

SYNTHESIS OF A SERVICE-BASED  
ARCHITECTURE DESCRIPTION LANGUAGE

BY

MICHAEL WILLIAM RENNIE

A Thesis

Submitted to the Faculty of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Computer Science

University of Manitoba

Winnipeg, Manitoba

©September 11, 2005

**THE UNIVERSITY OF MANITOBA**  
**FACULTY OF GRADUATE STUDIES**

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a MSc thesis entitled:

**SYNTHESIS OF A SERVICE-BASED ARCHITECTURE DESCRIPTION  
LANGUAGE**

submitted by: *Michael W. Rennie*

in partial fulfillment of the requirements for the degree of: *MSc*

---

*Dr. Vojislav Misic, Advisor*  
*Computer Science*

---

*Dr. Yanni Ellen Liu*  
*Computer Science*

---

*Dr. Ekram Hossain*  
*Electrical and Computer Engineering*

Date of Oral Examination: *September 9, 2005*

The student has satisfactorily completed and passed the MSc Oral Examination.

---

*Dr. Vojislav Misic, Advisor*  
*Computer Science*

---

*Dr. Peter R. King*  
*Chair of MSc Oral*

---

*Dr. Yanni Ellen Liu*  
*Computer Science*

---

*Dr. Ekram Hossain*  
*Electrical and Computer Engineering*

(The signature of the Chair does not necessarily signify that the Chair has read the complete thesis.)

**THE UNIVERSITY OF MANITOBA**  
**FACULTY OF GRADUATE STUDIES**  
\*\*\*\*\*  
**COPYRIGHT PERMISSION PAGE**

**Synthesis of a Service-Based Architecture Description Language**

**BY**

**Michael William Rennie**

**A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University  
of Manitoba in partial fulfillment of the requirements of the degree  
of**

**MASTER OF SCIENCE**

**MICHAEL WILLIAM RENNIE©2005**

**Permission has been granted to the Library of The University of Manitoba to lend or sell copies of this thesis/practicum, to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film, and to University Microfilm Inc. to publish an abstract of this thesis/practicum.**

**The author reserves other publication rights, and neither this thesis/practicum nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.**

## ABSTRACT

The service-based computing paradigm is rapidly gaining acceptance as a viable option for the creation of modern software systems. To effectively design and implement service-based systems, proper tool support is required. In this thesis we present our design and implementation of an architecture description language (ADL) which is tailored to the specification of service-based systems. The design of our ADL involved the creation of a formal language and the conversion of the language to an XML schema, which is further used for validation of specifications. The implementation of our ADL provides simple code generation for Jini for Java.

## ACKNOWLEDGMENTS

I would like to thank Dr. Vojislav B. Mišić for his continued support, most importantly though I would like to thank my wife Jennifer who always provided the support I needed to keep me focused on finishing my research.

## CONTENTS

1. <i>Introduction</i> . . . . .	11
2. <i>Related Work</i> . . . . .	16
2.1 Service-based Computing . . . . .	17
2.2 Software Architecture . . . . .	24
2.3 Architecture Description Languages . . . . .	29
2.3.1 C2SADL . . . . .	30
2.3.2 DAOP-ADL . . . . .	33
2.3.3 Darwin . . . . .	35
2.3.4 $\pi$ -ADL . . . . .	38
2.3.5 Rapide . . . . .	41
2.3.6 Weaves . . . . .	43
2.3.7 Wright . . . . .	45
2.3.8 xADL . . . . .	47
2.4 Infrastructure . . . . .	50
2.4.1 Jini . . . . .	52
2.4.2 OSGi . . . . .	55

2.5	Tool Support . . . . .	56
2.5.1	ACME . . . . .	57
2.5.2	ArchJava . . . . .	59
3.	<i>Service-based Computing</i> . . . . .	61
3.1	Component- vs. Service-based computing . . . . .	61
3.2	Component location and access . . . . .	65
3.3	Service search and selection . . . . .	68
4.	<i>The Language</i> . . . . .	70
4.1	Architectures . . . . .	71
4.2	Services . . . . .	73
4.3	Components . . . . .	74
4.4	Service Messages . . . . .	76
4.5	QoS Guarantee . . . . .	77
4.6	Resource Requirement . . . . .	78
4.7	Required Services . . . . .	78
5.	<i>The Implementation</i> . . . . .	82
5.1	Packages and Structure . . . . .	84
5.2	Package interaction . . . . .	89
5.3	Code Generation . . . . .	96
6.	<i>A Test Case</i> . . . . .	102

6.1	Defining the test case . . . . .	102
6.2	Using our ADL to describe the test case . . . . .	106
7.	<i>Conclusions</i> . . . . .	118
7.1	Future work . . . . .	118
	<i>Appendix</i>	121
A.	<i>XML Schema Definition</i> . . . . .	122
B.	<i>Notes on Jini</i> . . . . .	132



## LIST OF FIGURES

2.1	Simple architecture and component decomposition. . . . .	28
2.2	Simple C2 architecture. . . . .	31
2.3	Simple Darwin architecture. . . . .	35
2.4	Lazy instantiation. . . . .	37
2.5	Direct dynamic instantiation. . . . .	38
2.6	Simple $\pi$ -ADL architecture. . . . .	39
2.7	Simple Weave [14]. . . . .	44
2.8	Dynamic code insertion or 'Splicing' [14]. . . . .	45
2.9	Components provide services. . . . .	52
3.1	Components provide services. . . . .	62
3.2	Systems are built with components. . . . .	63
3.3	Systems can be built from services. . . . .	64
3.4	Components are used to provide services. . . . .	64
4.1	Architectures can include architectures. . . . .	71
4.2	Services might depend on others to provide their services. . . . .	79
4.3	Different components provide the same service with differing requirements. . . . .	80

5.1	Package description of ADL implementation. . . . .	84
5.2	Aggregation within the Structure package . . . . .	86
5.3	Arrangement of classes in the Structure package. . . . .	87
5.4	Arrangement of classes in the GUI package. . . . .	88
5.5	The class in the XML package. . . . .	89
5.6	Arrangement of classes in the Code package. . . . .	90
5.7	Overall sequence of ADL. . . . .	91
5.8	Creating a new ArchitectureDef instance. . . . .	92
5.9	Saving a specification. . . . .	94
5.10	Generating code for a specification. . . . .	96
5.11	Arrangement of sections within a codefile. . . . .	99
6.1	UAV can acquire/provide services from/to others in the fleet. . . . .	104
6.2	UAVs can have similar service types, with differing attributes. . . . .	105
6.3	Service selection can vary UAV system reconfiguration. . . . .	106
6.4	The architecture defined with a starting service. . . . .	107
6.5	The new services added to the UAV architecture. . . . .	108
6.6	Partial tree view of our UAV design. . . . .	109
A.1	Schema element <i>specification</i> . . . . .	122
A.2	Schema element <i>architecture</i> . . . . .	124
A.3	Schema element <i>starting-with</i> . . . . .	124
A.4	Schema element <i>service</i> . . . . .	125

A.5	Schema element <i>component</i> .	126
A.6	Schema element <i>service-message</i> .	127
A.7	Schema element <i>parameter-list</i> .	128
A.8	Schema element <i>parameter</i> .	129
A.9	Schema element <i>qos-guarantee</i> .	129
A.10	Schema element <i>required-resource</i> .	130
A.11	Schema element <i>required-service</i> .	131
B.1	Searching for a service.	132
B.2	Finding and acquiring services.	133
B.3	Jini proxy usage.	134

## LIST OF TABLES

2.1	SBC requirements, current systems and their associated problems. . . . .	25
3.1	Options for service locations and their accessibility. . . . .	66
5.1	Language elements and corresponding source artifacts. . . . .	98

## 1. INTRODUCTION

Consider the following scenario: A user has a document processing application and needs a custom spelling checker. The traditional solution to this problem would be to buy another application with the required functionality, or perhaps to develop it themselves if the available spelling checkers do not fit the requirements or budget (or both). More often than not, installation requires shutting down the base application (sometimes even the entire system) and restarting it afterwards; quite often, some manual interventions are needed too. Either way, the user will have to invest money and effort, and receive in return an application which might not be needed when the current job is done.

If the user is, in fact, a large company with hundreds or even thousands of employees and high-volume site-wide licenses, this translates into large amounts of money and extended time periods: not only to purchase the appropriate licenses, but also to install the newly acquired application where it is needed, to educate and train the users, and to provide support through extended periods of time. In summary, even a small addition to the existing functionality requires considerable expenditures, and the functionality may not be needed beyond a single project. Most likely, not all of the functionality of the basic document processing application is needed either.

It should come as no surprise, then, that a noticeable shift in user attitudes has appeared in recent years. Instead of purchasing a product to use only a part of its functionality for a limited period of time, and then leaving it on the shelf until it is needed next time, users are beginning to view software as a service. A service that should be available when the user needs it, but not longer than is needed; once the user is done with the service it does not persist on their computer.

The scenario outlined above might, then, look like this: upon recognizing that a major part of functionality is needed, the user would notify the application about it. The user might also specify some criteria regarding details of functionality, performance, security, price, and other characteristics of the service that implements that functionality. The listing of available services that fulfill the criteria specified would be presented to the user, so that the most suitable one can be selected (based on the user's preferences). The service is then downloaded and used, or used as a remote service. Depending on the terms of the license for the particular service, if downloaded, it may be deleted afterwards or remain stored on the user's computer. In the latter case, it may remain available for a specified time period or for a specified number of uses.

In summary, the user could purchase only the base functionality of an application. When extended functionality is needed, the components that provide that functionality can be acquired, dynamically bound to the base application, used as appropriate, and disposed of. More companies are moving towards a service-based approach to development and use of software products, as such an approach, commonly referred to as *service-based computing*,

can be shown to result in many benefits, technical as well as economical.

- Economy-wise, the user gets only the functionality they want, not a pre-packaged collection of functions that are used only rarely or never at all. Applications can be smaller and consume less resources. The impact in terms of performance is obvious, as are the economic benefits.
- The user is free to choose the service that best fits their needs. What's more, services from different vendors can be mixed and matched to build applications that can be customized as needed.
- In addition to that, the user can always have access to the latest version of the required functionality, rather than being bound to obsolete versions that are slowly becoming incompatible with everything else.
- The focus of development techniques shifts from large, monolithic applications with ever-changing requirements that are plagued by bugs for years, towards smaller components with a well defined scope that may be developed within a shorter time span and to strict quality requirements.
- Maintenance activities, for developers and support personnel alike, are simplified considerably.

However, advances in several areas are needed before the service-based paradigm can find widespread use. One such area is the design of service-based applications. Note that we are not referring to the design of individual components that provide services which do

not differ much from the design of similar units – classes, modules, packages, and the like – in the traditional approach. The differences become critical at the system level, where the service-based paradigm should be adhered to in the design right from the start, rather than applied later as an afterthought. In particular, the modeling formalisms must take dynamic reconfiguration into account (a system must be reconfigurable at runtime); so far, we have been successful only when designing systems with static architectures.

An important step in that direction is the development of architecture description languages (ADL) specifically designed to support the service-based approach. The availability of suitable ADLs will not only enable designers to specify the architecture of service-based applications in sufficient detail, but also to evaluate the design alternatives and validate them against the requirements, thus providing a firm foundation upon which service-based applications can be developed and implemented.

The shift to the service-based paradigm requires new and improved tools for the sufficient treatment of the software life cycle for this paradigm. Considering this shift and the focus on dynamic system reconfiguration, we now need software tools which can be used to create, validate, maintain and evolve systems of this type. The object of this thesis is how to describe software architectures that may be used to develop service-based, dynamically reconfigurable software systems. To solve this problem we have created an ADL and accompanying tool support that allows us to create, specify and validate architectures for the service-based paradigm. Our ADL includes a core set of requirements for such systems, and also provides a view and creates a style for service-based systems.



This thesis is organized as follows. Chapter 2 examines existing ADLs, with specific focus on those that support dynamism, followed by a brief discussion of infrastructure and tool support. Chapter 3 provides an in depth discussion of service-based computing, contrasted with traditional component-based systems, and some issues that require attention. Chapter 4 describes the language we created for our ADL, with both the structural and conceptual aspects of our language examined. Chapter 5 provides a comprehensive examination of the implementation of our ADL. Chapter 6 outlines a test case and uses our ADL to construct the specification for the test case. Lastly, in the accompanying appendix, we provide the complete XML schema for our language as well as technical notes on Jini [48].

## 2. RELATED WORK

Over the years there have been many advances in the field of software engineering. Such advances include the study of software architectures (SA) coupled with architecture description languages (ADL), providing a mechanism for describing software systems. The use of both SA and ADLs has gained popularity in the description of static systems, but, for the most part, still lacks the ability to describe systems that are dynamically reconfigurable, specifically those of the service-based paradigm. Such systems require specification methods that allows for descriptions of highly dynamic (both constrained and unconstrained) systems, that may be accessed remotely or locally. While quite a few ADLs have been created, and progress has been made in the areas of software architecture and service-based computing, we still lack an appropriate design time tool to facilitate the description of systems tailored to the service-based paradigm. The use of SA tightly coupled with an ADL tailored for the SBC paradigm would aid in the creation of a formal design, validation of the design and (possibly) code generation to execute the design.

Service-based computing (SBC), as mentioned above, changes how users view applications. Instead of purchasing large monolithic programs, that include functionality likely to rarely be used (if at all), users could instead purchase only the *base* functionality of an

application; when extended functionality is needed, the components that provide that functionality can be acquired, dynamically bound to the base application, used as appropriate, and disposed of. This approach to computing can be shown to result in many benefits, technical as well as economical.

This chapter is arranged as follows: In Section 2.1, we discuss the service-based computing paradigm, its requirements, what it is, and how it differs from traditional paradigms. Section 2.2 examines software architecture, its processes, what it is, and how it relates to traditional design paradigms. Section 2.3, we review existing ADLs, with focus on how they address any of the requirements of the service-based paradigm and dynamic system reconfiguration. Finally in Sections 2.4 and 2.5, we discuss infrastructure and tool support for ADLs to achieve dynamic system reconfiguration, and address the requirements of the service-based computing paradigm.

## *2.1 Service-based Computing*

Chances are good that you have been exposed to, or utilized some form of service-based computing and not even known it, like the Linux kernel, or Microsoft Office. The premise of the SBC paradigm, is that any computational service you require is (possibly) available when you require it. More specifically, the SBC paradigm is an approach where computational services required by the user are acquired and used only on demand. Services, in this instance, can be remotely utilized, much like web services [4, 5] or as a service downloaded and installed on your computer. While having the ability to use services on demand is

idyllic, there are associated drawbacks to the SBC paradigm, like the lack of appropriate security. Both the advantages and the disadvantages of this paradigm are examined in the remainder of this section.

Giving users, whether they are individuals or companies, the ability to utilize a host of different services based on personal criteria, and possibly from alternative vendors, is a giant leap forward from the traditional computing paradigm. Let us now examine the benefits of the SBC paradigm.

**Dynamic system reconfiguration.** Imagine having complete control over the parts or behavior of any program on your computer; such that you could update, change or completely reconfigure it to do whatever you wanted, whenever you wanted. Currently though, this level of unconstrained dynamism requires the aid of some form of infrastructure or internal framework [21, 50]. While system reconfiguration can take place at the users behest, it can also happen simply as a service has been updated by the vendor. Since services are typically only loaded on demand, if a newer version of a service has been created by a vendor, then that newer version would be loaded when the service is requested.

**Service selection.** Whose to say that every service must come from one vendor? In the SBC paradigm services can come from a multitude of service hosts and or vendors. There could be hundreds of services that all do the same thing, but that provide their service in a different way, or with other additional features, etc. Now, dependence upon individual service vendors has been removed; if you don't like the service you are using, you can update it or switch to another service that offers the same, or better, service.

**Increased performance.** Since software systems would now be created from a simple core program with separate services providing all other functionality, the performance of a software system could be improved. An example improvement could be defined as; (1) a decrease in the amount of memory required for the program, (2) a decrease in the amount of processing power needed, (3) a decrease in the amount of storage space required for the program and services, and (4) any combination of the previous three points. If a user wants a computationally fast service that uses very little memory, then that is what they get, provided such a service is available.

**Service reusability.** The development of new systems, or even the maintenance of old systems, can be intensive. Consider now that systems were composed of many services, each working together to make up one system. To update or maintain such a system would typically only require the redesign and implementation change, or refactoring of one service, and not the system as a whole. Also, consider the development of a new system in which the reuse of old, or (possibly) services from alternate vendors could decrease the time and effort for development of the entire system. The idea of extensive reusability is not a new idea, and has been utilized in many other paradigms, such as the component-based paradigm [11, 49].

So far, the SBC paradigm appears very attractive for use and design. There are currently design practices being developed to include the SBC paradigm, most notably, creating service-oriented architectures (SOA) [34], and utilizing dynamic architecture description languages [28] to formally design these systems. Like other paradigms, SBC has its share

of drawbacks, which hinder its use in large mission-critical applications in industry. The identification and resolution of these drawbacks, is what guides current research and development of the SBC paradigm. In the following we examine some of the drawbacks to SBC.

**Interoperability.** Consider a simple scenario; we have four companies independently making services that might be used by others. There is no inter-company communication on exactly what each service does or requires. Each service should be able to communicate with a variety of clients, not specifically those used by the developing company. Although there is no mandate forcing services to listen or respond to clients' requests, the utilization of a service could be severely decreased if it cannot communicate with many clients. Web services, an approach within SBC, have made strides to all but eliminate this problem, with the use of XML-compliant messages transported over standard TCP family protocols, know as the Simple Object Access Protocol (SOAP) [52].

**Security.** With such a large variety of services available, from so many different vendors, how do you know which service provider to trust? How can you as the user be assured that the service you select will actually perform only what the provider claims? Security issues like the aforementioned, severely hinder the wide spread use of SBC. In an age of hackers, trojans, and the like, users need to be reassured that what they are getting is what they want and is safe to use. Since the SBC paradigm requires the decoupling of services from where they are run (either remotely or locally), there must be some form of framework, which can help securely and reliably marshal the deployment of services.

Currently though, there is no such framework, and it is this lack of security that leaves service vendors free to create services that do anything the vendor desires, not necessarily legally or trustworthy.

**Quality of Service.** Closely tied in with security is the problem of quality of service (QoS). QoS is typically defined as the guaranteed minimum level of performance (provide what they advertise). Consider an example: you download a virus scanner that has a QoS indicating it will complete a system scan in no more than 10 minutes; with 10 minutes being the minimum length of time taken by your virus scanning service, any more and the QoS is meaningless, any less and that is a bonus.

To ensure QoS is as promised it can be gauged based on feedback obtained from clients of services [51], making the use and evaluation of services a community effort (similar to E-Bay and the like). The use of the experiences of other service users to either confirm or refute the QoS claims of a provider, would greatly increase the overall reliability and trust in the QoS statements of services. Other than the evaluation of services; QoS can also be introduced into the selection process of services. With QoS a user can request a listing of services, and apply additional constraints in the form of QoS requirements to refine their search and selection. A framework like this in which the user has absolute control over the service selection process is desired, but unwieldy. How would a selection mechanism like this be deployed over both multiple platforms and networks?

**Flexible pricing schemes.** With the ability to use all of these different services from different vendors at different times, there must be a method to pay for these services.

Traditionally, you would pay for volume licensing of monolithic software systems, which can be very expensive, now however, only fees for services actually used would be incurred. Pricing by service used, benefits companies, but in the same token benefits individual users, as they now only pay for what they *want*.

**Tool and Development Support.** The SBC paradigm will likely not be adopted as a design or development process until there is sufficient tool and developmental support. Until now we have been speaking strictly in terms of *using services*, but to be able to use services they first must be designed and created. Without supporting tools both the design and development of services is extremely difficult. A perfect example is the adoption of the object oriented paradigm; until there were languages (like Java) and other developmental tools available, it also did not find widespread acceptance.

Most developers are hesitant at best to adopt, or even try an immature design paradigm, unless there have been many tools, and other systems successfully created for it. This is an unfortunate circumstance, as it seems most developers wait for more mature releases or standardization before adoption of new paradigms. This "waiting for a better version" attitude taken by most developers is just as much a hindrance to the development of new paradigms as the problems of the paradigms themselves. If developers could, as a community, dedicate some effort to resolving problems of new design paradigms, there would be more frequent developments (a steady maturation process), and subsequent developmental tools will follow; i.e. the Eclipse Project [18].

Many of the issues of SBC, such as interoperability, platform independence, QoS and



security are tightly coupled with the client-service communication infrastructure. There have been considerable attempts made to combat these problems such as the development of Jini for Java [48] (as we will see in Section 2.4.1).

The aforementioned issues comprise the disadvantages of the SBC paradigm. More specifically though some of the disadvantages include the following.

- Dynamic system reconfiguration requires that there be some form of either infrastructure or framework available, which can work in a platform independent manner to marshal services to and from clients. Without such a system, dynamic reconfiguration utilizing remote service providers would be impossible.
- Interoperability is perhaps the largest of the disadvantages. Since services are made available to users, regardless of the platform they are using we require a mechanism to provide services in a platform independent manner. This problem is tightly coupled with the first one, in that it would likely be the associated infrastructure that would have to handle such interoperability issues. This problem could also be dealt with if there was an existing infrastructure which was external to a framework; for example a service providing framework that used the Java Runtime Environment (JRE).
- Security. In the age of hacking and exploitable security vulnerabilities, there must be a system in place to securely and safely provide services to users. Currently there is no such security system, and as such services can be provided by anyone for any purpose.

- Quality of Service, or does the service provide what it claims it does. This problem can be tied to security, in that we would require a systems for gauging if services do provide what they advertise.

We have now seen what the SBC has to offer, and what some of its drawbacks are Table 2.1, summarizes what we have, what we need and the associated problems to address what we need.

With a clearer idea of what we have and what we need for the SBC paradigm, let us now examine software architecture in Section 2.2.

## 2.2 *Software Architecture*

Software architecture (SA) according to Shaw and Garlan [47] is the description of system elements, the interactions of those elements, and patterns that guide arrangement and interaction of elements. Simply; SA is used to describe a system from a high up abstract view looking down. It is not concerned with the fine compositional details of classes, aggregation, polymorphism and the like, but instead, focuses on the decomposition of a software system into logical processing entities, their interactions [6] and their overall arrangement. This process can dramatically simplify any complex system [17], allowing for an easily understandable description of the system as a whole represented as a 'box and line' diagram [31] with notes describing system specifics.

To represent the architecture of any system is a simple process, with four main concepts; (1) components, represented as a 'boxes', (2) connections between components, represented

Requirements	What we have now	Problems
Dynamic system reconfiguration	DLL libraries, frameworks like the Java Runtime Environment (JRE)	Programs either interpreted or use frameworks to marshal dynamic interaction.
Service variation	Program plugins and web services	Plugins are specifically targeted. Web services are remote with no acquisition available.
Service reusability	Object oriented programming languages	Proper implementation.
Increased performance	Good programming practices and proper design	Design and implementation pose challenges to the inter-relation of services. Any frameworks, etc. in use to facilitate service interaction could be an impedance.
Interoperability	XML and XML-using protocols like SOAP	With many services from many vendors across many platforms, how can services be effectively used?
Security	Secure SOAP, digital signatures, trust authorities.	How can services be secured? Can we extend existing security mechanism, or do we need entirely new ones?
QoS	Not applicable	How can we ensure services provide what they advertise?
Tool/development support	Not applicable	Tool and development support is necessary for SBC adoption.

Tab. 2.1: SBC requirements, current systems and their associated problems.

as 'lines', (3) styles, or predefined arrangements that the architecture will follow (e.g. pipe and filter), and (4) a topology, or the overall arrangement of the architecture.

Architectural representations make it is easy to understand and describe component-based systems, where a simple mapping of actual system components to components represented in the architecture yields an architectural representation [10, 11]. The use of SA in component-based design is of course not its only intended use and it can be applied to the description of any system. SA is typically used when we describe systems in a top-down fashion, and initially consists of alot of prose and diagrams [47], but can also be used to represent an existing system. The simplicity of describing a system from an architectural perspective makes it appealing to many developers, juxt-opposed to this view are developers who feel that it is not descriptive enough to capture useful system information, like using UML [7] during the design phase to capture system information.

There are however, more formal aspects of architectural design which can be applied to constrain an architecture and allow for a more detailed description:

1. *Styles*. Any style can be applied to an architecture, and typically are used to constrain the types of interactions that are allowed within the system [1]. Pipe and Filter, for example, is a style of architecture that is prevalent in Unix-based systems. In this case, each component of the architecture would have an input and output, only an input, or only an output. Styles can be applied to an architecture, and an architecture could have many styles, even hybrids of other styles. In essence a style is a more formal way of describing how a component communicates with, and is oriented with respect

to, other components in the architecture.

2. *Formalizing Connections.* Interconnections between components aid in describing the behavior of components by focusing on what it takes as input and what it returns as output [3]. Moreover, behavior of components can be described through their interactions and connections. The main idea is to represent connections with a type of formal logic while treating components as black boxes, and through this process, realize how the components involved in these connections behave externally. Formalizing connections also allows developers to apply more stringent constraints on connections, which then behave as input or output constraints on the components involved in the connection.

3. *Architecture Description Languages.* Architecture description languages (ADL) are specialized languages used to formally describe system architectures. ADLs are based on formal languages that constrain the arrangement of an architecture (aspects of components, connections, style, topology, etc), via the definition of the language itself. For example, if we have a language that states all connections can only have two components participating, then we would not be able to represent, for example, two clients sharing the same connection to one server.

There are many different ADLs, each focusing on a different area of system design. What they all share is a mechanism for formally capturing and specifying design semantics of a system architecture. More detail is provided in the following section on ADLs, Section 2.3.

When describing the architecture of a system, an architecture can represent a component of another architecture and so on, recursively applying architectures to other architectural components, to achieve a finer description of the system and its components. For example, Figure 2.1 shows that an individual component of a system can be decomposed yielding a corresponding architecture with its own components, connections and connection constraints, just as its parent one.

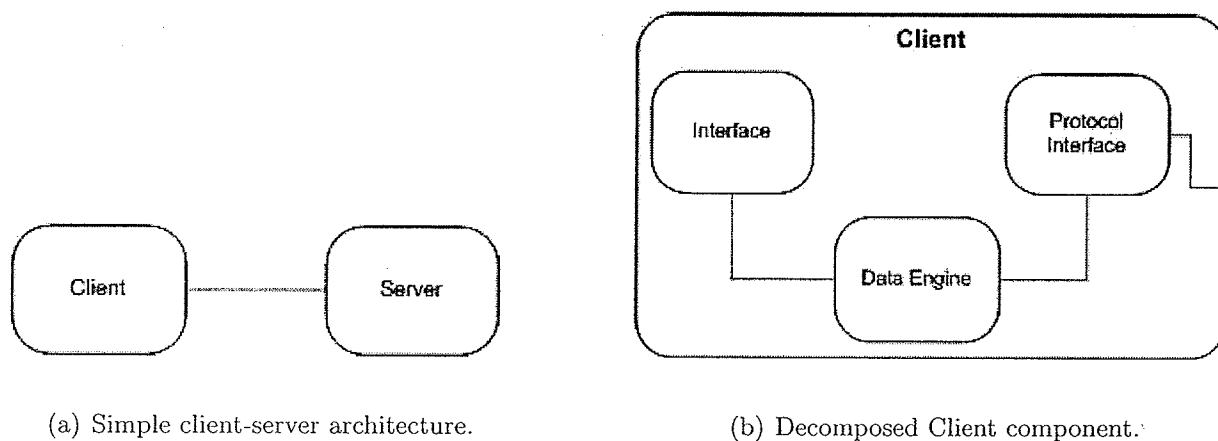


Fig. 2.1: Simple architecture and component decomposition.

This approach to modeling systems is not new by any means, and is simple to both understand and interpret at a glance. Simply, to further describe any component within an architecture, break it down into its own architectural description. Of course the description of architectures and their components is more complicated than simply drawing boxes and lines, causing some researchers to examine the feasibility of using existing design tools to effectively describe architectures [16, 45].

In Section 2.3, we examine existing ADLs that explicitly address system dynamism and what (if any) aspects of these ADLs could be considered as a starting point for my ADL.

## 2.3 *Architecture Description Languages*

Architecture-based design are increasingly recognized as a major phase in the design of large software systems [6, 47]. As part of this effort, a number of ADLs have been proposed [12, 32] which provide a more formal approach to describing system architectures. While most efforts have been focused on static architectures and related problems, some researchers have also discussed the more interesting (and also more difficult) problems of dynamic architecture specification and reconfiguration [25, 28, 30, 41]. Dynamism aside, the guiding premise of ADLs in general is to provide a formal manner to describe the abstract structure of a software system using components and their connections. While there are currently many ADLs available, they lack consistency of how they describe systems. It would seem that while each ADL follows simple requirements for any ADL, such as descriptions with components and connectors, each ADL has a different focus, like dynamism or formal representation of an architecture using a certain logic. An example of this is Weaves [14, 50], which focuses on existing system artifact integration. For the scope of this thesis, though, we need only consider the ADLs that have in some way addressed the issue of system dynamism, and dynamic system reconfiguration.

Let us now discuss ADLs and features of those ADLs that support dynamic composition and reconfiguration of components, services and systems.

### 2.3.1 C2SADL

C2SADL [30] (referred to as C2 for brevity) is an ADL that fully supports dynamic unconstrained system reconfiguration, in that, a system in C2 can grow and shrink, without any prior knowledge of dynamic changes. In C2, both components and connectors are considered first-class entities, with message passing through connectors used to link components to one another, subject to certain restrictions. Furthermore, connectors may perform optional filtering of messages from one component to another. Unlike other ADLs, C2 supports many object oriented (OO) concepts for the creation of new components, such as typing, sub-typing, generalization/specialization and type conformance. The inclusion of these OO concepts allows for a large variety of system configurations with typed components, sub-components, parent components, etc.

Each component described in C2 has two inputs (incoming requests/notifications) and two outputs (outgoing requests/notifications), which use the connections between components to pass messages to one another. It is through this message passing mechanism that C2 allows for dynamic system reconfiguration.

In the following example in Figure 2.2, we see a representation of an architecture description in C2. Architectures in C2 are connected to message buses (that are simply connectors), which facilitate components communicating to one-another.

C2 allows dynamic binding (referred to as ‘welding’) of components to connectors, as well as unbinding (unwelding). All possibilities for reconfiguring or rewiring of an architecture are allowed:



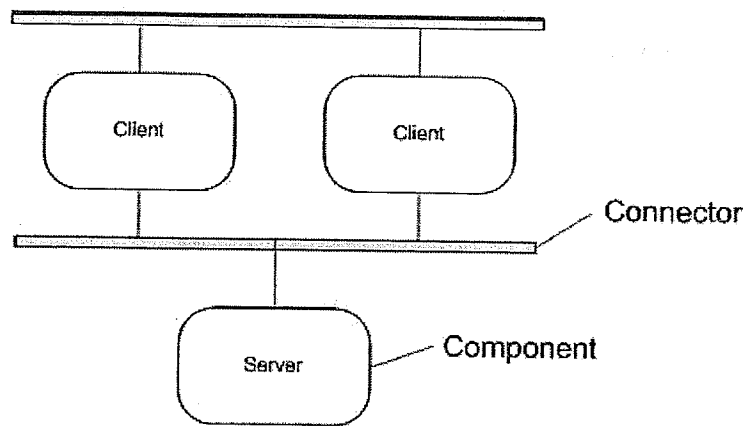


Fig. 2.2: Simple C2 architecture.

- a component can be unwelded from the architecture
- it can be rewelded in a different place
- it can be left unwelded and persist within the system
- or be removed from the system

Dynamic reconfiguration is accomplished using message passing between components, i.e., in the same way as the ordinary communication between them. This communication takes place through the connections between the components and the resident infrastructure; the Architecture Construction Notation (ACN) [30, 21]. With the ACN and the associated infrastructure, C2 provides an API interface to help marshal the welding/unwelding process between components. With the use of the ACN, C2 also supports the upgrading of existing components through sub-typing, modifying the subtype, adding the subtype to the architecture, and then removing the original component from it.

Despite its advanced capabilities and the associated API, a number of issues have not

been resolved in C2, mostly in relationship with unwelding. Some of the more prominent problems associated with unwelding are:

- how to unweld components that are currently in use?
- what happens to a dependent component when their parent is removed?

The language constructs for C2, unlike some more formal languages, are fairly simplistic and easily understandable. In the case of the C2 Architecture Description Notation (ADN) and the ACN for communication, both are represented in the same manner, conformant to Backus-Naur Form. The following examples show the description of a software architecture in the C2 ADN [22] and the weld/unweld description in the ACN [21].

```
system ::=  
  
    system system_name is  
  
        architecture architecture_name with  
  
            component_instance_list  
  
    end system_name;
```

The above example of a description in C2 is very straightforward, with a system encapsulating an architecture, which in turn holds components of the system. The next example demonstrates that even the description of a welding action is simple. The weld operator is called with a pair of architectures to be operated on, which are termed the 'welded pair' in C2 parlance.

```

architecture_welding ::=
    architecture_name.welding_operator
    welding_pair;

```

While the dynamism of C2 certainly fits well within the requirements of service-based architectures, the need for an external facility such as the ACN, shows that to achieve full dynamism would require some form of middleware or framework to broker service interactions. For our purposes, we consider middleware like that used in Web Services, which has been expanded in [33] to facilitate brokering interactions between services. By using middleware to act as broker for component interactions, we can further achieve system dynamism, by removing the need for either one of the end hosts to explicitly handle any processing related to system reconfiguration.

### 2.3.2 DAOP-ADL

DAOP-ADL [44] is an XML based ADL, that is oriented towards aspect- and component-based architecture descriptions. An architecture described in the Dynamic Aspect-Oriented Platform (DAOP) has three main elements; (1) components, (2) aspects and, (3) plug compatibility rules between (1) and (2).

DAOP-ADL was designed specifically to be used in the DAOP platform, as such, an architecture created using this ADL can be loaded into DAOP to determine dynamic connections.

In DAOP-ADL, aspects and components are defined separately with the use a public

interfaces. Each element described has at least two public interfaces; a provided interface, which describes what the output of the component is and a required interface, which describes what kind of input the component can receive.

Unlike the description of components, aspects do not have a provided interface, instead they have an evaluated interface. The evaluated interface describes which messages of the component the aspect is part of *can be evaluated*. The required interface for aspects describes any output messages and can also describe output events. Since aspects are coupled with components, they can also capture the events thrown from components with a target events interface, which describes which events can be captured.

There are three other main categories of information used for the description of components and aspects other than interface definitions [44]:

- Implementation classes. These classes must implement the provided interface of candidate components to maintain interoperability
- Properties. Input and output properties are used to allow components and aspects to communicate. Properties are in a {name, signature} format, allowing for simple translation. Furthermore properties can have one of two accessibility options: (1) usersite; only available to components/aspects of the same instance scope, and, (2) serversite; available to all components/aspects of the same distributed application.
- Dependencies. Simply, this is a listing of names from the required interface.

Thus far, we have discussed the first two parts of DAOP-ADL: interfaces of components and aspects. What we need now is to know how connections are handled. The ability of

components and aspects to connect, is handled by describing composition rules, which use elements from the component and aspect interfaces to determine communication suitability. This is done for one express purpose; to allow dynamic system reconfiguration, without the need to recompile altered interfaces. More specifically if composition rules were not used, every time reconfiguration took place a new interface would be created and then compiled, prior to the reconfiguration completing.

The dynamism of DAOP-ADL and the use of XML and XML schemas [54], are definitely a step in the right direction. The use of XML especially allows for easy data interchange, and removes the need for a separate parser and compiler for their ADL, as any one of the freely available XML parsers will handle this task.

### 2.3.3 Darwin

Darwin [28] supports components and connectors, using sub-classing to build more specific components from generic ones. Each component is described in terms of services it provides and services it requires. Darwin also allows composite components obtained from instances of simpler ones. This design emulates the object oriented style of design, with the inclusion of sub-classing and aggregation.

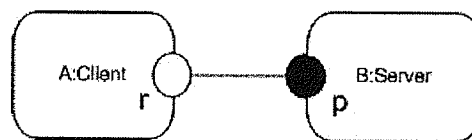


Fig. 2.3: Simple Darwin architecture.

Individual services are specified using the  $\pi$ -calculus [38, 39, 40], while the configuration

is specified by binding services requested by one component with services provided by another one. A binding can occur only if the type, what is actually provided, by a service matches the type, what is required, of the requesting service. Thus, the specification of the structure is decoupled from the specification of individual components and their services, as required by the principle of separation of concerns.

The client and server shown in Figure 2.3 could be described in  $\pi$ -calculus as follows:

$$Client(r)^{def} = (\forall o)(REQ(r, o) \mid Client'(o))$$

$$Server(p)^{def} = (\forall s)(PROV(p, s) \mid Server'(s))$$

$$System^{def} \equiv (\forall a_r, b_p)(Client(a_r) \mid Server(b_p) \mid BIND(a_r, b_p))$$

Instantiation and binding of services can occur either a priori (code statically added before run time) or dynamically at run time. In the latter case, two options are available. Lazy instantiation, is the process where a component is not instantiated until another component wants to use its services. However, the types of participating components and the binding(s) between them must be specified beforehand. In this manner, a system can evolve at run time, but only in a predictable manner which is fixed at design time, also known as constrained dynamism. However, Darwin constrains dynamism by not allowing cycles in the system, in which components directly or indirectly reference themselves.

The premise of lazy instantiation is that there is a place holder for the lazily instantiated component, but the component is not actually loaded until it's services are required. Figure 2.4 is an example of lazy instantiation. We can see in this example that *Client* is

the resident component, and that *Server* is only (at this stage) a placeholder available to *Client*. Once required, the services of *Server* will be loaded and become available to *Client*.

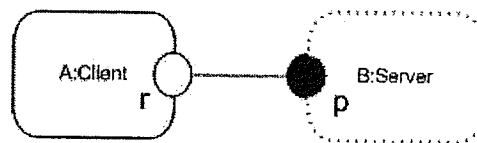


Fig. 2.4: Lazy instantiation.

The other option, for system dynamism, is known as direct dynamic instantiation, which allows arbitrary structures to evolve at run time. However, if the dynamically instantiated components are to interact with each other directly, they must exchange the relevant information (i.e. service references) through a third party, which is not part of the language per se. To support these types of dynamic architecture changes, an infrastructure is needed to broker these changes and handle components and their interactions throughout those changes.

The following example in Figure 2.5 shows two resident components, *Client* and *Service Provider*. In this case, depending on the service that *Client* requires, a new type of server might have to be instantiated to access the specific service provider. Simply, in the aforementioned instance, a new *Server* can be created for any given *Service Provider*.

In either of the aforementioned cases, bindings are permanent and cannot be undone. The goal of Darwin designers was to keep the notation declarative, and the introduction of an unbind operator would violate that requirement [24]. In other words, architectures specified in Darwin may grow (subject to certain restrictions), but they cannot shrink. Furthermore, the communication between system elements is subject to restrictions, and

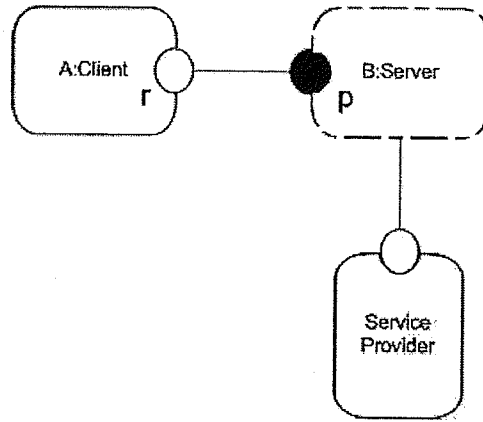


Fig. 2.5: Direct dynamic instantiation.

the help of an external agent may be required to achieve the full potential of an architecture specified in Darwin. Darwin integrates architecture dynamism very well, but, in terms of service-based dynamism, Darwin lacks (1) an unbind operator and (2) unconstrained dynamism, or the ability to dynamically change with no restrictions (such as the no cycle constraint).

#### 2.3.4 $\pi$ -ADL

$\pi$ -ADL [39] is an ADL specifically designed to describe dynamic and mobile architectures. Unlike most ADLs  $\pi$ -ADL is focused more on formally describing architectures, and not so much on the structural aspect.

Aside from formal descriptions,  $\pi$ -ADL is intended to capture the runtime aspect of architectures. The description of any given architecture in this ADL is similar to many other ADLs, in that it models components and connectors, with components containing ports that define connections. More specifically (1) components are considered computational



entities, which contain ports available for communication, (2) ports are connection points between other components and/or their environment, (3) connections provide a mechanism for components to communicate with one another and, (4) connectors, which are special components that marshal the connections between normal components.

Figure 2.6 is an example of a basic architecture description in  $\pi$ -ADL. This example follows the client server example used thus far, notice the special Connector component required for connections between normal components. The Connector component is *required* for any communication to take place between other normal components.

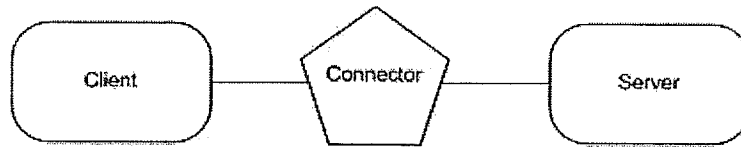


Fig. 2.6: Simple  $\pi$ -ADL architecture.

Components are connected on matching ports utilizing a value passing [39] mechanism, where all information passed between components and connectors are considered 'values'. Once a connection is made a wide range of data can be passed through it, from component communication data to architectural specifications. Furthermore, this notion of components and connectors is expanded to include composite creation of new elements from other component/connector combinations. Simply, a component can be comprised of other component/connector combinations, with the ports of the specialized component acting as gateways to the internal components. In conjunction with compositionality, ports can also be declared as restricted, limiting access to internal composite components. The premise of compositionality, is not constrained to only component specifications, but in fact, whole

architectures in  $\pi$ -ADL can be composed of other architectures.

$\pi$ -ADL is described as a layered system with the following layers [39]:

- The base layer, where behavior and connections are described. Since behaviors are described by corresponding connections, and subject to type restrictions, this layer allows the component connections of an architecture to be described as well as the description of abstractions.
- The first order layer, which is a refinement of the base layer. Typing options, such as data typing, are available here, and connection mobility is introduced.
- The Higher-ordered layer, where first class citizenship of elements is introduced, as well as behavior mobility.

Although  $\pi$ -ADL is designed to describe both static and dynamic architectures, we only need consider dynamic architectures and how they are represented within  $\pi$ -ADL. To describe a dynamic architecture,  $\pi$ -ADL uses the concept of an abstraction (from the base layer), which is used to describe any element within the ADL. An abstraction is analogous to a class in Java, and can be instantiated ('created' in  $\pi$ -ADL parlance) at any time. Once an abstraction has been created, it must be composed with a connection so that it can communicate with the rest of the architecture.

The following is an example of the client abstraction shown in Figure 2.6

```
component Client is abstraction(x:Natural) {  
    port is (connection is OutPort is out(Natural))
```

```

behavior is {
    ....
}
}

```

Coupled with the formalism of  $\pi$ -calculus and the ability to describe dynamic architectures,  $\pi$ -ADL is an ideal starting point in the creation of our ADL. While  $\pi$ -ADL supports compositionality of both components and architectures, it lacks the ability to include non-resident components and architectures. Non-resident components and architectures are those which physically reside in an alternate architecture. This is powerful idea, which gives us the ability to create new components or architectures from almost any existing component or architecture.

### 2.3.5 *Rapide*

Rapide [27] is an event-based concurrent language used to define and simulate system architectures. An architecture in Rapide is an executable specification, and the system may contain several such architectures at different levels of abstraction. An architecture is made up of a class of systems, which in turn are composed of interfaces, connections and constraints. The interfaces are simply modules or components of the system, which describe the operations made available by a module and what the module requires from other modules. Furthermore, an interface can describe the abstract behavior of modules using reactive rules [26], which define how the component will react to certain information being

received. Connections are used to describe the interactions between interfaces. They have the ability to describe either synchronous or asynchronous data communications between interfaces. Finally, to ensure correctness, constraints are described as specific restrictions placed on various aspects of connections and interfaces.

During the development of an architecture in Rapide, we have the ability to gradually instantiate modules into the system; in other words, the system can evolve dynamically at runtime. When a module is assigned to an interface in this manner, the interaction mechanism of the interface becomes the constraint for the module. Rapide also allows us to assign a reference architecture to an interface, rather than just modules. Architectures can thus be embedded, which greatly expands the composability of architectures in Rapide.

To handle relationships between (reference) architectures, Rapide employs a system of event pattern mappings [26] to describe the relationship. These mappings describe how executions of an instance architecture are mapped to that of a reference architecture. Furthermore, as Rapide architectures utilize the partially ordered event set (POSET) execution model, module constraints can be checked as messages are passed between modules, and before messages are sent from modules.

While the composability, architecture execution and the dynamic loading of modules in any order exemplify service-based concepts, the requirement of interfaces a priori for any given module detracts from its overall suitability in the service-based computing paradigm.

### 2.3.6 Weaves

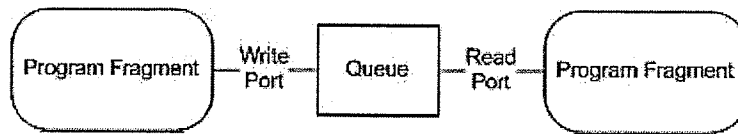
Weaves [14, 50], while not specifically an ADL, is a framework that allows for dynamic system evolution at run time. Weaves accomplishes this task by taking source code (of any kind) and wrapping it into intra-process modules. It then uses these modules to weave together applications or 'tapestries' from dynamically executing code. In this manner, any legacy or future code can be used to dynamically create applications.

The actual structural components of the framework are simple, and are broken down into five categories [50]:

1. Modules, which are the object code file(s). Each module also has (1) a data context, which is the state of the module within the scope of the other object files in the module. (2) a code context, or the code in the files. Each context can also have multiple entry and exit points.
2. Bead, or an instance of a given module. There can be many beads of a given module within a tapestry. Each bead has its own data context, but can share it's code context with other beads.
3. Weaves, or collections of data contexts that belong to beads of different modules. With the collections data contexts, multiple namespaces can be created within one address space, which is the foundation of the ability for dynamic reconfiguration.
4. String, which is a thread of execution. A string though, can only operate in a single weave, although multiple strings can execute simultaneously.

5. Tapestry, which is a set of weaves that describes the created system.

Figure 2.7 is an example of a simple tapestry, which contains four intra-process module instances or 'beads' in Weaves parlance. As mentioned the encapsulated code can be any kind.



*Fig. 2.7: Simple Weave [14].*

While any code can be wrapped in modules, it is important to note that the code is unchanged, and that no special code must be inserted to allow for the framework. The treatment of code in this manner allows Weaves to be as generic as possible, not requiring any particular coding or language paradigm.

Once the modules have been created within the framework, they then share attributes of both the traditional process and threading model of computing; (1) like processes, the modules allow for state separation, (2) like threads, the modules allow for code sharing and fast context switching.

From the generality of Weaves and the treatment of wrapped code, two advantages are immediately evident: we can compose and recompose legacy applications without modifying them, and runtime dynamic system reconfiguration is achieved through the use of check-pointing, dynamic code insertion and recovery. Figure 2.8 is an example of dynamic code insertion or 'splicing' in Weaves parlance.

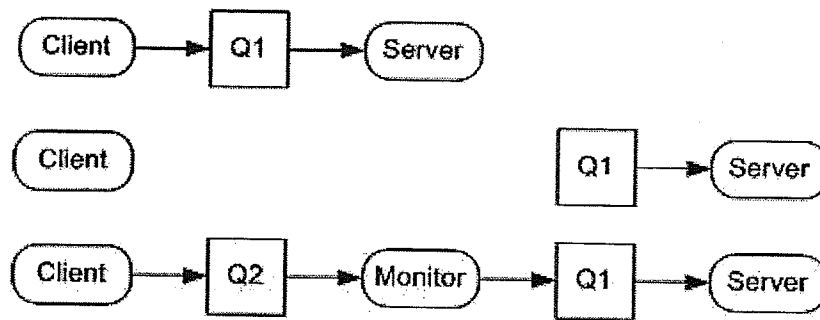


Fig. 2.8: Dynamic code insertion or 'Splicing' [14].

Weaves handles composition and recomposition of systems very efficiently, allowing for a great deal of system flexibility and dynamism. The only drawback to these features is that the code to wrap the components must be manually processed to utilize the framework.

### 2.3.7 Wright

Wright [47] was designed in an attempt to apply a more direct specification and analysis of architecture designs. This is accomplished by direct specification of interaction relationships among system components as protocols. Where protocols characterize the nature of interaction among components.

An architecture in Wright contains a description of components, connectors and the system. Each component is described using a collection of ports, which in turn describe how the ports are used computationally. Each component includes input and output ports, sufficient to describe it's interaction within the system to which it belongs. Not only do the ports describe what the component can provide to the system, but they also contain the description of what is required by the component from the system. Simply, ports provide a complete description of the requirements (both input and output) of a component.

Connectors are composed of a set of roles and a glue specification [3]. Roles describe the behavior of the components involved in the connection, while the glue specifies how the roles interact among themselves. This is done to ensure port-role compatibility when using the connector. There is no limit to the number of roles that any one connector can take on, with each role requiring a glue specification to define the interaction of the role(s). Glue specifications can in fact describe the interactions of many roles at once, and are not constrained to describe the interactions of only the two roles associated with a specific connection.

Lastly, the system itself is described by using a collection of instances and attachments. The list of instances represents instances of both components and connectors within the system specification. The attachments however, provide the topology of the system architecture, describing how the instances are laid out physically within the system. The following example shows what a specification for a system could look like in the Wright language.

System ClientServer

    Component Client

        port in [protocol]

        port out [protocol]

        comp spec [specification]

    Component Server

    ....



```

Connector LAN [specification]

....

Instances

....

Attachments

....

end ClientServer

```

While the specifications of systems are simple, they do not allow for any form a dynamism or system reconfiguration at runtime. Specifications in Wright are simply that of static architecture descriptions represented as topologies of the 'box and line' style. Even though Wright is not dynamic, it still does provide a good example of a well-structured formal static language.

### 2.3.8 *xADL*

While *xADL* [23] is not strictly a stand-alone ADL in its own right, it is however, an ADL interchange language, which can be used to wrap other ADL specifications. *xADL* works by having a predefined set of XML tags and links between them using the XPointer [53] mechanism. With predefined types it can decompose alternate architectures into the *xADL* style. The underpinning for this effort was to enable non-specific tool integration into ArchStudio [20], an IDE for the creation of C2 style system architectures. Although *xADL* was created to work with ArchStudio, it is easily expanded to include specific notations

and semantics from any ADL, as it has currently been expanded to suit C2, producing the specification xC2 [23].

To maintain ease and usability, there are only five element tags available; `<Component>`, `<Connector>` `<Architecture>`, `<ComponentType>` and `<ConnectorType>`. As an abstract base, these five tags are sufficient to describe any architecture with an architecture made up of components and connectors (in essence). For any given specific ADL though, these tags are likely not sufficient, which is why XML was used, to allow for customizable expansion of these tags to suit specific applications.

By default, each of the elements described above has links associated with it, in the form of XPointer definitions. Each of the aforementioned elements has one or more XPointers; Architectures have a Links pointer, Components and Connectors both have Supports pointers and, ComponentTypes and ConnectorTypes both have Interface pointers, which in turn have Parameter pointers.

The pointers from each element are used to link together differing tags within the specification. Below is a listing of the XPointer types for each element, and what they link together, taken from [23]:

- *Architecture*  $\rightarrow$  *Links* is a logical link between a component and a connector respectively
- *Component*  $\rightarrow$  *Supports* is the specification of acceptable names and types to the component instance
- *ComponentType*  $\rightarrow$  *Interface* is the specification of both Name and Method inter-

faces for this specific component type

- *ComponentType*  $\rightarrow$  *Interface*  $\rightarrow$  *Parameter* is the specification of input and output parameters for the specified component interfaces
- *Connector*  $\rightarrow$  *Supports* is the specification of the names and type(s) acceptable to this connector instance
- *ConnectorType*  $\rightarrow$  *Interface* is the specification of both Name and Method interfaces for this specific connector type
- *ConnectorType*  $\rightarrow$  *Interface*  $\rightarrow$  *Parameter* is the specification of input and output parameters for the specified connector interfaces

xADL goes a long way to making ADLs interchangeable, in acting as a language and wrapping utility. Even though any other ADL can be described in xADL and xADL can be extended, custom error and well-formedness checking must be done outside of xADL, as it is only validated against its own DTD *as is*.

As an ADL and (more so) an ADL extension, xADL suits the service-based paradigm perfectly, due to the fact that it is created and maintained as XML, removing it from specific processing, and allowing it to be integrated into online transactions using SOAP, EJB or the like.

The following section describes different forms of tool and infrastructure support both used and available to aid in achieving full dynamism when considering runtime system reconfiguration.

## 2.4 Infrastructure

All of the ADLs reviewed rely on external support to achieve true dynamism. Darwin needs it to support communications of dynamically instantiated components; C2 needs it to provide the requisite tasks related to welding and unwelding of components on demand; Rapide needs it to be able to execute its architectural specifications; and Weaves needs it to provide the operating environment in which individual modules can run. Therefore, it seems safe to conclude that true dynamism in software architecture requires the support of the proper runtime support infrastructure. In some scenarios, runtime support may be built in the generated code, not unlike Java bytecode – which requires the Java runtime environment to execute on a remote platform. In others, a separate runtime infrastructure is needed, or the necessary infrastructure may already be available on the execution platform. Middleware such as CORBA, RMI, or more recently Jini, offer significant potential for facilitating the runtime support for service-based systems, although many problems, especially in terms of interoperability and portability, remain to be solved [29].

There are many examples of infrastructures available that facilitate system dynamism, such as Common Object Request Broker Architecture (CORBA) and Java Remote Method Invocation (RMI).

CORBA [37] is an infrastructure that facilitates remote method invocation. Through the use of remote interfaces, CORBA can broker remote method requests from clients that have copies of corresponding remote interfaces, and stub objects for services. The interfaces and stubs are required by potential clients in order to know what methods can

be utilized, what they take as arguments, and what they return. In CORBA all method invocation takes place at the remote location, and there is no option to acquire the object for local invocation. RMI works much like CORBA, in that it uses interfaces to describe the capabilities (in this case methods) of a remote object to be accessed. Unlike CORBA though, RMI has the capability to transport objects from one location to another for invocation, through a process called serialization. Serialization is simply the wrapping of an object, in this case an interface, and sending it to a client.

Both infrastructures (CORBA and RMI) provide a solid foundation for creating dynamic systems, with one major flaw: the interfaces must be made available to a client at compile time, and if changes are made to the service being provided, new stubs must be created and given to the client before communication can take place. This does not facilitate the dynamic selection of services as desired by the SBC paradigm, as you would first need to download and examine all of the available interfaces to determine suitability for your purposes.

There has however been much work in this area to alleviate this interface dependence when using middleware like CORBA and RMI. Two such projects Jini for Java [48] and the Open Services Gateway Initiative (OSGI) [43], both provide infrastructures that allow services to be searched for and selected on the basis of criteria which are either explicitly specified by the developer or derived from the implementation of the service itself.

For this thesis we take specific note of Jini, which is fully described in Section 2.4.1. Services in Jini can be searched for, loaded dynamically, or downloaded to a client and

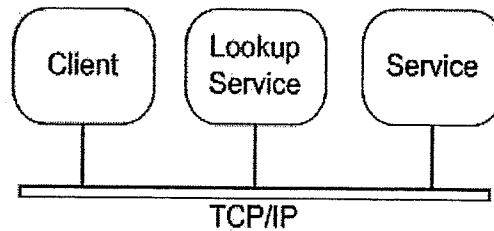


Fig. 2.9: Components provide services.

used locally all from within Jini itself. It provides all of the tools and tool support required to create, deploy and use services; such as a locator service, leasing service, and object transportation service. Furthermore, as Jini is implemented in Java, all that is required to use Jini is a Java Runtime Environment (JRE) and the Jini libraries.

#### 2.4.1 Jini

So far we have expressed the need for frameworks or infrastructures to facilitate the marshaling of service-components to and from clients, and Jini [48] provides just such a framework. Jini is a networking technology that allows devices, services and cooperative programs to interoperate seamlessly over standard network technologies like TCP/IP. However, there is no distinction drawn between different resources connected to a Jini network; everything that comprises the network is considered a service. Simply, Jini provides a method for a variety of services to exist, co-exist, discover and interact with each other on a network in a platform independent manner [35, 36]. Jini uses the Java Virtual Machine (JVM) on client machines to provide platform independence, and a simple lookup service for clients to locate services. A simple example of a Jini network is shown in Figure 2.9.

Everything in Jini, whether it is a part of Jini or an external service, all connects to

the network. Through the network Jini brings together clients and services through its lookup service (an in-depth description of this process is available in Appendix B). The mechanism within Jini that facilitates the lookup and selection process is called the lookup discovery manager. The lookup discovery manager allows criteria to be added to the service search process in the form of *Entries*. Entries are a serializable representation of attributes of a service, which are passed to the lookup manager at the time service searching takes place. The addition or removal of Entries can be used to either refine or broaden the search scope for a service. Jini provides a default set of entries intended to be the most commonly used when searching for services; such as Vendor, Version, Name, ServiceInfo, ServiceType, Location, Status and Comment [36]. Each of the aforementioned entries are simple classes that perform no operations and only store service attribute information.

Services in Jini are built upon the Java RMI API and allow services to be run remotely (much like web services) or marshaled to the client for local execution. Furthermore, Jini allows descriptive metadata to be added to services, which enhances searching for and finding desired services. For a better explanation, let's consider our word processing example. Imagine that we need a specific spell checker for our word processor, to find one, we would simply connect to a Jini lookup service and browse for one based on whatever criteria we wanted. Of course there are restrictions, such as, our word processor would have to be able to connect to a Jini lookup service, and it would have to be able to execute services within the JVM. Once a service is found, a spell checker for example, it can be sent to the client to execute locally, use resources from external JVMs or even use whole

other services (either remotely or locally). Jini does this by extending Java RMI and using its ability to marshal objects from one JVM to another through a process of serialization.

To provide its functionality and ability, Jini contains a comprehensive set of tools and services for its framework. Some of these tools include:

- A web server which is used to host objects on the network.
- *Reggie* a lookup service, which provides the ability to look up and search for services.
- *Mahalo* a transaction manager, that allows multiple operations to be treated as a single atomic operation.
- *Fiddler* a lookup discovery service, which will perform discovery operations on behalf of clients.
- *Mercury* an event mailbox, which will hold events for clients that are currently off-line until they come online again.

The interaction capabilities that Jini provides suit the service-based paradigm extremely well, as we can have services on a network which can be searched, selected, run locally or remotely, and have internal or external dependencies that may be local or remote. A drawback to Jini is the fact that it depends on the Java JVM, so if one is not present Jini cannot work. Also the environment and tool support necessary to run Jini in an effective manner are extremely complicated to run and maintain. There is an alternative that provides similar support to that of Jini, the Open Systems Gateway Initiative (OSGi) [43], with emphasis placed on mobile devices rather than an infrastructure to provide services.



### 2.4.2 OSGi

An alternative to Jini for providing a framework for dynamic services is the Open Systems Gateway Initiative (OSGi) [43]. The OSGi framework provides a component-based execution environment for networked components. This is similar to Jini, except that OSGi places heavy emphasis on components and component-based design instead of considering all networked entities as services.

The framework itself is designed in a layered fashion with four layers; the Execution Environment layer which defines the JRE environment for OSGi, the Modules layer which define class loading policies, Life cycle Management layer that creates bundles for dynamic usage and the Service Registry layer which provides dynamic interaction points between bundles. Where bundles are simply a collection of Java classes that are physically grouped together to provide some service.

The OSGi provides standard services which are simply Java interfaces and which reside in the service registry. These services are *not* self contained existing services that can be run in a standalone fashion unlike the provided services for Jini. To make use of any one of the standard services from OSGi a bundle must implement the service interface so that it may then be searched for by a client. There are many standard services available for implementation as of OSGi Release 3, some of which include [43]: Framework Services used for administration privileges, System Services that provide system independent functions, Protocol Services that map external protocols to OSGi, and Miscellaneous Services which provide a variety of alternative services. Most notably of the standard services is the

Protocol Service, which allows Jini services to be recognized and used by OSGi.

OSGi provides an extensive framework for dealing with system dynamics. Since we consider all components as *only* vehicles to provide services OSGi is not considered in this thesis as it is strongly component-centric in both services creation and provision.

## 2.5 Tool Support

The problem with most ADLs is that each have different focus in terms of architectural description. For example Darwin focuses on the formal description of systems with its use of  $\pi$ -calculus, whereas Rapide focuses on being able to execute and examine system descriptions. Unfortunately, most of those ADLs are not interoperable with one another. In essence, if you choose an ADL to aid your design you (typically) do not have the option of modifying it in another ADL. To do so would require the complete redesign of the architecture to suit the new ADL's format.

When working with architectures in ADLs, typically you would like to perform other tasks than strictly specification and validation, such as: generating source code, executing generated code and converting to other ADL formats. This requires that an ADL has automated tool support, which is why tool support is considered to be an essential part of any ADL [32]. Of course, different focus requires a different set of tools. Leaving it up to the creator of the ADL to decide what if any automated tool support is available.

### 2.5.1 ACME

ACME [13], is an ADL interchange language that proposes to bring a level of interoperability to ADLs. ACME is not designed to make the designs of one ADL work within another, but instead it promises to interchange common architectural knowledge between ADLs, while being tolerant of ADL specific knowledge. Simply, it takes common architectural knowledge, like component and connector information and makes it available in a generalized format for other ADLs (and their toolsets) to recognize, while not considering any ADL specific information about an architecture. The reasoning behind the creation of ACME consisted of five main goals [13]:

1. Provide an ADL interchange format - a mechanism to interchange information that can be understood by more than one ADL.
2. Provide architectural representation and analysis - allow architectures to be represented in ACME and then provide some level of analysis of those architectures.
3. Provide a foundation for new ADL development - ACME has a simple ontology which describes common elements of an architecture description, therefore any ADL *should* have these in their language.
4. Work towards standards for architectural representation - to try and standardize what basic elements should be included with any architectural description from any ADL.
5. Provide architects with meaningful expressive descriptions - make architectural descriptions and their analysis meaningful, human readable and easy to understand.

The ontology of ACME mentioned above, has seven basic elements: components, connectors, systems (combinations of components and connectors), roles, ports, representations and rep-maps. Components and connectors both have a set of interfaces that describe them. The component interface describes ports (which describe input and output), and the connector interface describes the roles of the connector; where roles are how the components involved interact with the connector. Lastly ACME has representations, which are simply graphs describing the topology of an architecture, and rep-maps, which describe how interfaces are interrelated.

With its ontology ACME can wrap any other architecture description, leaving out any ADL specific information. However ACME does provide a mechanism to also allow ADL specific information to be included. This mechanism is called ACME Properties, and it allows ADL specific information to be included as additional information. Consider the following example where a simple client component is described in ACME, but which contains other ADL information as properties.

```
Component client = {  
    port send;  
  
    Properties{C2-style : style-id = client-server  
        source-code : external = "client.cpp"  
    }  
  
    ...  
}
```

The inclusion of specific information in this manner allows C2 and other C2 aware ADLs (in this example) access to this information, but yet still allows non-C2 aware ADLs to read the structural information and ignore these properties. There are other uses for properties than simply storing ADL specific information, since *anything* can be placed in a properties declaration. For example you could include actual source code snippets, comments, etc..

To work with and create descriptions in ACME, there is a freely available tool called AcmeStudio [46], which is built upon the Graphical Editor Framework (GEF) [19] of the Eclipse platform [18] to provide simple component creation and manipulation, as well as rich model editing. As mentioned proper tool support for ADLs is ideal, and the creation of AcmeStudio facilitates the correct usage of ACME, especially as it handles the interchange of information from one ADL to another.

### 2.5.2 ArchJava

ArchJava [2], provides another form of strongly desired tool support – code generation. Other than infrastructure and interoperability, automated design and development tools [32] such as code generation only enhance the capabilities of ADLs to be more descriptive. In particular, code generation tools are needed to convert the architecture from a design to an executable form. In doing so, architectures can be built and then physically run to evaluate the design. ArchJava in particular, uses standard executable Java code, to describe and evaluate designs, with specific focus on maintaining communication integrity. The use of Java code, then allows architectures created within ArchJava to be run on any machine that has a JRE.

With the benefits of Jini serving as a framework to marshal system dynamism, in Chapter 5 we consider the use of Jini as our framework, and extend the idea of Java code generation for our ADL.

In the following chapter we discuss service-based computing contrasted with traditional component-based computing, followed by conceptual requirements for the SBC paradigm.

### 3. SERVICE-BASED COMPUTING

Service-based computing is the idea that software systems are made of interacting services, as opposed to components. Traditional systems are composed of local components with (typically) no external requirements, whereas service-components of service-based systems are composed of services that can be either local or remote with any number of external requirements. Furthermore, not all of the services are known a priori, as services can be dynamically selected, loaded and unloaded.

This is a shift in the thinking behind computing, as we move from large self contained software systems, to those which are the sum of a collection of services. In the following discussion, we examine how service-based systems are different than their traditional component-based counterparts.

#### *3.1 Component- vs. Service-based computing*

Software systems currently can be built according to the component-based design philosophy, where components are first-class entities from which systems are built. There are many benefits to component-based systems, like reusability and reduced refactoring [49]. What component-based systems lack though, is the ability to change, in that, to upgrade

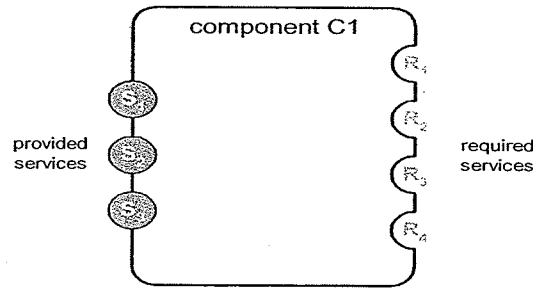


Fig. 3.1: Components provide services.

or improve such a system typically requires patches to be applied or whole new versions to be acquired. The reason for this is that within a component-based system a component provides or implements some services, which in turn may require input from another component, as shown in Figure 3.1. Components can also require no input or have no output, and can themselves be composed of other components interacting through appropriate inputs and outputs.

Whole systems can be made from one or more components that are interacting (or connected). When a user starts such a system, be they human users or external applications, one or more components of that system are utilized; those components in turn utilize others, and so on. In this paradigm, components are the essential units for both packaging and execution, while the services they provide or require are basically inputs and outputs (plugs and outlets), that serve to connect those components into a working system, as shown in Figure 3.2.

It should be noted that the interconnections of components is not dynamic, meaning that for one component to utilize another (or several) it must be aware of the type and structure of the input it will receive, requiring prior knowledge of the component when



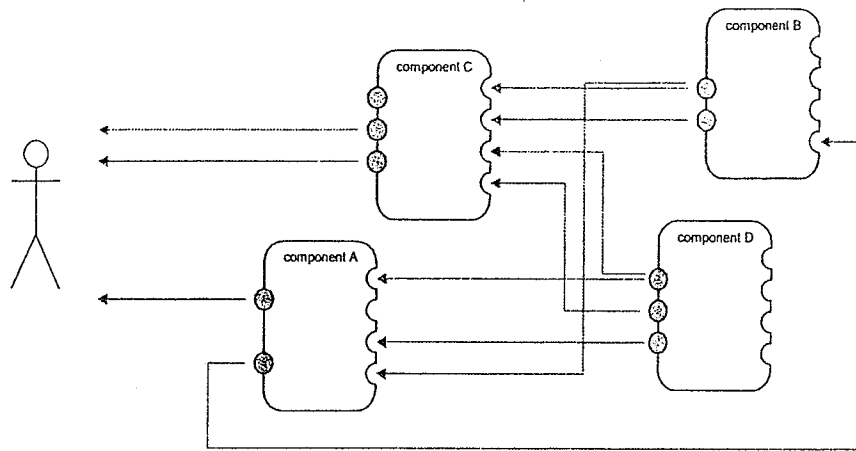


Fig. 3.2: Systems are built with components.

the system is created. There are of course exceptions to this general rule, like the use of shared object files in Linux or Dynamic Link Libraries (DLL) in Windows; components can use the services they provided dynamically, but they still must know the output of such a component when the system is created.

Software systems, however, can be designed and built in other ways, for example we can shift our thinking from components being first-class entities to using services as first-class entities. We still require components to act as the vehicles to provide services, meaning that the composition and interaction of services now dictate the structure of a system. For example Figure 3.3 shows interconnecting services which form a system.

While components are still needed, as shown in Figure 3.4, their role is strictly to implement or provide services. In this role, one component can now provide one or many services, which in turn may require other services. Alternate implementations can be substituted *at any time* as long as they implement the same externally observable behavior. With the ability to change services, yielding alternating implementations, we achieve a very

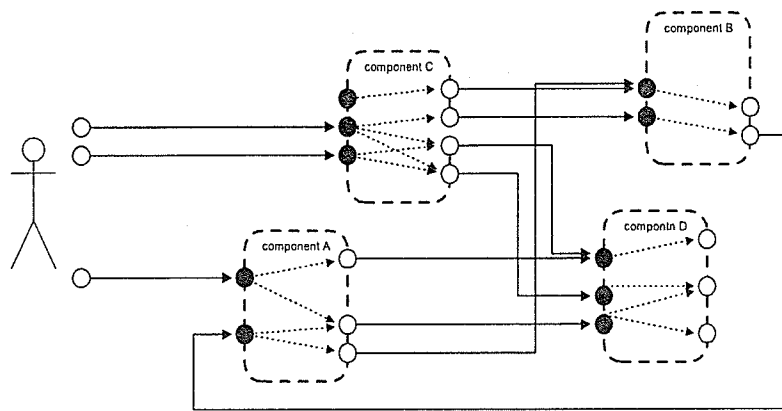


Fig. 3.3: Systems can be built from services.

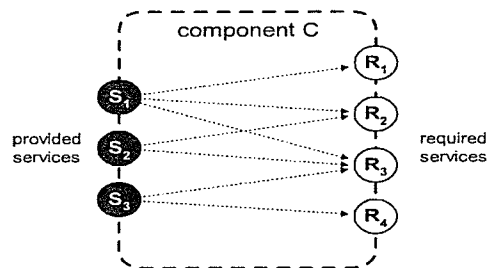


Fig. 3.4: Components are used to provide services.

high level of reusability, as we can reuse and change only individual services as desired, instead of having to change entire components. It is the dynamic use and substitution of services that provides the foundation for the service-based paradigm.

To summarize, component-based systems are composed of interacting components that provide some service via inputs and outputs (plug compatibility) of components, whereas service-based systems are composed of interacting services which are provided by components and can be dynamically used and substituted as desired.

Aside from the fundamental composition of systems, service-based systems, as opposed

to component-based ones, can have services or requirements that exist outside of the system in which they are defined. For example, if we have a system A that contains service S1, S1 could require a service or resource from A or from another system B, or C, and so on; S1 is not constrained to the services and resources of A. Component-based systems on the other hand, must know about all existing resource and component requirements ahead of time, and those requirements must be within the available scope of the component, including for example DLLs, which still must be available when the system is created (although they can be called dynamically when the system is running).

In the following section we discuss another difference between component- and service-based systems, which is the location of components, services and resources, and how they are accessed differently.

### *3.2 Component location and access*

In traditional component-based systems all of the components and their required resources are located within the same system, meaning that everything needed by such a system is readily available, with no external facilities required to locate or access any components or resources.

However, when we consider service-components, we now must realize that not all of the required components or resources are immediately present or contained within a single system. This is why location and accessibility of service-components is extremely important, especially if one system or service has heavy dependencies upon others to provide it with

Location	Manner of Access	
	local	remote
local	traditional or service-based	service-based (server-side)
remote	service-based (client-side)	service-based (incl. Web Services)

Tab. 3.1: Options for service locations and their accessibility.

services.

Component-based systems typically have local components which are accessed by local systems. In some cases though, for example Yellowdog Updater Modified (YUM) a package management system of Linux, we can have dislocated component-based systems, where components reside in alternate locations and must be downloaded and installed for use. With service-components we now have a much larger range of accessibility, which includes both local and remote access. Table 3.2 provides us with a simple taxonomy of components, their residency, and methods for accessing them. As we see with normal components, we only have one accessibility method, whereas with service-components we can access local and remote components, both locally and remotely.

If we now consider implementation aspects of service-components, it becomes apparent that there must be a common communication mechanism through which client-service interactions may take place; where a client can be considered either an end-user, or another system as a whole. In simpler usage scenarios, service providers can reside on local fixed storage from which they may be loaded when necessary, not unlike DLLs or their equivalents under other operating systems, like shared objects in your favorite flavour of UNIX.

In more complex scenarios services can be accessed through a LAN or the Internet. In this case, the service-component can run at a remote site, similar to the approach adopted by Web Services, with its services accessed remotely; alternatively, the service-component can be downloaded to the local site and installed, or installed from the remote site. In the former case, the interaction will in fact consist of a series of messages exchanged between the client and the service provider. In the latter case, once the external service-component is installed, it becomes indistinguishable from a local one. However, when the interaction is over, the component will be removed from memory, and possibly even deleted from local non-volatile storage.

Scenarios like those just described necessitate the presence of an infrastructure capable of managing the interactions described - either as part of the original client application, or independently of it. Such an infrastructure may be embedded in the operating system and thus made available to all the applications, or it may be made to run as part of the actual application, or even act in a third party capacity, helping to marshal dynamic services, but not part of either the operating system nor the application itself.

With service-components (possibly) from external sources, and users having the ability to switch them dynamically as desired, a mechanism for searching and selecting services is required. The following section describes the requirements for both searching and selection of services.

### 3.3 *Service search and selection*

As mentioned in the previous section, if service-components reside at remote locations (service providers) and there exists a infrastructure in place to access them, we still require a mechanism to search for and select the ones we want. This search-selection process would involve a registry of available services, making the process very similar to that of the Web Services paradigm, involving the universal description, discovery and integration protocol (UDDI) to search for available services.

In the Web Services approach, services are located via their signatures and simple descriptions written in web services description language (WSDL) [15]. While simple and efficient, this approach suffers from a major drawback: namely, it requires the designer to know in advance the signature of the service to be invoked. By extension, this also means perspective clients must possess detailed knowledge of the component that will provide the service.

This, however, does not fit well with the dynamic and ever-changing nature of service-based applications. Service-components are not known beforehand and they need to be accessed on the basis of their advertised capabilities for providing particular services, rather than according to a name and signature. To provide this level of searching, service-components would have to provide a significant amount of information about themselves. The minimum information required from service-components would consist of the names and signatures of the services provided, much like Web services. Additional descriptive information such as; functional constraints (i.e., inputs and required outputs, non-functional

constraints (memory and timing requirements), and security- and access-related features, as deemed necessary for the selection and/or use of that particular service-component.

In order to make the search and selection process of services as descriptive and generalized as possible, we need some kind of service lookup facility, which contains sufficient information about services and their requirements, and which can also provide the services to the user once selected. Such a facility would have to consider any provided QoS or requirements of a service, as these can be the determining criterion for the selection or rejection of a service. If for example we wanted a spell checker service, but only if it can complete in 30 seconds, then we do not want to see all spell checking services from the lookup provider, we only want to see those that match our service description *and* those that match our requested QoS property. If supplied, we would also only want to see those that matched the previous criteria *and* any additionally specified requirements.

Lookup and selection of services will be supported within our ADL and our implementation will rely on the infrastructure of Jini to provide this capability. In Section 2.4.1 and Appendix B, we further examine Jini and how it allows services to be described with specific documentation, as well as providing a lookup and marshaling mechanism for services, making it a viable choice as a framework for service-based system development.

Now that we have discussed the differences in component- and service-based computing, and differences in location, access, selection and searching, we must create a language for our ADL that captures or allows for these requirements as best as possible. In the following chapter we provide the design and rational of the language for our ADL.

## 4. THE LANGUAGE

The language provides the ordering of and information about component relation for definitions added to an ADL. Our ADL is tailored to the service-based paradigm, such that it describes the relationships of architectures and their service-components, and how service-components interact to provide services.

Unlike the languages of other ADLs, like C2 for example, that make the connections between components explicit, in our language we leave the expression of connections implicit, being derived from the location of service-components relative to the services they provide and the architectures they are defined within. This is done mainly because one component can provide many different services, within one or many architectures. On the other hand many components can implement one service from one or more architectures. To try and explicitly describe all of these possible connections adds a lot of additional, non-essential information to our language, so we opted to create a language that could be converted to an XML schema (see Appendix A for full XML schema description) and then described in XML. Using XML to implement our language we could then, through the use of parent-child relationships and the use of global elements implicitly describe connections between components and services, by where the component and service element appeared.



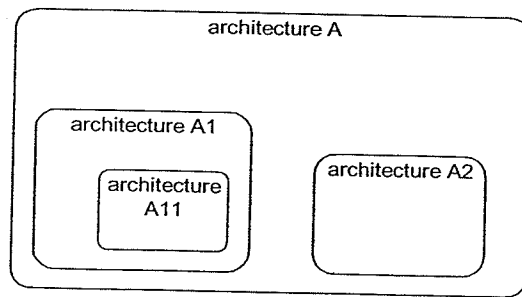


Fig. 4.1: Architectures can include architectures.

## 4.1 Architectures

The specification of a service-based system starts with the definition of an architecture. Architectures can also be composed of whole other architectures, providing a high level of reusability, as shown in Figure 4.1. When interacting with adjacent or nested architectures, the openness of the architecture determines if services from one architecture can be shared. More specifically, if one architecture can use the services of another as required services, or its own services can be used in a similar manner, we would define it as *open*. Alternately, if an architecture does not require any outside services, nor allow access to any of its services it is defined as *closed*.

The definitions of services contained within an architecture describe what the architecture does, with the provision to have these abilities extended by adding services as needed. In the case of nested architectures, services from higher-level architectures can have access to all services from lower level ones. At the same time, services from a lower level architecture can access services from a higher level one if and only if (a) the lower level architecture is not declared to be closed, and (b) such services are explicitly designated as global, in the

manner that will be outlined below.

The grammar representation for services has no limitation on the number of services that an architecture may contain, except that there must be at least one service definition which acts as the so-called starting service, i.e., the service which is to run initially when the architecture is loaded. Other services may be loaded at the same time when an architecture is loaded; the list of such services is determined automatically by the infrastructure when it loads the architecture.

Specification :=

Architecture+

Architecture :=

Openness "architecture" Name "is"

"starting" Service

["contains" Service]\*

["includes" Architecture]\*

"end-architecture"

Openness := "open" | "closed"

## 4.2 Services

Each service within an architecture is provided (i.e., implemented) by one or more software components. In this model, components are derived entities which can be extracted from the service definitions, rather than being primary entities that provide services. Furthermore, the definition of a component that provides a service can be located inside *or* outside of the definition of a service which it implements. This however is not the same as a service being external; which means the service is entirely defined outside the scope of an architecture. When we consider a service as being external, we are referring to the fact that the service and its implementing component are provided outside the definition of the current architecture.

If not external, every definition of a service must contain an implementing component. Components in any case act as implementations of the services they provide (of which there can be more than one), similar to implementing interfaces in Java. For example, you may want a quick but not-quite-accurate spell-checker, or a slower but fully accurate one.

The language allows the client to distinguish service provider components and select the best one. The accessibility, or where it can be accessed from, can allow other services of nested architectures to have access it, provided the accessibility is not local. Most services are accessible globally (by default), to other services both within the architecture and outside of it; in some cases, services may be restricted to client services within the same architecture only, by setting their accessibility to local.

Service :=

```
Accessibility "service" Name "is"
```

```
["provided-by" Component] |
```

```
["external"]]
```

```
["provides" ServiceMsg]+
```

```
"end-service"
```

```
Accessibility := "local" | "global"
```

### 4.3 Components

A component is defined with a name and an availability specification. As components physically implement services, the availability specification refers to the implementation itself:

- A private component provides its services to clients within the same architecture only; such services are always local.
- A protected component is accessible to services residing on other hosts as well, and the interaction is performed through message exchanges via an appropriate communication link (think of Web Services); services provided by a protected component may be either local or global.
- Finally, a public component may have its executable image (e.g., a Java jar file or equivalent) available to be transferred to the remote host for execution; in this case, any accessibility restrictions on the services provided are meaningless.

While components can have differing availability, it is important to note that a component defined as public does not make the service it implements mobile, only movable; meaning that it can be sent between clients and executed, but not partially executed on one client, suspended, transferred to another client and have the execution resume from where it was suspended.

Component :=

```
Availability "component" Name  
"end-component"
```

Availability :=

```
"private" | "public" | "protected"
```

Access restrictions imposed by the components are mapped onto services. In this manner, a service can have different implementations, some of which are global while others are protected, and possibly some of which are local as well.

The default accessibility level is public; protected takes precedence over public, and private takes precedence over either of them (similar to Java). It may happen that a single service provider component is labeled with different access restrictions within different service definitions. In this case, the most restrictive qualifier will be used, eliminating inconsistencies from the processing of architecture definitions.

Note that the outward extension of an architecture depends entirely on the service provider definitions, whereas the inward extension depends on the openness of the archi-

ecture. In other words, an architecture defined as *closed* can declare that it will not seek help from others, but individual service provider components may be made accessible or available to the public on a case by case basis, independently of the architecture extendibility setting.

#### 4.4 Service Messages

Each service is invoked via an appropriate message. To carry through the relation to Java, consider the methods defined in an interface as service messages; they describe the behavior of the service, which are in turn implemented by the service-component. A service message specifies functional information such as the service name and a list of parameters in parentheses (similar to a method signature), while the other aspects describe definitions that define guarantees and requirements of the service message. A guarantee could be that processing will take only take X amount of time, or accuracy of task if 98% or better, whereas requirements could mean alternate resources or services that this one requires to provide its service, such as memory or processing constraints.

ServiceMsg :=

    Name "(" ParameterList ")

    ["with" QoS Guarantee]+

    ["at" ResourceReq]\*

    ["requiring" RequiredServ]\*

ParameterList := [Datatype [, " Datatype]\*]

Datatype := "int" | "double" | "string" | "void"

## 4.5 QoS Guarantee

The operational information of a service is formatted as a list of QoS guarantees. Each guarantee consists of a property and an associated value. This list of guarantees provides a mechanism for the search and selection process of services by the user. Example: a user wants a service that can calculate their tax return in 3 minutes, a search of QoS guarantees of available services is done, and if there is one, it is selected. If from that same example more than one matching service was found, then the user has the option to additionally refine their criteria until the exact service they want is found. Refinement can be the addition or deletion of guarantees to or from existing search criteria. Coupled with the use of resource requirements (defined in the following section), a user has very specific control over what services are searched for and what services are selected. A QoS guarantee can be anything, ranging from guaranteed computation time to download speed.

QoSGuarantee := Property "of" Value

Property := "anything"

Value := "anything"

## 4.6 *Resource Requirement*

In order to meet the guarantees for a service, a component may also have some requirements of its own. Two main types of requirements can be readily identified: resource requirements and service requirements. Resource requirements are similar to QoS guarantees, except that they spell out what are the properties of the operational environment that the client must provide in order for QoS guarantees to be met. Such requirements may include minimum available memory, minimum CPU speed, compatibility with specific versions (or range of versions) of the operating system and/or other services, and other related information.

ResourceReq := Resource "of" Value

Resource := "anything"

Value := "anything"

## 4.7 *Required Services*

Service requirements, on the other hand, identify a number of other services that are (or may be) needed in order to fulfill the obligations. For example, a tax processing service might need additional services to process returns based on the return type; furthermore it may require certain QoS guarantees of the required services in order to meet QoS guarantees of its own.

Note that any given component may actually be able to provide one or more services, as



is the case in the component-based paradigm. However, instead of a component requiring a set of services regardless of which particular service it provides, our language allows more precise modeling of dependencies. Namely, it is possible to define a subset of required services that correspond to each of the provided services, as shown in Figure 4.2.

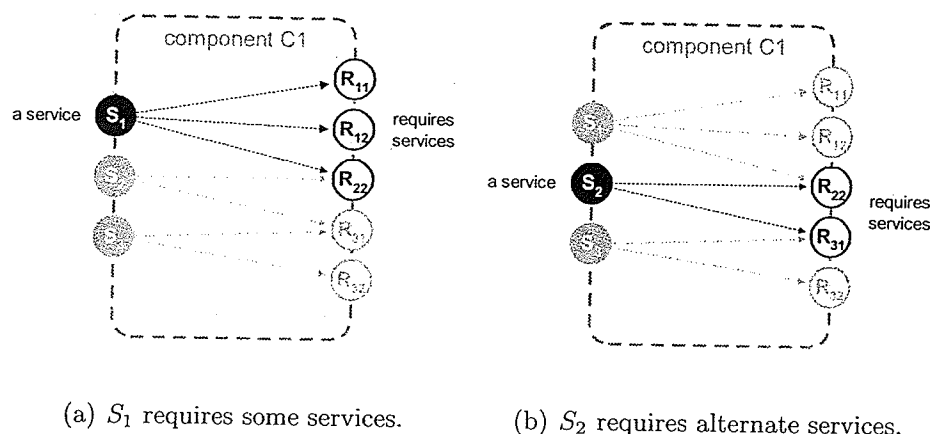
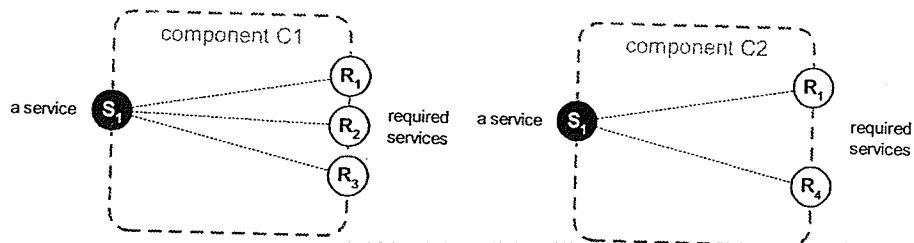


Fig. 4.2: Services might depend on others to provide their services.

This facility provides an additional selection criterion and allows for finer control of system execution (and, consequently, performance) at runtime. Namely, the client may prefer to get services from a component which needs less resources; in some cases, less resources may mean fewer required services, in particular in cases where some of those services are not available locally and, thus, have to be accessed or even downloaded from a remote location prior to being used to provide the original service.

We have mentioned above that any single service can be provided by more than one component. The components that provide the same service may differ in terms of their QoS guarantees. They may also differ in terms of other services they require; for example, component C1 may require services  $R_1, R_2$ , and  $R_3$ , in order to provide service  $S_1$ , whereas component C2 may require services  $R_1$  and  $R_4$  to do the same, as shown in Figure 4.3(a)

and Figure refreqc2.



(a) C1 may require  $R_1$ ,  $R_2$  and  $R_3$  to provide  $S_1$ .  
(b) C2 may require only  $R_1$  and  $R_4$  to provide the same service,  $S_1$ .

Fig. 4.3: Different components provide the same service with differing requirements.

These services may be limited to services available locally, i.e., those defined within the architecture as well as those downloaded from other architectures. The default option is local as well as remote ones.

Furthermore, a required service can be labeled immediate, in which case it must be made available prior to execution (by whatever means available) or optional, which may defer acquisition because the component may not need it at all, or is ready to wait until it becomes available.

RequiredServ :=

Location Immediacy

Name ["(" ParameterList ")"]

["with" QoS Guarantee]+

Location := "local" | "remote"

`Immediacy := "immediate" | "optional"`

The infrastructure that manages the architecture will initially load the starting service, as well as its immediate required services (subject to resource limitations, of course); optional services will be loaded when they are actually invoked.

We note that the list of required services is an optional part of the language. The definitions of services and components that implement them are necessarily local; if a service is accessed via a remote host, no guarantees can be given as to the services it may require.

An alternative to this service-centric design, is to turn the architecture definition from inside out, and obtain a more common component-centric definition, similar to the one created by Cervantes et. al. [8, 9] which pertains to OSGI [43] technology. More specifically, in their design, they consider dynamic components relationships (loading, transporting, etc), which is very similar to our notion of dynamic services. The difference is, that they consider parts of software being transported and dynamically used, whereas we view all of a system's components as services.

## 5. THE IMPLEMENTATION

With the complete design of our language and the language mapped to an XML schema (see Appendix A, an implementation of our ADL was completed using Java 1.5. Java was chosen as the vehicle for implementation for several reasons; the most important outlined below:

- Java provides a very diverse and comprehensive API for working with XML. XML documents can be read, written and validated with very few lines of actual code. XML documents can be searched and modified simply with Javas' implementations of XPath [55] and XPointer [53]. Furthermore the platform independence of its XML APIs contribute significantly to its suitability.
- As mentioned in Section 2.1, to achieve full dynamism of the scale the service-based paradigm requires, would force us to either implement or utilize a framework to marshal services. Java again provides the solution to this problem with Jini [48] (Section 2.4.1); an API that allows services, described as the implementation of remote interfaces, to be located, used remotely, serialized and acquired, dynamically activated, and dynamically removed. Jini has the ability to communicate with its lookup services located on the local machine, local network, or over the internet. The

only drawback is that Jini may have difficulty finding services, as its lookup service does not forward requests, so each lookup service must be queried individually.

- With Java, we can run our implementation on any platform with a Java Runtime Environment, without having to recompile or modify the code in any way. Even though the implementation was done in Java 1.5, it was compiled to be backward compatible to Java 1.3, providing some flexibility in JRE versions.
- The OO capabilities of Java allowed us to create general abstract classes and interfaces for elements and code tools, cutting down on the amount of duplicated code through sub-classing and polymorphism.

Once the programming environment was decided upon, the first step was to begin our design by breaking our program down into logical package descriptions, or which Java packages would handle what processing. Once the sectioning of the design was complete each of the packages was further decomposed into individual classes, represented as class diagrams with their interactions modeled via sequence diagrams. To help illustrate how we envisioned our ADL functioning we then created use-case diagrams and state diagrams.

The goal of our design was to create an ADL that was menu-driven and had a dialog-based graphical user interface (GUI) that allows elements to be added and removed from the overall specification. As a specification within our ADL progresses, the overall form and element organization is represented to the user as a tree for ease of (re)organization and understanding. Once an element has been added to the design, it can then be modified (via it's properties), removed, have children added to it, or depending on the element, have

it's source code generated and viewed. The layout of the GUI for our ADL is comprised of a tree view on the left hand side and a dynamic content tab pane on the right for viewing source artifacts.

## 5.1 Packages and Structure

Figure 5.1 shows the overall package description of our ADL implementation. The Structure package contains all of the classes that represent elements of our XML schema, the GUI package contains all graphical interface classes, the XML package contains all classes used to read/write/parse XML files, and the Code package contains all classes used to generate code.

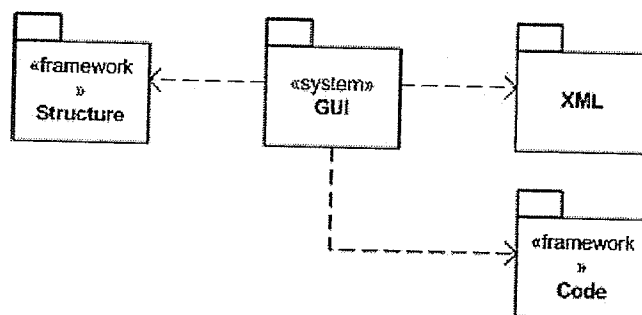


Fig. 5.1: Package description of ADL implementation.

With the use of XML and XML schema, we were required to create a class representation of our schema to enforce the constraints of the schema within our code. This was done by creating one class for every element present in the schema, with the exception of the *starts-with* element (as it is nothing but a wrapper element). Figure 5.3 shows the classes within the Structure package, and how they are related to one another.

The structure classes use aggregation to maintain the proper schema mapping. More specifically, all of the structure classes are arranged and composed of one another in such a way that they directly represent the form of the XML schema (see Figure 5.2). Aggregation is handled within each class using a *java.util.TreeMap*, which is similar to a hashtable, but provides performance guarantees. The use of TreeMaps was further enhanced with the use of *generics*, a new concept to Java 1.5, which allows return types of collections to be specified at creation time, removing the need for type-casting.

Each of the classes of this package are also used in a corresponding dialog, which allows users to make instances of them, for example the *NewArchitectureDialog* from the GUI package is used to create instances of the *ArchitectureDef* class, and so on.

The GUI package, as mentioned, includes all of the classes that make up the user interface for our ADL. The main window for the application, along with all of the user input dialogs, and other customized swing classes are contained in this package. One of the things to note here is that the GUI package depends on the other packages, but there is no dependence on the GUI package from any of the others. The implementation was designed this way to keep the data model of our ADL loosely coupled to the user interface. This way if future expansion was desired, in either the data model or user interface, refactoring is minimized. The overall arrangement of the GUI package is shown in Figure 5.4.

All XML processing is handled by the XML package. All constants used for element names, reading, writing, parsing, and validating of XML is done by the XML class in this package. To ensure processing is as trivial as possible, the Document Object Model

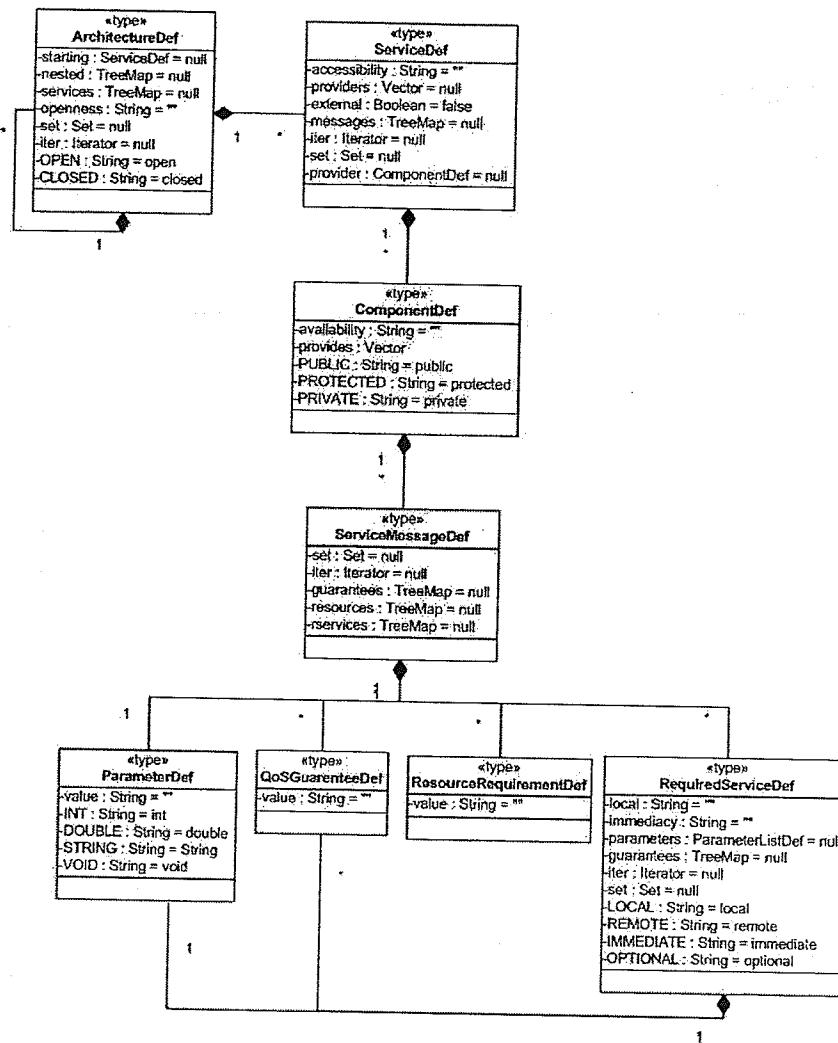


Fig. 5.2: Aggregation within the Structure package

specification version 2(DOM2) was used for all XML operations. DOM2 provides simple methods to handle XML, for both reading and writing. To read an XML document using DOM2 is as simple as:

```

docbuilderfact = DocumentBuilderFactory.newInstance();
docbuilder = docbuilderfact.newDocumentBuilder();
docbuilder.setErrorHandler(new DefaultHandler());

```



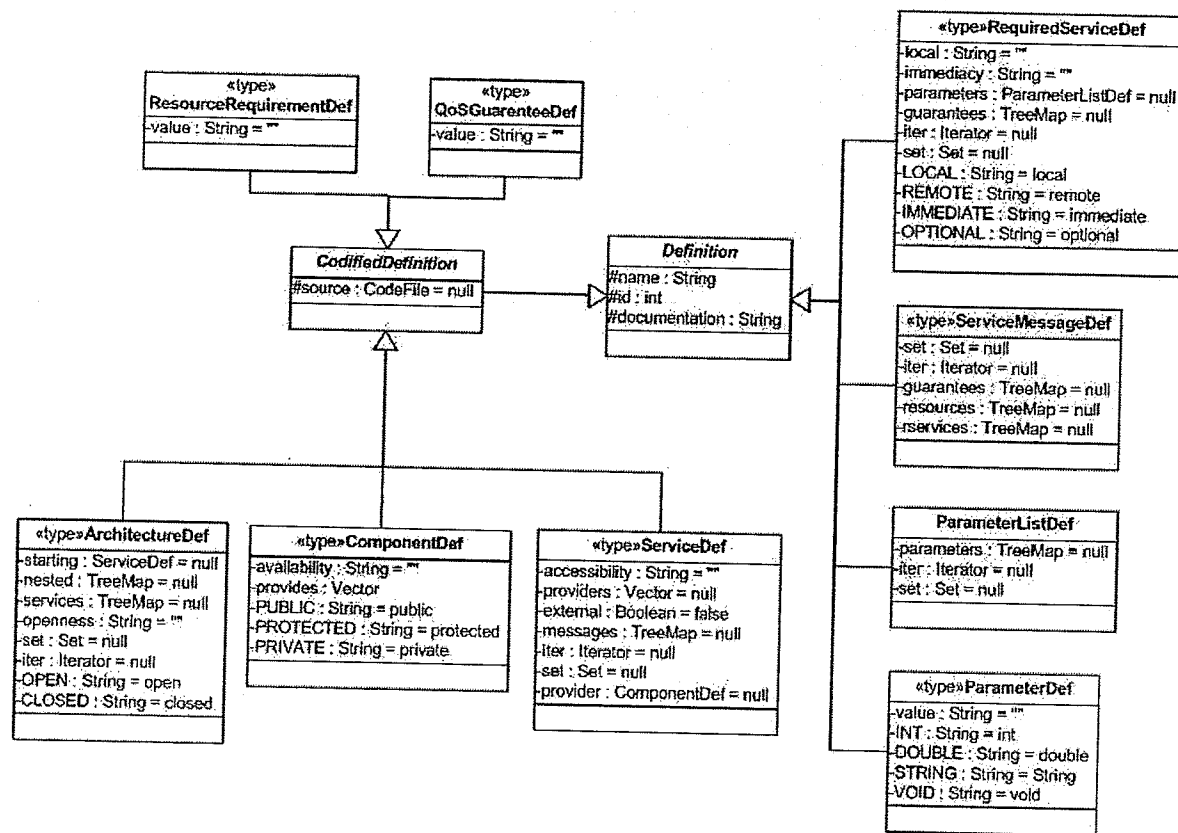


Fig. 5.3: Arrangement of classes in the Structure package.

```

try {

    document = docbuilder.parse(xmlfile);

} //end try

catch(SAXException spe) {spe.printStackTrace();}

catch(IOException ioe) {ioe.printStackTrace();}

```

While parsing an XML file, DOM2 also provides validation of the XML document, allowing us to use the same mechanism for reading an XML file to provide validation support in our ADL. Since DOM2 represents an XML document as a tree in memory, we opted to use a JTree in the main user interface to display our specifications. In doing so we

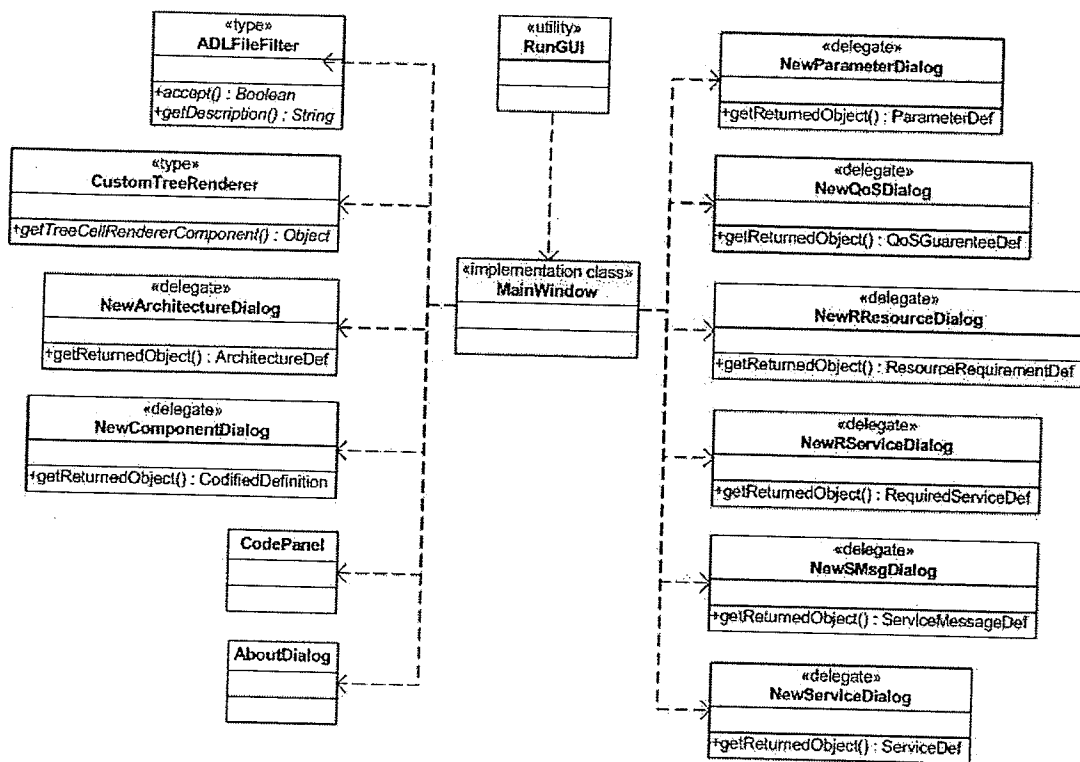


Fig. 5.4: Arrangement of classes in the GUI package.

could use simple recursive methods to read and write XML to and from the JTree. With the code to work with Java trivial, the associated XML package is also trivial consisting of only one class, as shown in Figure 5.5.

Code generation for our ADL is contained within the Code package (more details on code generation in Section 5.3). Figure 5.6 shows the relationships between the code generation classes. The Generator is the main class that actually performs the code generation using simple recursive methods to walk through the JTree in the main user interface, and generate corresponding source artifacts.

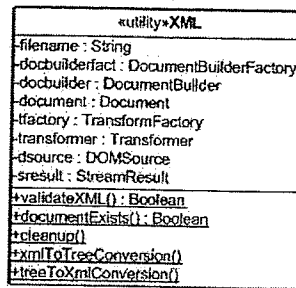


Fig. 5.5: The class in the XML package.

## 5.2 Package interaction

In the previous section we discussed how all of the packages and classes are arranged. Now let us discuss how the packages and classes interact with one another. We show in Figure 5.1 how our ADL is arranged and how each of the classes within each package are related. Before we discuss how the packages interact, we must first understand the basic processes of our ADL and how they relate to one another. In Figure 5.7 we view the basic states of interaction and show how they are related to other tasks that can be performed.

With our ADL designed to be a dialog-based system we ensure that all of the states in Figure 5.7 must return to the idle state after processing is complete aside from a few exceptions. The ADL will not proceed with any task until the user explicitly activates it, which can take place through either a menu click or a dialog. There are exceptions where one task depends on another prior to its execution; for example code generation will never take place until the specification has been saved. Such dependencies only exist where consistency is required –we only want generated code for the most recent specification, which requires that it be saved beforehand.

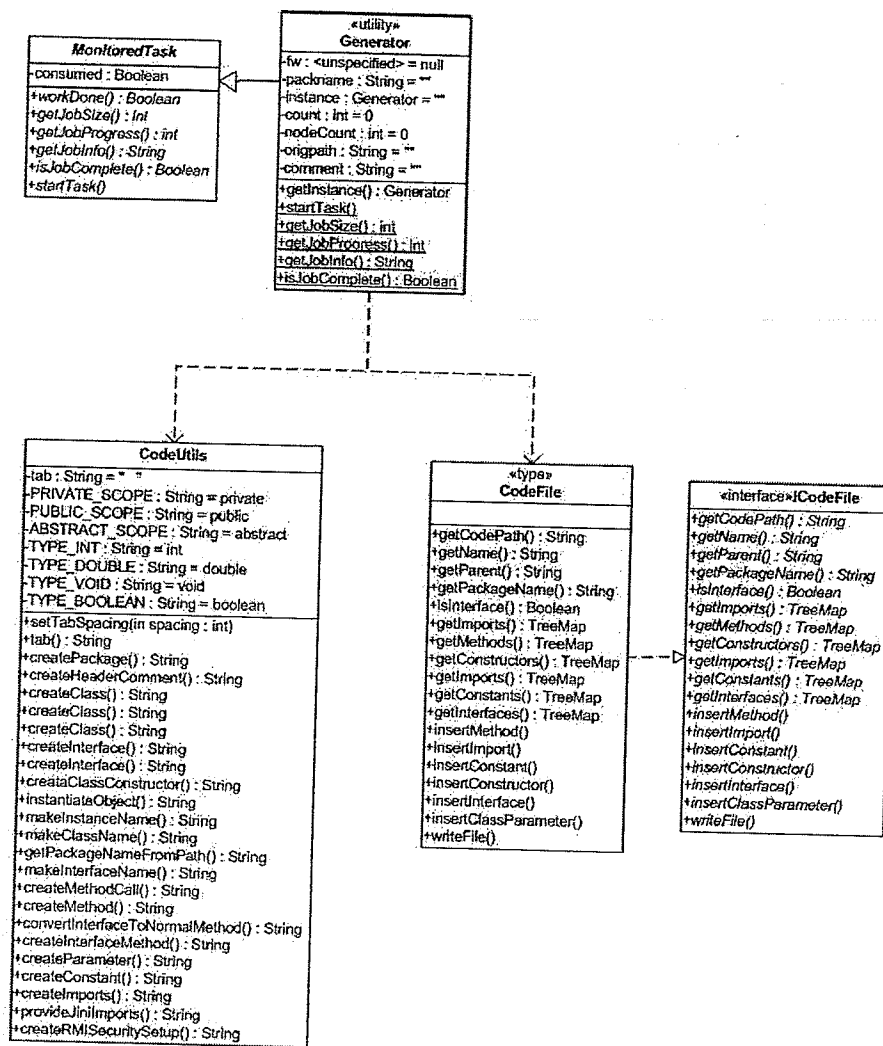


Fig. 5.6: Arrangement of classes in the Code package.

Figure 5.7 describes how the states of our ADL are related, what it does not show is how the packages of our implementation interact to provide the processing of those states. To describe this interaction we use sequence diagrams and specific processing tasks from our ADL.

All interaction involves the GUI package and at least one of the remaining packages. The decomposition of the problem makes each of the packages, except for the GUI package,

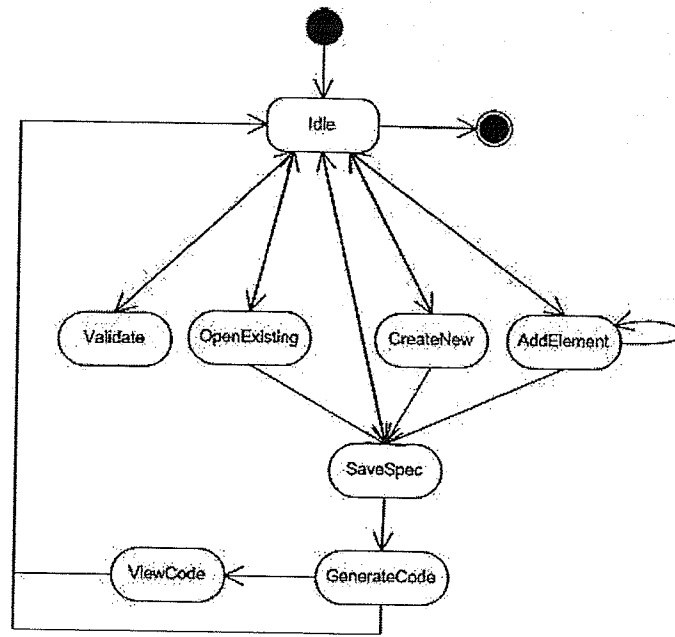


Fig. 5.7: Overall sequence of ADL.

self reliant; such that they do not require any of the resources of any other packages. The GUI package depends on all of the other packages because it is used to represent these packages to the user in a graphical manner. In essence the GUI package acts as the portal to the other packages, in that, every interaction takes place through the GUI package which then uses resources of another package to perform a task.

With a general idea of how the packages interact, let us more specifically examine how the packages mentioned in Section 5.1 interact with one another from the perspective of the class *MainWindow* from within the *GUI* package. We need only consider this view as all of the packages interact only through *MainWindow* – even then, none of the other packages such as *XML* or *Structure* interact with one another, only with *MainWindow*.

As mentioned each of the elements from our XML schema have been mapped to a

corresponding class, each of these classes in turn have a corresponding dialog that is used to create instances. This design pattern is carried through all of the classes in the *Structure* package, each one class representing an element with a corresponding dialog. Therefore, we can examine only one example of the interaction between the *MainWindow* and the structure classes to understand how they are used. Figure 5.8 shows the interaction between the *MainWindow* class and the structure class *ArchitectureDef*, and how the corresponding dialog *NewArchitectureDialog* is used to create instances.

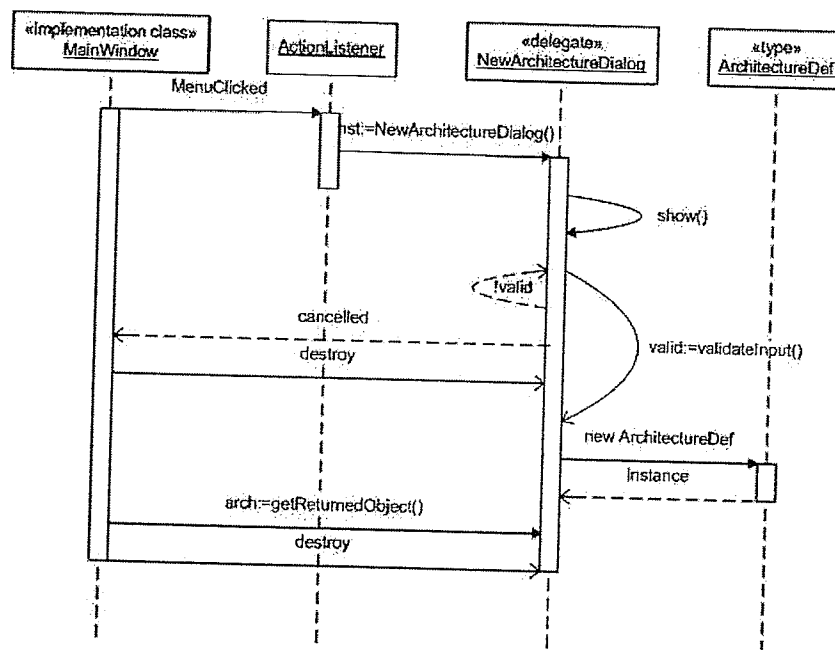


Fig. 5.8: Creating a new *ArchitectureDef* instance.

To create an instance of any one of the element classes, a menu item is clicked which then presents the corresponding dialog to the user. The user can at any time cancel creating an instance, or continue providing information until an instance can be created. Each of the corresponding element dialogs provide error checking to ensure that the information

provided by the user is correct and will not cause inconsistencies within a specification. Also, constraints such as elements that have dependencies on others, like an architecture that must have a starting service, are handled by the corresponding element dialog. For example, to create an *ArchitectureDef* instance the user is forced to also create the *ServiceDef* that will act as the starting service, before the *ArchitectureDef* instance is created.

In terms of state semantics, Figure 5.8 shows movement from the idle state to the *addElement* state and back again (refer to Figure 5.7 for state details). Some of the corresponding element dialogs also allow more than one instance to be created; for example when creating a new *ServiceDef* the user has the option of adding as many *ServiceMessageDefs* as they desire, yielding the cycle on the *addElement* state.

Next we look at the how a specification is saved, describing how the *XML* package interacts with *MainWindow*. The process for saving is the same regardless of whether the user initiates it directly, or it is performed automatically. More specifically when the user initiates the save process through a menu click an abstracted method is used to perform the actual save operations. This way the same save operations are used anytime a save is required, providing flexibility for auto-saving (timed or otherwise) and to maintain consistency (saving before code generation). Figure 5.9 shows the interactions involved in saving a specification. Since the *XML* package is only used for handling XML, this is one of three places it is utilized. The other two uses are for loading a saved specification and for validating an existing specification.

With a specification represented by an instance of the *XML* class, we have further

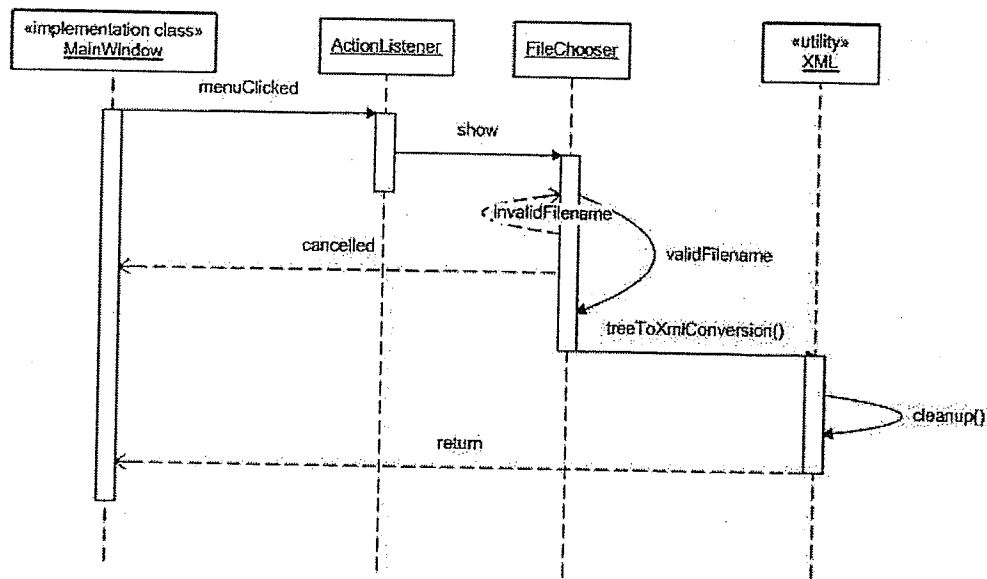


Fig. 5.9: Saving a specification.

simplified handling a specification between memory and disk. From Figure 5.9 we see an example of this with the calling of the method `treeToXmlConversion`, which is a simple and efficient recursive method that converts the tree view from *MainWindow* to a valid DOM2 representation in memory. Alternatively when loading a specification another simple recursive method, `xmlToTreeConversion` is used to load the specification from disk. To initiate the save process a user simply clicks the save menu item or initiates a task that auto saves prior to execution (like code generation). This in turn either displays the file dialog or not depending on various factors and then creates the XML representation of the specification in memory. The XML in memory is then written to a standard XML file on disk. The process of loading a specification is the exact opposite; a file is read from disk and loaded into memory, then the tree view is refreshed based on the XML in memory.

Finally we look at the interaction of the *Code* package. Like *Structure* and *XML*, this



package is only used within the *MainWindow* class and does not have any dependencies on any other packages. There are some notable exceptions in the manner with which this package is arranged for interaction. Mainly the *Code* package provides all of the resources to generate specification source code. To accomplish this however, can be an intensive process requiring a lot of computational resources. To alleviate the work load for *MainWindow* multi-threading was introduced. Using multiple threads meant that the interaction with the *Code* package had to thread safe; which was accomplished using Java synchronization primitives.

Figure 5.10 describes the interaction of the *Code* package. Notice that for the most part it begins the same as for the other packages, in that a user clicks a menu item which then begins the process, the exception here is that when the generation process is started a new thread is created which runs the code generation methods. The separate thread use is characterized by the asynchronous message from the status dialog which continually asks the thread what its progress is so that the user can be notified in the form of a progress bar.

Aside from the use of an extra thread, the *Code* package also has a restriction on how it can be used. Unlike all of the other classes in this project the *Generator* class –the class that actually runs the code generation, is a singleton class. Using the singleton design pattern we can ensure that only one *Generator* instance ever exists to increase thread safety and prevent more than one set of source files from being generated at the same time.

There is much more to the code generation process, which we discuss in the following

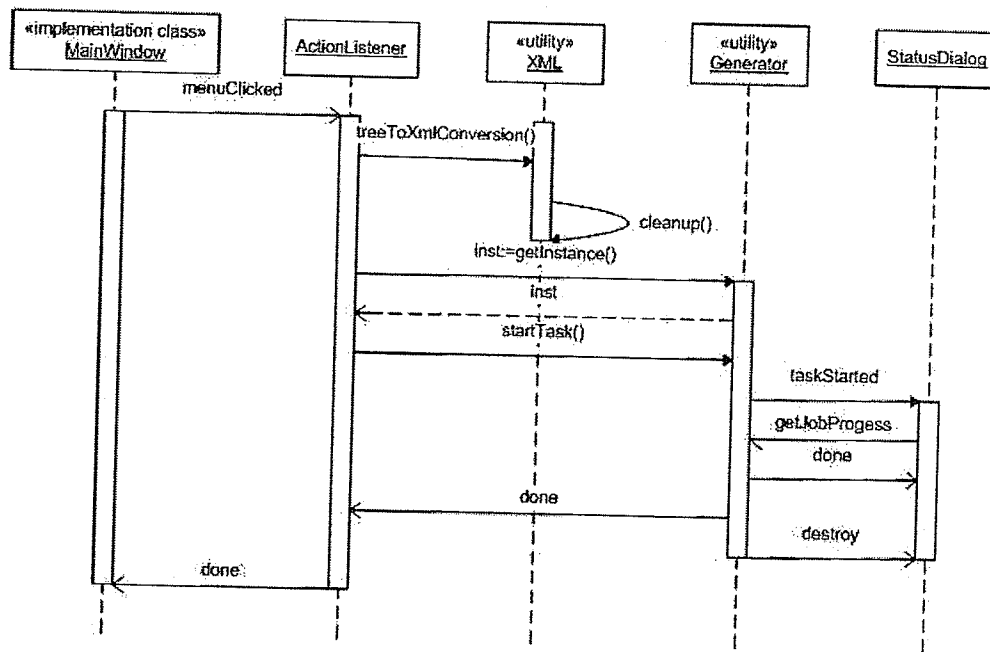


Fig. 5.10: Generating code for a specification.

section.

### 5.3 Code Generation

To aid the design process we have included a source code generation toolkit, which generates the complete source code skeleton for a design. The generated source code is backward compatible to Java 1.3, provides Javadoc compliant commenting for easy documentation, and is compatible with Jini 1.0 and above.

Generated source is placed in a directory called "Source" located in the design save directory. As the source code is generated it is separated into packages based on architectures within the design; i.e., each architecture is its own package. The separation of architectures into packages allows us to maintain openness restrictions from one architecture to

another, with the use of import statements in code. Although each architecture creates a new package, all of the sub-elements of an architecture in the design are constituents of the corresponding package. This way, we can enforce availability and accessibility through packaging as well.

Consider an example: If we have a specification that contains two architectures, say A1 and A2, they form two distinct packages, A1 and A2. Inside each of these packages we would find all of the source code for all of the services, components, interfaces, etc that are child elements of that architecture. If we have the case where A2 was a nested architecture of A1, then A2 would become a sub package of A1.

With an understanding of how generated source code is arranged, let us discuss how the source is generated from each of the elements in the design. Since our ADL works with Jini, we have opted to create all source artifacts as skeleton services that work within Jini. To elaborate, let's examine what part of the source code each element in our language corresponds to; as shown in Table 5.1.

Not all of the elements of a specification become independent source artifacts, only those that extend *CodifiedDefinition* (from Figure 5.3). The other elements become part of the next closest codified parent element, which will always be either an Architecture, Component or Service. Each one of the codified definitions contains within it a copy of its source file, removing the need to explicitly handle directories and location on a specification-wide scale. This then provides better access to source artifacts through the tree view of our GUI, as each node in the tree represents a language element, and therefore allows access

Language Element	Source Artifact
Architecture	Facade providing Jini service and lookup
Service	Remote interface that contains Service-messages as methods
Component	Implements remote interface(s) from the service(s) it provides
ServiceMsg	Included as method definition in service interface
ParameterList	Part of the signature of a ServiceMsg
QoSGuarantee	Implements AbstractEntry interface
RequiredServ	Jini service definition included in the providing facade
ResourceReq	Implements AbstractEntry interface

Tab. 5.1: Language elements and corresponding source artifacts.

to its sourcecode (if applicable).

Each of the source files that are generated are stored in memory as class type *CodeFiles* which extends *RandomAccessFile*. Extending *RandomAccessFile* provides us with all of the capabilities of a normal file, but also allows us to seek forwards and backwards within the file to add or remove text. With the ability to seek in the file, we can write code files, and seek backward or forward to predetermined sections to insert or remove code. Currently though, as we do not support the editing of code files in our ADL, we do not need to use the seeking functionality of the *RandomAccessFile* class. Instead we currently write codes files in a top-down fashion as shown in Figure 5.11.

Pseudo-code for how we produce a code file is outlined below.

Begin

write-package

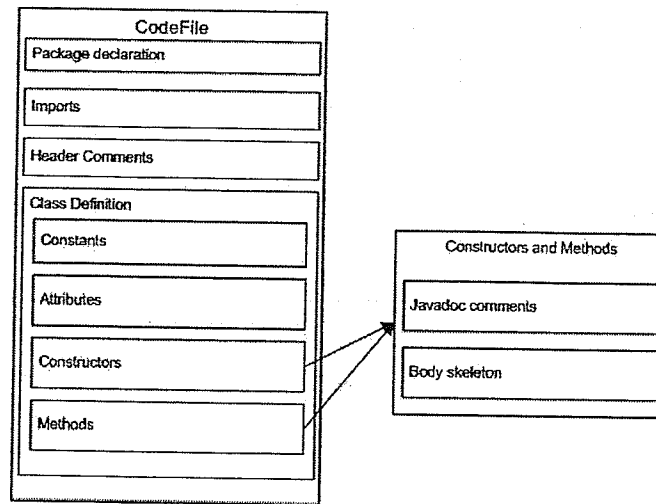


Fig. 5.11: Arrangement of sections within a codefile.

```

write-imports

write-header-comment

write-class-header

write-parent-class

write-implemented-interfaces

while(more)

write-constants

while(more)

write-attributes

while(more)

write-constructors

write-skeleton-body

while(more)

```

write-methods

write-skeleton-body

End

The actual process of generating the sourcecode is handled within the *Generator* class contained in the *Code* package (see Figure 5.6 for more package details). As each class from the structure package contains its associated source file and these classes are arranged in a tree, we recursively traverse the tree to create source artifacts. This way each node in the tree that we examine, that can have code generated for it, already has the code file accessible for writing to. The remaining nodes that do not have associated code files, simply return their information up one level in the tree to the appropriate codified parent node. In the special case of both QoSGuarantee and ResourceRequirement, they are stand alone codified definitions that have no children and do not pass any information to a parent node. They are however used in the service provider as searchable entries, for more information about entries refer to Section 2.4.1. An example of generated sourcecode artifacts is available in Chapter 6.

There are however some limitations to our source generation package. At present, the source code generated by our ADL is not editable within our ADL, the reason being that we do not provide any parsing tool support, which would be required to ensure only valid Java code was inserted. All source that is created though, is fully Java compliant and provides Javadoc recognizable commenting of classes, interfaces and methods. Generated source is only a skeleton representation, meaning there is no functionality for methods,

etc., it is only the valid code representation of the specification. Any functionality for the sourcecode must be added externally to our ADL.

The following chapter provides an example test case that is described in our ADL and its accompanying sourcecode.

## 6. A TEST CASE

This chapter provides an in depth example that exemplifies the service-based paradigm, in that it has a strong dependence upon the remote acquisition of services to carry out a required operation. We start the example by describing the parameters and terminology of the example followed by the specification of the example and viewing the generated XML document, ending with generated sourcecode examples from the specification.

### 6.1 *Defining the test case*

Let us consider an Unmanned Air Vehicle (UAV) example, where we have a threat assessment system contained within an autonomous flying vehicle that can be dynamically reconfigured. A UAV, in this example does not perform its task alone, and is in fact part of a larger fleet of UAVs each with varying threat assessment services. As a UAV encounters threats, if it does not have the required threat assessment service loaded, it can ask one of the other UAVs for it. If available, a UAV can acquire an appropriate assessment service from another UAV and dynamically load it for use.

From the perspective of only a single UAV, this test case demonstrates dynamic system reconfiguration on various levels. A UAV can change only one service or all of its services,



either partially or completely changing its capabilities. Although UAVs can dynamically change their composition depending upon threats encountered, in this test case we have imposed some constraints on a UAV's ability to do so.

Consider the following formal constraints on the UAV system:

1. the small finite memory is defined as having only 200MB of space
2. the differing input stimuli are defined as: Air to Air (AA), Ground to Air (GA), Air to Ground (AG), and Ground to Ground (GG)
3. there is no time lost for the acquisition of services from differing sources
4. it takes 1 second to remove an active service
5. it takes 1 second to acquire a new service
6. it takes 1 second to start a new service

Next, we examine the dynamic reactivity of the UAV to differing input stimulus. Reaction is measured by the UAV dynamically acquiring services that it needs for threat assessment from input stimulus. This example has four types of input (or threats), with each of the available threat deterrent services of a varying size. Each of the services all take the same amount of time to run or provide their service, although this might not always be the case. Below we outline the services available to the UAV.

- AA 100MB: an air to air detection service that requires 100 MB of memory
- AG 90MB: an air to ground detection service that requires 90 MB of memory

- GA 25MB: a ground to air detection service that requires 25 MB of memory
- GG 150MB: a ground to ground detection service that requires 150 MB of memory

Figure 6.1 a proposed UAV and how it can interact with other UAVs in the fleet to acquire and provide services. Each UAV has a simple operating system acting as broker for sensor stimuli and services to and from the active memory. The main memory available to the UAV is broken up into four 50 MB sections, with the ability to combine these sections to make larger sections if needed.

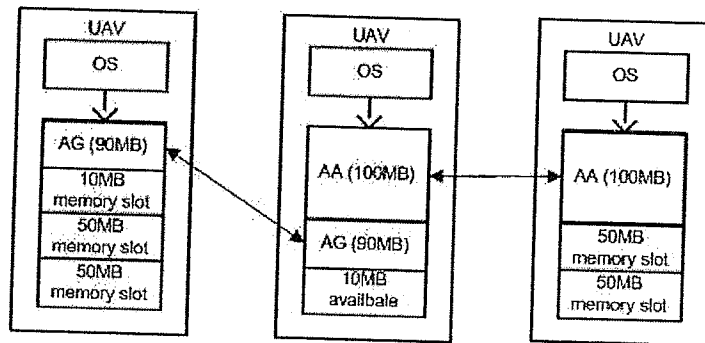


Fig. 6.1: UAV can acquire/provide services from/to others in the fleet.

With a single UAV being part of a larger fleet, not all UAVs will share the exact same services as shown in Figure 6.1. Each UAV could potentially have similar services, but which are provided with differing attributes such as QoS guarantees, memory requirements or executions times. Depending on what a UAV requires for a service it might search for and select one specific service over another. To demonstrate the difference of adding in additional service information, consider Figure 6.2, showing more than one UAV with the same service, but with different attributes.

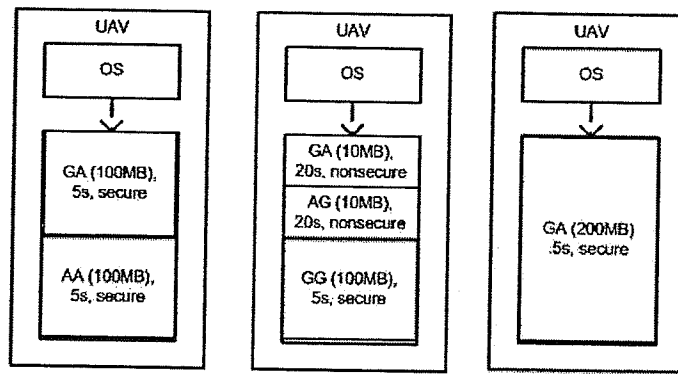
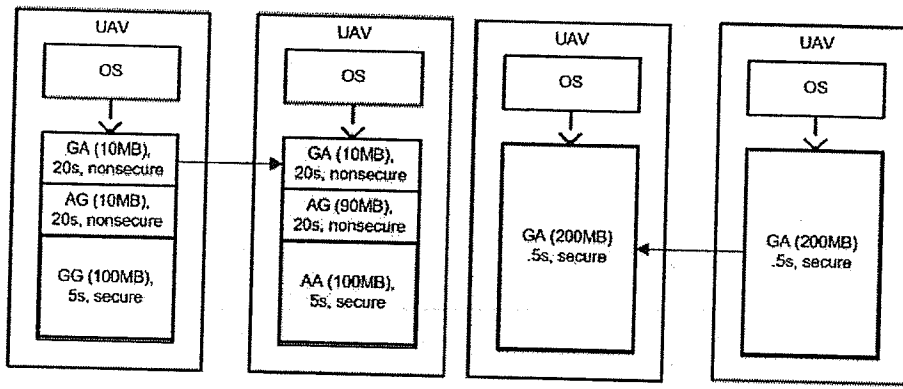


Fig. 6.2: UAVs can have similar service types, with differing attributes.

To illustrate the selection problem, consider that the UAV has AA (100MB) and AG (90MB) services loaded utilizing 190MB of the max 200MB of memory, leaving 10MB of memory free. Now consider that the UAV encounters a GA threat, and needs the GA service. In this case we will assume there are many choices available to the UAV depending on what its criteria are for the required service. If the UAV only required a GA service and did not care about memory, execution speed or security, it could simply take the 10MB GA service, with no further reconfiguration required. If however the UAV had additional criteria, requiring the service to execute in no more than one second, it would then need to select a service and reconfigure itself accordingly to accommodate the new service. In the latter case, depending on which service is selected, the UAV would be partly or completely reconfigured. Figure 6.3 shows a UAV and the choices of services it has available with the level of reconfiguration based on the service selection.

In the following section we create a single UAV specification within our ADL, providing visual references, XML and generated code examples.



(a) No reconfiguration.

(b) Full system reconfiguration.

Fig. 6.3: Service selection can vary UAV system reconfiguration.

## 6.2 Using our ADL to describe the test case

Based on the description of a single UAV above, we can create this example as a single architecture called UAV. We consider service variation by including QoS guarantees and resource requirements for the services of this particular UAV. We also consider that one service might require another to provide its capabilities, we therefore included required services in our UAV.

When creating a new architecture for our UAV, we ensure the presence of the required starting service by forcing one to be created at the time an architecture is created. In this case we will make the OS the starting service. Figure 6.4 shows the newly created architecture with its starting service *OS*.

With an architecture defined, we can then begin adding the services available within our UAV. In this example we will consider that all elements of the UAV are services, even the OS. This allows us to specify the *OS*, *AA\_100MB*, *AG\_90MB*, *GA\_25MB* and the *GG\_150MB* as services of the UAV architecture. However, the OS was added previously as

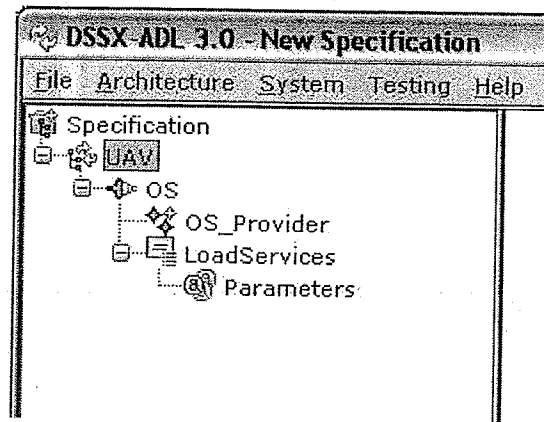


Fig. 6.4: The architecture defined with a starting service.

the starting service for the UAV, so we only have to add the remaining services, and that for this specification we ignore the 200MB memory constraint from the previous section. When we create a service, it must have at least one implementing component and at least one service-message, unless specified as external. Those constraints are enforced when the service is created, by not allowing the creation of a new service until a component and at least one service message have been defined, unless defined as external. We will assume that none of our services are external, meaning they all must contain an implementing component, which will be named with the service name it provides concatenated with the word 'Provider' (example: *AA\_100MB\_Provider*). Lastly, before our services can be created we must include at least one service message. In this case we assume each service (in the beginning) has only an AssessThreat service message, which we define when we create our service. Figure 6.5 shows the new services added to our UAV architecture.

With all of the services for our UAV defined, we can then go back and add more service messages to non-external services. We also have the ability to add QoS guarantees, resource

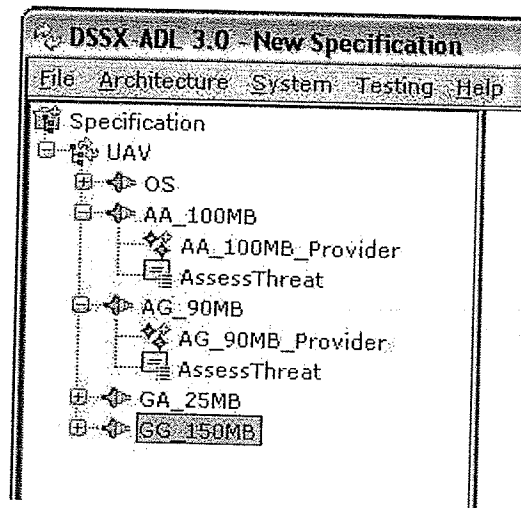


Fig. 6.5: The new services added to the UAV architecture.

requirements, parameters and required services to service messages, while also being able to add QoS guarantees and parameters to required services. In this example we have assumed that each service has its own implementing component and *AssessThreat* service message. Now, for each *AssessThreat* service message we will add *Time* and *Distance* parameters, *SecureComm* and *AssessmentTime* QoS guarantees, a *Memory* resource requirement, and a *ParseThreat* required service.

Lastly for this example, we will add a *ThreatType* parameter and a *Timing* QoS guarantee to the *ParseThreat* required service. Now that we have described our UAV, we can see what the complete specification looks like in Figure 6.6.

Once our specification has been saved it is written to an XML document. The corresponding XML for our UAV example is provided below; some portions have been omitted for brevity.

```
<?xml version="1.0" encoding="UTF-8"?>
```

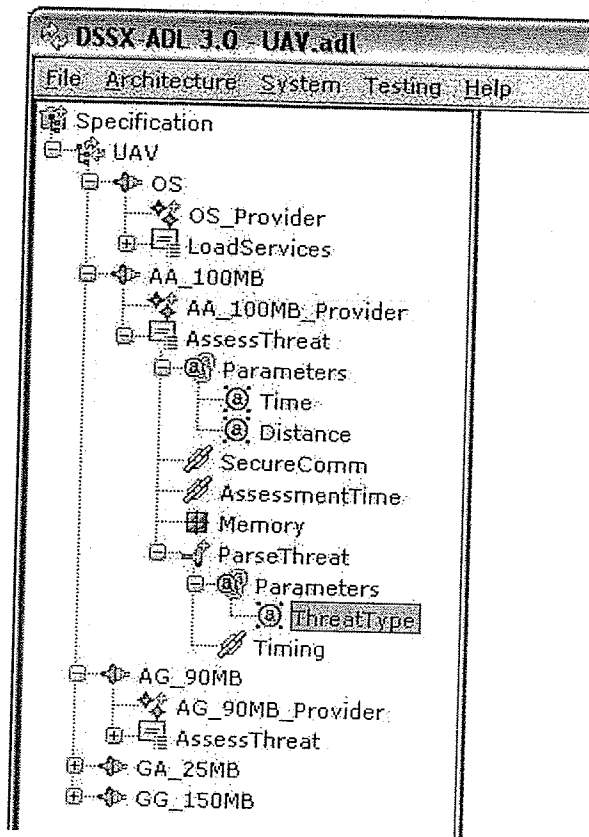


Fig. 6.6: Partial tree view of our UAV design.

```
<!--This document is generated by DSSX-ADL version 3.0-->
<Specification xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://www.cs.umanitoba.ca/~softart/schema-final.xsd">
  <architecture name="UAV" openness="closed">
    <starting-with>
      <service accessibility="global" external="false" name="OS" >
        <component availability="protected" name="OS_Provider" />
        <service-message name="LoadServices">
          <parameter-list name="Parameters"/>
        </service-message>
```

```

    </service>

</starting-with>

<service accessibility="global" external="false" name="AA_100MB" >

    <component availability="private" name="AA_100MB_Provider" />

    <service-message name="AssessThreat">

        <parameter-list name="Parameters">

            <parameter name="Time" value="double"/>

            <parameter name="Distance" value="double"/>

        </parameter-list>

        <qos-guarantee name="SecureComm" value="true"/>

        <qos-guarantee name="AssessmentTime" value="&lt; 5 seconds"/>

        <required-resource name="Memory" value="20 MB of RAM"/>

        <required-service immediacy="immediate" location="local" name="ParseThreat">

            <parameter-list name="Parameters">

                <parameter name="ThreatType" value="String"/>

            </parameter-list>

            <qos-guarantee name="Timing" value="&lt; 1 second for result"/>

        </required-service>

    </service-message>

</service>

<service accessibility="global" external="false" name="AG_90MB" >

    <component availability="private" name="AG_90MB_Provider" />

.....

</service>

</architecture>

</Specification>

```



With the specification saved and the XML file written, our ADL provides support for code generation. In this example, when run, the code generation tools create one package for the UAV architecture, one remote interface for each of the services and one implementation file for each of the components (see Table 5.1 and Section 5.3 for more details on code generation). The generated code files can be viewed within our ADL, following is an example of the generated remote interface for the *AA\_100MB* service.

```
package UAV;

import java.rmi.Remote;
import java.rmi.RemoteException;

// This document is generated by DSSX-ADL version 3.0
/**
 * This file describes the skeleton for AA_100MB
 *
 * COMMENTS:
 */
public interface IAA_100MB extends Remote {

    public static final String ASSESSMENTTIME = "< 5 seconds";
    public static final String MEMORY = "20 MB of RAM";
    public static final String SECURECOMM = "true";

    /**
     * interface method AssessThreat
```

```

* @todo implement method AssessThreat in alternate class file
*
* @param Time double
* @param Distance double
* @throws RemoteException
*/

public void AssessThreat(double Time, double Distance) throws RemoteException;

} //end interface

```

An example of the generated sourcecode of the providing component, *AA\_100MB\_Provider* for the *AA\_100MB* service is shown below.

```

package UAV;

import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import net.jini.lookup.ServiceIDListener;
import net.jini.core.lookup.ServiceID;
import net.jini.core.discovery.LookupLocator;
import net.jini.lookup.JoinManager;
import net.jini.discovery.DiscoveryGroupManagement;
import net.jini.core.entry.Entry;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lookup.entry.Name;
import net.jini.lookup.entry.ServiceInfo;
import net.jini.discovery.LookupDiscoveryManager;

```

```

import net.jini.lookup.entry.Comment;

import java.rmi.RMISecurityManager;

import java.io.Serializable;


// This document is generated by DSSX-ADL version 3.0

/**
 * This file describes the skeleton for AA_100MB_Provider
 *
 * COMMENTS:
 */

public class AA_100MB_Provider extends UnicastRemoteObject implements
    IAA_100MB, Serializable, ServiceIDListener {

/**
 * This is the default constructor
 */

    public AA_100MB_Provider() {
    } //end constructor


/**
 * interface method AssessThreat
 *
 * @todo implement method AssessThreat in alternate class file
 *
 * @param Time double
 * @param Distance double

```

```

    * @throws RemoteException
    */

    public void AssessThreat(double Time, double Distance) throws RemoteException {
        return null;
    }

    /**
     * skeleton of method ServiceIdNotify
     * @todo complete method ServiceIdNotify
     *
     * @param serviceid ServiceId
     */
    public void ServiceIdNotify(ServiceId serviceid) {
    }

    /**
     * skeleton of method getServiceId
     * @todo complete method getServiceId
     *
     * @return ServiceId
     */
    public ServiceId getServiceId() {
        return null;
    }

    /**

```

```

    * skeleton of method toString

    * @todo complete method toString

    *

    * @return String

    */

    public String toString() {

        return null;

    }

} //end class

```

Finally the example sourcecode for how we implement our QoS and resource requirements. As mentioned they are implementors of the *AbstractEntry* interface from within Jini, which then allows them to be used as both searchable criterion for clients and descriptors for service providers.

```

package UAV;

import net.jini.entry.*;

import net.jini.lookup.entry.*;

// This document is generated by DSSX-ADL version 3.0

/**

    * This file describes the skeleton for SecureComm

    *

```

```

    * COMMENTS:

    */

public class SecureComm extends AbstractEntry implements
    ServiceControlled {

private String value = null;

/**

    * This is the default constructor

    * @param value String

    */

    public SecureComm(String value) {

        this.value = value;

    } //end constructor

/**

    * skeleton of method toString

    * @todo complete method toString

    *

    * @return String

```

```
*/  
  
public String toString() {  
    return null;  
}  
  
} //end class
```

## 7. CONCLUSIONS

The problem of ever increasing monolithic software systems, and the push more and more to service-based computing, drives the need for tools to create these systems.

In my thesis I have created a service-based architecture description language for use to help design, implement, validate, refactor and evolve these service-based systems. Specific focus of our ADL was given to flexibility in service-based systems, and to providing full support for dynamically reconfigurable systems.

We maintained the focus on flexibility by taking the component-based system view and changing it, to stipulate that components are only vehicles which provide implementations for services. In doing so, we made every part of a system a service, which could have various locations and resource requirements. This then captured the essence of the service-based paradigm, in that all systems are composed of interacting services.

### 7.1 *Future work*

With the current version of our ADL we can create specifications, edit and validate them and even have some sourcecode created. From the results of test examples, like Chapter 6.2, we have uncovered some remaining issues within our first iteration. Identifying



and addressing these areas of improvement and correcting any outstanding issues is the direction for future work on our ADL. Some of these issues and what our solution will be in the next iteration are outlined below.

- *Runtime Environment*: We need to include a runtime environment for testing the specification in a 'live' setting. This would require the ability to start and configure Jini services to run in conjunction with our ADL. We must be able to demonstrate the dynamism of our specifications in terms of loading, unloading and searching for services. With Jini, this is handled for us, as services can be local or remote, searched for, downloaded (or not), and (de)activated at runtime. Using the Java classes for reflection, we will include the ability to run the Jini environments by running the associated jar files as though they were run from the command line. A path collection mechanism will be created to allow users to specify where the required libraries are located to run Jini services. Another option would be to use the tool provided by Jini that presents a graphical mechanism for starting Jini services.
- *Interoperability*: Our ADL is represented in XML, which can make it interoperable with other ADLs and frameworks like: xADL [23] - which then provides interoperability with ArchStudio [42], ACME [13], xC2 [23], and OSGi [43]. To achieve interoperability, we will use XSLT transforms, giving us the ability to convert our XML representations to that of any other ADL that uses XML or similar markup languages.
- *Expanded Code Generation*: Currently users can generate and view source skeletons

for selected elements, but this is not sufficient for a useful design environment. We must expand our code generation and handling to include the ability to edit the code within our ADL, parsing of code and insertion of code snippets via element properties; meaning the operations of methods could be declared during design time, and be inserted into the code for the user. This issue will be addressed by either porting the ADL to Eclipse, or by using the Eclipse editor libraries within our current implementation.

- *Running Generated Code:* With generated code and the addition of the Jini runtime environment, having the ability to actually execute the generated code to evaluate its suitability. This would require the compilation of code within our ADL as well as accessing any external resources for running the compiled code. In our case that would mean the inclusion of the Java compiler (javac) and launching the JRE from within our ADL to run code.
- *Expanded Code Support:* While currently our code generation is targeted towards Jini, it would be ideal if there was more than one type of code that could be generated. Perhaps the inclusion of code support for other infrastructures such as CORBA could be incorporated.

The ultimate goal of this work is to take our ADL and code generation tools and deploy them as a plugin for the Eclipse Project [18]. Such a plugin would include the use of perspectives within Eclipse as well as a graphical editor for specifications using the Eclipse Graphical Editor Framework (GEF) [19].

## APPENDIX

## A. XML SCHEMA DEFINITION

Following is a full description of the XML schema representing our ADL language. It is broken down into the individual elements of the schema for easier understanding. The schema was designed using global elements to eliminate duplicate definitions, and includes an id system to allow for ID/IDREFS to correlate certain elements.

The specification element is the root element of the schema and only contains one child type, an architecture.

```
<xs:element name="Specification">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="architecture" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

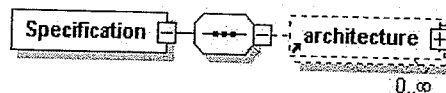


Fig. A.1: Schema element *specification*.

Architecture elements are composed of up to three types of children, there is one

starting-with child which must always be present, and there can be zero or more service or architecture children.

```
<xs:element name="architecture">

  <xs:complexType>

    <xs:sequence>

      <xs:element name="starting-with">

        <xs:complexType>

          <xs:sequence>

            <xs:element ref="service"/>

          </xs:sequence>

        </xs:complexType>

      </xs:element>

      <xs:element ref="service" minOccurs="0" maxOccurs="unbounded"/>

      <xs:element ref="architecture" minOccurs="0" maxOccurs="unbounded"/>

    </xs:sequence>

    <xs:attribute name="id" type="xs:ID" use="required"/>

    <xs:attribute name="documentation" type="xs:string"/>

    <xs:attribute name="openness" use="required">

      <xs:simpleType>

        <xs:restriction base="xs:string">

          <xs:enumeration value="open"/>

          <xs:enumeration value="closed"/>

        </xs:restriction>

      </xs:simpleType>

    </xs:attribute>

    <xs:attribute name="name" type="xs:string" use="required"/>

  </xs:complexType>

</xs:element>
```

```

</xs:complexType>

</xs:element>

```

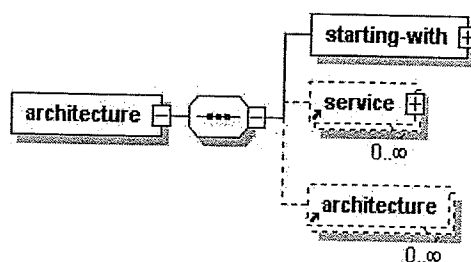


Fig. A.2: Schema element *architecture*.

There is only one starting-with element per architecture element, and it is used to declare the service that acts as the starting service for that particular architecture.

```

<xs:element name="starting-with">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="service"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

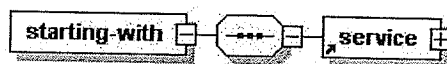


Fig. A.3: Schema element *starting-with*.

Services are described as components and service messages, there must one component child, and there must be at least one service-message child.

```

<xs:element name="service">

  <xs:complexType>

    <xs:sequence minOccurs="0">

      <xs:element ref="component"/>

      <xs:element ref="service-message" maxOccurs="unbounded"/>

    </xs:sequence>

    <xs:attribute name="id" type="xs:ID" use="required"/>

    <xs:attribute name="documentation" type="xs:string"/>

    <xs:attribute name="accessibility" use="required">

      <xs:simpleType>

        <xs:restriction base="xs:string">

          <xs:enumeration value="local"/>

          <xs:enumeration value="global"/>

        </xs:restriction>

      </xs:simpleType>

    </xs:attribute>

    <xs:attribute name="external" type="xs:boolean" use="required"/>

    <xs:attribute name="provided-by-component" type="xs:IDREFS" use="optional"/>

    <xs:attribute name="name" type="xs:string" use="required"/>

  </xs:complexType>

</xs:element>

```

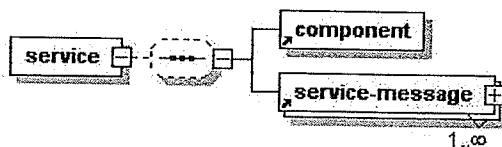


Fig. A.4: Schema element *service*.

Components do not have any children, and act simply as an information store.

```
<xs:element name="component">
  <xs:complexType>
    <xs:attribute name="id" type="xs:ID" use="required"/>
    <xs:attribute name="documentation" type="xs:string"/>
    <xs:attribute name="availability" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="private"/>
          <xs:enumeration value="public"/>
          <xs:enumeration value="protected"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="provides-service" type="xs:IDREFS" use="optional"/>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
```



Fig. A.5: Schema element *component*.

Service-messages are children of only services, with the information they store describing the remote methods of the parent service.

```
<xs:element name="service-message">
```



```

<xs:complexType>

  <xs:sequence>

    <xs:element ref="parameter-list"/>

    <xs:element ref="qos-guarantee" minOccurs="0" maxOccurs="unbounded"/>

    <xs:element ref="required-resource" minOccurs="0" maxOccurs="unbounded"/>

    <xs:element ref="required-service" minOccurs="0" maxOccurs="unbounded"/>

  </xs:sequence>

  <xs:attribute name="id" type="xs:ID" use="required"/>

  <xs:attribute name="documentation" type="xs:string"/>

  <xs:attribute name="name" type="xs:string" use="required"/>

</xs:complexType>

</xs:element>

```

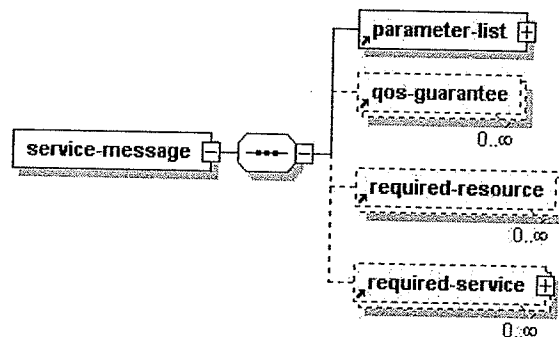


Fig. A.6: Schema element *service-message*.

The parameter-list element behaves similarly to the starts-with element, in that it only acts as a wrapper, in this case for Parameter elements.

```

<xs:element name="parameter-list">

  <xs:complexType>

    <xs:sequence>

```

```

    <xs:element ref="parameter" minOccurs="0" maxOccurs="unbounded"/>

</xs:sequence>

<xs:attribute name="id" type="xs:ID" use="required"/>

<xs:attribute name="name" type="xs:string" use="required"/>

<xs:attribute name="documentation" type="xs:string"/>

</xs:complexType>

</xs:element>

```

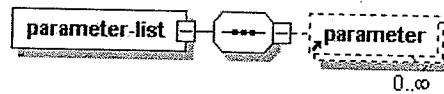


Fig. A.7: Schema element *parameter-list*.

The parameter element describes a name type pair like (int, arg) and is contained only as a child of a parameter-list element.

```

<xs:element name="parameter">
  <xs:complexType>
    <xs:attribute name="id" type="xs:ID" use="required"/>
    <xs:attribute name="value" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="int"/>
          <xs:enumeration value="double"/>
          <xs:enumeration value="String"/>
          <xs:enumeration value="void"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>

```

```

    </xs:simpleType>

  </xs:attribute>

  <xs:attribute name="name" type="xs:string" use="required"/>

  <xs:attribute name="documentation" type="xs:string"/>

</xs:complexType>

</xs:element>

```

**parameter**

Fig. A.8: Schema element *parameter*.

A QoS-guarantee describes a searchable constant which provides information about either service-messages, or required-services.

```

<xs:element name="qos-guarantee">

  <xs:complexType>

    <xs:attribute name="id" type="xs:ID" use="required"/>

    <xs:attribute name="value" type="xs:string" use="required"/>

    <xs:attribute name="name" type="xs:string" use="required"/>

    <xs:attribute name="documentation" type="xs:string"/>

  </xs:complexType>

</xs:element>

```

**qos-guarantee**

Fig. A.9: Schema element *qos-guarantee*.

Required resources are similar to QoS-guarantees, in that they describe information about a service message. In this case though, they specifically describe any additional resources that may be required by a service message.

```
<xs:element name="required-resource">
  <xs:complexType>
    <xs:attribute name="id" type="xs:ID" use="required"/>
    <xs:attribute name="value" type="xs:string" use="required"/>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="documentation" type="xs:string"/>
  </xs:complexType>
</xs:element>
```

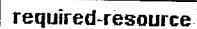


Fig. A.10: Schema element *required-resource*.

Service-messages can have external requirements as well, and those can take the form of other services that are required to provide a service for another service. The required-service element describe such dependencies.

```
<xs:element name="required-service">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="parameter-list"/>
      <xs:element ref="qos-guarantee" maxOccurs="unbounded"/>
    </xs:sequence>
```

```

<xs:attribute name="id" type="xs:ID" use="required"/>
<xs:attribute name="documentation" type="xs:string"/>
<xs:attribute name="location" use="required">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="local"/>
      <xs:enumeration value="remote"/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
<xs:attribute name="immediacy" use="required">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="immediate"/>
      <xs:enumeration value="optional"/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
<xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>
</xs:element>

```

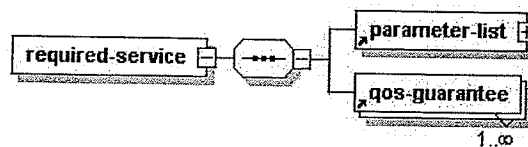
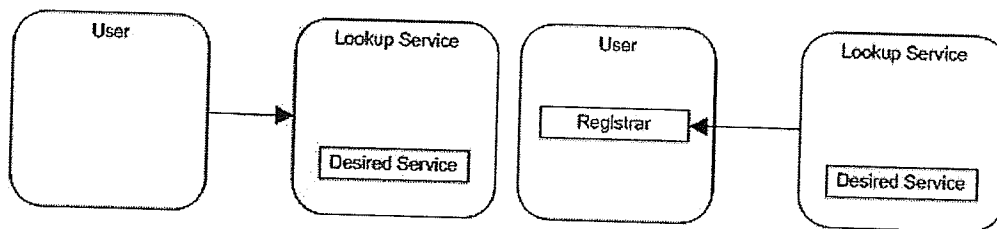


Fig. A.11: Schema element *required-service*.

## B. NOTES ON JINI

There are many technical aspects of Jini, most important to our research is the search and selection process of the Jini lookup service *Reggie*. Reggie, exists on the network – either a LAN or the internet, as itself a service, which provides the capability for users (whether they are human or other services) to lookup and acquire services. The following four figures outline the basic process of getting the lookup service and then getting a desired service. Simply, this process involves the user requesting a service locator –Figures B.1(a) and B.1(b), using the locator to find the desired service –Figure B.2(a) and acquiring the service –Figure B.2(b).



(a) Asking for a locator.

(b) Getting a locator.

Fig. B.1: Searching for a service.

The services that are provided for the lookup service is handled in an almost identical way, except that service providers provide their services to the lookup service, instead of acquiring services for use. An important point here is that there is nothing stopping a

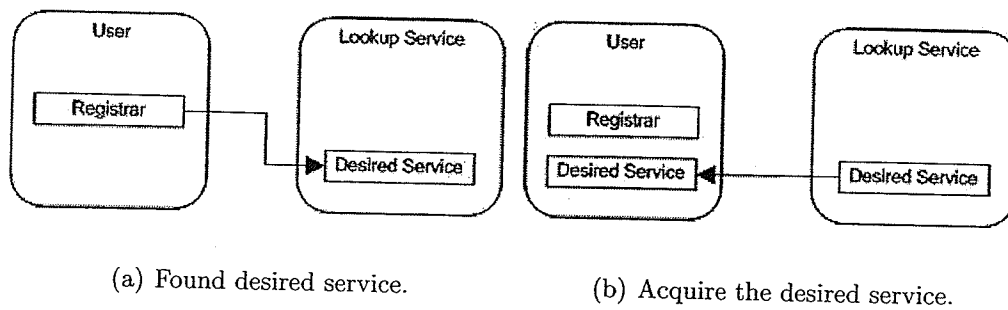


Fig. B.2: Finding and acquiring services.

service provider from being a user as well. The basic idea is the same for providing services; first the provider asks for a locator and then uses the locator to make the provider's service available on the lookup service.

Another provision is the use of proxy servers. Using the lookup service in this manner closely resembles how clients and service providers interact with the UDDI in web services. Figure B.3 outlines this interaction. First the user and the lookup service communicate, and so do the provider and the lookup service. Once the service request and provider have been satisfied, the user is then directed to the service location, at which point the user proxy interacts directly with the desired service.

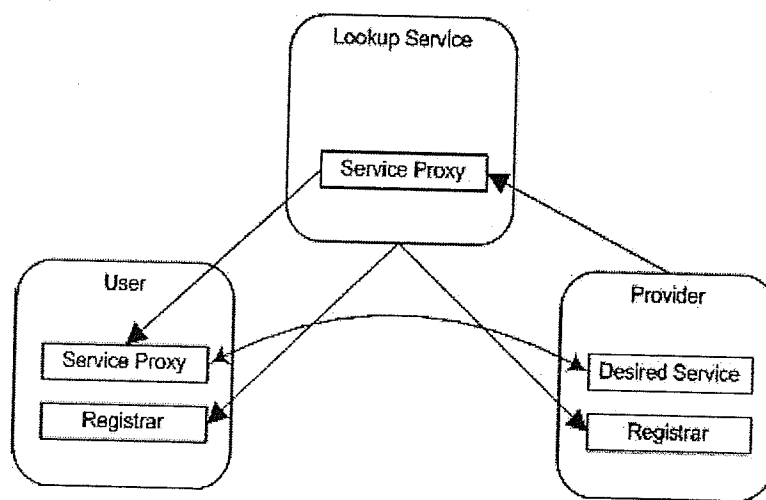


Fig. B.3: Jini proxy usage.



## BIBLIOGRAPHY

- [1] G.D. Abowd, R. Allen, and D. Garlan. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Trans. Softw. Eng. Methodol.*, 4(4):319–364, 1995.
- [2] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, May 2002.
- [3] R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71–80, Sorrento, Italy, May 1994.
- [4] R. Anzbock, S. Dustdar, and H. Gall. Software Configuration, Distribution, and Deployment of Web Services. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 649–656, Ischia, Italy, 2002.
- [5] K. Ballinger. *.NET Web Services: Architecture and Implementation*. Addison Wesley, Menlo Park, CA, 2003.
- [6] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. The SEI Series in Software Engineering. Addison-Wesley, Reading, MA, 2nd edition, 2002.

- [7] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide*. Addison-Wesley, Boston, MA, 2003.
- [8] H. Cervantes and R.S. Hall. A Framework for Constructing Adaptive Component-Based Applications: Concepts and Experiences. In *Proceedings 7th International Symposium on Component-Based Software Engineering*, pages 130–137, Edinburgh, UK, May 2004.
- [9] H. Cervantes and R.S. Hall. Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model. In *Proceedings of the 26th International Conference on Software Engineering*, pages 614–623, Edinburgh, UK, May 2004.
- [10] C. Chaudet, R.M. Greenwood, F. Oquendo, and B.C. Warboys. Architecture-driven Software Engineering: Specifying, Generating, and Evolving Component-based Software Systems. In *IEEE Software Proceedings*, pages 203–214, 2000.
- [11] F. Chen, Q. Wang, H. Mei, and F. Yang. An Architecture-based Approach for Component-oriented Development. In *Proceedings of the 26th Annual International Computer Software and Applications Conference*, pages 450–455, 2002.
- [12] P. Clements. A Survey of Architecture Description Languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 16–25, Paderborn, Germany, March 1996.
- [13] D. Garlan, R. Monroe, and D. Wile. ACME: an Architecture Description Interchange

- Language. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research*, pages 7–21, Toronto ON, Canada, November 1997.
- [14] M. Gorlick and R. Razouk. Using Weaves for Software Construction and Analysis. In *Proceedings of the 13th international conference on Software engineering*, pages 23–34, 1991.
- [15] K. Gottschalk, S. Graham, H. Kreger, and J. Snell. Introduction to Web Services Architecture. *IBM Systems Journal*, 41(2):170–177, 2002.
- [16] C. Hofmeister, R.L. Nord, and D. Soni. Describing Software Architecture with UML. In *Proceedings of the First Working IFIP Conference on Software Architecture*, pages 1–10, 1999.
- [17] R.C. Holt. Software Architecture Abstraction and Aggregation as Algebraic Manipulations. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, pages 5–9, 1999.
- [18] IBM. The Eclipse Project. <http://www.Eclipse.org>, 2004.
- [19] IBM. The Graphical Editor Framework Project. <http://www.eclipse.org/gef/>, 2004.
- [20] Institute for Software Research. ArchStudio. <http://www.isr.uci.edu/projects/archstudio>, 2004.
- [21] Institute of Software Research. ISR Software Architecture Research. <http://www.isr.uci.edu/architecture/adl/ACN.html>.

- [22] Institute of Software Research. ISR Software Architecture Research.  
<http://www.isr.uci.edu/architecture/adl/ADN.html>.
- [23] R. Khare, M. Gunterdorfer, P. Oreizy, N. Medvidovic, and R.N. Taylor. xADL: Enabling Architecture-centric Tool Integration with XML. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, page 9, January 2001.
- [24] J. Kramer and J. Magee. Distributed Software Architectures. In *Proceedings of the 19th International Conference on Software Engineering*, pages 633–634, Boston MA, USA, 1997.
- [25] D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, Boston, MA, 2002.
- [26] D.C. Luckham, J.J Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–354, April 1995.
- [27] D.C. Luckham and J. Vera. An Event-based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.
- [28] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering.*, pages 3–14, San Francisco, California, USA, 1996.
- [29] N. Medvidovic. On the Role of Middleware in Architecture-based Software Develop-

- ment. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 299–306, Ischia, Italy, July 2002.
- [30] N. Medvidovic, D.S. Rosenblum, and R.N. Taylor. A Language and Environment for Architecture-based Software Development and Evolution. In *Proceedings of the 21st International Conference on Software Engineering ICSE1999*, pages 44–53, Los Angeles California, USA, 1999.
- [31] N. Medvidovic and R.N. Taylor. Separating Fact From Fiction in Software Architecture. In *Proceedings of the 3rd international workshop on Software architecture*, pages 105–108, 1998.
- [32] N. Medvidovic and R.N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.
- [33] N.K. Mukhi, R. Konuru, and F. Curbera. Cooperative Middleware Specialization for Service Oriented Architectures. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 206–215, 2004.
- [34] E. Newcomer and G. Lomow. *Understanding SOA with Web Services*. Addison Wesley, Boston, MA, USA, 2004.
- [35] J. Newmarch. Jan Newmarch's Guide to Jini Technologies.  
<http://pandonia.canberra.edu.au/java/jini/tutorial/Jini.xml>, 2004.

- [36] S. Oaks and H. Wong. *Jini in a Nutshell: A Quick Reference*. O'Reilly, Sebastopol, CA, 2000.
- [37] OMG. Common Object Request Broker Architecture: Core Services, Revision 3.0. OMG Document no. 91.12.1, Object Management Group, December 2002.
- [38] F. Oquendo. Formally Refining Software Architectures with  $\pi$ -ARL: A Case Study. *ACM SIGSOFT Software Engineering Notes*, 29(5), September 2004.
- [39] F. Oquendo.  $\pi$ -ADL: An Architecture Description Language based on the Higher-Order Typed  $\pi$ -Calculus for Specifying Dynamic and Mobile Software Architectures. *ACM SIGSOFT Software Engineering Notes*, 29(4), May 2004.
- [40] F. Oquendo.  $\pi$ -ARL: An Architecture Refinement Language for Formally Modeling the Stepwise Refinement of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, 29(5), September 2004.
- [41] P. Oreizy, M. Gorlick, R.N Taylor, D. Heimburger, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An Architecture Approach to Self-adaptive Software. *IEEE Intelligent Systems and Their Applications*, 14(3):54–62, May 1999.
- [42] P. Oreizy, N. Medvidovic, and R.N Taylor. Architecture-based Runtime Software Evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE98)*, pages 177–186, Kyoto, Japan, April 1998.
- [43] OSGi Alliance. Open Services Gateway Initiative . <http://www.osgi.org/>, 2005.

- [44] Mónica Pinto, Lidia Fuentes, and Jose María Troya. DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-based Development. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 118–137, Erfurt, Germany, 2003.
- [45] J.E. Robbins, N. Medvidovic, D.F. Redmiles, and D.S. Rosenblum. Integrating Architecture Description Languages with a Standard Design Method. In *Proceedings of the 20th International Conference on Software Engineering*, pages 209–218, 1998.
- [46] B. Schmerl and D. Garlan. AcmeStudio: Supporting Style-centered Architecture Development. In *Proceedings of the 26th International Conference on Software Engineering*, pages 704–705, 2004.
- [47] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River NJ, 1996.
- [48] Sun Microsystems, Inc. Jini Network Technology. <http://www.sun.com/software/jini>, 2004.
- [49] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, Reading, MA, 2nd edition, 2003.
- [50] S. Varadarajan. The Weaves Runtime Framework. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, pages 197–205, April 2004.
- [51] V. Misic, D. Ibrahim, and M. Rennie. Towards Community-Based Web Service Discov-

ery and Selection. In *Proceedings of the Web Services Workshop MWSW05*, page 1, 2004.

[52] W3C. Simple Object Access Protocol (SOAP) 1.2. online documentation, W3C, 2004.

[53] World Wide Web Consortium (WC3). XML Pointer Language.  
<http://www.w3.org/TR/xptr>, 2003.

[54] World Wide Web Consortium (WC3). XML Schema.  
<http://www.w3.org/XML/Schema>, 2004.

[55] World Wide Web Consortium (WC3). XML Path Language.  
<http://www.w3.org/TR/xpath20/>, 2005.