Acknowledgments

First of all, I would like to thank my supervisor Prof. Bengt Lennartsson for commenting on my work and turning my, sometimes, wild ideas into more realistic ones. I also would like to thank Prof. Peter Fritzson for accepting me as a Ph.D student and for continuous support throughout these years, and Prof. Jan Bosch for comments on my work and numerous discussions on software architecture.

A very special "thank you" to the people at the Department of Mathematics, Statistics, and Computer Science at Växjö University, especially Ulf Cederling, Mathias Hedenborg, and Gösta Sundberg.

Then I would like to thank all friends and colleagues at PELAB, especially Tobias Ritzau that made traveling and being away from home more bearable and for all "joyous acclamations" at the office.

I also would like to thank all that contributed with comments on this work. Especially Richard Olson that helped me with proof-reading the thesis and for asking the questions no one else would ever do, Patrik Nordling for input from "the real world", and Tommy Persson and David Byers for help with layout and LaTeX.

I would also like to express my appreciation to Bodil Mattsson-Kihlström, Gunilla Norbäck, and Lillemor Wallgren, helping me solving all administrative problems.

Finally, and most important of all, thank you, Susanne for all support and encouragement even though I have been away from home too much, and my family for believing in me and for supporting me.

Jesper Andersson
Linköping, April 1999

ABSTRACT

Advances in software engineering technology continue to assist engineers in building more complex systems than before, but even if these systems are more complex they sustain the same problems of previous developments. First, they are never error-free and second, they need continuous improvement to meet users requests for more and improved functionality. These changes often result in downtime, where users can not use the system. In other types of systems downtime can be interdicted, for instance the switching systems in the telecommunications domain. A dynamic reconfigurable system that updates or replaces software without stopping the executing applications is needed.

This thesis discusses support for the modeling and implementation of software systems with a dynamic architecture. Using software architectures as a basis for reasoning about run-time modification makes it possible to use a different approach, dynamic reconfiguration at the architectural level. Introducing architectural agents allows for developers to specify reactiveness in architectural specifications.

The introduction of the architectural agent, adds a new dimension to traditional architectural specifications. Agents allow designers to express run-time evolution which is initiated and governed internally in the executing application. The methodology for architectural agents is simple yet capable of describing systems with complex dynamic behavior.

CONTENTS

1 Introduction **1**
1.1 Mastering Complexity . . . 2
1.2 Software models . . . 3
1.2.1 Structured models . . . 3
1.2.2 Object-oriented models . . . 4
1.2.3 Software Architecture . . . 5
1.3 Research question . . . 5
1.4 Structure of the thesis . . . 6

2 Software Architecture **7**
2.1 Architectural models . . . 8
2.1.1 Perry and Wolf (1992) . . . 8
2.1.2 Garland and Shaw (1993) . . . 9
2.1.3 Boehm et al. (1995) . . . 9
2.1.4 Soni, Nord, and Hofmeister (1995) . . . 10
2.1.5 Krutchen (1995) . . . 11
2.1.6 Bushmann et al. (1996) . . . 12
2.1.7 Bass, Clements, and Kazman (1997) . . . 12
2.1.8 Conclusion . . . 13
2.2 Architecture Description Languages . . . 14
2.2.1 The ADL classification framework . . . 14
2.2.2 UniCon . . . 19
2.2.3 ACME . . . 21
2.2.4 C2 . . . 22
2.2.5 SADL . . . 23
2.2.6 Wright . . . 24
2.2.7 Darwin . . . 24
2.2.8 Conclusions . . . 25
2.3 Architectural styles . . . 25

2.3.1 Some architectural styles . . . 26
2.4 Architecture Design Processes . . . 28
2.4.1 ATAM . . . 28

3 Related Technology **33**
3.1 Aspect Oriented Programming . . . 33
3.2 Software components . . . 35
3.2.1 Component definitions . . . 35
3.2.2 Existing component infrastructures . . . 37
3.3 Frameworks . . . 44
3.4 Patterns . . . 44
3.5 Conclusions . . . 47

4 Adding reactiveness to architectures **49**
4.1 The simulation architecture . . . 50
4.2 Dynamic reconfiguration . . . 51
4.3 Architectural reconfiguration . . . 52
4.3.1 Structural reconfiguration . . . 53
4.3.2 Updating reconfiguration . . . 53
4.3.3 Initiate dynamism . . . 54
4.4 Support in existing ADLs . . . 56
4.4.1 Wright . . . 56
4.4.2 Darwin . . . 57
4.4.3 C2 . . . 57
4.4.4 Rapide . . . 58
4.4.5 Conclusion . . . 58
4.5 Requirements for a development environment . . . 58
4.5.1 Design support . . . 58
4.5.2 Implementation support . . . 59
4.5.3 Run-time support . . . 59
4.6 Architectural Agents . . . 59
4.6.1 The Structure of Agents . . . 60
4.6.2 Agent–Architecture interaction . . . 61
4.6.3 Configuring agents — rules . . . 63
4.6.4 Cooperating agents . . . 63
4.6.5 Agents in specifications . . . 64
4.6.6 Agents in implementations . . . 65
4.7 A reactive specification . . . 67
4.7.1 Invocation of the optimizer . . . 68
4.7.2 Status change on Workstation . . . 69

4.8 A reactive implementation 69
4.8.1 More about the IBM Agent Building Environment 70
4.8.2 A simple versioning agent 73

5 Conclusions and Future Work 77
5.1 Conclusions 77
5.2 Future work 78
5.2.1 Methodology 78
5.2.2 Validation 80

Chapter 1

INTRODUCTION

As the complexity of software systems has increased, the need for more efficient methods and tools has emerged. Software reuse has been proposed as one way to produce systems faster and with improved quality.

A vision of the software engineering community has been composition of reused components, developed in-house with standard off-the-shelf components [1, 2]. Several researchers have claimed that software composition will be one of the bases for the software industry in the future [3]. One problem with this approach is the lack of support in methodology, languages, and tools for more complex compositions [4]. The standard composition mechanism has been an ordinary subroutine call.

In the area of software architectures a system is described in terms of components and connectors. Components are assembled into systems using connectors. The developer can choose from a pre-defined set of connectors when constructing the system. Complex connectors have been defined and both tools and composition languages have been developed. Existing work in the software architecture community is focused on static descriptions.

Requirement changes, environmental changes, system defects, and unexpected changes in a system's operational environment often require overall system changes. In certain critical situations, dynamic architectural reconfiguration is an absolute requirement.

In this thesis we are interested in support for the modeling and implementation of software systems with a dynamic architecture. Using software architectures as a basis for reasoning about run-time modification makes it possible to use a different approach, dynamic reconfiguration at the architectural level.

1.1 Mastering Complexity

Complex systems have interested scientists for thousands of years. Some scientists have been principally interested in understanding and describing particular complex systems themselves while others have been more interested in the philosophical aspects of underlying complexity itself. For example, Simon [5] is interested in the "architecture of complexity", looking for patterns in complex systems. The key property of complex systems identified by Simon is the hierarchical structure, or a system is a composition of sub-systems. This hierarchy is used by humans to better understand, describe and construct complex systems. Starting at a high level, using abstractions to represent underlying function, and proceeding to delineate higher functions by breaking them into their sub-functions has been found best approach to understanding and describing complex systems.

Software systems are often classified as complex. Parnas [6] identifies important software structures and lists a set of goals for a modular structure. In particular he underlines the importance of abstraction in software development. Abstractions are described by interfaces and he emphasises that to fully understand a system at a given level of abstraction we do not need to understand the detailed implementation of a function, only its interfaces.

F.P. Brooks identifies four inherent properties of software systems and development projects contributing to the complexity [7, p. 182]. Firstly there is the *complexity* of the software itself. Brooks claims that this is always present. He claims that the reason for the inherent complexity is that no two entities in a system are alike. The second property is *conformity*. Software exists as a part in larger systems and must exhibit a number of different interfaces to surrounding systems. This leads to the fact that small changes in the execution environment or adjacent components often require a component to be modified. Third, software is invisible. When software is developed it is impossible to directly study how the development is proceeding. This can only be based on descriptions of the system, such as design documents or source code. And fourth, software often changes rapidly. Not only when it is under development but throughout its entire life-cycle.

The tools for mastering complexity are abstraction and decomposition. These techniques are based on the general concept of splitting complexity into several parts, each part being less complex than the total. Abstraction is defined by the IEEE as " A view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information" [8]. Abstraction and decomposition are key techniques used when developers create their software models, the description of what is to be developed.

1.2 Software models

"Software is invisible", was one of Brooks conclusions. Unveiling the hidden structures of software has been and still is the main concern for the software engineering community. Software development concerns building and refining models. Starting with abstract problem descriptions, these are gradually refined into intermediate models which finally results in the production of an implementation model in a specific programming language.

Over the years new programming languages have introduced the capability of using behavioral abstractions, i.e. subroutines and functions. Another common abstraction mechanism involves the use of data abstractions, which allow a developer to extend the type system of a language. Data abstraction was later extended into abstract data types, where a type was packaged together with a set of operations on that type. Object orientation expanded abstract data types with a capability to organize types in hierarchies reflecting commonality among different types.

Parallel to language development, new modeling methods have been developed. Each method includes notations for expressing different models, a process description that describe the activities step-wise, and a set of associated tools. There are two classes of models, static and dynamic. Static models capture the invariant structures of software. Dynamic models, such as Petri-nets and finite state machines have been used to capture dynamic aspects, for instance concurrent operations.

To improve the quality of software products more advanced techniques — better software models — for describing software and its operations has, and continues to be, a major goal. As time has passed, the expressiveness, the capability of fuller and more precise description, has been greatly improved, following behind but paralleling the improvements in programming languages themselves. However, much remains to be done.

1.2.1 Structured models

Languages developed in the 60's and early 70's introduced abstractions for control, data, and functions . These constructs aided programmers in the design of algorithms. Programmers used *flow-charts* to graphically depict and document the structure of an individual algorithm. Design and formulation of algorithms was regarded as a complicated process [9, p. 125]. The general approach used was to divide the process into a set of sub-processes that together provided the problem solution. This algorithmic or structured decomposition technique was later expanded and generalized by Jackson [10]. Jackson presented a graphical notation based on three elementary constructs; sequence, iteration and selection, all supported in existing languages. These where used to create JSP-diagrams, graphical models that depicted the structure of an application. JSP diagrams are hierarchical and each process is divided into sub-processes, these sub-processes are further divided into additional sub-process etc.

Subsequently David Parnas' ideas on modularization and encapsulation [6] where each module revealed as little as possible about its implementation to other modules in the system nourished the development of abstract data types. Abstract data types where influenced by the class concept in Simula [11], but there are some major differences. Simula classes describe co-routines, i.e. a reentrant procedure with internal state and several entry-points while a abstract data type describe a data structure and operations on this structure.

The language Mesa [12], developed at Xerox PARC in the mid-70's implemented these ideas. Mesa inspired both Nicklaus Wirth and his work on Modula and the design group behind the Ada programming language. Developing systems using abstract data types increased the developers' needs in terms of modeling capabilities. Although JSP and flowcharts were still used to describe algorithms the concept of modules introduced by Mesa and similar languages required new modeling capabilities for depicting modules and how they where interconnected.

Deremer and Kron [13] discuss two different types of languages, one supporting programming-in-the-small and the other programming-in-the-large.

Programming-in-the-large concerned structuring individual modules into a system. A module was seen as collection of resources, for instance variables, constants or subroutines. The demand for tools capable of expressing modules and their interconnections were addressed in Module Interconnection Languages [14] (MIL), and later task-level description languages as promoted by the Ada community during the 80's, see Durra [15] in this regard. The first module interconnection language MIL-75 [13] and the subsequent languages were capable of describing modules and their interfaces, and how modules were connected. These models described systems from a different point of view compared to algorithmic models such as JSP-diagrams. The focus on overall structure instead of module internals provided developers with tools which were more powerful in terms of both handling complexity and describing large software systems.

1.2.2 Object-oriented models

In the late seventies and early eighties new languages, such as Ada and Smalltalk became popular and widely used. In 1982 Rentsch [16] prophesied that "object-oriented programming will be in the 1980's what structured programming was in the 1970's". The focus was no longer on algorithms or data as abstractions. The new ideas combined data specifications with algorithm specifications in abstract data types. This new way of thinking required new models supporting development of object-based or object-oriented systems.

Grady Booch outlined a development method in [17] where objects are the fundamental abstraction in decompositions and models. Subsequently, new and more elaborate modeling techniques have been developed, taking object-oriented modeling to where it is today.

But even if object-based models are more powerful in terms of abstraction than pure algorithmic models researchers and practitioners perceived that as the complexity of systems being developed continued to grow, yet another abstraction level where even more abstract models could be created was needed.

1.2.3 Software Architecture

One of the more recent attempts to model software at a new level of abstraction is software architecture. The motivation for software architecture is that systems today are so complex that there is a need for describing them at a high level, but still with a capability to address specific aspects, such as non-functional requirements. The origin of software architecture is Module Interconnection Languages where researchers and practitioners discussed the module architecture of an application. Perry and Wolf [18], predicted in 1989 that the 90's "will be the decade of software architecture".

1.3 Research question

Software architecture is a relatively young area, rapidly changing and becoming more mature. It is clear that important progress is yet to be made in this area. A detailed survey of work-in-progress on an international scale was initiated in order to find significant areas that needed to be further explored.

In parallel to this survey we discussed software engineering issues with some Swedish companies on a more general level. While the initial focus was on reuse and maintenance issues and how improved software architecture tools and descriptions might improve the situation, it was soon discovered that an underlying issue was that of the dynamic evolution of software wherein systems where modifiable on a fundamental level online. Further, it was discovered that in this area there are no tools or techniques properly capable of supporting the design and development of dynamic software architectures. consequently, to make progress in this the following research goal has been formulated.

The goal is to develop a methodology for the development of software systems with a dynamic architectures. This is expanded into three parts, using Booch's description of what a methodology include [19, p. 23]:

1. *Notation:* How to include dynamic aspects in software architecture descriptions?

2. *Process:* Which set of process specific issues should be considered.

3. *Tools:* Which set of tools will improve the designers and implementor's capability to create good dynamic architectural designs?

1.4 Structure of the thesis

Chapter 2 introduces software architecture. The area is relatively young and we believe that going through different existing definitions, description techniques and design methods is useful. In Chapter 3 we describe some areas related to software architecture and our work. Chapter 4 is devoted to dynamic architectures; outlining the problem, presenting the state-of-the-art, and introducing our contribution the architecture agent. Chapter 5 conclude and discuss future work.

Chapter 2

Software Architecture

The structure of software applications has always drawn the interest of researchers and practitioners. Applications has been described and classified based on internal structure and inter-module communication, for instance client-server and message-passing applications. Parnas' early works on modules and program families [6, 20], as well as McIlroy's ideas on component based software development [1] focused on how to compose systems out of pre-developed parts. Object-orientation and the seemingly never ending growth of application complexity has increased the need for a technique to describe and reason about an application's structure at a high level.

This requirement is partially answered by software architecture. Software architecture is a relatively young discipline within the software community, even though most practitioners and some researchers claim that they have used such a concept for a long-time under other aliases. This chapter will introduce the area of software architecture. As this research area is still an immature one, both the presentation and the conclusions drawn will, in all probability, differ somewhat from what other researchers in the area.

In the first section we define the conceptual boundaries of software architecture by reviewing some of the most influential architectural models. Continuing, then we present our own more refined definition.

Another important subdivision of the general topic, how architectures can be specified and described, is described in section 2. We recapitulate a framework for architectural description languages (ADL's) for short, and give a brief overview of some of the most influential and widely used ADL's.

The third section is devoted to architectural styles, i.e. recurring structural patterns at the architectural level. The concept of styles is defined, important styles are presented, and how styles can be used to support software development is described.

The final section of this chapter focuses on process models. Architectural design must be supported in the development process by methodology and techniques for validation and verification to move architectural design more securely into industrial practice. We present a method for architectural design, and the verification and validation techniques it employs.

2.1 Architectural models

There are several definitions of software architecture. In this section we present some of the more influential definitions in the chronological order in which they were developed. This allows the demonstration of how strongly earlier definitions have influenced later ones.

At the end of this section we consider these definitions in toto and conclude reflect on the presented definitions and conclude. We then develop our own more refined definition, which we use as a major basis in subsequent chapters.

2.1.1 Perry and Wolf (1992)

Perry and Wolf present a definition based on structure in [21].

> "...a software architecture is a set of architectural (or, if you will, design) elements that have a particular form. We distinguish three different classes of architectural elements: processing elements; data elements; and connecting elements."

The architecture is seen as a triad containing elements, form and a rationale. The elements can be of one of three classes; computing elements, data elements and connector elements. The form is a set of parameters constraining the set of available components as well as the composition. Beside the compositional constraints the parameters also confine the evolution of the architecture. The architecture rationale serves as the logical basis for the existence of the architecture.

Perry and Wolf also compare software architecture to the traditional architecture of buildings. Via this comparison they emphasize four additional factors; (1) views, (2) styles, (3) style and engineering, and (4) style and materials.

The concept of views is important to the software architect. In order to communicate different aspects of an architecture to different users a simplified view concerning only the aspects that are important to the specific user need to be presented. Even though the authors stress the importance of using different views they concentrate on only one view, the implementation view.

The notion of style is useful to a software architect. A particular style assembles several architectural elements arranged according to some formalized rules.

The concept of architectural style can be used both as a set of rules during architectural design and as a concept used for communication something similar to the way design patterns are used. The similarity of architectural styles and design patterns is discussed in more depth in Section 3.

2.1.2 Garland and Shaw (1993)

In [22], Garlan and Shaw suggest that software architecture should be regarded as a new level of design.

> "...beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives. This is the software architecture level of design."

They continue on to present a structural view of an architecture as a set of components, linked together with connectors. This somewhat simplified definition is used only to consider architectural style and distinguish one style from another. This is the main focus of the author's work.

An architectural style is a pattern of structural organization which defines a family of systems. A particular architectural style provides a set of components and connectors that can be used, and a set of constraints for how elements can be combined. Even though the authors in their definition of software architecture, present several aspects that are of interest at the architectural level of design, they do not discuss any other architectural views than the structural view.

2.1.3 Boehm et al. (1995)

Boehm et al. take a more pragmatic approach to software architecture in [23]. They argue that it is not possible to give a complete description of the software system without taking the different stakeholder's needs into account. The architecture is seen as a media for communication as opposed to a pure description of a system's internal structures as proposed by Perry and Wolf; and Garlan and Shaw. The architecture is not only interesting in the design phase, but should be used throughout the process. According to the authors a software architecture includes:

- a collection of components, connections, and constraints. Components can be both hardware and software;

- a collection of needs, expressed by the system stakeholders; and
- a rationale. The rationale demonstrates that an implementation of the structural architecture will satisfy all needs expressed by the stakeholders.

The authors have no definition of architectural styles. Views are important as they are used to extract and present architectural information for a specific stakeholder. If a set of design documents, graphic or non-graphic, describes an architecture, some subset of these documents represents a view of particular interest to a stakeholder. The authors stress the importance of a rationale, which will ensure that an architecture will satisfy user needs.

2.1.4 Soni, Nord, and Hofmeister (1995)

In [24], Soni et al. provides an definition of software architecture based on industrial experience from Siemens Corporate Research. The definition is based on structures and views, where each view describe the system from a particular perspective. In their preliminary discussion they use the following definition.

> "Software architectures describe how a system is decomposed into components, how these components are inter-connected, and how they communicate and interact with each other."

This definition is later extended, based on their case-studies of industrial applications. The utility op presenting different views becomes clear to the authors in the course of these studies. According to them there are four important views (or architectures) which they have identified and clarified via the case-studies:

- The *conceptual architecture*, describes a system by means of critical design elements: how they are related to each other.
- The *module interconnection architecture* comprises two orthogonal structures. Functional decomposition is one structure which depicts the decomposition of a system into subsystems. The second structure involves layers, which are used to reduce dependencies, e.g. provide interfaces for hardware.
- The *execution architecture* captures the dynamic structure of an application during execution. The main benefit of the execution architecture lies in the ability to describe specific dynamic aspects of a system such as performance, location, and migration.
- The *code architecture* organizes source code modules, libraries, etc. This architecture is influenced by the implementation languages used, version control, and configuration-management strategies and tools applied, etc.

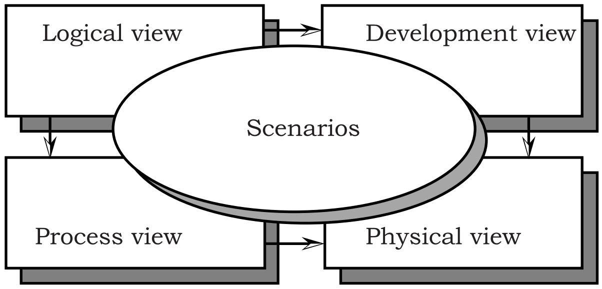


Figure 2.1: The 4+1 model of software architecture.

Each of the different architectures plays a specific role in a development project. For instance, the code architecture is important to configuration managers while the execution architecture is important to system operators.

Styles and description languages are not important to the authors. It is worth noting that, according to them, a module interconnection architecture always involves a layered style.

2.1.5 KRUTCHEN (1995)

Krutchen provides no formal definition of architecture but further elaborates the view concept in [25]. His work focuses on object-oriented systems and is greatly influenced by Grady Booch [19].

According to him there is a need for at least five different views of the software system architecture in order to cover all system aspects. These different views are directed towards different stakeholders. The views are, a logical view, a development view, a process view, a physical view, and scenarios. The systems architecture is developed using four of the views while the fifth (scenarios) is used for illustration and validation. The structure and relationships between the five views is depicted in Figure 2.1.

In his work, Krutchen addresses both notation and style. He concludes that no notation or style is suitable for every view. Instead, he proposes specific notations (derivatives from Booch'94) and discusses how different styles described by Garlan and Shaw [22] can be applied in each of the different views. The *logical view* describes a set of key abstractions in the system, which mainly implement functional requirements. The object-oriented style is used and a simplified variant of the Booch'94 notation. For the *process view*, an extended notation is employed. He discusses different styles that can be applied and suggests the pipe-filter and client-server styles. The process view captures how processes in a system behave during execution and addresses different non-functional aspects, such as performance, concurrency, and synchronization. The *development view* focuses on the organization of source modules. Elements like sub-systems and libraries are prominent entities in the notation used for this view. Krutchen recommends the layered style for this view. The *physical view* also addresses certain non-functional requirements, such as availability and performance. This view describes how different elements in the logical, process, and development view are mapped onto the system environment. The final view is the *scenario view*. It consists of a set of use-case instances [26, pp 41], which show how elements from the four previously described views work together.

Each view is directed towards a specific group of stakeholders, for instance the development view is mainly for programming purposes and is tailored for use by programmers and configuration managers.

2.1.6 Bushmann et al. (1996)

In the book Pattern-Oriented Software Architecture [27], Bushmann et al. provides the following definition of software architecture.

> "A software architecture is a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show relevant functional and non-functional properties of a software system. The software architecture of a system is an artifact. It is the result of the software design activity."

This definition can be viewed as a concluding definition based on the definitions presented previously. The authors stress the importance of multiple views and hierarchical architectures. Subsystems become an important part of the architectural definition. As the title of the book implies, the authors' major concern lies in architectural patterns. But the concept of patterns for software architecture should not be mixed up with architectural styles. According to them every architectural style can be described as an architectural pattern, but patterns for software architecture span the range from high level patterns, such as architectural styles, to more implementation dependent patterns. They describe how different patterns are applied and composed in the design activity resulting in a software architecture for a system which pertains to a specific architectural style.

2.1.7 Bass, Clements, and Kazman (1997)

In [28], Bass et al. presents a compressed definition of software architecture.

> "The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationship among them."

This definition includes a view concept, although that is not immediately apparent. The quote "structure or structures", is a hint that there is not a single description (or structure) that makes up the complete architecture.

Style in not mentioned as an important concept. Instead, they focus on the externally visible properties of the components in the architecture. This distinction is important to the authors since their interest is to analyze architectures for specific properties, such as performance and manitainability, and hence, the characteristics of the participating components becomes important.

2.1.8 Conclusion

Above, several different definitions of software architecture, based on the specific interests and varying viewpoints of different authors has been presented. We have seen how these definitions have evolved over time, pointing towards the day when a general consensus regarding a description of software architecture and its conceptual subdivisions will be reached. To close this section we present our definition constructed by combining, refining, elaborating, and extending the concepts delineated in the previous definitions.

We see an architecture as a description of a system. The description contains multiple views in order to break down complexity and simplify understanding and reasoning. The majority of the views depict structure; static, dynamic, or both. To create these representations two basic building blocks are used, components and connectors.

Component is either a functional abstraction or a data abstraction accessible through an interface that specifies all external dependencies and services provided by the component. Connectors are connection abstractions with three major capabilities: connection, transformation, and transportation.

A connector must be able to connect to a component. Connecting a connector to a component sounds trivial but is problematic including difficulties such as type mismatches requiring type conversions. Other more trivial connecting activities could be, turn a component into a subscriber to specific messages, reflecting a a message passing connector. The second area of responsibility is transformation. A connector should be able to handle different, more or less, complicated conversions. For instance, different types of data transformations to handle differences in data representations etc. Finally a connector is responsible for moving information within the connector. For instance, broadcast a message to all subscribers or invoke a method in a remote object.

Components and connectors are assembled into configurations. A configuration can be static or dynamic depending on the type of application being developed. In order to handle dynamic configurations and be able to describe these properly we will have to make the descriptions dynamic as well. To achieve this we introduce "architectural level" components and connectors, which are used to model the dynamic behavior. These components does not necessarily reflect the actual architecture of the application and are used for modeling purposes only.

2.2 Architecture Description Languages

In order to introduce architecture-based software development and use it effectively developers must be able to express architectural constructs and communicate these clearly within and without the development team. A common language is needed to avoid problems due to misinterpretations. The language to be used, in order to minimize these risks and, most importantly, to create the capability of developing and using architectural description support tools and to make the language usable in support tools, has to be formal to a certain extent.

Several architecture description languages (ADLs) for describing software architectures have been developed. Every language proposed is based on the author's own perception of what an ADL should include. This has resulted in a multiplicity of languages, wherein often a language focuses on one specific architectural aspect, excluding others , of interest to the architectural community. Additionally, the level of formality in each of the existing languages varies, often being quite minimal. Since formality is the basis on which the use of analytical architectural tools are built, this fact raises a special difficulty.

As mentioned, there exist several languages for describing architectures and we discuss the more influential ones in this section. Prior to that we provide a brief overview of properties in existing languages, by relating a framework for classification and comparison, developed by Medvidovic and Taylor. There is no consensus within the software architecture community of what an ADL is or should be.

2.2.1 The ADL classification framework

In order to define what an ADL is, one should start from the definitions presented in Section 2.1. Medvidovic and Taylor surveys several languages in [29, 30], identifies characteristic properties of individual languages, and synthesizes a framework based on their findings.

First they identify two important aspects of architecture description languages supported by every language in the survey, the ability to model architectures and tool support.

The ability to model architectures is founded on the assumption that an architecture consists of components and connectors combined in architectural configurations. This is not controversial statement and is based on the surveyed architectural models presented in the previous section. Medvidovic and Taylor also present some desired features that each modeling element should provide support for. For instance, a connector description should support typing and semantic descriptions among other features. It is additionally that support for dynamism, i.e. dynamically changing architectural configurations, and non-functional properties be present in the architecture configuration notation. We will return to these specific features later in this section.

Tool support is emphasized as the other important aspect of architecture description languages. The fact is that every language, technique, or process proposed is to be judged on the availability and usability of tools that can be associated with it. Every ADL has some kind of tool support, but the features are continuously changing and being added to. This complicates the problem of comparisons.

In Figure 2.2 we depict the taxonomy of Medvidovic's and Taylor's framework. Every feature presented in the framework is discussed more extensively below.

Architecture Modeling Features

The architectural modeling features listed in Figure 2.2 require some further explanation and motivation. We relate Medvidovic's and Taylor's description found in [30].

There are several definitions of a component. Perry and Wolf [21], use "processing and data elements". Another, derived from Garlan and Shaw [31] defines a component as a "primitive or composite unit of data storage or computation". These two definitions are similar. Even though they are not connected to a specific architecture description language, most ADLs' use a similar component concept, but with certain extensions, such as defining other classes of components [31, p. 149], e.g. the manager component and the controller component.

Components

One of the fundamental principles of software architecture is that an architectural element should provide an *interface* to other elements in the architecture. Interfaces consist of a set of services that a component provides. But component interfaces should not only describe the total set of services

ADL
- ARCHITECTURE MODELING FEATURES
  - **Components**
    - Interface
    - Types
    - Semantics
    - Constraints
    - Evolution
    - Non-functional properties
  - **Connectors**
    - Interface
    - Types
    - Semantics
    - Constraints
    - Evolution
    - Non-functional properties
  - **Configurations**
    - Understandability
    - Compositionality
    - Heterogeneity
    - Constraints
    - Refinement and tracability
    - Scalability
    - Evolution
    - Dynamism
    - Non-functional properties
- TOOL SUPPORT
  - Active Specification
  - Multiple Views
  - Analysis
  - Refinement
  - Code Generation
  - Dynamism

Figure 2.2: ADL classification and comparison framework.

available. They should also be capable of describing the specific set of services needed by the component. Interface descriptions are similar to the class interfaces in object-oriented languages and other package specifications in imperative languages, such as Ada. Another aspect of components involves *types*, or the ability to handle abstract specifications that can be multiply instantiated in an architecture description. In an architecture, a specific pattern can reappear in a set of components. Parameterized types, allow for specifications of types, which later can be instantiated to actual types. An important activity, which architecture is intended to support is formal analysis. A limited analysis can be performed based on information gathered from a component interface or type, but to allow for more extensive and exhaustive formal analysis the components *internal semantics* must be provided.

Constraints are used to certify that a component is not misused. Constraints include implementation constraints (e.g. performance constraints) and state constraints. Components reflect design elements, therefore components should be able to evolve as design elements do. Typical mechanisms for *evolution* are sub typing and feature modification. Simulating architectures and begin able to study dynamic behavior is most important. In order to support this, a language should provide means to specify certain *non-functional properties* for a component. For instance, throughput properties and performance properties.

Connectors

Connectors need support for the same set of features as components do. In order to handle the wiring of connectors to components, *interfaces* are needed. The interface for a connector describes how components which attach to a connector can utilize it. *Types* are important to connectors, in the same way as they are for components. Connector instances of the same type will reappear in the architecture. In order to perform extensive analysis, the internal *semantics* of a connector must be provided, in a similar manner as is required with components. Connector *constraint* descriptions need also to be provided for to be able to ensure that a connector is not misused. Additional desirable features include support for *evolution* in the form of refinement and modification of existing connectors and the ability to model *non-functional properties*.

Configurations

Clearly, as with any type of design, a software architecture should be easy to understand. This property *understandability* is an absolute necessity if a design is to meet the fundamental requirement of serving as a vehicle for communication. In order to comply with this requirement several languages have an accompanying graphical notation. Another required feature is *compositionality*, i.e. support for description at different levels of abstraction. This is certainly useful in large system development where hierarchical descriptions is employed to deal with complexity. Another aspect of compositionality involves support for the composition of heterogeneous components, or *heterogeneity*. Components can differ in many ways, for instance the implementation language used with one component can differ from that used in others. Additionally, the external support needed by components can vary. *Constraints* are useful in order to restrict the number of possible configurations and to guide the system architect in making proper decisions. An example of the use of constraints would be that an architectural language require that every port on a connector be connected to a port on a component.

An architecture is expected to support the entire design process, from early conceptual modeling to implementation. Therefore, one important feature needed is support for architectural *refinement* including traceability between the different refinement levels. Systems will also expand or otherwise change over time. Therefore, support for ease of scaling (*scalability*) and *evolution* is most important.

The architecture should be capable of describing and properly handling non-functional properties of the design, for instance modifiability and performance properties. Some existing languages allow for the specification of a limited number of such non-functional properties. In important instances as will be seen below in Section 4, architectures need to change dynamically. For such systems, the ability to describe dynamic aspects (dynamism) of the architecture is most important.

Tools

Every surveyed language has some tool support, although the intended use and usefulness vary. Most languages provide one or another tool that supports modeling or construction of architectural descriptions and, for some languages, generation of code. Languages with a formal basis provide tools for static analysis of descriptions for specific architectural properties. Another type of tool supports external dynamic reconfiguration of architectures.

2.2.2 UniCon

UniCon [32] is an architectural description language intended to support development of software architectures and streamline the transformation of architectural models into source code.

The building blocks in a UniCon architecture are components and connectors. Component specifications can be found in *interfaces*, while a connector specification is referred to as a *protocol*. For each element, a special section in the specification describes the implementation.

An interface specifies the properties of a component including the type of a component, type invariants, functionality, other characteristics, and *players*. UniCon uses the term player when referring to a named entity, which is visible in the interface specification. A player is specified in the interface with a signature, functionality, and other properties which are useful when interacting with a particular, "player". UniCon provides a set of predefined component types including Module, SharedData, Filter, and Process.

The connectors have protocol specifications that define connector properties. Each protocol includes a type specification. In earlier versions of UniCon, the number of available connector types was fixed. The pre-specified types where Pipe, FileIO, ProcedureCall, DataAccess, PLBundler, RemoteProcCall, and RTS-Scheduler. In recent developments the capability to add new, user defined connector types, has been added. In the protocol specification it is also possible to specify other properties, such as performance. Equivalent to players for components is the specification of *roles* for connectors.

Besides the interface or protocol specification, every UniCon element specifies its implementation. An implementation in UniCon can be either primitive or composite. Primitive implementations are indivisible, i.e. cannot be divided into other primitive implementations. Composite implementations instantiate and configure a set of components and connectors (primitive or composite). UniCon does not explicitly model configurations, instead the component construct is used to specify a complete architecture, using a composite implementation.

Illustrating a simple Unix Pipe-Filter architecture is easy in UniCon using the built-in component and connector types Filter and Pipe. In Figure 2.3 a new component is created which accepts input in the form of ASCII strings, and produces a sorted output with all characters in the interval a-g as capitals. The component is a composite, composed of two components, "upcase" and "sorter", connected with a standard Unix pipe P.

```
COMPONENT UPCASESORT
   INTERFACE IS
      TYPE Filter
      PLAYER input IS StreamIn
          SIGNATURE ("line")
          PORTBINDING (stdin)
      end input
      PLAYER output IS StreamIn
          SIGNATURE ("line")
          PORTBINDING (stout)
      end output
      PLAYER error IS StreamIn
          SIGNATURE ("line")
          PORTBINDING (stderr)
      end error
   END INTERFACE
   IMPLEMENTATION IS
      USES capsup INTERFACE uppercase
      END capsup
      USES sorter INTERFACE sort
      END sorter
      USES P PROTOCOL Unix-pipe
      end P

      BIND input TO capsup.input
      CONNECT capsup.output TO P.source
      CONNECT sorter.input TO P.sink
      BIND output TO sorter.output
   END IMPLEMENTATION
END UPCASESORT
```

Figure 2.3: Specification of a UniCon component.

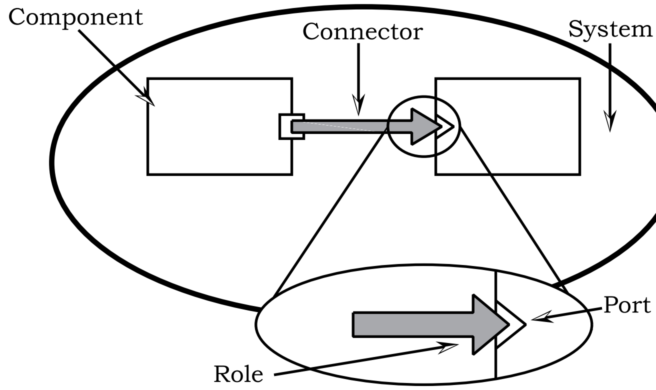


Figure 2.4: Architectural structure in ACME.

2.2.3 ACME

The primary purpose and intended use of ACME [33] is as a architecture interchange language for architectural development environments. ACME is based on seven fundamental architectural modeling elements: *components*, *connectors*, *systems*, *ports*, *roles*, *representations*, and *rep-maps*. The first five are used for modeling the structure of an architecture as depicted in Figure 2.4. A component provides at least one interface. The set of ports included in an interface defines the interface type. A port is a single point of interaction, which can refer to simple interactions, such as a single function-call, or more complex interactions, such as multi-cast operations. The connector interface is composed of a set of roles. A pipe-connector, for instance, has two roles, reading and writing. More complex connectors may require more than two roles. An example of this would be where an event driven system uses an event mechanism to which several clients subscribe. The connector representing this will use a single publishing role and an arbitrary number of receiver roles. Finally, components and connectors are configured into complete systems.

ACME supports hierarchical description of architectures, in the form of single or multiple representations. In an hierarchical description the top level ports and roles must be mapped on ports and roles in the lower level descriptions. ACME provide a mechanism for this in the form of rep-maps.

Architecture is not completely about structure. Other properties of an architecture exist and are important. The purpose behind ACME was to develop an interchange language where properties of interest to a set of description languages could be accommodated. As mentioned in the previous section, the greatest common divisor for existing ADLs lay in the modeling capabilities. Every feature presented was not available in a single language. To manage this and accommodate properties from different architecture languages ACME uses a *property* construct which annotates the structural description with additional information. Properties are not interpreted by ACME, instead the interpretation is left to the language or tool which understands and supports a specific property. As an example, imagine a simple pipe-filter architecture where the pipe-protocol is formally defined in Wright (see below for a description of Wright). The ACME description is annotated with a property construct, protocol, that Wright can understand and manipulate. To handle properties, a set of language dependent property lists must be developed. These are interpreted only by tools that understand them and for which they are useful.

2.2.4 C2

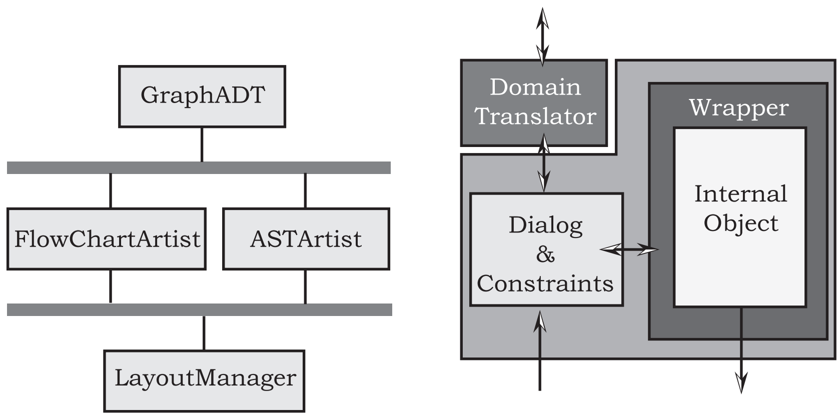
C2 [34] is fundamentally not a architectural description language. C2 originated as an architectural style[1] suitable for design and description of graphical user-interfaces in applications. Later improvements added both tools and a notation (C2 SADL), similar to a traditional ADL.

Architectures expressed in the C2 style divide a system into layers, each layer constituted by a set of connectors. A central component of the semantics for C2 style architectures involves the principle that no component be aware of any other component residing at a lower level. Components can be connected to each other via connectors, and the number of components that can be connected to a connector is unlimited. C2 also allows for two or more connectors to be connected directly. The structure of a simple C2 architecture is displayed in Figure 2.5(a).

Communication within a C2 architecture involves a message-passing mechanism passing requests upwards in the architecture and notifications downwards. Requests are directives from a component at a lower level while notifications indicate a state change in a component's internal state. The set of requests sent upwards and notifications accepted from above constitute a component's top domain. The bottom domain is defined by the set of notifications emitted from a component and the requests accepted from components that reside at a lower level. The internal structure of a C2 component is shown in Figure 2.5(b).

The primary responsibility for the connector is to route or broadcast notifications and requests to the receiving components. The connector can also implement filtering mechanisms, such as a priority strategy, and buffers.

[1] Read more on architectural styles in Section 2.3.



(a) C2-style architecture. (b) C2-component internals.

Figure 2.5: C2 architecture style.

Even though the description above is that of an architectural style, C2 provides a notation for components and connectors, configured into systems, similar to other pure ADLs. The modeling ability is somewhat restricted since the interface and architectural semantics are limited to the C2 style.

2.2.5 SADL

The Structural Architecture Description Language [35] (SADL), is a formal description language with a focus on description of hierarchical architectures. SADL uses two types of hierarchies, *vertical* and *horizontal*. A vertical hierarchy is similar to the traditional analysis, design, and implementation model hierarchies used to bridge the gap between a high level design model and a more detailed implementation level model. The different architectural models in a vertical hierarchy use different vocabulary in each model description, while the horizontal hierarchies use the same vocabulary for all models.

SADL's intended use is the description of several architectural models at different levels in a hierarchy and to describe mappings from one level to another. When an architecture at a given level is analyzed, the mapping can be used to determine if the same properties hold for the corresponding architecture at another level. SADL offers the possibility to model specific architectures as well as generic architectures. Mappings, architectural styles, and refinement patterns can also be specified. A style consists of a vocabulary, a set of design elements, constraints on the configuration, semantic constraints used when refining architectures, and semantics for the connectors in the style.

The SADL language has been used in several projects for analyzing architectural properties, such as safety, statically. The formal notation is supported by a graphical notation.

2.2.6 WRIGHT

The Wright [36] language is intended for formal analysis of architectural properties with a focus on semantics for connectors. Wright provides the capability to formally specify connector semantics in a CSP [37] subset.

A system in Wright consists of three parts. First, all component and connector *classes* are specified. A component is specified in terms of its *interface*, i.e. as a collection of *ports*. A component specification can also include a description of a computation.

Connectors are specified by a set of connection points or *roles*, and a *glue* specification. The roles specify how the connector behaves while the glue coordinates the roles, similar to the computational specification of a component.

Wright has been used in several industrial applications [36] where architectures described in Wright have been analyzed. The choice of CSP for the formal semantics allows the use of standard off-the-shelf tools for CSP.

2.2.7 DARWIN

Darwin [38] focuses on describing of distributed systems. The concept of component in Darwin is similar to that found in other languages. A component provides a specified set of services but also requires additional services in order to fulfill them. Each “service” port and “require” port is typed, i.e. the type of data is specified in the interface. Darwin also allows composite components composed from set of sub-components. A system is viewed as a composite component.

The connectors in Darwin are primitive. There exist no explicit connectors in the language. At configuration time, the system architect *binds* “require” and “service” pairs. Darwin has formally specified semantics (Π-calculus) and it is possible to statically analyze architectural properties. Darwin also supports dynamic instantiation of components, i.e. components can be created at run-time. This is a frequently occurring event in distributed systems, for instance, in order to achieve proper load-balancing.

Darwin sits on top of the Regis system [39], and is supported by tools for modeling, analysis, and code generation from the architecture models.

2.2.8 Conclusions

We have presented a brief overview of some of the more influential and successful architecture description languages available. As we have seen, the expressiveness varies from language to language. Our criticism of these languages lies in the lack of support for describing dynamic configurations. Some languages have external tools that can be used to change a configuration at run-time. But there is no direct support to describe such an event in the languages. Another problem lies in the usefulness of certain languages by less skilled developers. In order to provide analysis capabilities, some languages require a degree of formality which preludes use by such types of developers.

2.3 Architectural styles

The use of patterns is common in every engineering discipline. This knowledge inherent in such patterns has been developed over time and includes that stemming from direct, practical experience. Often, such knowledge is transferred from generation to generation, in a non-formal way. In software engineering, idioms and patterns are used both for modeling and design [40]. At the architectural level, with a focus on system structure, “pipe-filter” and “layered” architectures are two examples of patterns. Recurring patterns at the architectural level of design are referred to as architectural styles, an analogy to architectural styles for buildings.

A style is defined in a similar manner as a design pattern. A style specifies a set of component types and connector types that can be used in the style. It provides a set of structural constraints on how to configure components and connectors into a system, and some of the style invariants. A definition should also includes, or should include, additional information that simplifies the process of understanding, such as a description of the underlying computational model, some examples of how the style is used, and a list of advantages and disadvantages from using the style.

Applications can use several styles at different levels of abstraction. In a hierarchical architecture, the top-level architecture can be a “pipe-filter” architecture while one of the filters can use another style internally, the same also holds true for connectors. It is additionally possible to mix several styles at the same level [41].

2.3.1 Some architectural styles

In their seminal work Shaw and Garlan [31] list twelve architectural styles. They classify the styles based on the underlying computational model. While this classification is valid in some cases we believe that the inclusion of styles like "main program and subroutines" and "OO systems" is unfortunate. These two styles are examples of implementation styles, used to implement some of the other styles. For instance a pipe-filer or client server architecture can be implemented with these two styles. A categorization of styles into *design styles* and *implementation styles* we belive is more appropriate. In the design patterns community we can se the same differentiation into design and implementation patterns. A problem with this classification is that is sometimes context dependent, i.e. some styles are both design and implementation styles, for instance the OO-style.

Design styles

Pipe-Filter
: In a pipe-filter style architecture each component has a set of inputs and a set of outputs. A fundamental constraint is that every component should at least have one input and one output. A component reads data from its inputs, transforms it, and writes it to the outputs. The connector's responsibility is to transport data from one end of the connector to the other. A style invariant is that no component or connector can share a state with any other component or connector. A common example of such is the pipe-line. A command in the UNIX shell is one example of an instantiation of a pipe-line. Here some input is generated at the beginning of the pipe-line, transformed in some filter, and the final output is to be found at the end.

Client-Server
: A client-server architecture style is a variant of a distributed system architecture where components represent processes that can be divided into three major classes; client processes, server processes, and actor processes.

  The client components require and use some service from a server component. A server component provides a set of services to clients. Finally, the actor components work as both client and server.

  One important characteristic of the server components is that they do not know of which clients or the number of clients that will use the their services prior to run-time. On the other hand, client components, must know the server identities. This knowledge can be coded into a client or a special server component can be used to look up specific services and provide the server's identity.

The connectors used in a client-server architectural style can come in many forms. The most common ones are synchronous or asynchronous remote message calls, similar to connectors used in a traditional distributed system. In recent system development, the client-server style has been used in single machine application. For instance, the COM architecture implements a client-server architectural style in the Windows operating system.

Implementation styles

OO-Systems

An OO-system — or data abstraction — architectural style represents all applications that use abstract data-types, a data representation, and a set of elementary methods for manipulating the data-structure.

It is not easy to explicitly say what is a component and what is not in this style. During design the components could be classes or some other type of module description, but a component could also be a object. The same holds for implementation. At run-time all components will be objects. In the original description of Shaw and Garlan this characteristic is neglected.

Later architectural models, such as the 4+1 model (Section 2.1.5), introduce views. With the concept of views, we can introduce a logical-static view for instance. In this view we describe our static-conceptual design and our components become classes. In a logical-dynamic view we describe the systems behavior with objects expressed as components.

Another important characteristic of components in an OO-system style is that they can support more than one interface, be multithreaded, and represent mobile entities. Depending on the underlying object-model, the components can exhibit other useful properties as well.

Connectors expressed in the OO-systems style are not explicit, i.e. modeled separately. At composition time one object calls a method in another object. This creates a web of communicating objects. An important aspect is that a component must know of the receiver of a method call. This is the obverse of the situation with the pipe-filter style.

We have chosen to classify the OO-system style as an implementation style. Due to the tight coupling of the OO-system's style to the implementation language this style is not implementation independent.

Main program with subroutines style

The main program with subroutines style provides components and connectors available in imperative languages. The components reflect some functional abstraction, while connectors are function call connectors. One component is distinguished as the main-program component, acting as a driver of the computation by calling other subroutine components in the system.

2.4 Architecture Design Processes

In earlier works on software architectures the focus has been on description and analysis, while people have paid less attention to the problem of providing sufficient support for design. Methods that have been proposed have been ad-hoc and based on subjective evaluation.

Abowd et al. [42] has proposed an evaluation of certain non-functional requirements for a set of candidate architectures. They present two different categories of evaluation; questioning and measuring. Techniques for questioning are informal and result in a subjective evaluation of the architecture. Scenarios, questionnaire, and checklists are examples of techniques in this class. Formal evaluation requires measurement making the result quantifiable.

Certain other methods use basically the same approach. The activities involved are, develop a candidate architecture, evaluate the architecture using prioritized quality requirements as evaluation criterions, accept the candidate architecture or modify it, and re-evaluate it. Below we present one of the more mature and promising approaches to a controlled architecture design process, which is based on the process skeleton described above. The method has been developed and tested in tight collaboration with several industrial projects.

2.4.1 ATAM

The Architecture Tradeoff Analysis Method [43], or ATAM for short, is a method for evaluating architecture-level designs for a set of different functional qualities such as performance, reliability, security and maintainability. It is a spiral process model [44] for design at the architecture-level. ATAM has emerged from work on the Software Architecture Analysis Method (SAAM) [45] at the Software Engineering Institute at Carnegie Mellon University.

SAAM consists of five steps; develop candidate architectures, develop candidate scenarios, evaluate scenarios, expose scenario interactions, and finally an overall evaluation. Each scenario is prioritized reflecting the importance of the scenario for a given application. SAAM has been used to analyze a number of different industrial systems. For instance, a global information system, a air traffic control real-time system, and a commercial revision / configuration management tool.

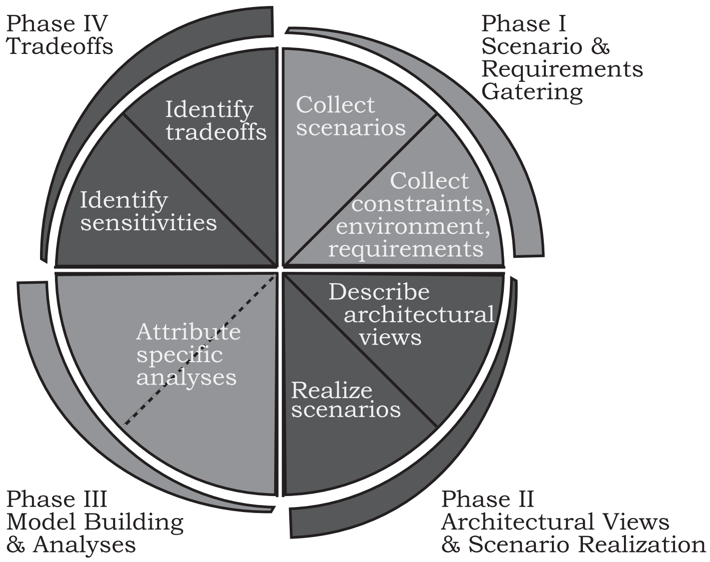


Figure 2.6: The ATAM process model.

The ATAM process, depicted in Figure 2.6, consists of four major steps; (1) Collect Scenarios and Requirements, (2) Architectural Views and Scenario Realization, (3) Build models and analyze, and (4) Tradeoff identification. Each phase is further divided into several sub-activities, which are described below.

Collect Scenarios

Scenario specification is a widely used technique in requirements elicitation. Scenarios stimulate and facilitate communication among different stakeholders which is important in the first two steps of ATAM. For each key functional and quality requirement, scenarios should be either collected or developed. The scenarios can be written down in a sentence or two. Making the descriptions too long is an indication of poor understanding of the problem and makes communication more difficult. Below we present scenarios guiding the analysis of an imaginary system requiring dynamic modification. The scenarios represent two common dynamic changes for the system where *dynamic modification* is a key requirement:

- each plug-in server simultaneously publishes a new plug-in within fifteen seconds after upload.

- the application downloads and installs a plug-in component.

Collect Requirements/Constraints/Environment

In the specification activity we group requirements, constraints, and environment descriptions related to a specific quality attribute. The design space constraints and a description of the execution environment is important since they will have an impact on the analysis of a specific attribute. For the analysis of the dynamically modifiable requirement we collect the following requirements and constraints, and describe the execution environment:

- Req: The class-server should store plug-ins in a version controlled system.
- Req: The application should query the server every second minute for new plug-ins .
- Req: If a new plug-in appears, the application should down-load and install the new class within 2 minutes.
- Con: No more than four plug-ins to be published per hour.
- Con: The application has three operational modes; high, normal, and low.
- Env: The analysis should consider download and installation of at least ten components for each of the three operational modes.

Describe Architectural Views

The ATAM process relies on comparison of multiple architectures. Each candidate architecture will describe how functionality is distributed over a configuration of components and the different quality attributes are described in views. Multiple views are important since no single architectural description is sufficient for all attributes. For each attribute view additional relevant architectural information required for the analysis should be provided.

Attribute-Specific Analyses

When the candidate architectures are described they should be evaluated for every isolated attribute. The analyses produces a set of statements about every candidate architecture for every attribute. For instance the following statements where made for one candidate architecture for the dynamically modifiable application:

- the server checks in a plug-in and the version server signals every subscriber.

- the client checks and down-loads new plug-ins in an average of 75 second during high operational load.

Identify Sensitivities
The objective for identifying sensitivities is to find attributes that are sensitive for architectural modifications. By varying some attributes and studying the resulting modified architecture, statements made during analysis should not be affected significantly. If some statements are affected by modifying some structural element these elements are weak points in the architecture. For instance, in our dynamic system the number of plug-in servers could be varied. If this affects the possibility for applications to down-load and install plug-ins within the given time-frame, the number of servers is a sensitive element.

Identify Tradeoffs
Identifying architectural tradeoffs aims at finding two or more conflicting sensitivities. In our example system the number of plug-in servers where important for the application. If we decrease the number of servers the two minute requirement might be endangered. On the other hand, several servers require replication of data which will make it more difficult to assure that all servers publish the new plug-in simultaneously.

After several iterations adding information and improving the architectural descriptions based on the analysis, sensitivities, and tradeoffs the resulting architecture partially meets the attribute and functional requirements. However, even if ATAM improves an understanding of the architecture, it is impossible that all requirements can be provided for at the architectural level since many only pertain to the implementation level. But still, the resulting architecture will provide designers with a solid foundation to build on in the following detail design.

Chapter 3

RELATED TECHNOLOGY

This section briefly presents some topics related to software architecture that are important to take note of. First, Aspect Oriented Programming, (AOP), a new approach for system development, is discussed. All applications consist of a set of functions and a set of "aspects" describing the non-functional behavior of these functions. This novel approach involves the description of each aspect independently and the subsequent "weaving" of aspects and functions into a single application. Second, component based software engineering, wherein maximum utility is made of software reuse. is mentioned. This reuse capability has been very high on the list of desirable software system qualities for many years, high on the "silver bullet" list, since McIlroy proposed in the late 1960's that mass-produced components [1] could end the software crisis. Thirdly, frameworks the object oriented (OO) variant of software libraries, wherein not only code is reused but also structure or architecture of the framework itself. Finally, we will describe design patterns in more depth. Design patterns are related to architectural styles, describing common solutions in terms of structure, but at another level of abstraction. At the end we summarize and conclude.

3.1 Aspect Oriented Programming

The traditional way of approaching software development is to deal with the functional requirements first. After a functional decomposition the developers consider the implementation of all necessary quality requirements, aspects, including memory management, concurrency, etc.

This approach has several drawbacks. Functional decomposition results in a system wherein the different quality aspects are randomly distributed and spread over any number of functional components in no clear fashion. This makes it

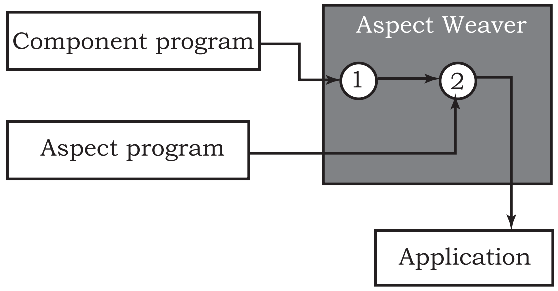


Figure 3.1: Aspect Oriented Programming.

extremely difficult to maintain these aspects. For instance, if we would like to change one aspect, e.g. a synchronization policy, we might find it easier to rewrite the complete application instead of trying to find all places where this aspect is affected or in some way implemented.

Aspect Oriented Programming [46] proposes another approach for managing aspects. The concept involves describing different aspects in separate specifications and then weaving them together with the functional behavior into one unit. The technique is depicted in Figure 3.1. A component program (or language) describes the required functionality in the application. The component description is sent to the aspect weaver and woven together with the aspect program, represented in an aspect language. With this approach, aspects and functionality can be created and maintained separately.

There are several examples of aspects for different types of domains. For instance, distributed systems include aspects such as distribution and synchronization policies, fault-tolerance mechanisms, and distributed garbage collection. In order to support aspect oriented programming for this class of applications, we have to develop one more aspect language which can be used for describing these aspects in this class of applications. The weaver must also be developed and tailored for each aspect and component language.

The weaving process rewrites the functional parts of the application, adding all the necessary aspect related code. The resulting code is most often extremely difficult to understand and manipulate directly, but since the different descriptions of aspects and functionality are separately maintained, it is these that can be used for later modifications.

Aspect Oriented Programming promises great rewards in terms of understandability and simplification in the process of developing complex software systems. However a significant amount of additional work is needed, particularly in regard to both aspect languages and weaving technology. Existing example systems where AOP has been applied are somewhat limited both in terms of complexity and number of implemented aspects.

3.2 Software components

For more than thirty years, utilization of standardized software components within a system has been frequently listed on the software engineering communities silver-bullet [7] "top-ten" list. This component approach, used in many other disciplines such as electronic and civil engineering, has never reached the same degree of success within the software engineering field.

This section presents some of the more widely used definitions of software components from the last three decades of work. Each definition has been influenced by the techniques used in that decade. We also present an overview of existing component technology.

3.2.1 Component definitions

McIlroy (1968)

McIlroy's concept of "Mass produced software components" [1] was written in response to the call for the famous NATO conference on software development held in Garmish in 1968. This conference gave birth to some of the most well known terminology in the software systems field including *"software crisis"*, *"software engineering"*, and *"software components"*.

McIlroy began with a comparison of the software industry and other industrial areas and stated that software developers were working at the craft level, while developers in other areas had advanced to working at the modern industrial mass production level. He identified what he believed to be the missing link, "the absence of a software components sub-industry". His definition of components as source code "routines" was natural for that time since subroutine libraries made up the principal type of functional abstractions used in the then current software development.

McIlroy also addressed the need for flexible and versatile routines, and proposed "families of routines for any given" job. Each family was to include variants of the same functional abstraction where different characteristics, such as algorithm (e.g. different sorting strategies), performance, precision, and robustness were to be taken into account. As an example, he used the implementation of a simple trigonometric sine routine which resulted in a family of over 300 implementation variants..

An implementation of a family requires considerable effort. As a consequence, McIlroy foresaw an emerging market wherein organizations adopting component technology would be forced to buy standardized components instead of developing them. He emphasized domain knowledge as an essential part of the process and proposed some certain application areas which had reached a level of maturity allowing it to be both possible and profitable to create components (routines) for them.

Even though McIlroy's proposal seems naive today, is should be kept in mind that subroutine libraries have been successful as saleable "components" for both imperative and object-oriented languages.

Cox (1986)

In the early 80s new languages, such as Ada and Smalltalk, gave birth to new interpretations and definitions of the software component concept. In [2, p. 26], Objective-C's creator Brad Cox sought "a sword" to cut the the Gordian knot of software engineering in the same area as McIlroy did, in hardware manufacturing.

> "The silicon chip is the unit of hardware productivity boom. Might the Software-IC concept do the same for software?"

According to Cox, a component – called a "software-IC", was a binary package combining different characteristics separately drawn from subroutines and Unix "Pipe-Filters". Developers using binary IC's would only consider the external interface, similar to the usage of Unix filters. Software-IC's, implemented in his Objective-C language, would communicate via messages passed between components.

In comparison to McIlroy's components, Software-IC's have several advantages. First, they are binary, which improves reusability since implementation details are hidden from the user. Second, each component is self-contained, i.e. it does not depend on any specific external component.

Software IC's never became a success. Firstly, the C++ language from Bell labs, which implemented similar (OO) concepts expanded its' dominance in the OO filed while Objective-C never reached the same popularity resulting in a very small market for Software IC's. Additionally, no interface standards where decided upon, leading to mismatches when developers tried to combine components from several vendors.

Szyperski et al. (1996)

In the last ten years, many new techniques have entered the market. New object-oriented techniques, new design methods and notations, and new languages. Development via "software components" has often been used as a buzz-words to catch the market's attention. However, very few practical attempts to implement the concept have been made.

Academic work on software components has continued and questions the previous IC analogy. Software components are subject to late integration, which puts additional requirements on both components and their supporting frameworks, not something taken into account in either McIlroy's work or Cox's IC's.

The new ideas and advances in technology have lead to a slightly modified definition of software components, formulated most directly by Szyperski et al. at the International Workshop on Component-Oriented Programming (WCOP'96):

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

This definition covers both technical issues and market aspects. Firstly a component has to be self-contained in terms of essential functions. All communication, both inbound and outbound, has to be clearly specified in explicit interfaces. Use of interfaces and making components self-contained make components independently deployable and composable by third-parties.

The requirement for a true market for software components is heavily stressed by Szyperski et al. Without the existence of such a component market, no component technology no matter how good it is will survive.

3.2.2 Existing component infrastructures

A component infrastructure is a supporting middleware with a capability of hosting software components and facilitating communication among these. In the early 1990's, Microsoft introduced the COM technology and the Object Management Group (OMG) launched CORBA. These component techniques were widely accepted and several vendors and software houses started development projects. But no single product has been completely successful and the market have been more or less equally shared among the competitors.

A few years ago, in the mid-nineties Java arrived. Java has platform independence as a primary design goal, but extensions and additions to the language, especially Java Beans and RMI, make Java a major competitor.

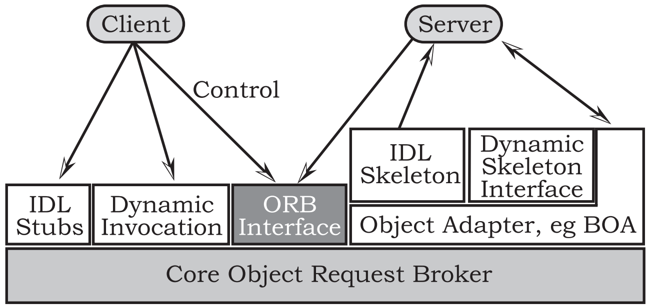


Figure 3.2: The CORBA core architecture.

CORBA

In 1989, several prominent organizations formed the *Object Management Group* (OMG). OMG initially focused on architectures for compound documents which was an important topic at the time. Later, the OMG switched from this top-down approach and focused their work on a more general support architecture for object based applications. This resulted in the release of the CORBA specification, which was adopted in October, 1991.

The initial goal for CORBA was to support development of applications with a complex execution environment. Support for *distributed components*, *multi operating systems*, *different programming languages*, and *legacy systems*, were to be included. With these goals in mind, the OMG designed an architecture which supported object oriented applications of objects with transparent communication mechanisms. The architecture is depicted in Figure 3.2.

Language interworking was one of the design goals for CORBA, and is achieved with the *Interface Definition Language*, (IDL). IDL specifications are mapped or compiled into a specific programming language, e.g. Java, Ada, C++, and C. The compilation process will create two pieces of code, a *stub* and a *skeleton*. This is similar to SUN's RPC mechanism. The stub represents the client-side while the skeleton represents the server-side. The stub and skeletons are only part of an execution if the two communicating objects are in different address spaces.

When a client object sends a request to the Object Request Broker (ORB) via an IDL stub, it invokes a statically referenced object. Clients also have another option when invoking server objects, which is to dynamically lookup a suitable object, construct a request, and pass it to the ORB. This dynamic invocation requires the ORB to have a database of interfaces that are available. This database is referred to as the *Interface Repository*.

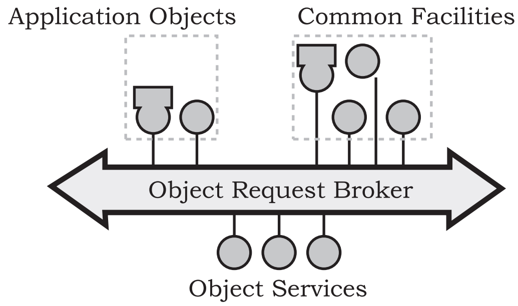


Figure 3.3: The Object Management Architecture.

At the server-side, a static request is passed from the object adapter via the IDL skeleton to the server object. Dynamic requests passes through the Dynamic Skeleton Interface. A dynamic request can require the creation of a server object, if that is the case the object adapter will lookup an implementation in the *Implementation repository* and create a server object before passing the request on.

CORBA and two extensions, CORBAservices and CORBAfacilities constitute the *Object Management Architecture* (OMA) which is displayed in Figure 3.3. Within the OMA there is a third category, application objects, which supports a specific application domain.

OMG CORBAservices include services and utilities useful, for objects in general, and distributed objects in particular. There is a *naming service*, which allow objects to reference other objects by name, and a *security service*, which implements protection for either a single object or for groups of objects. Among other services, there are database related services, such as a *transaction service* and a *query service*, and more general services, such as a *time service* and a *licensing service*.

CORBAfacilities [47] are divided into two categories, *horizontal* facilities and *vertical* facilities. Horizontal facilities support application development in different domains, while vertical facilities are more domain specific. The horizontal, or common, facilities are further divided into four major domains, identified so far; User interfaces, Information management, System management, and Task management. The vertical facilities have not yet been standardized.

CORBA is by far the most developed component infrastructure, but OMA does not go beyond objects and, thus, does not embody all fundamental requirements of components.

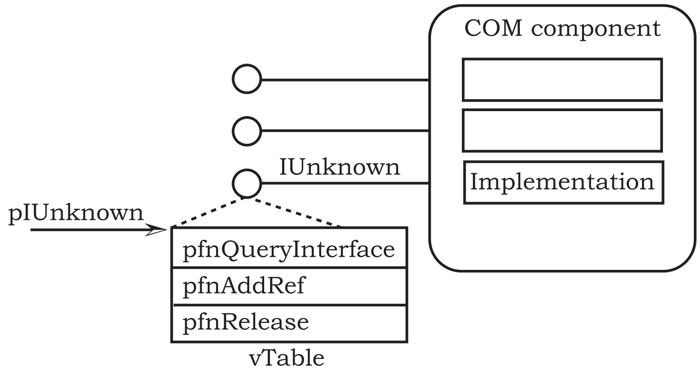


Figure 3.4: A COM component and its interfaces.

COM/DCOM/Active X

Microsoft component infrastructure COM was introduced in 1993. It stems from Microsoft's compound documents technology *OLE* and Visual Basic extensions (VBX). At that time, Microsoft did not worry about networking in general or the Internet in particular. COM therefore was designed and developed as a *single machine* infrastructure. Objects did not pass messages from one to another via some distributed system middleware, instead messages had the limitation of being passed via the internal message loop in the Windows operating system. If two objects resided in different processes, a simple interprocess communication mechanism facilitated the communication.

Except for limitation of restricting objects to communication within the same operating system, COM had several similarities with the initial CORBA standard, focusing on platform and language independence, and binary components.

A COM component uses the same separation as a CORBA component does, that of interfaces and implementations. COM objects interact via interfaces, which are collections of functions. All interfaces can be referred to via a name, which is not guaranteed to be unique, and a reference. A reference is a *Globally Unique Identifier*, GUID and is generated via an algorithm which guarantees this uniqueness. Classes are also given unique id's. The id's for for the class and the interfaces that the class implements are embedded into the COM component.

A COM object is created from a COM class. The class implements one or more interfaces as depicted in Figure 3.4. Every COM component (or class) must implement at least the *IUnknown* interface which allows other components to query

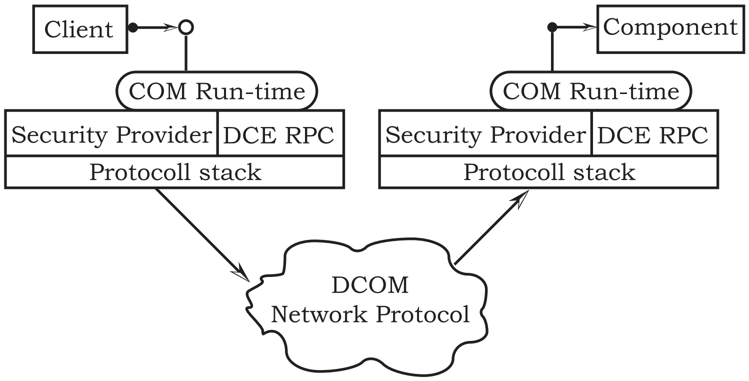


Figure 3.5: The DCOM architecture.

a component for any other interfaces that this component provides. The *IUnknown* interface also specifies two reference counting methods that other objects call when they have a valid reference to that object and when they no longer are to maintain the reference. Interfaces have a table with pointers to the functions constituting the interface. Interfaces can also be outgoing, specifying call-back functions. Clients using the component have to provide and register call-back functions in the server before using it.

COM does not support interface inheritance, so existing interfaces cannot be extended. Given that each interface has a unique id, no interface can be modified. Every extension to an existing interface must be in a completely new interface. This requirement make it easier to handle the otherwise difficult problems involving component versions and backward compatibility.

Shortly after the introduction of COM, Microsoft realized that the future lay in distributed computing. But COM, designed for local use only, required an extension to the model to handle this. The work on an extension resulted in DCOM. To make the description of DCOM a simple one, one can see DCOM as “COM with a wire”. The IPC communication used for objects in different processes was extended to a simple remote procedure call mechanism. The resulting architecture is depicted in Figure 3.5. Naturally DCOM also handles all local and cross-process calls.

OLE and Active X controls are important concepts in Microsoft’s component model. These are entities built upon COM / DCOM which rely on the provided infrastructure. A component must qualify for these categories by implementing a predefined set of interfaces common to all OLE / Active X components.

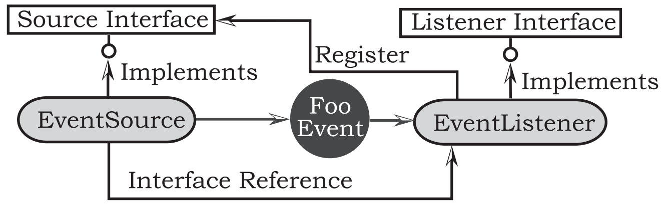


Figure 3.6: Overview of the event model.

Java Beans

SUN Microsystem's infrastructure for components is built on Java Beans [48]. Around Java Beans new services and technologies pop-up sources that never seem to be exhausted.

The basic building block, the Java Bean, is a set of classes with related resources. What is somewhat confusing is that a bean is both a design and run-time entity. Entities created from the bean are also called beans. Beans are designed with graphical composition in mind. This is reflected in SUN's naive definition of a bean.

> "A Java Bean is a reusable software component that can be manipulated visually in a builder tool."

The bean model describes a set of aspects on a bean that distinguishes a bean from other Java classes. These aspects are *events*, *properties*, *introspection*, *customization*, and *persistence*.

Events
: All beans display a set of access-points where other beans can hook up and receive events fired from the bean. Events are used to notify all clients and propagate the change state. The event model, described in Figure 3.6, is extremely simple. A client bean that would like to listen to another bean is required to implement a predefined *listener interface*. The client registers the listener at the source bean. When an event is generated in the source bean the specific event handling method is called for in all registered beans. Information can be passed to clients in an *EventObject*. Event objects contain specific event information, such as the event type and other state related information.

Properties
: Properties are named attributes in a Java Bean and are used to customize

appearance and behavior. Each property has a modifier method, which is used for setting the value, and a selector method which accesses the current value of a property. Java Beans supports *bounded* and *constrained* properties. A constrained property is checked for bound violations for every modification. For the check, the bean sends out a *veto event* to every registered listener. If a client opposes the change, it “throws” (sends) an exception which the bean “catches” (receives) and reverts to the previous value. A bound property causes a *property changed event* to be fired. In this manner all registered listeners will be notified when a change is made.

Introspection
Introspection concerns the ability to find out which methods, properties, and events a bean component supports. This is an extremely powerful technique which can be used in, for instance a design tool or at run-time. Java introduces and supports reflection on Java classes with the *Core Reflection API*. Java Beans proposes and supports two types of introspection. The simplest type is via programming conventions. For instance, the property `bar` will have a modifier `setBar` and a selector `getBar`. The alternative type requires that programmers specify all sufficient information in a separate *BeanInfo* class.

Customization
When a developer or user composes beans into an application in some design support tool, one part of the task involves customizing the individual beans. In order to support this, the Java Bean specification proposes two types of customization support. For simple beans it is most often sufficient to export properties. A design tool “introspects” (inspects) the bean and creates a *property sheet*, which is used for customization. The alternative, for more complex beans, is to have the bean provide a *customizer class*. The customizer class is graphical component that controls the customization of a bean.

Persistence
Instances of Java Beans can easily be stored on secondary storage. The specification requires a bean to support the *Java Object Serialization* mechanism, which provides support for saving and restoring the internal state of a bean. The persistent behavior can either rely on the simple mechanisms provided by the serialization specification or as customized in the bean, called *externalization*. For instance, a design tool can externalize a set of customized beans so the requested apparence and behavior is guaranteed.

In combination with Java’s model for distributed objects, RMI [49] and the object serialization service, Java Beans has become a true competitor to COM and

CORBA. Java Beans still have some glitches but new developments and modifications continuously improve the model.

3.3 Frameworks

A framework is a set of interacting and cooperating classes that a user can reuse via specialization or direct instantiation in an application. Some of the classes in the framework can be abstract, i.e. only interface specifications. This is our interpretation of the many definitions available [50, 51]

Frameworks provide an abstract design for a set of related problems. The applications must not necessarily be in the same domain, but at some conceptual level the basic requirements should be the same. The two classes of frameworks can be described as *vertical* frameworks that provides domain specific support, and *horizontal* frameworks providing more general application support for use in many different domains.

A common misunderstanding is that frameworks are “object libraries”, which is not true. Frameworks are not function libraries in a traditional sense, since the communication within a framework is bi-directional. A method in a framework can call an abstract method, which is implemented in the application specialization of a class. For instance, many graphical user-interface frameworks use callback routines for the implementation of different actions. The callback routines are implemented outside the framework.

One of the most famous object-oriented frameworks is the Smalltalk MVC framework [52] for graphical user interfaces. This framework provides a set of classes for *views* and *controllers*, which can be reused or specially adapted for an application. If the MVC framework exemplifies a vertical domain specific framework the C++ Standard Template Library [53, pp. 427-688] (STL) represents a horizontal framework. STL includes, among other things, container classes, algorithms, and iterators.

3.4 Patterns

The idea and notion of design patterns originated in architect Christopher Alexander’s work on a language of patterns facilitating communication and simplifying design of homes and urban districts. Alexander’s idea was to let ordinary people participate to a large degree in the design of their homes and surroundings.

For software engineers patterns became a natural continuation on the path of using abstractions to simplify understanding and construction of applications. Beginning with code abstractions, continuing with data abstractions into abstract data-types and classes, design patterns take the abstraction capabilities to a new level.

The idea of applying design patterns to software was first presented by Erich Gamma et al [54] in 1993. In the original presentation, a template was proposed to be used for the documentation of design patterns. A somewhat modified template, consisting of thirteen sections was later presented in [40].

Header
: The header consists of the name of the design pattern, a jurisdiction and a characterization. The name is important since this will be added to the design vocabulary and should comprehend the essence of the pattern. Jurisdiction in short, describes where this pattern applies while the characterization concerns what the pattern does. We will describe these more below.

Intent
: This section presents the issues that this particular design pattern addresses and what the design pattern does. It should also include a rationale that describes the underlying foundation for the pattern.

Also known as
: Sometimes a design pattern can appear several times with different names. This naming problem occurs since the same pattern could be applied to several different problems. If the pattern is known under some other name or names, a reference to these patterns is created.

Motivation
: The motivation section describes the existence of the pattern and demonstrates its application to a particular problem illustrating how the class structures and object structures solve the problem. This “example of use” aims to assist the reader in understanding the latter an more abstract parts of the pattern description.

Applicability
: In which situations can a design pattern be applied? Recognizing a potential problem spot in a design and finding an appropriate design pattern that would solve this problem is difficult. The applicability section is intended to describe typical situations wherein a particular design pattern is useful and how to detect these.

Structure
: This is a graphical representation of the pattern in some well-known notation, such as the Unified Modeling Language (UML) [55]. This section can include both a view that describes the static relationships among the

participating classes and some type of interaction diagram describing how objects interact dynamically in the pattern.

Participants
: This section presents the participating classes or objects in more detail, listing the responsibilities, and gives a rationale for a participant.

Collaborations
: The participating classes or objects collaborates in order to exhibit the expected behavior. This section presents and explains both the static relationships among classes and the dynamic collaboration among objects.

Consequences
: No pattern is always a perfect match for a particular problem. As information for a pattern user the consequences, tradeoffs, and risks should be presented and explained. That is done in this section.

Implementation
: Implementation specific consequences for a particular implementation language or class of languages is described separately. This section can also include hints and specific idioms that can be used when a pattern is implemented.

Sample code
: This gives examples of fragments of code illustrating the implementation of a pattern in a specific language. The choice of language depends on the organization, but it should be a well-known object-oriented language.

Known uses
: This section presents where this pattern is used in real-world applications. At least two different examples are to be included.

Related patterns
: Different patterns can often be used together to solve a more complicated problem. If it is common to apply this pattern in combination with some other patterns, a reference to those is extremely helpful. If a user searches for a particular pattern to solve a problem but really should look for a combination of patterns, a explanation of that in this section can greatly limit the effort involved.

In the header of a pattern description there is a classification section that classifies the pattern in two different categories. First we have the *jurisdiction* classification, which include three different categories; class, object, and compound. Class jurisdiction covers different static relationships among base classes and their specialization's, while object jurisdiction describes relationships among objects. The third category, compound jurisdiction, refers to patterns dealing with relationships among groups of objects. The second classification dimension is concerned with pattern behavior or intended use, and provides three different classes; *creational*, *structural*, and *behavioral*. Creational patterns concerns creation of objects, structural ones how classes and objects are composed, and behavioral ones how classes or objects interact to provide a specific behavior.

3.5 Conclusions

We devoted this chapter to related technologies that are of great importance to better understand how software architectural topics can fit into a development project.

Even though aspect oriented programming has no direct relation to software architectures except that both address the problem of handling quality requirements, architectural design can be regarded as aspect oriented design, wherein views extract and display relevant constructs to developers considering a particular quality or set of qualities. In subsequent sections we present a technique, where aspects can be modeled and described separately in architectural designs. Our focus will be on aspects that are somewhat related to reconfiguration, such as reliability and load balancing.

To effectively use the new technologies for component based software engineering, the developers must be able to reason at a higher level of abstraction. Here architectural description comes into consideration. One of the fundamental and still unsolved problems with component based application development is how to introduce non-functional requirements into an application. Since many of the non-functional requirements are spread over the system, i.e. impossible to derive from a specific component, architecture and the capabilities to create designs that partially meets these requirements become extremely important.

The connection between an architecture and a framework is easy to see. A framework implements an abstract design, which has an architecture. This abstract architecture restricts the design space for detailed designs. For instance, many GUI frameworks use message passing as the fundamental connector for components. Every application that uses this framework has to adapt to that architectural style. Frameworks can also be used to transfer architectures within a program family. For instance, a domain specific architecture can be implemented in a framework which is used in all development projects for that family.

The relation between design patterns and architectural styles is also obvious. One problem is that sometimes they are too similar, making it difficult to really see the difference. But there are important differences including level of abstractions and intended use and if these are not taken into account a variety of design problems can result.

Chapter 4

ADDING REACTIVENESS TO ARCHITECTURES

In this chapter we introduce the fundamentals of dynamic reconfiguration of applications. We discuss why it is natural to address dynamic reconfiguration at the architectural level and characterize the dynamic changes possible in an architecture. We will also discuss various other important issues concerning the initiation and control of dynamic reconfiguration. Further, an overview of current approaches and support for specification of dynamic architectures in existing description languages is given. Finally we present architectural agents, an important part of our approach to managing dynamic architecture efficiently during design. At the end we present some "proof of concept" applications where architectural agents play a prominent role.

Advances in software engineering technology continue to assist engineers in building more complex systems than before, but even if these systems are more complex they sustain the same problems of previous developments. First, they are never error-free and second, they need continuous improvement to meet users requests, such as demands for more and improved functionality. These changes often result in downtime, where users can not use the system. As the number of users grow and daily work more and more depend on access to applications these downtimes causes irritation and users do not tolerate these in the long run. In other types of systems downtime can be interdicted, for instance the switching systems in the telecommunications domain. A dynamic reconfigurable system that updates or replaces software without stopping the executing applications is needed.

Raising the level of abstraction is a well known strategy to manage complex problems and dynamic reconfigurations adds an additional dimension to the already complex questions inherent in the design of , themselves, complex systems.

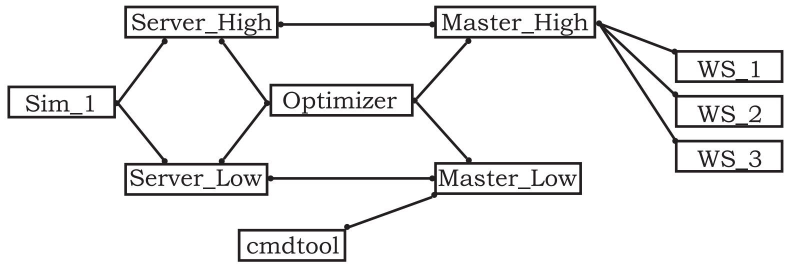


Figure 4.1: Architecture of a simulation application.

It is advantageous to pay attention to the dynamic aspects early in the system design and development process, in the general design phase, much prior to the detail design or implementation phases. However that is not to say that architectural considerations are not prominent in all development phases. Dynamic reconfiguration is best discussed via the components and connectors concepts introduced earlier. It is especially important that early in the design process a common understanding within the design group is reached about this issue in order to properly reflect the consequences and effects in later design and implementation. Consequently, we strongly recommend taking dynamic reconfiguration issues into account at the highest levels of design abstraction.

4.1 The simulation architecture

In this section, in order to bring the topic directly to the fore, we will present a real world example of an application with a dynamic architecture.

This application is a distributed simulation application supporting fail-safe simulations on heterogeneous clusters of workstations. The application has been used in several development projects by a Swedish company. It runs on several workstations as long as these are idle. When a workstation returns to application operation, the simulation immediately stops executing on that workstation.

An overview of the high level architecture is shown in Figure 4.1. The figure does not display every connector between the participating components. This instance of the simulation runs on three workstations and accepts jobs on two priority levels.

Given the architecture depicted in Figure 4.1, one could get the impression that all connectors shown are valid at any given point in time. But this is not true. In order to fulfill the fail-safe requirement, the developers have chosen to isolate every component in the architecture, especially the “workstation” components and the “simulation master” components. This means that no connector exists, except when the system is to actually use them. After a connector has been used, it is removed. For instance, when a user returns to work and generates significant load on the workstation, the workstation opens a connector to the “Master” component and signals that no new simulation jobs should be scheduled on this workstation.

We will later return to this application and show how we (1) improve the architectural description by introducing architectural agents and (2) model the dynamic aspects of this particular configuration more explicitly in the specification.

4.2 Dynamic reconfiguration

Producing high quality software, on time, and keeping costs within reasonable bounds have been three major goals from the very beginning of software engineering as an engineering science. To achieve this, several theories, methods, and tools have been developed, all supporting some part of the total idealized design process. Even though software developers have a wide range of tools and techniques available today, there are still severe problems. These are most directly associated with the increased complexity inherent in modern software systems. As a result, hardly ever are deployed systems either error-free or fully functionally satisfactory. The result is that, once brought into operation, systems undergo a series of “patches”, “fixes”, modifications, and changes in a high pressure environment resulting in not only system unavailability, but quite often a severe diminishment in logical clarity of the total system. This degradation continues throughout the often quite long operational cycle of a system as incremental changes in functions or the adding of functions or the adding of functions to adapt it to new operational requirements. In order to reduce operational costs, if not to considerably extend the life of the system, it is most important to design systems so that maintenance enhancements maintain the logical clarity of the system. That has to be kept in mind when recommending adding an additional consideration, dynamic reconfiguration, to the architectural design repertoire.

At the same time, increasingly many functions dependent on software in business, industry, and the home are depended on to be immediately available. Having these services unavailable du to updating the software is annoying for users. And there are other types of systems – real-time systems – that cannot afford to be taken down, for instance, telecommunication systems, command-and-control systems, and any other systems requiring continuous operation.

In a not-so-far-distant future, there will be an increasing number of applications with massive distribution and mobility characteristics adding considerably to that class of system which must be modified during operation or “on the fly”. Applications will no longer be huge monoliths, on the contrary, the current trend is towards thin clients where functionality is plugged-in on “when–needed–only” basis. For these types of applications, we often find other problems wherein dynamic configuration is a strong consideration, such as synchronization of updates so we are able to guarantee that software executed on different hosts (nodes) are compatible.

For the systems described above, a partial solution lies in dynamic reconfiguration wherein changes are introduced dynamically and incrementally without affecting the state or considerably reducing the performance of the system.

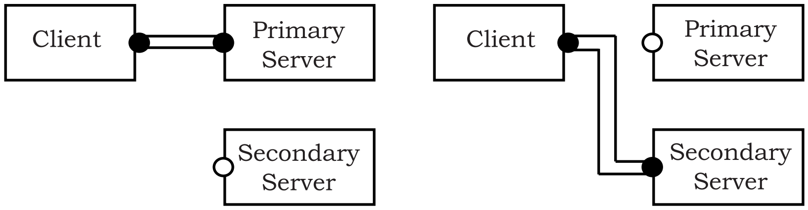
The idea of dynamically modifying certain lower level elements of an application is not new. In 1976, Fabry [56] presented a technique for replacing types dynamically. Over the years several strategies have been developed and implemented, spanning the range from hardware solutions with redundant processors to software based systems supporting distributed applications. A more detailed overview of previous work is to can be found in Segal et al. [57].

4.3 Architectural reconfiguration

A software architecture is a configuration of components and connectors as stated any number of places previously. This perspective is most useful when considering the design of the dynamic aspects of a system, since reconfiguration concerns the manipulation of structures. The systematic architectural representations, most easily and clearly expressed as combinations of components and connectors, provides designers and system developers with a logical foundation upon which to make appropriate design decisions, perform testing, and implement system functionality.

There are a range of dynamic architectural reconfigurations that can be made. First, new components and connectors can be dynamically added to the architecture. Adding a component typically also involves adding at least one connector. Second, connectors and components can be removed from the architecture. These two basic operations can be combined into a third type of reconfigurations wherein a component or a connector is replaced. This type of reconfiguration is practicable as long as the new component’s or connector’s interface is consistent with that of the replaced entity’s. If one or more interfaces have changed further modifications must be applied.

Other reconfigurations preserve the set of participating architectural elements but affect the configuration’s operation. Connectors can be re-wired from one component to another. In some applications components can be temporarily



(a) Before reconfiguration. (b) After reconfiguration.

Figure 4.2: Structural reconfiguration.

shifted about, if not removed from operation, within the architecture. An example of this is to be found in functions designed to achieve load-balancing.

Finally, we have the most complex reconfiguration, the change of architectural style. Reconfiguring an architecture into another style requires applying more minor modifications in a highly structured and controlled manner.

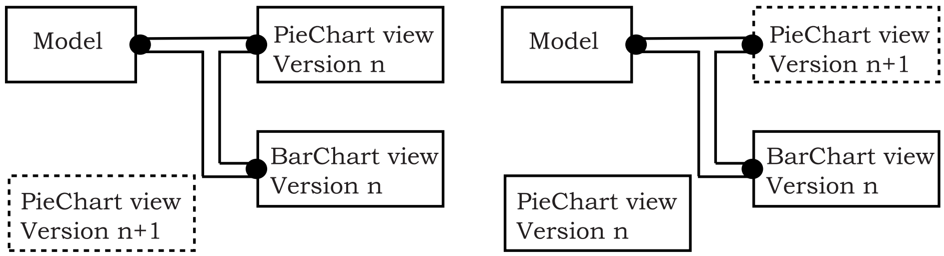
Fortunately for clarity's sake, we can make a distinction between structural reconfiguration and updating reconfiguration. The reason for doing this division is that all structural reconfigurations will require additional analysis of different architectural properties, such as conformance to a specific style. For instance, if we add a filter and a set of pipes to a "pipe-line" architecture, we must analyze the resulting architecture and confirm that no loops have been introduced.

4.3.1 Structural reconfiguration

Structural reconfigurations include dynamic changes to the inter-component communication pattern which is delineated by the connectors. The communication pattern will also change when components (or connectors) are added or removed. These types of changes to an application can easily be described and visualized at the architectural level. In Figure 4.2 the client component is re-linked to the secondary server dynamically.

4.3.2 Updating reconfiguration

The updating reconfigurations include replacement of existing components and-/or connectors with new versions. The architectural description is sufficient for describing these activities as well. In an architectural description the compo-



(a) Before reconfiguration. (b) After reconfiguration.

Figure 4.3: Updating reconfiguration.

nent's interface and the connector protocols are formalized. Therefore, we do not consider the component or connector internals. The component or connector can be upgraded to new versions as long as the component-connector interfaces are consistent with previous versions. If the interface is modified, a more complex reconfiguration must take place, most often a combination of structural and updating modifications to the architecture. In Figure 4.3, the GUI-component PieChart View is updated dynamically and replaced by a new version of the same component.

4.3.3 Initiate dynamism

Another important aspect of architectural dynamism is when and how the modification tasks are initiated. In Figure 4.4, we see two types of events that cause reconfigurations. Examples of external events are hardware failures and communication problems. Internal events can be load-balancing or other exception events in the application. Previous work has focused on what can be called *planned reconfiguration* where the activity is initiated and managed from the outside. The alternative, self-governed or internal reconfiguration, comes into play when the application itself is responsible for the reconfiguration, both its initiation and implementation. These two strategies have different considerations to be taken into account when considering response time or the efficient handling of reconfigurations, both internal and external.

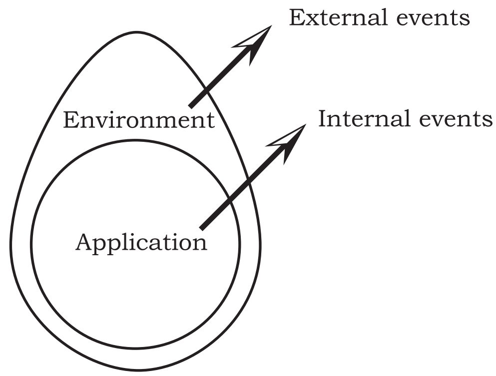


Figure 4.4: Events initiating reconfigurations.

External reconfiguration

External reconfiguration is both initiated and conducted from the outside. For instance, a system operator opens a dialogue with the application, sending a command sequences controlling the reconfiguration. This approach is simple and straightforward, but there exist some drawbacks.

External initiation is suitable for planned reconfigurations, for example upgrading software. Other events which do not require direct action to be taken can also be handled from the outside.

Some events, external and internal, are not easy to predict and plan for and some require direct measures in order to satisfy system requirements. These types need a different approach where the software itself detects these events and performs architectural reconfiguration without external involvement. For instance, a robust system sometimes needs dynamic routing to guarantee continuous access to data. The system must react directly to these events and cannot afford to wait for a manual reconfiguration.

Another problem with external reconfiguration lies in the amount of information that a human operator can be reasonably expected to have available and in mind. If an external operator is ,conscienteous he or she must keep all the knowledge about the current application configuration available in order to be able to make correct decisions. For large systems this is practically impossible. In order to properly manage a reconfiguration, operators do need some type of support system assisting them, providing the necessary information so the operator can complete the task.

Self-governed reconfiguration

The alternative to external reconfiguration is internal or self-governed reconfiguration. Here the application include some reactive functionality with the capability to react to events, external and internal, each previously specified event triggers a reconfiguration task.

The advantage over external reconfiguration comes in terms of faster response time. Certain events must be treated immediately, for instance failures in mission-critical processes described in the previous section.

Another important advantage lies in access to information. The application can access the current configuration and use this information to make appropriate decisions given that the systems stores the current configuration. Keeping adequate and up-to-date information is not a trivial problem but this information is up-to-date and can be used to perform reconfigurations spanning the range from simple to moderate complex reconfigurations. For extremely complex reconfigurations external intervention is preferable. But if the complex reconfigurations have been tested thoroughly, these also can be managed by the application.

The drawback with internal reconfiguration is that we must include more functionality that increases the complexity of the application. But introducing these capabilities in a structured and controlled manner can reduce the overall system complexity. Other drawbacks involves faults in the logic controlling the configuration, which causes failures with severe effects on the system and difficulty of managing complex reconfigurations.

4.4 Support in existing ADLs

Dynamism was one property touched upon in the framework for classification of architecture description languages in Section 2.2.1. In this section we review the approaches taken in four of the important current specification languages concluding by pointing out some essential missing elements for describing dynamic architectures. The review presented below is cursory and does not cover all aspects concerning dynamism in detail.

4.4.1 WRIGHT

Extended specifications in Wright [58, 59], add event based reconfiguration to ordinary Wright specifications. This technique provides the means for exhaustive static analysis. Nevertheless, there are several limitations on the expressiveness of dynamic aspects in Wright. One problem is that there is no clear separation of concerns. It is difficult to see when an event occurs and what will happen. The authors introduce a separate, system-wide, configuration component which will govern all reconfigurations in the system. This is due to the fact that the specification becomes a mixture of the static structure and the dynamic behavior. Another problem is that the support from Wright is limited to the design level. There is no direct support for the implementors so the gap to bridge when implementing the system is still wide. In an extremely flexible system where components are added or removed in an unpredictable manner, Wright can't be used since the set of possible configurations must be decided upon beforehand.

4.4.2 Darwin

Darwin is another architectural description language, where the implementations supports dynamic aspects of an architecture. The intended application area is that of distributed systems. Darwin provides support for dynamic reconfiguration at run-time in the form of a reconfiguration manager. A supervisor can send directives in a script language to the system that invoke dynamic reconfiguration. Darwin has no explicit support for dynamism during design. The implementation phase gets support from the distributed system support framework. The loosely coupled components and connectors constitute the foundation for dynamic reconfiguration.

There is no way of expressing dynamism in Darwin specifications, neither when a reconfiguration takes place nor what will happen. But the generality of the elements in a Darwin implementation makes it possible to perform dynamic modifications. In a Darwin implementation, every element can be either added, removed, or replaced dynamically but from external tools only.

4.4.3 C2

An approach similar to that in Darwin is used for architectures implemented in the C2-style. The C2-framework introduces components and connectors as implementation entities. A component or connector corresponds to a class in the implementation language. At run-time system maintainers use a shell-like tool ArchTool [60]. ArchTool implements a language where the manager can express modifications (e.g. add a component) and constraints. Applications in the C2-style are aware of the architecture, and the meta-information is directly accessible from ArchTool.

The reconfiguration manager approaches are valid for both updating reconfiguration and structural reconfiguration, but support for unplanned events is absent.

4.4.4 Rapide

Rapide [61], is an architecture definition language were events are first class constructs. In the specification phase a designer can describe the set of events that a process can generate and a set of reactive statements, these latter describe how the process is to react to events generated by other processes. Additionally Rapide provides a conditional expression for connections, e.g. if an airplane is within communication range the control center can open a connection. This can be used to model dynamic architectures. Still, the Rapide model lacks in expressiveness, mixing dynamic and static behavior in one specification.

4.4.5 Conclusion

This section has described dynamic aspects of existing architecture description languages. There is a need for two major improvements.

First, no language supports explicit specification of dynamic behavior. These properties are either mixed into the ordinary static descriptions or not specified at all. Second, no language supports the proper specification of when certain reconfigurations should take place. This can be expressed in Wright and Rapide but only intermixed in the static description. Rapide provides an event mechanism but this can be used in all specifications not solely for dynamism which makes it difficult to find where a certain reconfiguration is specified.

4.5 Requirements for a development environment

To fully support the development of dynamic architectures there are some issues that must be addressed during design, some during implementation, and some at run-time. Current approaches support some of these partially, but no tool or description language covers the lot.

4.5.1 Design support

The design support in a tool should pay attention to the following three issues. First, there must be support for expressing dynamic aspects of an architecture in the description language, i.e. what will happen and when. The "when" part is the reactiveness. It is important to separate the dynamic aspects from the structural specification since this will simplify the process of understanding of what happens dynamically.

Second, there should be means by which the designer can express events, both internal to the architecture and external. It must be simple to connect to explicit reconfiguration tasks handling the events.

Finally, the description of the architecture should be executable, i.e. the designer should be able to simulate the architecture, trigger events to test specific reconfiguration functions, and display the resulting architecture in animated form. Snapshots of the evolving architecture should be available for different analysis, such as those of completeness and style conformance.

4.5.2 Implementation support

To simplify the process of translating a design into an implementation, several aspects must be regarded. First, it is important to separate the reconfiguration functionality from the overall application functionality, as stated in [58]. This separation will simplify maintenance since parts can be replaced independently.

Second, another important aspect is the reconfiguration logic that is introduced in a reactive architecture. This logic must describe how the application reacts to certain events, e.g. a broken communication link will result in a re-link of a connector to the backup-server.

Finally, the environment should provide the programmer with sufficient support for mapping the specification of a reactive architecture into an implementation language. This can be in the form of a support framework with a direct mapping from elements of the architectural description to elements in the application language.

4.5.3 Run-time support

At run-time the run-time system must include support functionality for dynamic architectural reconfiguration. This is necessary for many reasons, for instance performance and reliability purposes. In an object oriented architecture, the replacement of a class (component) can affect multiple objects in the executing application. These objects must be migrated to the new class representation in an controlled and effective manor. The run-time support should provide the necessary support functionality to handle these kind of activities. A more in-depth description of this problem in reference to the Java language can be found in [62].

We cannot ignore that at times a configuration will be initiated and controlled by system operators. Therefore a system should provide a suitable interface whereby an operator directs and controls dynamic modifications.

4.6 Architectural Agents

Previous sections in this chapter discussed different aspects of dynamic applications and architectures. What we have seen is that there is a lack of support for

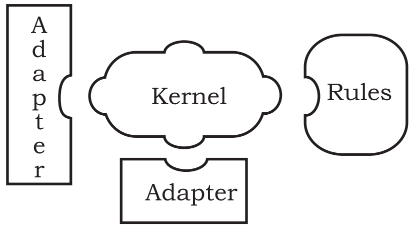


Figure 4.5: The Agent structure.

capturing dynamic aspects of architectures in current description languages. In order to fully support analysis, specification, design, implementation and operation of system with dynamic architectures we introduce *architectural agents*.

Architectural agents are an extension and modification of the software agent concept. Software agents work as assistants performing tedious tasks, such as information retrieval and indexing. The extensions and modifications we specify adapt agents to these purposes and streamline their inclusion in existing architectural representations.

Agents are capable of reacting to, external or internal application events. The main task is to decide if the application should be reconfigured due to the event and govern the actual reconfiguration.

In this section we give a full and detailed account of the operation of architectural agents. We discuss how to apply them in development projects, how to configure agents, how agents interact with applications, and conclude with examples.

4.6.1 The Structure of Agents

An agent consists of three major parts, depicted in Figure 4.5; an engine, adapters, and a rule base. The structure we use has been partially adapted from the development that led to the IBM Agent Building Environment (ABE) [63]. This is in-itself an excellent examples of a flexible and easily dynamically modifiable architecture.

The active part of an agent is the engine. An engine reacts to events, interprets rules, and takes actions. There are several different types of engines, spanning the range from very simple inference engines to more advanced engines utilizing reasoning and planning. In our work we use simple inference engines which allows us to specify simple reactive agents.

Rules are used to “program” an agent. A rule describes how the agent should react to specific events if specified conditions pertain. Rules are divided into three parts. Firstly the event part, describing the event or set of events that trigger the rule. Secondly, the conditional part which is used to further constrain the activation of actions. Finally, there is the action part which prescribes how the agent is to react to an event.

Our agents are meant to deal with architectural factors; However, they must also be capable as required of interacting with other systems, such as log viewers and databases. In order to provide a uniform and configurable interface between the agent and the architecture or other external systems, we use adapters. An adapter consists of a set of incoming and outgoing interactions points that components use for trigger events and agents use to act on a external system, such as the architecture.

In subsequent sections we describe the adapters and the rule-base more in-depth.

4.6.2 Agent–Architecture interaction

The agent interacts with the architecture via triggers, sensors, and effectors, as displayed in Figure 4.6. The communication is bi-directional with incoming triggers and outgoing sensors and effectors. The set of triggers, sensors, and effectors constitute the interface between a domain and an agent. The interface component is called an adapter. For each domain in an agent enabled application, a domain specific adapter is used, i.e. every external domain that an agent will interact with must be abstracted in an adapter.

The triggers are “proxies” used by a domain for event signaling. For instance, an application generates a event if a communication time-out exception is raised internally. When the application invokes the trigger a corresponding internal event in the agent will be generated. In event driven systems, these “triggering events” are handled separately and do not affect the application event handling mechanisms. In applications one could utilize from connecting the two event-mechanisms and use agents as configurable exception handlers. Imagine a client-server system, the communication objects throws “time-out” exceptions if the communication fails. If these “language” exceptions are re-directed to the agent, the agent can handle these exceptions and take proper actions, for instance reconfigure the application.

Sensors are used by the agent for monitoring an application’s state. For instance, a sensor can be used to monitor the condition of a components internal state. This mechanism would be most frequently implemented in the conditional part of a rule. Imagine a component which is about to be moved from one host (node) to another. If the component is active we have to wait until processing

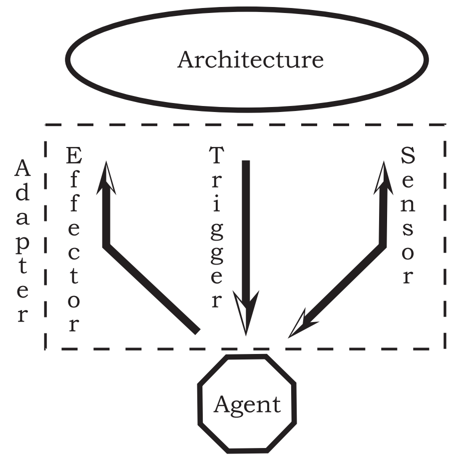


Figure 4.6: The architecture adapter.

stops before we safely can move it, in this situation the agent uses a sensor and polls the component checking, if it has gone idle. Sensors can be either binary and return a Boolean value or used to set unbound rule-variables. These variables are bound to a value and can be used in subsequent conditionals and actions in the rule.

The third type of interaction point is the effector. It is used by the agent to act on a domain. Effectors are typically outgoing proxies for functionality available in the domain. Some run-time systems, for instance the C2 framework, provide functions for dynamic reconfiguration. Other domains can provide functionality that can be useful in additional situations. Imagine a system where system operators are to be notified when the system has been reconfigured. In this case, the final action of a successfully processed rule could invoke an email application, via an email-adapter, and send a notification.

Introducing adapters will make the domain transparent to the agent. For instance, we can use a “specification adapter” during specification and a “run-time adapter” at run-time. It these two adapters provide the same interface, we can reuse the rules from the specification for the run-time agents. We can specify a C2 architecture and describe the dynamic aspects in agents. Then we reuse the rules and replace the C2-specification-adapter with a C2-runtime-adapter in the implementation.

```
// this is the adapter inclusion part
use system
use time

// this is the event part

event intervalAlarm internal->
      time::SetInterval(20,"seconds","intervalAlarm");

// here are all rules for the agent
intervalAlarm!->system::out("Interval alarm generated")
```

Figure 4.7: A simple ARL specification.

4.6.3 Configuring agents — rules

Our agents use simple inference engines. A rule consists of three parts; a event or set of events, a antecedent expression, and a consequent expression.

$$event!\, antecedent \;\Rightarrow\; consequent$$

The antecedent expression corresponds to the conditional part of the E/C/A-rule and logically implies the consequent part corresponding to the action. We continue to work on the full formalization of the *Agent Rule Language* (ARL). In our specifications we use a simple syntax with simple semantics. The ARL specification is given for each architectural element (component, connector, or configuration) and consists of three parts; adapters, events, and rules. We present a simplified example of the syntax in Figure 4.7. First, we specify which adapters need to be available to the agent; second, we specify the set of events that the agent reacts to and handles internally; and finally we specify the rule set. This simple specification creates an agent which prints a message every 20 seconds.

4.6.4 Cooperating agents

In more advanced applications several agents will coexist. They will constitute an agent architecture. This will add a new dimension to the design of a reactive system. Agents must be able to cooperate with functions being divided among them. For example one agent could be responsible for a sub-architecture or responsible for a specific functional quality, e.g. reliability. If one agent is responsible for handling one aspect in the complete system, it's a vertical agent. On the other hand, if an agent has several responsibilities within a restricted sub-architecture it's a horizontal agent. These two types of agents, horizontal and vertical, can be mixed in an application.

4.6.5 Agents in specifications

In order to use agents as a vehicle to specify dynamic architectures, we have to take into account a number of necessary additional support mechanisms. These include ones allowing the expression of typical modifications at the architectural level. Medvidovic proposes a special language for expressing such constructs in [64]. Our language, the Architectural Modification Language (AML) covers the necessary basic operations. Another important mechanism involves the simulation of specification's operation. Given the specification of a dynamic architecture, one should be able to run a simulation, trigger events, and review the resulting architecture. In this situation we can utilize two or more adapters working on different domains but providing the same interface. If we develop adapters that implement the AML and work on several different domains, e.g. a graphical representation and a text representation, we will be able to "simulate" different representations of the description.

Architectural Modification Language (AML)

The Architectural Modification Language implements the basic operations that can be applied on an architecture in order to perform a reconfiguration. Since the operations are low-level, works in progress to develop a standard set of more complex operations and user-defined operations based on these. The syntax and semantics of our AML is not yet completely formalized; however for the examples in this thesis current operations are sufficient. We outline the operation signatures and the informal semantics below.

`Load(Entity,Location)`
: Load a component or connector representation.

`UnLoad(Entity,Location)`
: Unload a component or connector representation.

`Connect(Entity,withEntity)`
: Connect a connector to a component or other connector.

`Disconnect(Entity,withEntity)`
: Disconnect a connector form another entity.

`Enable(Entity)`
: Enable a component or connector.

`Disable(Entity)`
Disable a component or connector.

`Signal(Entity,Message)`
Send a message to a component or connector representation.

Architecture description

We embed agents in static architectural descriptions. For our specifications we have chosen the ACME description language. ACME allows additional information to be added to a description via the *property* construct. We create a new property – Agent, which includes a definition of the agent that will work on this architectural entity. Future work will allow for the separate specifications of agents since we need to represent the case where an agent cooperates with more that one entity. However, the current approach is suitable for simple applications. In Figure 4.8 we create a simple Client component and add the agent as a property.

In this example we have modeled the dynamic behavior of a safety critical application. The Client accesses mission-critical data on a server, `Server1`. When the communication line fails, a time-out event is generated. This triggers the internal event `timeout` in the agent. The agent acts according to the specified rules and disconnect the `Server1Connection`, enables `Server2`, connects `ToServer` to `Server2Connection`, and finally signals `retry` to the `Client`.

With this example we have demonstrated how to use the ARL and AML in combination with an existing ADL. ACME is able to be used since it is the only language which allows for user defined extensions.

4.6.6 Agents in implementations

When implementing systems modeled with agents one can choose either to include agents in components and connectors or maintain the separation. In the best of worlds the only work needed should be replacing specification domain adapters with implementation domain adapters automatically when a developer presses the button to generate application code.

If the project uses an architectural framework, similar to the C2 framework, as the basis for implementation; agents could be included as a separate component type. If the system is to be implemented without support from an architectural framework, agents could be offered as classes.

At run-time, agents will work either as stand-alone objects or as integral parts of other objects. When an agent performs a reconfiguration it will require support from the architectural framework and / or other support structures. Agents need meta-level information in order to make appropriate decisions.

```
Component Server1 : Server  = new Server;
Component Server2 : Server  = new Server;
Connector Server1Con : RPCConnector = new RPCConnector;
.
.
Component C1 : Client = new Client extended with {
  Properties {
    Agent={
      use acme

      event timeout external->acme::Signal(Agent,timeout);

      timeout!->acme::disconnect(ToServer,Server1Con);
                acme::unload(Server1Con,local);
                acme::load(Server2Con,local);
                acme::enable(Server2);
                acme::connect(ToServer,Server2Con);
                acme::connect(Server2Con,Server2);
                acme::enable(Server2Con);
                acme::signal(Client,retry);
    }
  };
  Port ToServer : MessagePort = new MessagePort;
};
```

Figure 4.8: The Agent Property in a ACME specification.

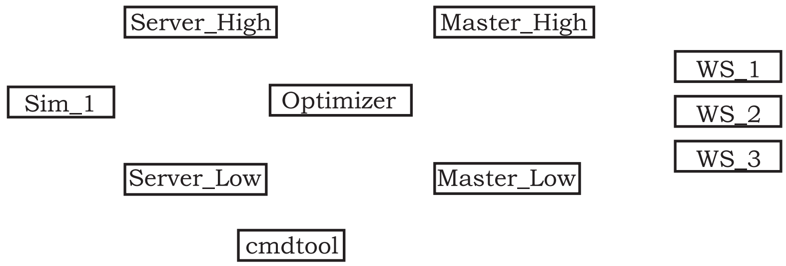


Figure 4.9: The "normal" configuration.

4.7 A reactive specification

In our work we have specified the simulation architecture described in Section 4.1. The designers of the simulation application expressed the difficulty communicating how the application really worked. The particular instance of the application supported jobs with two priority levels, high and low.

The architectural description in Figure 4.1 included all connectors in the architecture. But this view on the configuration is not especially interesting for our modeling purposes as it depicts all possible connections active, which is most often not reflecting the current configuration. The non-functional requirements imposed on the system required that the simulation tasks performed on the workstations should never terminate as a result of another process's termination. The designers made a design decision that no connector should be available if it was not used. A better description of the normal configuration of the simulation architecture is given in Figure 4.9.

Here no component is connected to any other connector. The connector will be created and instantiated dynamically on a when-needed-only basis. In this presentation we focus on the description of two events that cause the system to reconfigure itself dynamically:

1. *Optimize:* The optimizer is invoked every twenty minutes. It opens up connections to the simulation masters and servers, schedules jobs, and assigns them to processors. Then the connectors are closed and removed.

2. *Status change:* When a workstation is no longer idle (other processes generates evident load), the workstation component opens a connection to the simulation master component and signals that no new processes should be scheduled on this workstation.

```
Component Simulation_Optimizer : Optimizer_T {
  Properties {
  };
  Port getStatus;
  Port getComputationalProfile;
  Port asSlavesHigh;
  Port asSlavesLow;
};
```

Figure 4.10: ACME specification of the Optimizer component.

4.7.1 Invocation of the optimizer

In Figure 4.10 we show the specification for the optimizer component. The dynamic reconfiguration is initiated when an external timeout event is triggered. When this happens the optimizer communicates with the servers and masters respectively.

We add the reactive behavior in the Agent property. Extracts of the agent specification can be found in Figure 4.11.

```
Agent={
 ...
 event sim::start_optimize external->
  Signal(Agent,optimize);
 event sim::stop_optimize external->
  Signal(Agent,s_optimize);
 optimize!->
  connect(getStatus,getStatusHigh.toRole);
  connect(getStatusHigh.fromRole,HighServ.getStatus);
  connect(asSlavesHigh,asSlavesHigh.toRole);
  connect(asSlavesHigh.fromRole,HighMaster.asSlaves);
  ... // enable connectors
  sim::Continue_Optimization;
  s_optimize!->disconnect(ToServer,Server1Con);
  ...
 };
```

Figure 4.11: The agent specification for the Optimizer component.

In the specification we begin with connecting external events from the *sim* domain with the agent's internal event-mechanism. We have two external events, one which is generated when the optimizer is about to start the optimization and one which is generated when it has finished. The agent will react to the external *"optimize"* event by setting up the connections needed by the participating components.

In this setting the optimizer component will communicate with the simulation server for high priority jobs and both simulation masters. When the communication is setup, the agent uses the *Continue_Optimization* effector in the *sim* domain. When the "optimizer" component receives this notification it will continue and perform the actual optimization tasks. When it has finished it will generate a *stop_optimize* event which results in an internal–agent–event whereby the agent performs some housekeeping procedures and disables and removes all connectors that were set up before the optimization.

4.7.2 Status change on Workstation

When a workstation is idle the simulation system will use this computational resource. When a user returns and generates evident load the simulation must stop sending additional jobs to this resource. In Figure 4.12 we specify the component representing a workstation in the simulation system. The component has one port, *statusChanged*.

When non-simulation processes generates load on the workstation the workstation is notified by a external event. As a reaction to this event the workstation component should set up a connection to the simulation server and signal that no additional jobs should be sent. We have captured this dynamic behavior in an agent outlined in Figure 4.12. In the same manner the workstation component will signal the simulation server when the workstation is idle.

4.8 A reactive implementation

An important part of our work involves how to reuse agent rules, developed in the specification phase, in the implementation phase. We demonstrate how agents can control reconfiguration activities at the architectural level with a simple implementation that uses the C2 framework and the IBM Agent Building Environment (ABE). The C2 framework offers the capability to dynamically modify the architectural configuration. In this section we explain the architecture and configuration of an ABE agent and present a simple system where components are updated as soon as new versions appear on a version server.

```
Component Simulation_WS1 : Beast_Simulation_Slave_T {
  Properties {
   Agent={
    use acme
    use sim
    event sim::ws_not_idle->
     acme::Signal(Agent,statuschange);
    statuschange!->
     acme::connect(statusChanged,
                   WS1ChangedStatus.toRole);
     acme::connect(WS1ChangedStatus.fromRole,
                   HighServ.changeStatus);
     acme::enable(WS1ChangedStatus);
  };
 };
  Port statusChanged;
};
```

Figure 4.12: Reactive specification for a Workstation component.

4.8.1 More about the IBM Agent Building Environment

The IBM Agent Building Environment (ABE) is a toolkit providing the user with a general architecture to ease the use of agents in applications. ABE is based on a general architecture composed of five different components. Each component is based on a specialized framework, which provides a set of classes for agent programmers. We delineate the architecture in figure 4.13. We explain the characteristics and properties of the individual components in more depth below.

Agent Control
The agent control component is responsible for the initialization and overall control of a running agent. During initialization the control component initializes the library, different adapters, and engines according to separate configuration files. The agent control framework consists of a set of classes that provides an API which an application can use to dynamically configure an agent. By using the API the application also dynamically can load and store new or updated rules.

Adapters
The adapters are domain specific interfaces for the agent to the application and the applications environment. ABE provides a set of standard adapters for direct use (e.g. Time-adapter and File-adapter). The ABE adapter frame-

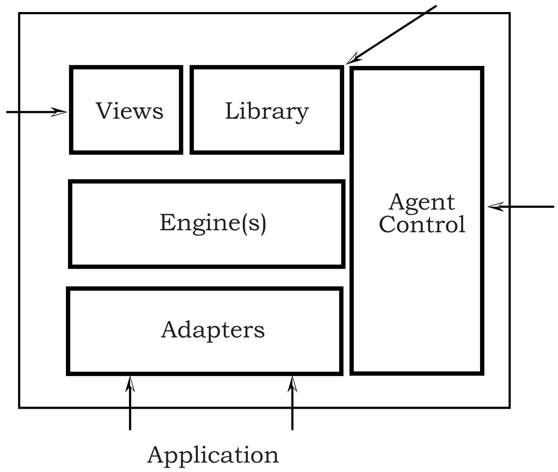


Figure 4.13: The ABE basic architecture.

work provides an interface for application programmers to write their own, domain specific adapters. Finally, ABE comes with a set of domain specific adapters, such as the Email-adapter and HTTP-adapter. These adapters are not applicable in all types of applications.

Engines

The heart of the agent is the engine. The engine will interpret all instructions to the agent. The instructions consist of rules and facts. ABE comes with a simple rule-based engine which is sufficient for many agent applications. The engine framework can be used to develop other types of engines, such as a learning engine or an analyzing engine. We will give a more detailed description of the engine, how to construct rules, and facts below.

Library

The library is the agent warehouse were the application can store rules and facts. The engines retrieve rules and facts for processing. The application will uses the Views component for access to the library.

Views

Views are used for insertion and manipulation of rules and facts. The framework components can be used for the development of different tools for library access. Currently the ABE provides a simple rule editor.

Rules and Facts

The rule based engine in ABE work with a subset of the Knowledge Interchange Format (KIF) [65]. KIF is a proposed standard for the representation and exchange of inference rules. A rule consists of two parts or expressions. In the first instance there is an antecedent-expression which logically implies a consequent-expression.

$$antecedent \;\Rightarrow\; consequent$$

Normally KIF uses a prefix notation.

$$(\Rightarrow\; (antecedent)\,(consequent)\,)$$

Both the antecedent and consequent expressions are built from logical atoms. A logical atom consists of a predicate followed by a list of terms. The list of terms is composed of zero or more terms. A term is either a variable or a constant. Variable terms are initially unbound, constants replace variables when these are bound. Constants and variables can be symbols, strings, integers, or real numbers.

KIF also allows the combination of atoms using logical AND and OR operators. These connectives can be nested and form an expression tree. When the logical implication in the root node is added, we have a complete KIF rule. As an example consider the statement “ If Beth is taller than Bill, then she should move to the front seat of the car”. This will result in the following KIF rule.

$$(\Rightarrow (IsTallerThan\ Beth\ Bill)\,(Move\ Beth))$$

The inference engine can combine rules and facts when evaluating expressions. A fact is an atom where all terms are bound. In ABE there are two types of facts, long term and short term. Long term facts are manipulated using Views and stored in the library. Short term facts, on the other hand originate from the adapters and are generated dynamically.

It is beyond the scope of this thesis to completely cover this topic. We will use and explain several rules in the example in section 4.8.2.

Adapters

The adapters work as the agent’s interface to the application or the application environment. The adapter represents a specific domain. The agent is configured with a set of adapters which constitutes the agent domain. Within ABE there are two types of adapters, core adapters and application adapters. A core adapter is useful for many applications (e.g. file-adapter), while the application adapter can represent anything that can be described in terms of events, conditions, and actions.

In the process of designing an application adapter, one should focus on events that are related to that particular domain. The adapter should handle all possible events within a domain, and if appropriate conditions are satisfied, provide means for the agent to take proper actions. In the next section we will examine a domain specific adapter, the architecture-adapter, in more depth.

4.8.2 A simple versioning agent

We demonstrate the use of architectural agents using a simple, but still useful, update system. Figure 4.14 shows the structure of the system. The update system has two components. First, we have the component server where components are stored. Applications can connect to the server over the network and down-load components. Each component is stored in a versioning system. Clients can, when down-loading a component, specify which version of the component is preferred. The second part is the application. In this case a variant of the simple C2-style StackArtist application presented in [64], with two different presentations of a stack.

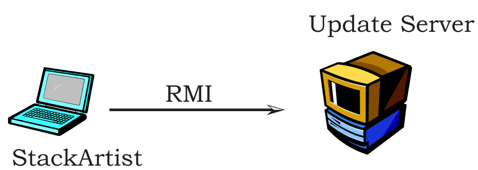


Figure 4.14: The Update Server system structure.

To get dynamic behavior we place an additional requirement on the application wherein we state that the application should continuously check if new versions of the used components appear on the component server. If a new version appears, the application should download the modified component and dynamically replace it.

When implementing this system we can choose to code the update requirement by hand. This will work but will lead to several drawbacks. First, we have to break up encapsulation of the used components to modify them. The result will be components with application specific functionality for dynamic download and updates. The reconfiguration functionality and logic will not be separated from the application functionality. If we, on the other hand, choose to implement the requirement using architectural agents we will not run into these problems.

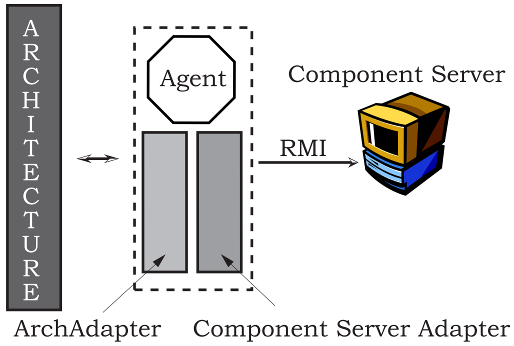


Figure 4.15: The Component Server and Update Agent.

In this simple example one agent is sufficient. We attach the agent to the application, as shown in figure 4.15. The agent is configured with two application specific adapters. First, we have the component server adapter which implements an interface for the update server. The component server adapter uses RMI [49] for communication and contains sensors for retrieving version information and different effectors for the downloading of components. The second adapter is the architectural adapter.

The architectural adapter is specific for C2 and can add and remove components and connectors, weld and unweld elements and extract information about the architecture for the agent. Since it is specific for C2 it will not be usable with any other architectural support framework.

By connecting and configuring the agent properly with adapters, facts, and rules, we will get the required dynamic behavior. The "facts" provided to the agent is network addresses and other environment specific information. The rules are of much more interest. In Figure 4.16 we present two rules. We use an "infix" notation to make the agent specification more readable.

The first rule sets an alarm which is generated every two minutes. The second rule is triggered by the alarm and performs the actual update of the component. First, the agent queries the component server to get the version number of the latest version available. Then it compares this with the version number of the component in the executing application. If it is a new version, the antecedent expression is evaluated as "true" and the consequent expression is processed. In the consequent expression the agent will download the new component. After unwelding and finishing the old component the new component will be added, welded, and started.

```
EventName("AGENT:STARTING") =>
    SetIntervalAlarm( 120, "seconds", "Check", "Check")
EventName("Time:Alarm") AND
AlarmId("Check") AND
CompCheckVersion("StackPieArtist", ?version) AND
ArchCheckVersion("StackPieArtist", ?version) =>
    CompDownload("StackPieArtist", ?classname) AND
    ArchUnWeld("MainBus", "StackPieArtist") AND
    ArchUnWeld("StackPieArtist", "BindingBus") AND
    ArchFinishComponent("StackPieArtist") AND
    ArchAddComponent("StackPieArtist", ?classname) AND
    ArchWeld("MainBus", "StackPieArtist") AND
    ArchWeld("StackPieArtist", "BindingBus") AND
    ArchStartEntity("StackPieArtist")
```

Figure 4.16: Rules for the Update Agent application.

With this simple example we have shown how to use agents for architectural dynamic reconfiguration. The agent is separated from the application and can easily be reused. All applications based on the C2 framework can reuse the update agent as a whole directly, with no modifications to the source. Naturally, rules and facts will have to be modified to apply to the new environment. The different adapters which the agent uses is also individually reusable, for instance the C2 adapter can be reused in any application built using the C2 application framework.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The software technology shift towards component-based and distributed systems will create a new type of application wherein applications modify their internal structures independently. In the near future mobile terminals will be able to configure their software in different ways, depending on in which environment the terminals are currently working in . This great leap in technology requires a new viewpoint and support mechanisms for software developers. Systems will no longer be huge monoliths executing on the stand alone desktop computers. By no means do current methodological support, tools, and techniques address these topics sufficiently.

Dynamically changing applications will be more difficult to model. The internal structures will no longer be static. On the contrary, these structures will evolve rapidly. For instance, one basic class model used in a mobile application, may change several thousand times in one day if the carrier of the mobile terminal changes environments several times every day and functionality is plugged-in on a "when-needed-only" basis. A defined process for design, efficient support for implementers, and embedded support in the run-time system will facilitate the development of applications with dynamic structures.

In this thesis we have presented the discipline of software architectures and argued that it is advantageous to capture the dynamic aspects of an application at the architectural level of design. In our survey of architectural models and description languages we found that there was a lack of support for describing software systems where the application itself initiated and controlled reconfiguration.

Our contribution, the architectural agent, adds a new dimension to traditional architectural specifications. Agents allow designers to express run-time evolution which is initiated and governed internally in the executing application. We have created a simple modeling technique yet one capable of describing systems with complex dynamic behavior.

Agents can be included in existing static architectural descriptions without requiring modifications of these descriptions. We have also shown how to develop the rule sets that control agents and how rule-sets developed during design can be reused in implementations.

We also believe there are other application areas where agents can be useful, for instance in component based development. We believe, based on our current observations, that architectural agents can aid the design and implementation of component based systems, especially design for non-functional requirements. During design, agents can be used as vehicles for modeling dynamic modifications, and supervise application state and data transmission. Agents can pinpoint special requirements, put on specific components, and guide developers as they choose pre-developed components or design new ones. In this fashion, agents support the development of different aspects separately, much as Aspect Oriented Programming does but with a much simpler and cleaner structure.

5.2 Future work

We plan to continue our investigation and development efforts in the area of dynamic software architectures. The work will be focused on two major activities.

First, we will elaborate and develop the methodological beginning outlined in this thesis and construct a comprehensive methodology for development of software systems with dynamic architectures. Based on the assumption that every methodology consists of a notation, a process description, and a set of tools, our goal with this work will be to provide developers with this trio. We present the future work in these three areas in more detail in Section 5.2.1.

Second, we will validate architectural agents and their methodological use in a real-world setting. These studies are important both for verification and validation purposes and we will carry this work out in cooperation with industrial partners. We present the work planned and the validation phase in more depth in Section 5.2.2.

5.2.1 Methodology

Notation

Our future work on notation will focus on the two languages used in our specifications. The notation we used in this thesis is both incomplete and informal.

The Agent Rule Language (ARL) and the Architectural Modification Language (AML) will be further elaborated, extended, and formalized to meet expected requirements from developers.

We will also continue to refine the activity of including agents in architectural specifications. Currently, we plan to include agents as separate entities. This will improve both the reusability and understandability of a specification.

Another important issue to be gone into involves cooperating agents. In applications with more than one agent, agents can divide tasks among themselves, manage more complex tasks, and reduce specification effort. In order to provide this capability, a language which is used by agents for inter-agent communication must either be developed or adapted to fit our needs.

A minor task, although still important, is the development of a graphical tool notation.

Processes

In this thesis we have not addressed process related issues. In our future work we will include these aspects. Currently, we see nothing that indicates a need for a completely new, ground breaking process model just to accommodate dynamic architectures. On the contrary, we believe that minor modifications at the lower levels of process descriptions will be sufficient.

We plan to integrate the process into our tool-set. In this manner we will simplify for developers migrating to our methodology and avoid a steep learning curve.

Tool support

In Section 4.5 we listed important functionality that a tool supporting development of reactive architectures should implement.

The first step in this work will be integration of agent technology in a "standard architecture tool". In comparison to existing tools we take a slightly different approach covering a different area focusing on unplanned reconfiguration.

The designer will use a graphical editor to design the component architecture, picking components and connectors from a toolbox. In the design phase the designer also develops the rule-base for agents using rule editors and attach agents to the architecture.

Our work on development of software in general and development of software with dynamic architectures in particular focus on sufficient support for communicating models within a design group. We believe that tools and techniques can be used to improve communication and simplify the process of understanding a design. In the spirit of this conviction we will put lot of effort into making our specifications executable. Providing facilities for simulations, where designers can study what will happen dynamically, when their dynamic architecture is executing in a real setting.

In order to support migration from design to implementation we will have to develop a support framework, simplifying the implementor's tasks. This framework should include support for developing agents and adapters, and support for introducing agents in existing systems.

5.2.2 Validation

An important part of future work is to validate the architectural agents in an industrial setting. Some questions that will be asked; Are agents really useful? Do they reduce or add complexity? In order to answer these and similar questions we plan to conduct two studies; one general, with a focus on agent capabilities for specifying dynamic architectures in general, and one domain specific focusing on component based development and specific problems in that domain.

Systems with dynamic architectures

When we construct our more comprehensive methodology, we will, in parallel, work on specifying an industrial strength software system with a dynamic architecture. This will provide us with inputs on the rule language to be developed, which modification operations that are needed, and how a process and set of tools best supports the development team.

After this initial construction phase, we plan to move into a controlled validation activity and find a design team willing to use our methodology and specify the dynamic aspects of their system, preferably from a different domain. This phase of the validation is not planned in detail yet, but this will probably be conducted in an informal manner, based on discussions, providing feedback from actual users of our work.

Component based development

When designing systems with pre-built components, the architecture of the system will be the natural focus throughout development.

> "The goal of assembling applications from reusable components is still elusive because business applications require system-wide properties like reliability, availability, maintainability, security, responsiveness, manageability, and scalability (the "ilities"). Assembling components and also achieving system-wide qualities is still an unsolved

> problem. As long as the code that implements ilities has to be tightly interwoven with code that supports business logic, new applications are destined to rapidly become as difficult to maintain as legacy code"[1]

Several authors have shown that there is a connection between the configuration of the components in the architecture and the system wide properties that the system exhibits. The difference between these system wide properties and the functional properties is that while functional properties can most often be located within a single component or a tightly coupled group of components within the architecture, system wide properties are scattered throughout the system in multiple components. At the same time, even if an architectural configuration is created that fulfills many of the quality requirements, such as maintainability, there are still many such requirements that need support in run-time. For instance, fault-tolerance includes checking the state of objects at run time for deviations, resetting objects to safe states etc. In this thesis we have shown how dynamic reconfiguration can be used to implement such requirements.

The topic of using component architectures with binary components raises many interesting questions. How can system wide properties be included in systems where components can not be modified? Filman [66], observes that several of the quality requirements or "ilities" can be achieved by "systematically controlling the inter-component communications". The same observation is made by Szyperski and Vernik in [67], where they state that, system wide properties in component systems will "require dedicated support outside of the participating components". This provides even more reason for taking the approaches suggested in this thesis.

[1] This quote was taken from the Call for papers: Workshop on Compositional Software Architectures, Monterey, California January, 1998.

BIBLIOGRAPHY

[1] M. D. McIlroy. Mass produced software components. In *Software Engineering, Report on a conference sponsored by the NATO Science Committee*, 1969.

[2] B. J. Cox. *Object Oriented Programming — An Evolutionary Approach*. Addison-Wesley, 1986.

[3] O. Nierstrasz and T. D. Meijler. Research Directions in Software Composition. *Computing Surveys*, 27(2):262–264, June 1995.

[4] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch or Why it's hard to build systems out of existing parts. In *Proceedings of the 17th international conference on Software engineering (ICSE'95)*, pages 179–185, Seattle, April 1995. ACM-Press.

[5] H. A. Simon. *The Sciences of the Artificial*. MIT Press, 1981.

[6] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12), December 1972.

[7] F. P. Brooks. *The Mythical Man-Month*, chapter 16. Addison-Wesley, 1995.

[8] IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. The IEEE, 1983.

[9] N. Wirth. *Systematic Programming: An Introduction*. Automatic Computation. Prentice-Hall, 1973.

[10] M. A. Jackson. *Principles of Program Design*. Number 12 in A.P.I.C. Studies in Data Processing. Academic Press, 1975.

[11] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Number 8 in A.P.I.C. Studies in Data Processing. Academic Press, 1972.

[12] C. M. Geschke, J. H. Morris Jr., and E. H. Satterthwaite. Early Experience with MESA. *Communications of the ACM*, 20(8):540–553, August 1977.

[13] F. DeRemer and H. Kron. Programming-in-the-Large versus Programming-in-the-small. *Trasactions on Software Engineering*, pages 321–327, June 1976.

[14] R. Prieto-Diaz and J. M. Neighbors. Module Interconnection Languages. *The Journal of Systems and Software*, 4(6):307–334, 1986.

[15] M. R. Barbacci and J. M. Wing. Durra: A Task-level Description Language Reference Manual (version 2). Technical report, CMU / SEI, September 1989.

[16] T. Rentsch. Object Oriented Programming. *SIGPLAN Notices*, 17(9):51–57, 1982.

[17] G. Booch. Object Oriented Development. *IEEE Transactions on Software Engineering*, SE-12(2):211–221, 1986.

[18] D. E. Perry and A. L. Wolf. Software Architecture. Unpublished, revised in 1991, September 1989.

[19] G. Booch. *Object-Oriented Analysis and Design with Applications*. The Benjamin / Cummings Publishing Company, Inc., 2 edition, 1994.

[20] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[21] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *Software Engineering Notes*, 17(4):40, Oct 1992.

[22] D. Garlan and M. Shaw. An Introduction to Software Architecture. *Advances in Software Engineering and Knowledge Engineering*, 1, 1993. World Scientific Publishing.

[23] A. Abd-Ailah, B. Boehm, B. Clark, and C. Gacek. On the Definition of Software System Architecture. In D. Garlan, editor, *Proceedings of the 1st International Workshop on Architectures for Software Systems*, pages 85–95, Carnegie Mellon University Pittsburgh, PA 15213, April 1995. Technical report: CMU-CS-TR-95-151.

[24] D. Soni, R. L. Nord, and C. Hofmeister. Software Architecture in Industrial Applications. In *International Conference of Software Engineering (ICSE'95)*, pages 196–207, 1995.

[25] P. B. Krutchen. The 4+1 View Model of Architecture. *IEEE Software*, 12(6), November 1995.

[26] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[27] F. Bushmann, R. Meunier, H. Rohnert, P. Sommerlad, and Stal M. *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley & Sons, 1996.

[28] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1997.

[29] N. Medvidovic and R. N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In M. Jazayeri and H. Schauer, editors, *Software Engineering – ESEC/FSE '97*, volume 1301 of *Lecture Notes in Computer Science*, pages 60–76. Springer Verlag, September 1997.

[30] N. Medvidovic. A Classification and Comparison Framework for Software Architecture Description Languages. Technical Report UCI-ICS-97-02, Department of Information and Computer Science, University of California, Irvine, Irvine, California 92697-3425, 1997.

[31] M. Shaw and D. Garlan. *Software Architecture, Perspectives on an emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, 1996.

[32] M. Shaw, R. DeLine, and D. V. Klein. Abstractions for Software Architecture and Tools to Support Them. *Transactions on Software Engineering*, 21(4):314–335, April 1995.

[33] D. Garlan, R. T. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of the Centre for Advanced Studies Conference (CASCON '97)*. IBM / CAS, November 1997.

[34] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and Dubrow D. L. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Software*, June 1996.

[35] M. Moriconi and R. A. Riemenschneider. Introduction to SADL 1.0. Technical report, Computer Science Laboratory, SRI International, March 1997.

[36] R. J. Allen. *A Formal approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1997. CMU-CS-97-144.

[37] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[38] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *5th European Software Engineering Conference (ESEC 95)*, 1995.

[39] J. Magee, N. Dulay, and J. Kramer. Regis: A Constructive Development Environment for Distributed Programs. In *IEE/IOP/BCS Distributed Systems Engineering*, pages 304–312, September 1994.

[40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[41] A. Abd-Allah. Composing Heterogeneous Software Architectures. Technical report, USC Center for Software Engineering, April 1995.

[42] G. Abowd, L. Bass, P. Clements, R. Kazman, L. Northrop, and A. Zaremski. Recommended Best Industrial Practice for Software Architecture Evaluation. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, January 1997.

[43] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. The Architecture Tradeoff Analysis Method. Technical Report CMU / SEI-98-TR-008, CMU / Software Engineering Institute, 1998.

[44] B. W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, May 1988.

[45] R. Kazman, L. Bass, G. Abowd, and M. Webb. SAAM: A Method for Analyzing the Properties Software Architectures. In *16th International Conference on Software Engineering (ICSE 16)*, pages 81–90, May 1994. Sorrento Italy.

[46] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP'97*, Lecture Notes on Computer Science, pages 220–242. Springer Verlag, 1997.

[47] Common Facilities Architecture. Object Management Group, Inc., November 1995.

[48] Sun Microsystems. *JavaBeans*, 1.01 edition, July 1997. Available at: ⟨`URL:http://java/sun/com/beans`⟩.

[49] *Java Remote Method Invocation - Distributed Computing For Java*, March 1998. Available at: ⟨`URL:http://java.sun.com`⟩.

[50] R.E. Johnson and B. Foote. Designing Reusable Classes. *Journal on Object-Oriented Programming*, 1(2), June 1988.

[51] M. Mattsson. Object-Oriented Frameworks - A surevey of methodological issues. Technical Report LU-CS-TR:96-167, Department of Computer Science, Lund University, 1996.

[52] G. E. Krasner and Pope S. T. A Cookbook for Using the Model–View–Controller User Interface Paradigm in Smalltalk–80. *Journal of Object-Oriented Programming*, pages 26–49, August / September 1988.

[53] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3ed edition, 1997.

[54] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Abstractions and Reuse of Object-Oriented Design. In *7th European Conference on Object-Oriented Programming (ECOOP'93)*, 1993.

[55] M. Fowler. *UML Distilled*. Addison-Wesley Longman, Inc., Reading, Massachusetts, 1997.

[56] R. Fabry. How to Design A System in Which Modules can be Changed on the Fly. In *Proceedings of International Conference on Software Engineering*, pages 470–476. IEEE-CS Press, 1976.

[57] M. E. Segal and O Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 10(2):53–65, 1993.

[58] R. Allen, R. Douence, and D. Garlan. Specifying Dynamism in Software Architectures. In G.T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems Workshop*, 1997. Available at ⟨`URL:http://www.cs.iastate.edu/~leavens/FoCBS/`⟩.

[59] R. Allen, R. Douence, and D. Garlan. Specifying and Analyzing Dynamic Software Architectures. In *1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, March 1998.

[60] P. Oreizy. Issues in Runtime Modification of Software Architectures. Technical Report 35, Department of Information and Computer Science, University of California, Irvine, CA 92697, August 1996.

[61] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 12(9):717–734, Sep. 1995.

[62] J. Andersson, M. Comstedt, and T. Ritzau. Run-time Support for Dynamic Java Architectures. In *Proceedings of the ECOOP'98 workshop on Object-Oriented Software Architectures*, 1998. Technical report 13/98 University of Karlskrona/Ronneby.

[63] IBM Intelligent Agent Center of Competence, ⟨URL:http://www.networking.ibm.com/iag/iaghome.html⟩. *IBM Agent Building Environment Developer's Toolkit, OVERVIEW*, level 6 edition, June 1997.

[64] N. Medvidovic. ADLs and Dynamic Architecture Changes. In Laura Vidal, Anthony Finkelstein, George Spanoudakis, and Alexander L. Wolf, editors, *Joint Proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96)*, pages 24–27. ACM, 1996. SIGSOFT'96.

[65] J. M. Bradshaw, editor. *Software Agents*. MIT Press, Cambridge, Massachusetts, 1997. AAAI.

[66] R. E. Filman. Achieving "ilities". Presented at the OMG-DARPA-MCC Workshop on Compositional Software Architectures, January 1998. Avalable at: ⟨URL:http://www.objs.com/workshops/ws9801/papers/⟩.

[67] C. Szyperski and R. Vernik. A Case for Tired Component Frameworks. Presented at the OMG-DARPA-MCC Workshop on Compositional Software Architectures, January 1998. Avalable at: ⟨URL:http://www.objs.com/workshops/ws9801/papers/⟩.