is a significant issue for further investigation.

We note that there is a considerable literature and research extant on hot swapping that should be investigated in the context of this problem.

5.3.6 Constraint Checking

The final part of this discussion involves the use of constraints which we have made provision for in all schemas and which are the defining characteristics of an architectural style. In our simple examples, we successfully created applications without imposing any constraints or using a style. Experience with text processing applications shows that styles or style sheets are a powerful tool for implementing constraints when properly designed and used. So there is motivation to consider how they might be used in component assembly.

In using the style schema with the composition schema the style information allows the engine to verify that the domain elements and configuration specified by the composition code are correct. That is the configuration manifests an allowable topology, the components and connectors are of the correct type for the style and any constraints are logically consistent.

The ability of an engine to use constraint information in the form of predicates or rules is directly proportional to its intelligence and the information in its knowledge base if such an AI technique is used. A standard ontology of constraint properties would facilitate the operation of such an engine. An ontology learning capability would likely be a necessary component of this engine.

The alternative is to embed a limited ontology in the XML tag structure. A simple case-based pattern-matching algorithm can then process the constraint information in the XML file. We have included this option in our XML schema for the most likely

constraint features, the cases of component type and configuration topology. The advantage of this approach is its simplicity and presumed better performance over an AI approach. The disadvantage is that it is limited due to a fixed case structure.

We note that composition is hierarchical in nature where any component may itself be a complex composition of other components. Such a component may implement any style or group of styles that are consistent among themselves. However the encapsulation of this complexity inside a composite component prevents any interaction between the styles of different levels in such a hierarchy.

Finally, resolving a two element abstraction model of connectors and components to a single element implementation model should not present problems from the viewpoint of element type since the separate classifications should be orthogonal. In the case of any other constraint which would appear to be inconsistent between the two sets of elements, the inconsistency may be resolved by appending the element type to the constraint in question. Facilities built into the engine to handle constraints in general should easily handle style refinement.

5.3.7 Other Domains of Relevance

The use of a hardware example suggests an examination of the body of research on tools to support the development of hardware components and their assemblies, such as silicon compilers and VLSI design tools. The field of design of large-scale hardware composites from basic components is much more mature than the corresponding field of software component assembly. As a result, there may be ideas and implementations which would be transferable to the development of automated software composition tools.

Aspect-Oriented Software Development (AOSD). This work is described by the AOSD Steering Committee [AOS] as:

a new technology for separation of concerns in software development. The techniques of AOSD make it possible to modularize crosscutting aspects of a system [such as] design or architectural constraints, systemic properties or behaviors (e.g., logging and error recovery), and features.

This area appears to address the large number of concepts, terms, views and issues on software architecture arising from the plethora of viewpoints from which the authors cited in Chapter 2 write. As such it may have a contributory impact on our formalism and may lead to the development of new tools for architectural specification and development.

5.3.8 Tools

During the development of the prototype, it became evident that there was a need for certain tool to assist in writing the XML.

The first tool considered was that of a component packager. This would be a tool with a wizard that guides the component developer through the successive stages of building the XML code for a component. The developer would be prompted to specify the component attributes and constraints if any, and to supply the source code for the implementation. The tool should have an edit facility for modifying existing XML code packages. Finally, the tool should have a simple GUI for human users, but also a machine format interface so that it may be composed with other components to build applications such as a composition engine.

The second tool that would be useful is a composition code generator. This tool would also have a GUI with drag and drop capability. Menus of components could be created dynamically. The designer would then drag component icons onto a blackboard and draw directed edges to connect them. The tool could prompt for appropriate labels for component methods and port names. The output would be an XML CC.

Finally, there is room for a tool that would build constraints to support the XML schema proposed in Section 3.2.3. Consistency checking could be a subcomponent of the tool and would likely prove useful in engine development.

Appendix A

List of Acronyms

- ADL** Architecture Definition Language. A language with an appropriate vocabulary and syntax suitable for describing the architecture or structure of software systems.
- AOSD** Aspect-Oriented Software Development. A technology for separation of concerns in software development. The techniques of AOSD make it possible to modularize crosscutting aspects of a system.
- CBSE** Component-Based Software Engineering. A subdiscipline of Software Engineering focusing on the reuse of object middleware components.
- CC** Composition Code. An XML specification of the (engine) instructions for assembling a composite component.
- CORBA** Common Object Request Broker Architecture. The Object Management Group middleware specification that allows distributed objects to communicate over a network.

DARPA Defense Advanced Research Projects Agency. A department of the US government responsible for managing defense-related basic research. The agency responsible for the creation of the Internet.

DCOM Distributed Component Object Model. The Microsoft protocol that allows distributed objects to communicate over a network.

DOM Document Object Model. An interface that allows programs and scripts to dynamically access and update the content, structure and style of documents.

DTD Document Type Definition. Part of the XML specification providing instructions for determining the validity of an XML document.

FTP File Transfer Protocol. A protocol to promote sharing of files between two machines across a network.

IDE Integrated Development Environment. A set of tools such as editors, compilers, linkers, library managers, project managers and runtime operators, integrated to work together to provide a complete environment for the development of software programs.

IDL Interface Definition Language. A language used to specify the properties of the interface of an object or module of code.

JVM Java Virtual Machine. The Java language interpreter.

OO Object-Oriented. A programming paradigm supporting code encapsulation, inheritance and polymorphism.

- SAML** Security Assertion Markup Language. An XML-based security standard for exchanging authentication and authorization information.
- SAX** Simple API for XML. The standard interface for event-based XML parsing.
- SQL** Structured Query Language. A language for specifying queries made of the data in a database.
- SXML** A concrete instance of the XML Infoset (an abstract data set that describes information available in a well-formed XML document) in S-expression format particularly suitable for Scheme-based XML authoring.
- UML** Unified Modelling Language. A language for specializing, visualizing, constructing and documenting the artifacts of software systems.
- URI** Uniform Resource Identifier. A compact string of characters for identifying an abstract or physical resource.
- URL** Uniform Resource Locator. The subset of URI that identify resources via a representation of their primary access mechanism.
- VLSI** Very Large Scale Integration. In hardware design, this refers to the design of components containing in the hundreds of thousands of transistors.
- XKMS** XML Key Management Specification. A set of protocols for distributing and registering public keys.
- XML** Extensible Markup Language. The universal format for structured documents and data on the World-Wide Web

Bibliography

- [Act99] Active Network Working Group. Architectural framework for active networks, July 1999. <http://www.dcs.uky.edu/calvert/arch-docs.html>.
- [AG94] Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 71–80, Sorrento, Italy, May 1994.
- [Age] The Defense Advanced Research Projects Agency. Vision. <http://www.darpa.mil/ito/research/anets/vision.html>.
- [AKPS00] Matthias Anlauff, Philipp W. Kutter, Alfonso Pierantonio, and Asuman Sünbül. Using domain-specific languages for the realization of component composition. In *Fundamental Approaches to Software Engineering*, pages 112–126, 2000.
- [ALSN00] Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Piccola – a small composition language. In Howard Bowman and John Derrick, editors, *Formal Methods for Distributed Processing, an Object Oriented Approach*. Cambridge University Press, 2000.
- [AN 98] AN Composable Services Working Group. Composable services for active networks, September 1998.
<http://www.cc.gatech.edu/projects/canes/papers/cs-draft0-3.pdf>.

- [AOS] AOSD Steering Committee. home page.
<http://aosd.net/>.
- [Aoy98] Mikio Aoyama. New age of software development: How component-based software engineering changes the way of software development. In *Proceedings of the International Workshop on Component Based Engineering*, Kyoto, Japan, April 1998.
- [BM98] Markus Breugst and Thomas Magedanz. Mobile agents – enabling technology for active intelligent network implementation. *IEEE Network*, 12(3):53–60, 1998.
- [BMR⁺96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley and Son, 1996.
- [BPM⁺98] V. Benjamins, E. Plaza, E. Motta, D. Fensel, R. Studer, B. Wielinga, G. Schreiber, Z. Zdrahal, and S. Decker. Ibro3: An intelligent brokering service for knowledge-component reuse on the World-Wide Web. In *The 11th Banff Knowledge Acquisition for knowledge-Based System Workshop (KAW98)*, Banff, Canada, 1998.
- [Bro98] Alan Brown. From component infrastructure to component-based development. In *Proceedings of the International Workshop on Component Based Engineering*, Kyoto, Japan, April 1998.
- [BW98] Alan W. Brown and Kurt C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, 1998.
- [Cat95] Fabiano Cattaneo. An evaluation of architecture description languages. Technical Report 96.152, Politecnico di Milano, Dipartimento di Elettronica e Informazione, October 1995.

- [CHDG99] Qiming Chen, Meichun Hsu, Umeshwar Dayal, and Martin Griss. Multi-agent cooperation, dynamic workflow and XML for e-commerce automation. Technical Report HPL-1999-136, Hewlett Packard Software Technology Laboratory, 1999.
<http://www.hpl.hp.com/techreports/1999/HPL-1999-136.html>.
- [CR99] Cynthia D. T. Cicalese and Shmuel Rotenstreich. Behavioral specification of distributed software component interfaces. *IEEE Computer*, 32(7):46–53, July 1999.
- [Cra96] Stephen Crane. A framework for distributed interaction, November 1996.
<http://citeseer.nj.nec.com/18406.html>.
- [CWD00] Francisco Curbera, Sanjiva Weerawarana, and Matthew J. Duftler. On component composition languages. In *Proceedings of the 5th International Workshop on Component-Oriented Programming (WCOP 2000)*, Sophia Antipolis, France, June 2000.
<http://www.ipd.hk-r.se/bosch/WCOP2000/submissions/Weerawarana.pdf>.
- [Dep97] Department of Computing, Imperial College of Science, Technology and Medicine, London, UK. *The Darwin Language*, 3d edition, 1997.
<http://www.doc.ic.ac.uk/jsc/research/darwin-lang.pdf>.
- [Der] Luca Deri. Yasmin: A component based architecture for software applications. In *Proceedings of 8th IEEE International Workshop on Software Technology and Engineering Practice (STEP '97)*, pages 4–12, London, UK. IEEE Computer Society Press.
- [Der97] Luca Deri. *A Component-based Architecture for Open, Independently Extensible Distributed Systems*. PhD thesis, Institut Fr Informatik und angewandte Mathematik, Univeritt Bern, June 1997.

- [dSFY98] Sushil da Silva, Danilo Florissi, and Yechiam Yemini. Composing active services in netscript. In *DARPA Active Networks Workshop*, Tucson, AZ, March 1998.
- [Edd99] Guy Eddon. Com+: The evolution of component services. *IEEE Computer*, 32(7):104–106, July 1999.
- [GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 175–188, 1994.
- [Gar96] David Garlan. Style-based refinement for software architecture. In *Joint Proceedings of the Second International Software Architecture Workshop (ISAW2) and the International Workshop on Multiple Perspectives in Software Development (Viewpoints '96)*, San Francisco, CA, October 1996. ACM Press.
- [GMW97] David Garlan, Robert Monroe, and David Wile. ACME: An architecture description interchange language. In *Proceedings of CASCON'97*, November 1997.
- [Gut99] Erik Guttman. Service Location Protocol: Automatic discovery of IP network services. *IEEE Internet Computing*, 3(4):71–80, 1999.
- [Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [JZ98] Michael Jackson and Pamela Zave. Distributed Feature Composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24(10):831–847, 1998.

- [Kis01] Oleg Kiselyov. *SXML Specification*. Naval Postgraduate School, Monterey, California, October 2001.
<http://okmij.org/ftp/Scheme/xml.html>.
- [Kni99] Günter Kniesel. Type-safe delegation for run-time component adaptation. In Rachid Guerraoui, editor, *Object-Oriented Programming 13th European Conference (ECOOP '99), Lisbon Portugal*, volume 1628, pages 351–366. Springer-Verlag, New York, NY, 1999.
- [Lim96] A. S. Lim. Composable scalable enterprises with agile autonomous components. In *Proceedings of the Academia/Industry Working Conference on Research Challenges (AIWoRC '00)*, Buffalo, NY, October 1996. IEEE Press.
- [Lum00] Markus Lumpke. *A π -Calculus Based Approach for Software Composition*. PhD thesis, Institut für Informatik und angewandte Mathematik, Universität Bern, 2000.
- [LV95] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.
- [McI76] M. D. McIlroy. Mass-produced software components. In Naur, Randell, and Buxton, editors, *Software Engineering: Concepts and Techniques, Proceedings of the NATO Conferences*, pages 88–98, New York, 1976.
<http://cm.bell-labs.com/cm/cs/who/doug/components.txt>.
- [MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

- [Meg00] David Megginson. SAX 2.0: The simple API for XML, 2000.
<http://www.megginson.com/SAX/sax.html>.
- [Men00] David W. Mennie. An architecture to support dynamic composition of service components and its applicability to internet security. Master's thesis, Department of Systems and Computer Engineering, Carleton University, 2000.
- [Mic96] Microsoft Corporation. DCOM technical overview. Technical report, 1996.
- [Mil80] Robin Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science No. 92. Springer-Verlag, 1980.
- [Mil93] Robin Milner. The polyadic pi-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.
- [MKS89] Jeff Magee, Jeff Kramer, and Morris Sloman. Constructing distributed systems in Conic. *IEEE Transactions on Software Engineering*, 15(6), 1989.
- [MM99] P. Mckee and I Marshall. Behavioural specification using XML. In *Proceeding of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 53–59. IEEE Press, 1999.
- [MP00] David Mennie and Bernard Pagurek. An architecture to support dynamic composition of service components. In *Proceedings of the 5th International Workshop on Component-Oriented Programming (WCOP 2000)*, Sophia Antipolis, France, June 2000.
<http://www.ipd.hk-r.se/bosch/WCOP2000/submissions/mennie.pdf>.

- [MPW89] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes parts I and II. Technical Report ECSLFCS -89-85, University of Edinburgh, Computer Science Department, 1989.
- [MQR95] Mark Moriconi, Xialei Qian, and R.A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995. <http://www.sdl.sri.com/papers/tse95/>.
- [MS77] Kai-Uwe Mätzel and Peter Schnorf. Dynamic component adaptation. *Communications of the ACM*, 40(1), January 1977. <http://www.gridforum.org/>.
- [MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 28(1):70–83, 2000.
- [ND02] Ian Nunn and Dwight Deugo. Automated assembly of software components based on XML-coded instructions. In *To appear in Proceedings of the 17th ACM Symposium on Applied Computing (SAC 2002)*, Madrid, Spain, March 2002. ACM Press.
- [NR99] Elisabetta Di Nitto and David Rosenblum. Exploiting ADLs to specify architectural styles induced by middleware infrastructures. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 1999)*, pages 13–22, Los Angeles, CA, 1999. ACM Press.
- [Obj01] Object Management Group. OMG unified modeling language specification, Version 1.4, September 2001. <ftp://ftp.omg.org/pub/docs/formal/01-09-67.pdf>.
- [PD99] Nat Pryce and Naranker Dulay. Dynamic architectures and architectural styles for distributed programs. In *Proceedings of the 7th IEEE Workshop*

on Future Trends of Distributed Computing Systems (FTDCS '99), Cape Town, South Africa, December 1999.

<http://www-dse.doc.ic.ac.uk/~np2/papers/ftdcs99.pdf>.

- [PSSC98] Steve Pruitt, Doug Smart, Wonhee Sull, and T. M. Cook. The merit of XML as an architecture description language meta-language. Technical report, Microelectronics and Computer Technology Corporation, 1998.
<http://citeseer.nj.nec.com/pruitt98merit.html>.
- [PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.
- [SC97] Mary Shaw and Paul Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *Proceedings 21st International Computer Software and Applications Conference (COMPSAC97)*, pages 6–13, August 1997.
- [Sch] Drscheme version 103.
<http://www.cs.rice.edu/CS/PLT/packages/drscheme/>.
- [SG94] Mary Shaw and David Garlan. Characteristics of higher-level languages for software architecture. Technical Report CMU-CS-94-210, Carnegie Mellon University, School of Computer Science, December 1994.
- [SG96a] Mary Shaw and David Garlan. Formulations and formalisms in software architecture. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Development*, Volume 1000 of Lecture Notes in Computer Science, pages 307–323. Springer-Verlag, 1996.
- [SG96b] Mary Shaw and David Garlan. *Software Architecture Perspectives on an Emerging Discipline*. Prentice Hall, New Jersey, April 1996.

- [Sha01] Mary Shaw. The coming-of-age of software architecture research. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, pages 656–664a, Toronto, Canada, 2001. IEEE Computer Society.
- [(SS] XML-Based Security Services Technical Committee (SSTC). home page. <http://www.oasis-open.org/committees/security/>.
- [W3C98] Extensible markup language (XML) 1.0. Technical Report REC-xml-19980210, W3C, February 1998.
- [W3C01a] Document object model (DOM) level 3 core specification. Technical Report Version 1.0 W3C Working Draft 13, W3C, September 2001. <http://www.w3.org/TR/2001/WD-DOM-Level-3-Core-20010913/>.
- [W3C01b] Xml key management specification (XKMS) 1.0. Technical report, W3C, March 2001.
- [WRMT95] E. James Whitehead, Jr., Jason E. Robbins, Nenad Medvidovic, and Richard N. Taylor. Software architectures: foundation of a software component marketplace. In *Proceedings of the First International Workshop on Architectures for Software Systems*, pages 276–282, New York, 1995. ACM Press.