2004

Software architecture: styles and representational schemes

Somo Subro Banerjee
Louisiana State University and Agricultural and Mechanical College, sbaner2@lsu.edu

SOFTWARE ARCHITECTURE: STYLES AND REPRESENTATIONAL SCHEMES

A Thesis
Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
requirements for the degree of
Master of Science in System Sciences

in

The Interdepartmental Program in System Sciences

by
Somo Subro Banerjee
B.E., National Institute of Technology, Rourkela, India, 1999
August 2004

Acknowledgements

This work would have been incomplete without the help and significant contribution of some special people, all of whom I would like to thank sincerely.

First and foremost, I would like to thank Dr. Donald Kraft for his guidance and support. He has always been there to help whenever I needed him and has egged me on whenever I have felt stranded and lost. He has always given valuable tips and made insightful remarks that have led my research in the right direction. He has contributed significantly to the writing, editing and refinement of this document. He has proofed this document and suggested corrections and improvement and he has always been right.

I am extremely grateful to Dr. Sukhamay Kundu for exposing me to this exciting area of software architecture and creating my interest in this research direction. He is the person who has taught me all that I have learnt in the last couple of years. He has guided me in my research efforts, inspired me with new thoughts and ideas, and helped me in developing the academic skills required to carry out research at this level. He has encouraged me in my efforts, has been severely critical when I have been erroneous and has taught me the art of doing things right. From him I have learnt that the best way of doing things is to do them right, and that perfection can be achieved by sticking to a task and putting forth the best possible effort.

I would like to thank Dr. Doris Carver and Dr. Bijaya Karki for being members of my research committee and for giving me their time and helping me with my research.

Last, but not least, I would like to thank my wife for being such a supportive person. She has been with me in the highs and lows that accompany a research effort and never once let me lose my focus and direction. She has listened to my ideas patiently and provided valuable tips. She has helped me in refining this document by pointing out errors and typos, and helped me in searching and procuring online research material pertaining to my research area.

Somo Subro Banerjee

Table Of Contents

ACKNOWLEDGMENTS……ii

LIST OF TABLES……v

LIST OF FIGURES……vi

ABSTRACT……viii

CHAPTER 1. INTRODUCTION……1

CHAPTER 2. DEFINITIONS OF SOFTWARE ARCHITECTURE……6

CHAPTER 3. VIEWS AND STYLES……15
3.1 Stakeholders and Views……15
3.2 Architectural Styles……18

CHAPTER 4. ARCHITECTURAL REPRESENTATION SCHEMES……28
4.1 Pictorial and Textual Representation Schemes……30
4.2 An Improved DFD for Architectural Description: AND-OR DFD……35
4.2.1 Physical vs. Logical Presence……35
4.2.2 How Much to Show……35
4.2.3 Action Grouping……36

CHAPTER 5. ARCHITECTURE DESCRIPTION LANGUAGES AND UML……44
5.1 Architecture Description Languages……45
5.1.1 Components……46
5.1.2 Connectors……47
5.1.3 Configuration……48
5.2 The C2 Architecture Description Language……48
5.3 Unified Modeling Language (UML)……49
5.4 Modeling……52
5.5 UML for Software Architecture……54
5.6 Architecting Using UML: An Example……55
5.6.1 Conceptual View……57
5.6.2 Execution View: Two Representations and a Problem……59

CHAPTER 6. SUMMARY AND CONCLUSIONS……63
6.1 Summary……63
6.2 Opportunities for Future Work……69

REFERENCES……71

VITA……76

List Of Tables

1. Decomposition of Block Builder Software……11

2. Block Builder architecture based on Kazman's definition……12

3. Block Builder architecture based on Wolf's definition…… 13

4. Decomposition Model of Cost Optimizer……..25

5. Code segment for DFD example……33

6. Process Decomposition of C2 Generator……...40

7. Logical Decomposition of C2 Generator……56

List Of Figures

1. The “4+1” view model……....17

2. Pipeline architecture for Block Builder……..19

3. Example of Data Abstraction……20

4. Example of Communicating Processes…… 20

5. Diagrammatic Representation for Implicit Invocation……...21

6. Diagrammatic Representation for Repository (Blackboard)……..21

7. Diagrammatic Representation of Interpreter……..22

8. Diagrammatic Representation of Main Program and Subroutines…….22

9. Block Builder as a Layered Architecture……24

10. Layered Model of Cost Optimizer……..……..26

11. Boxes-and-Lines representation of Block Builder……30

12. Building Blocks of a DFD……....32

13. DFD for code segment in Table 5…….33

14. Two Control Flow Diagrams……34

15. Action Grouping and Representation for AND-OR DFD……37

16. Possible combination of iterative AND-OR groups…… 38

17. AND-OR DFD representation of C2 Generator (process view)……..41

18. FSM representation of C2 Generator……42

19. Mathematical Modeling……52

20. Software Modeling using UML……... 53

21. Component and Connector……...57

22. Conceptual View of C2 Generator……58

23. Sequence Diagram for a process view of C2 Generator……60

24. Another Sequence Diagram for a process view of C2 Generator……61

Abstract

Software architecture is being widely used today to describe a very high level design methodology of large software systems. Software architecture represents the overall structure of a system in an abstract, structured manner. A good architectural representation scheme holds the key to the effectiveness of a software architecture description and usage. In this work we look at architectural styles and architectural representation schemes. We propound the idea that the layered architectural model is a suitable candidate for a generalized architectural style and that it can cater to many different problem domains, other than the message-passing systems it has traditionally been used to model. We propose some rules by which the layered architectural style can be improved and modified in order to be able to model a wider problem domain. Then we evaluate different methods of architectural representations that have been used to model software architecture and analyze their strengths and shortcomings. We propose the use of a modified data flow diagram architecture representation scheme. This scheme is called AND-OR DFD method and is introduced and developed in this thesis. The main concept introduced here is a combination of components to form action groups to support multiple workflows and the relationships among them, without significant increase in the architectural complexity. Finally, we look at UML as a prospect for a generalized architecture description language and discuss its merits and demerits with examples.

Chapter 1: Introduction

Software architecture is being widely used today to describe a very high level design of large software systems. Software architecture expresses the overall structure of a system in an abstract, structured manner. The main goal of a software architectural representation of a system is to identify the major components that constitute this system, and the interactions between these components and represent them without saying too much or too little [19, 38]. It can be said that software architecture is an imprecise area of research in the sense that there is still no consensus on a complete definition of software architecture. Various definitions have been proposed [7, 25, 40, 53], each attempting to encompass the essence of software architecture and only partially succeeding in doing so. We will compare and contrast some popular definitions of software architecture in Chapter 2**.**

It might be interesting and useful to find out where software architecture fits in with the overall software engineering life cycle. Software engineering is, simply stated, the act of designing, building and maintaining large software systems. A formal definition given by Sommerville [48] states that 'Software engineering is concerned with the theories, methods and tools for developing, managing and evolving software products". Most models of the software life cycle include the following six processes: requirements engineering, design, programming, integration, delivery, and maintenance. During the first process, developers and clients meet to discuss ideas for the new software product. Developers use a variety of techniques in order to assess the real needs of the client. Unless the requirements engineering process is done properly, the resulting software will not be useful to the client even though it may run correctly. The requirements engineering process is completed when the specifications for the new software product are written in a formal document called the requirements specification document. The

next step has traditionally been the design of the software system. Software architecture now is the beginning of this stage. We can consider software architecture to be the high-level design stage and it is software design at the highest level of abstraction. It represents the "big picture" of the entire software system with its basic and fundamental components, and then these components can be subjected to traditional software design.

An important aspect of software architecture is representation. The representational clarity and power of a software architectural description is very significant and determines the completeness and hence success of the succeeding steps in the software development lifecycle which build on this description. Various methods of diagrammatic and descriptive representation have been used to describe software architecture of a system, but none without their own limitations and problems. Researchers have used text, boxes-and-lines [7], data flow diagrams and other pictorial and textual methods for this purpose. In Chapter 4, we take a look at some methods of representation of software architecture and their benefits and limitations. We propose an improved scheme of the DFD like representational scheme that takes into account multiple work flows. We discuss the advantages of this scheme, which is called the AND-OR DFD method, and the limitations that still remain. We then look at a more FSM-like representation of architecture, and compare it with the AND-OR DFD representation. However, these representations are informal and provide little information about the actual computation represented by boxes, the interface of the components, or the nature of their interactions [15]. In order to make such representations meaningful, one needs to append a textual description to the representation. This makes the diagram unwieldy and mostly difficult to use. While architectural concepts are often embodied in infrastructure to support specific architectural styles and in the initial conceptualization of a system configuration, the lack of an explicit, independently

characterized architecture or architectural style can significantly limit the benefits of software architectural design [4].

In order to address this problem, researchers in the past few years have come up with architectural description languages (ADLs) [3, 18, 31, 47] as notations for architecture models. An ADL for software applications focuses on the high-level structure of the overall system rather than the details of any specific component of the system [49]. ADLS provide constructs for specifying architectural abstractions in a descriptive notation. They provide mechanisms for decomposing a system into components and connectors, specifying how these elements are combined to form a configuration and defining families of architectures or styles [15]. Hence, they provide abstractions that are adequate for modeling a large system while ensuring sufficient detail for establishing properties of interest [35]. In Chapter 5, we compare and contrast some ADLs and review their strengths and weaknesses.

Though the different ADLs server their purpose satisfactorily, each of them embodies a particular approach to the specification and evolution of an architecture, with specialized modeling and analysis techniques that address specific system aspects in depth [11, 15, 29]. However, this emphasis on depth over breadth of the model can make it difficult to integrate these models with other development artifacts, because the rigor of formalisms tends to ignore the more day-to-day development concerns [35]. Also closely related to the ADL method of architectural representation is the concept of architectural views [7, 21, 25, 30]. According to [25], an architectural description is organized into one or more views. A view address one or more concerns of the system stakeholders, who are individuals, teams, and organizations with interests in, or concerns relative to, the system. Views help to reduce the amount of information to be dealt with, at one time, and thus enhances clarity and representational power. The software

architecture of a system can be therefore represented as a set of views, and a combination of these views comprises of the system as a whole. The major drawback of using the view approach is that the representation becomes view-centric; the side effect is redundancy among different views. These redundancies can cause inter-view mismatches, such as inconsistencies and incompleteness [12]. Thus, a good representational scheme should take view integration into consideration, try to be consistent between different views and minimize view redundancy.

UML has established itself as a leading Object oriented analysis and design methodology. It is a language for specifying, constructing, visualizing and documenting artifacts of software-intensive systems [1]. To do so, it uses a collection of views that capture some aspect of the system being developed. This similarity of approach of UML and software architecture, both representing a system as a collection of simpler views, has led researchers to believe that UML can be used to represent software architecture [1, 6, 12, 13, 14, 21, 23, 24, 27, 37]. In this work, we will try to illustrate how different views of UML can be used for a view-centric representation of software architecture, and address the issues of view integration and redundancy minimization. We will attempt to argue that UML can be extended suitably to develop a generalized ADL, and we discuss the strengths and weaknesses of this idea.

The rest of the thesis is organized as follows: in Chapter 2 we take a look at software architectural definitions. We compare and contrast two popular definitions in order to evaluate their correctness and completeness. In Chapter 3 we talk about architectural views and how they are explained by the definitions discussed in the previous chapter. We also talk about architectural styles and their capabilities with a special emphasis on layering. In Chapter 4 we look at architecture description schemes like DFDs and FSMs. We introduce an improved version of DFD representation of architectural representation. We also compare and contrast this

method with FSM representations. In Chapter 5, we discuss ADLs and look at the properties of an ADL, and expectations from a generalized ADL. We look at how UML can be used to represent software architecture and its different views in an efficient manner, and discuss the pros and cons of using UML as an ADL. Finally, in Chapter 6 we conclude the thesis and put forth some future research directions in this area.

Chapter 2: Definitions of Software Architecture

Software architecture has been widely researched both be academicians and by software architects in industry, and few fundamental concepts have evolved and been accepted by all. Software architecture expresses the overall structure of a system in an abstract, structured manner. Software architecture is the representation of a software system at the highest possible level of abstraction. It is the representation of the earliest design decisions that need to be made in order to build a software system. Software architecture is mainly a collection of components that make up the software system. These components are also called software elements. The components are held together by connectors that define the relationship between different components. The connectors can, in turn, be components themselves. The emphasis is on component purpose and interaction, and not on the details of what subcomponents make up the components or how the components work. Software architecture attempts to provide a framework for the software designers to work on, at the next lower level of abstraction.

There have been many different definitions of software architecture [7, 19, 25, 40, 53] and there is still the quest for a definition that is complete and agreed upon by one and all. But software architects have, by experience and examples, reached a consensus on what software architecture is, and what its roles and responsibilities are. We know, for instance, that software architecture represents a system as a set of components and connectors and defines the relationships between different components. In addition, a software architectural description of a system may have many different external agents, commonly called “stakeholders” [7, 10, 25, 30], and each stakeholder may have different expectations and requirements from the same system. It is imperative for the software architect that he takes into account all these different requirements of one or more stakeholders, while creating the architecture for the system.

There are several available definitions of software architecture [53], but we will talk about two of the most popular ones. One of the more commonly used definitions has been given by Kazman in [7], which states, "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 relationships among them".

From the above definition, we can infer that architecture defines components and interactions between components. It is an abstraction of a system that only presents details about components that affect how they use, are used by, relate to, or interact with other components. A component is an object with independent existence and can be a module, a process, a procedure, a library etc. The definition just implies that there can be more than one kind of component without mentioning any specifics about what they might be. The term "relationships" refers to the connections between these components, taking into account the behavioral aspects and the nature of dependencies. The behavior of each component is part of the architecture insofar as that behavior can be observed or discerned from the point of view of another component. This behavior is what allows components to interact with each other, which is clearly part of the architecture. This does not mean that the exact behavior and performance of every component must be documented in all circumstances; but to the extent that a component's behavior influences how another component must be written to interact with it or influences the acceptability of the system as a whole, this behavior is part of the software architecture. Again we see that the definition suggests that there can be different kinds of relationships, e.g., subdivision, synchronization, or message passing.

Reference to the "externally visible" properties of components reminds us of the level of abstraction at which the concept of architecture holds true. We are at the highest possible level of

abstraction. Noteworthy is the fact that the definition assumes that any software component of an architecture may be "large" enough to have its own architecture and so, there are many "internal" aspects of this component like its components and their interrelationships which we are not aware of, and of which we don't need to know at this level of abstraction. Only those functionalities of a software component that need to be known in order to define the interrelationships and dependencies between this component and other components of the architecture comprise of the externally visible properties of a software component. The internal behavior of a component is not an essential part of an architectural description.

One important point to note here is that the definition implies that every software system has an architecture because every system can be shown to be composed of elements and relations among them. In the most trivial case, a system is itself a single element - an uninteresting and probably non-useful architecture, but still technically an architecture. This definition mentions what software architecture is and what it attempts to do, but is too vague. For example, it doesn't mention whether there are any constraints on the kinds of components or whether the components should have a particular form. It also doesn't state if there is any rationale for choosing a particular component or a group of components. In fact, it mentions that a system can have many different architectural representations. That, in turn, gives rise to the issue of how to determine whether an architectural representation is good or bad. It is obvious that this question needs to be answered at the outset because an unsuitable architecture can lead to faulty system design and erroneous behavior of the system.

Another definition that merits mention here is given by Wolf in [40]. This definition states "Software Architecture = {Elements, Form, Rationale}

A software architecture is a set of design elements that have a particular form." Let us look at what this definition tries to say about software architecture.

- **Elements**: According to [40], there are three classes of software elements, namely processing elements, data elements, and connecting elements. The processing elements are those components that take some data and apply transformations on them, and may generate updated or new data. The data elements are those that contain the information to be used, transformed and manipulated. The connecting elements bind the architectural description together by providing communication links between other components. The connecting elements may themselves be processing or data elements, e.g., procedure calls, shared data, or messages. The connecting elements play a fundamental part in distinguishing one architectural representation from another, and have an important affect on the characteristics of a particular architecture or architectural style.
- **Form**: The architectural form consists of weighted properties and relationships [40]. The definition implies that each component of the architecture would be characterized by some constraints, generally decided by the architect, and some kind of relationship with one or more other components. Properties define the constraints on the software elements to the degree desired by the architect. Of course the architect may set no constraints on a component if he/she so desires. Weighting may refer to one of two things, either the importance of the property or relationship (allows the architect to distinguish between essential components and accessory components), or the need for selecting from a bunch of alternatives, giving preference to some over others (allows the architect to constrain the choice while giving some degree of latitude to the designers who must satisfy and implement

the architecture). Relationships constrain how the different elements will interact and how they are organized with respect to each other in the architecture.

- **Rationale**: The rationale explains the different architectural decisions and choices; for example, why a particular architectural style or element or form was chosen. Rationale is tied to requirements, architectural views and stakeholders. Probably all choices are governed by what the requirement is. There are many different external components that have an interest in the system, and expect different things from the same system. We therefore have to consider the different external demands and expectations that affect and influence the architecture and its evolution. A system has one or more stakeholders, or external components that have an "interest" in the system. A rationale demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders' need statements.

This definition, as we see, defines the basic kinds of software components, and what their roles and responsibilities are. Moreover, it explains the need for architect created constraints, and also the need to choose from a set of alternatives based on requirements. It also introduces the concept of stakeholders and the affect of external interests on the overall architecture of a system. This definition therefore goes further and makes software architecture more precise. But there are still some issues that remain unclear. We still cannot ascertain whether the architecture we made for a system is good or bad. Similarly, we have no means of saying that a chosen component is a good component.

In order to understand correctly what these definitions explain, we will take an example of a simple software system called Block Builder that takes as input a C program and identifies blocks in the same. The basic blocks for a single program (with no functions) are main, the loops

(e.g., for, while) and the conditionals (e.g., if, then, switch). The software identifies and marks the beginning and end of each block found in the input file. Then it builds a tree out of this code, with each node representing a block. The tree structure of blocks is obvious. It can display the code with all blocks identified. This software is simple but, by definition, has an architecture. Though software architecture is typically used for large, complex systems, this simple example will suffice as an example. Table 1 below shows a simple decomposition model [9] for this software.

The decomposition method is useful for presenting the functionality of a system as a set of modules. This method is used to show how system responsibilities are partitioned across modules and how those modules are decomposed into submodules. It however is a crude method for architectural decomposition because it does not show the connectivity between the modules.

Table 1: Decomposition of Block Builder software

Module Name	Submodule(s)
1) ReadInput	CreateSentenceForEvaluation
2) EvaluateSentence	IdentifyBlockStartSentence
	IdentifyBlockEndSentence
3) BuildBlockTree	BuildTokenTreeFromSentences
4) DisplayOutput	DisplayTokenTree

We see from table 1 that there are four main modules for the Block Identifier, and each of them has one or more submodules. Let us now describe this architectural representation based on each of the two definitions that were stated earlier.

Table 2 represents the architecture of Block Identifier based on Kazman's definition. We can see that there are four components. The Sentence Builder reads one line from the input file, and

cleans it up, removing non-relevant features like leading spaces and tabs, comments, double quoted strings, single quoted characters, special characters like '/;', '/"','/{' from the line. It also gets rid of empty lines that the programmer may have left for clarity and formatting. We can see that the externally visible property of this component is that it "converts one input line into sentence for evaluation" by the Sentence Evaluator, which is the second component. So this component is connected to the Sentence Evaluator module and it sends the sentence it created to that module. We see that Kazman's definition doesn't say anything about the nature of connectivity between the connected modules. The next component, the Sentence Evaluator is connected to two modules. It receives a sentence from the Sentence Builder module. It then checks this sentence to see if it has a valid token. A sentence has a valid token it indicates the start of a block, e.g., the sentence has an 'if' followed by the condition in a pair of parentheses followed by the '{', which indicates the start of the block. Needless to say, all sentences starting with an 'if' are not valid token sentences since some of them may be just one line long. Once this module has identified a statement which has a valid token, this sentence is sent to the Block Tree Generator module for further processing. Again, we cannot determine the nature of connectivity between this module and the other module.

Table 2: Block Builder architecture based on Kazman's definition

Components	"Externally Visible" Properties	Relationship (Element/Connection)
1) Sentence Builder	Converts one input line into sentence for evaluation.	Token Evaluator/Send a Sentence
2) Sentence Evaluator	Checks for valid token in the sentence.	Sentence Builder/Get a Sentence
		Block Tree Generator/Send a Sentence
3) Block Tree Generator	Builds the tree of valid blocks of code.	Sentence Evaluator/Get a Sentence
4) Block Code Displayer	Displays the Block Tree	Block Tree Generator/Get the Block Tree

The third module, the Block Tree Generator module, takes a sentence and builds a BlockTree, where each node represents a block of code, from start to end. So the Block Tree Generator is connected to the Sentence Evaluator module, and gets a sentence from it. The fourth module, the Block Code Displayer, is connected to the Block Tree Generator module, and gets the Block Tree from this module. It then displays the Block Tree, which is the output of this software.

Now let us look at the same architecture from the standpoint of Wolf's definition of software architecture. Table 3 displays the Block Identifier architecture based on this definition.

Table 3: Block Builder architecture based on Wolf's definition

Element	Type	Form (Properties/Constraints)	Rationale
1) Sentence Builder	Processing	Removes non-relevant parts of the input line and forms a sentence for evaluation. Needs an input file in order to proceed.	Satisfies the need for a sentence to be easily evaluated.
2) Sentence Evaluator	Processing	Checks for valid token in a sentence Needs sentences from Sentence Builder Module.	Satisfies the requirement that the start and end of blocks need to be identified.
3) Block Tree Generator	Processing	Builds the tree from valid tokens identified. Needs evaluated sentences to indicate start and end of blocks.	Satisfies the need that the tree structure of blocks inside the program needs to be identified
4) Block Code Displayer	Processing	Displays the Block Tree. Needs Block Tree.	Satisfies the need that the code with blocks needs to be displayed.
5) Input File	Data	Must be a 'C' program in a text file. Must be syntactically correct.	Provides the required input.
6) Sentence	Data	Must be simplified as required	Provides input to next module.
7) Evaluated Sentence	Data	Must indicate whether it is a Block Starter or a Block Ender or neither.	Provides input to next module.
8) Block Tree	Data	A new node is added to the leftmost available position in the current level of Block Tree.	Provides the output.

We can clearly see that this definition adds to Kazman's definition, because it distinguishes between different kinds of elements. So we have four processing elements, which are similar to the four components identified by Kazman's definition. But we also have four data elements, which satisfy the need for connectivity between modules. There are no specific connecting elements in this case, as we can see. The other difference that we see in this description is that for each element, we also indicate their form, e.g., properties of that module and constraints, if any. For example, for the Block Tree Generator module, the constraint is that it needs the evaluated sentences from the Sentence Evaluator module which should indicate the nature of the sentence, whether it indicates the start of a new block, or the end of the current block, or neither. It is also implied, though not explicitly stated, that we need a tree structure because nested blocks of code bear a parent-child relationship. So we know the reason why the architect decided to have a tree structure and not something else. The property of this module is that it builds a tree structure for the input program, and each node indicates the start of a block, and contains the entire body of that block.

Though Wolf's definition gives quite a reasonable architectural description for a software system, it still lacks in completeness. From the above description, it is quite impossible to say that for the Block Builder system the chosen architecture is the best one, or that something would go wrong if we decided to build the tree differently, for example.

This lack of completeness of these and many other well-intended definitions has led researchers to conclude that it is too difficult to precisely define software architecture. Nevertheless, researchers and software architects seem to understand what software architecture is and what it attempts to do.

Chapter 3: Views and Styles

Software architecture is about components, connectors and connections. Architecture-based software development involves assembling of a system from its smaller subparts or components. An important goal is to correctly identify the essential subparts of the system under consideration. Once the subparts are correctly identified, the next aim is to identify how different subparts gel with each other, how they communicate, what are the dependability and what are the connections. When the subparts and the dependencies are put together, we should have successfully assembled the required system with all its desired features.

3.1 Stakeholders and Views

When a software system needs to be built, there are many external components (people, other software systems, other hardware systems, etc) that have an interest in this system. In other words, the output(s) of a software system may affect one or more external components. For example, the output of the Block Builder software, the block tree, can be used by a larger software system which aims at making editing and debugging C programs easier. It can also serve as input for an automatic debugger. For the developer of the Block Builder system, the stakes might be purely monetary, as it might be able to be sold. So we see that different external components have their own interest in a system being developed, and each might view the system from its own perspective, and might be purely interest in its own stakes. These eternal components that may be affected by the performance of the system are called Stakeholders.

Different stakeholders have different interests in the system, and so their perspectives of viewing the same system are different. The stakeholders' requirements may overlap or may be different from each other. The architect has to balance all these different perspectives and then come up with an architecture that satisfies all stakeholders to the maximum possible extent. The

point to be noted here is that stakeholders have a significant influence on the architecture and their viewpoints are to be considered important while coming up with architecture for a system. This "stakeholder" influence on architecture makes it clear that there can be different perspectives of viewing the software architecture of one system. These perspectives have been traditionally called "architectural views".

A simple example is the design of a house. We can have a blueprint that shows the overall structure, with the position of rooms, doors and windows, ventilators, and so on. Another blueprint may concentrate on the wiring of the house, the position of the wires, the number of outlets in each room, and their purpose, safety devices like fuse, and so on. Yet another blueprint may be used to represent the plumbing features of the house, the position of taps, sinks and basins, the position of pipes, and the water supply, and so on. These blueprints serve the purpose of the mason, the electrician, and the plumber respectively, but are still the blueprints of the same house. There can also be one master blueprint that represents all the above features together.

The same holds with the architecture of a software system. Viewpoints of different stakeholders may lead to different views of the same system, and these views when integrated will form the complete architectural description of the software system being designed. So we see that architectural representation has two objectives; to be able to come up with different views based on the requirements; and, integration of these different views to form the complete architectural representation.

Different authors have proposed a different set of architectural views as the most suitable ones. The most popular among them is the "4+1" view model proposed by Phillipe Kruchten [30]. The four main views according to this model are:

- **Logical View (Module View)**: The logical view is concerned about the output(s) of the system and how it will affect the end users. For this purpose the logical view attempts to defragment the system into a set of abstractions, or modules. This decomposition serves two purposes; it enables functional analysis, and it also helps in identification of common mechanisms and design elements that are common across the system.
- **Process View (Component-and-Connector View):** This view deals with concurrency and distribution, system integrity, and fault tolerance [9]. This view tries to explain which components interact, and how they do so. It tries to explain the connections between different components from a runtime perspective.
- **Development View (Allocation View)**: This view describes the static organization of the software in its development environment. It deals with modules, work allocation, costs and planning. It also involves monitoring of project progress, software reuse, and security.
- **Physical View (Deployment View):** This view describes the mapping(s) of the software onto the hardware and reflects its distributed aspects. It looks into system requirements like reliability and performance. It maps the various elements identified in the above three views.

These views, combined with a 5th view, which is "**use cases or scenarios**", represents the entire architecture. Figure 1 represents Kruchten's "4+1" view model.

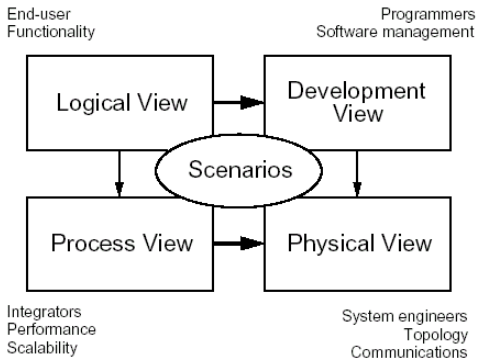


Figure 1: The "4+1" view model [30]

The architecture of a system is therefore a combination of three things: the components, their interrelationships, and the functional and behavioral constraints applied by the stakeholders' concerns. These, in turn, give rise to views. The different views blend together to form the architectural descriptions. Over the years, the architects and researchers have identified certain architectural styles and patterns. Each view can be developed using a particular style, depending on the suitability of the same for the representation of that view.

3.2 Architectural Styles

Software architects use a number of commonly recognized "styles" to develop the architecture of a system. Architectural style implies a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints that are implemented. Components, including encapsulated subsystems, may be distinguished by the nature of their computation (e.g., whether they retain state from one invocation to another, and if so, whether that state is available to other components). Component types may also be distinguished by their packaging—the ways they interact with other components. Packaging is usually implicit, which tends to hide important properties of the components. To clarify the abstractions we isolate the definitions of these interaction protocols in connectors (e.g., processes interact via message-passing protocols; unix filters interact via data flow through pipes). The connectors play a fundamental role in distinguishing one architectural style from another and have an important effect on the characteristics of a particular style [40].

The style of a specific system is usually established by appeal to common knowledge or intuition. Architectures have traditionally been expressed in box-and-line diagrams and informal prose, so the styles provide drawing conventions, vocabulary, and informal constraints (e.g.,

limiting topology or numbers of components of some type). An analysis of common architectural styles [43] suggests that they are discriminated by the following categories of features:

- Which kinds of components and connectors are used?
- How is control shared, allocated, and transferred among the components?
- How is data communicated through the system?
- How do data and control interact?
- What type of reasoning is compatible with the style?

Each of these styles is appropriate for some classes of problems, but none is suitable for all problems. To choose an appropriate style, two kinds of information are required: (1) careful discrimination among the candidate architectures and (2) design guidance on how to make appropriate choices. Architectural styles have also been referred to as architectural patterns [46]. Eight such patterns have been identified. We shall discuss them now, highlighting their merits and demerits and their specific areas of use.

- **Pipeline**: This style is suitable for applications that require a defined series of independent computations to be performed on ordered data. This pattern attempts to decompose the problem into a set of computations, or filters, with operations, called pipes to stream the data from one process to another. The filters interact only via pipes. The Block Builder architecture can be modeled as a pipeline, as seen in Figure 2. Each of the four components can be represented by filters, with the pipe between any two successive filters taking the intermediate output of the first filter and feeding it into the next filter.

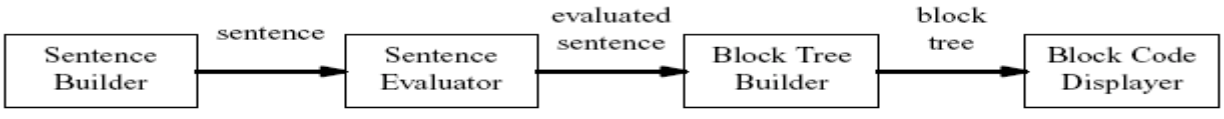


Figure 2: Pipeline architecture for Block Builder

- **Data Abstraction**: This style is suitable for applications in which a central issue is identifying and protecting related bodies of information, especially representation information. When the solution is decomposed to match the natural structure of the data in the problem domain, the components of the solution can encapsulate the data, the essential operations on the data, and the integrity constraints, or *invariants*, of the data and operations.

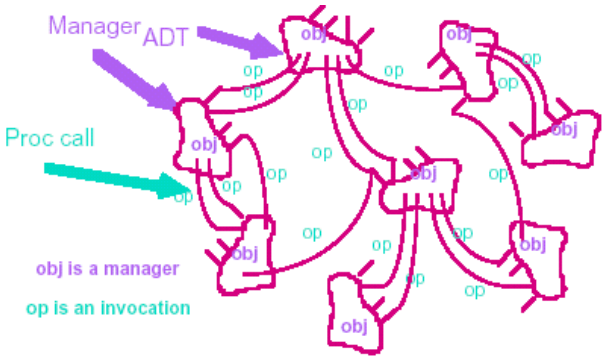


Figure 3: Example of Data Abstraction [46]

- **Communicating Processes**: This process is applicable for applications that involve a collection of distinct, largely independent computations whose execution should proceed independently. The computations involve coordination of data or control at discrete points in time. As a result, correctness of the system requires attention to the routing and synchronization of the messages. We can see clearly that the Block Builder system does not have an architecture of this style, because every next module depends on the output of the previous module.

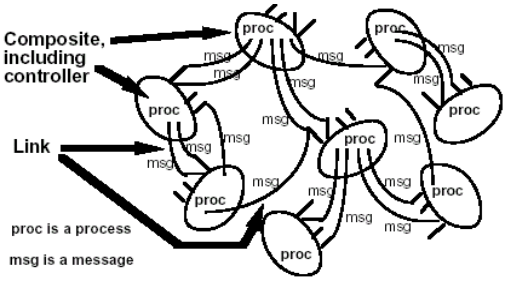


Figure 4: Example of Communicating Processes [46]

- **Implicit Invocation**: This style suits applications that involve coupled collection of components, each of which carries out some operation and may in the process enable other operations. These are often *reactive* systems.

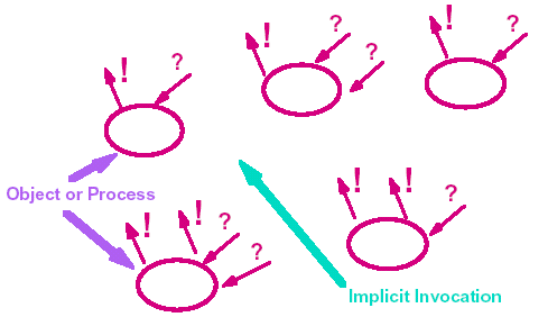


Figure 5: Diagrammatic Representation for Implicit Invocation [46]

- **Repository**: This style is suitable for applications in which the central issue is establishing, augmenting, and maintaining a complex central body of information [46]. Typically the information must be manipulated in a wide variety of ways. An example of a repository or blackboard architecture may the management of work allocation process in a company.

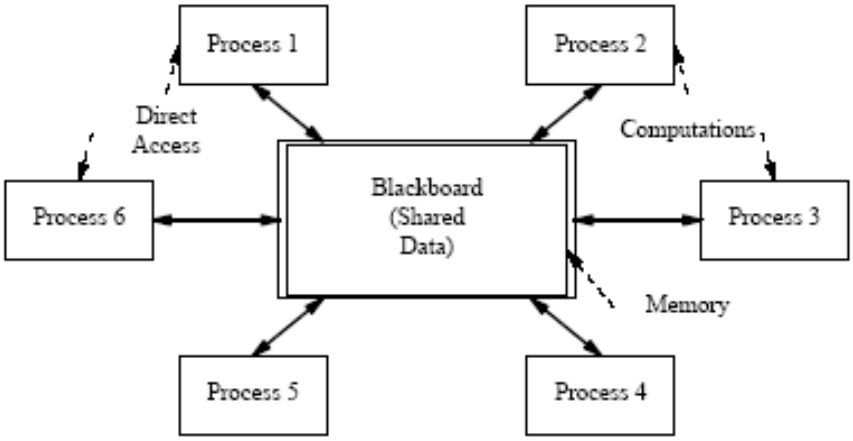


Figure 6: Diagrammatic Representation for Repository (Blackboard)

Another example of a repository is a Business-to-consumer (B2C) website where a person who has a job to be done posts his requirement on the website, and people interested in doing

that job can post their requests, and then the person who posted a job can allocate the job to one of these contenders depending on their qualifications and abilities.

- **Interpreter**: The interpreter architectural style is suitable for applications in which the most appropriate language or machine for executing the solution is not directly available. The pattern is also suitable for applications in which the core problem is defining a notation for expressing solutions, for example as scripts. The best example of this type of architectural style is the interpreter for any programming language.

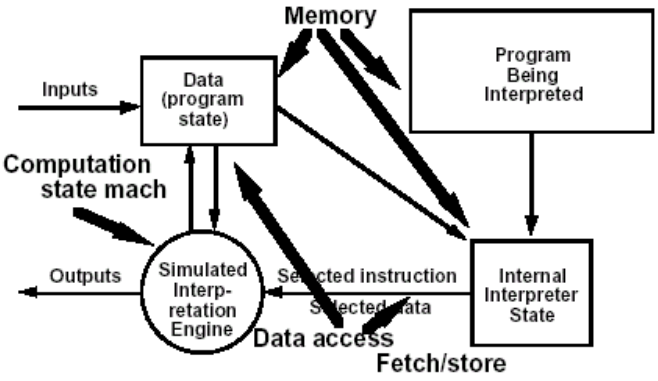


Figure 7: Diagrammatic Representation of Interpreter [46]

- **Main Program and Subroutines**: This style suits applications in which the computation can appropriately be defined via a hierarchy of procedure definitions. It is usually used with a single thread of control. The Block Builder can be definitely represented in this architectural style because the software is written in C, a procedural language.

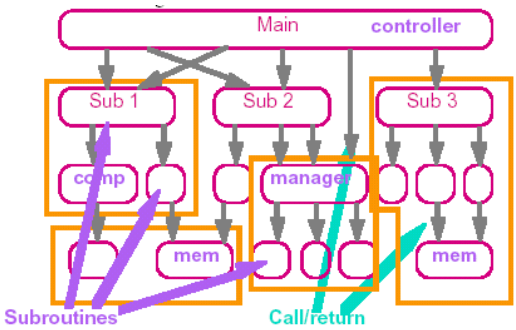


Figure 8: Diagrammatic Representation of Main Program and Subroutines [46]

- **Layered**: This architectural style is suitable for applications that involve distinct classes of services that can be arranged hierarchically. Often there are layers for basic system-level services, for utilities appropriate to many applications, and for specific tasks of the application. To date, the layered architectural pattern has been used mostly to model message-passing situations. But this architectural style has the capabilities to model other kinds of problems, for which it is not generally used. Almost all architectural representations of the pipeline style and the main program and subroutines style can be modeled by a layered style. So obviously the Block Builder system can be modeled as a layered architecture.

In our opinion, at the highest level of abstraction, most traditional software systems can be represented by a layered structure. Of course a C program with just the main() and no functions is a single-layered architecture, trivial but noteworthy. If there are functions that are called from main(), they form the next layer, the functions called by a function in a layer forms the next layer, and so on. The system model of a layered architecture will typically comprise a hierarchy of opaque layers. The components are generally a collection of procedures, and the connectors depend on structure of components and can be procedure calls, client-server, and such. The control structure is mostly single threaded, like the pipeline or the main program and subroutines style.

The strict layer structure as explained above is not too useful. But, by introducing some flexibility, the layered structure can be used to model many software systems. For example, we might allow the layers to be transparent, i.e., interfaces from the lower layers show through as opaque so that only the interface defined by a layer can be used by the next layer up. We can also define the method of interaction between processes in the same layer. In a strict layering model,

the components in the same layer cannot communicate, while in a lenient layering model such communication is allowed.

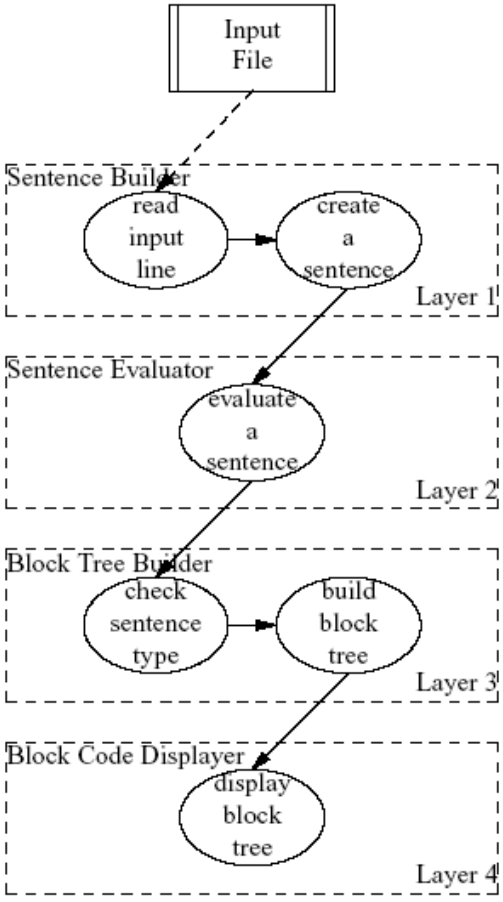


Figure 9: Block Builder as a Layered Architecture

We can also apply constraints on the direction of inter-layer communication direction. In a strict scheme, the inter-layer communication is unidirectional, while in a lenient scheme such communication can be bi-directional. Figure 9 shows the Block Builder system in the layered architecture style. The input to the software is a text file containing the C program for which we need to identify blocks. The architecture has four layers, one for each of the components identified in the decomposition model. We can see that the direction of communication between layers is unidirectional, but communication between the subcomponents in a layer is allowed, and this is again unidirectional.

Table 4: Decomposition model of Cost Optimizer

Module Name	Submodule(s)
1) LoadInput	LoadGraph
	LoadVirtualLoadTrafficMatrix
	LoadCostMatrix
2) ComputeInitialCost	AllocatePaths
	ComputePhysicalLoads
	ComputeCosts
3) OptimizeCost	ComputeShortestPaths
	OptmizeCostByReducingPathLengthAndCost
4) DisplayOutput	PrintPathsAndFlows
	PrintLinkTraffic

As another example of the use of layered architectural style for modeling, we shall look at a software system called the Cost Optimizer that computes the least expensive path for a message in a network. It finds the least expensive array of paths for a bunch of messages traveling from one node to another. The decomposition model for this software is shown in table 4.

We see from the decomposition model that there are four components, and each of them have subcomponents. The inputs to this software are the physical topology of the network, the load traffic, and the cost matrix. The software takes the inputs, allocates initial paths for the messages, and computes the loads and costs. It then optimizes the costs by computing shortest paths and by selecting links with lesser cost. Once it obtains the optimum path for the traffic, the display component displays the paths selected, and the link matrix for the network. This software can also be modeled in the layered style. Figure 10 shows a layered model of the Cost Optimizer software system.

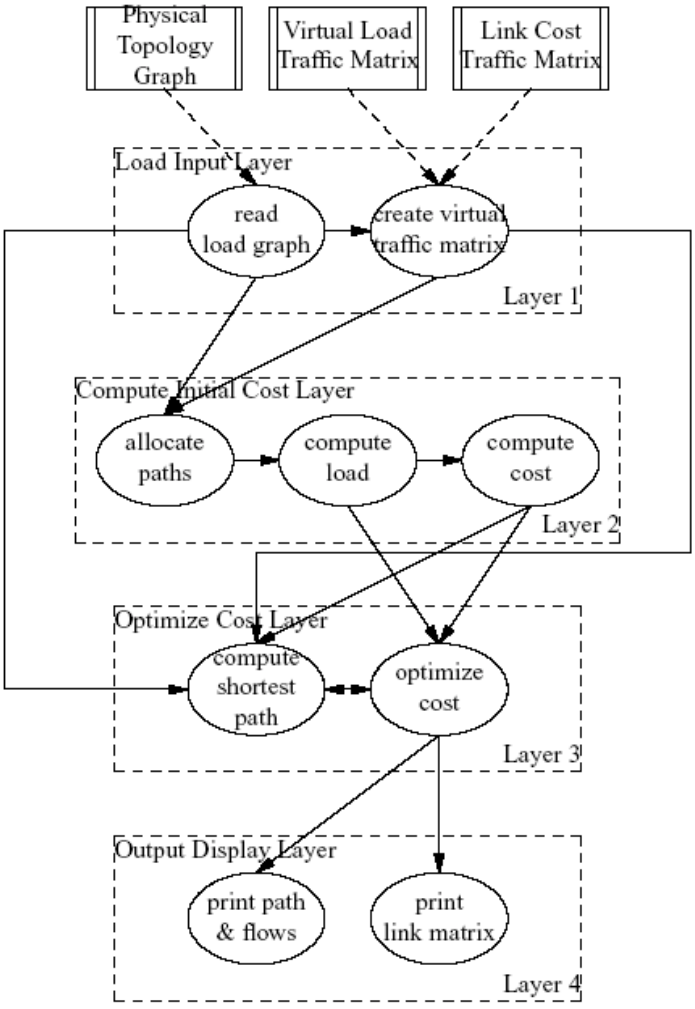


Figure 10: Layered Model of Cost Optimizer

We see from the decomposition model that there are four components, and each of them have subcomponents. The inputs to this software are the physical topology of the network, the load traffic, and the cost matrix. The software takes the inputs, allocates initial paths for the messages, and computes the loads and costs. It then optimizes the costs by computing shortest paths and by selecting links with lesser cost. Once it obtains the optimum path for the traffic, the display component displays the paths selected, and the link matrix for the network. This software can also be modeled in the layered style. Figure 10 shows a layered model of the Cost Optimizer software system.

This is an example of a very lenient layering. Communication is allowed between the subcomponents of a layer, and such communication can be bi-directional, as seen in the optimize cost layer. Moreover, layers can communicate to any layer that comes after it, and is not restricted to the immediately next layer, as seen by communication between the load input layer and the optimize cost layer. The flow of control is basically unidirectional, from the top to the bottom of the diagram.

It is easy to see that the layered style can be used to model many different kinds of software systems. In general, non-parallelism can easily be represented by a layered model. Even if there is parallelism of components, we can put the parallel processes in one layer, and the ones they communicate to in the next. In the case of a architecture that is modeled in the communicating processes style, we can create lenient Layering to allow bi-directional communication between any two layers. So by creating suitable constraints or relaxations on the layering style, we can use this style to model the logical view of many different kinds of software systems.

Chapter 4: Architectural Representation Schemes

Coming up with the architecture of a system is one matter, representing it by visual expression while maintaining its readability, clarity and representation power is another. Researchers and architects have found it very difficult to efficiently represent a software architectural description of a system on paper by means of diagrams and text.

The architecture of a software represents its "big" picture, like the architecture of other man-made systems (be that a machine, a bridge, a building, or a business policy). In that sense, a software architecture serves as a guide for the next lower level design in the development of the software and also as a high level structural description of the system. In particular, the architecture of a software system defines the major components (data and processing elements), and shows how it supports various work-flows or use-cases via the interactions among those components. This can be clearly seen for the case of a database application, where the schemas for the relations in the database can be thought of as the components of the architecture of the database application, providing basic data for use elsewhere in the application; the schemas immediately lays down the major steps in answering queries in terms how the results obtains from individual relations will be combined via projections, selections, and join-operations.

One can talk of designing an architecture just as one can talk of designing a data-format for input/output, designing algorithms for various computational tasks, data-structures to support efficient implementation of the algorithms, input parameters of functions, and so on. Any man-made system involves design at many different levels, depending on the complexity of the system. Likewise, designing the architecture of a software system takes on a more important role when the complexity of the system increases. Some decisions at the architectural level are aimed at providing greater flexibility at the lower level designs by isolating them from other

components, while other architectural decisions may constrain the choices for the lower level design decisions.

One of the simplest abstractions of a program is its flowchart, which groups operations that are performed together (all or none). The abstract data types provide another form of abstraction of data and operations on them to simplify descriptions of algorithms and programs. The dataflow diagrams go one step further in the abstraction level, staying above the level of function-parameters-and-calls; in particular, it chooses the abstraction level sufficiently high to avoid branching conditions.

One of the tasks in defining the architecture is to separate certain decision at a lower level that can be called "design" for the upper level architectural components. Just as one can create different database applications by filling the schemas with different actual relation names and attribute names, one can create different software systems by filling up the different architecture components with different functionalities. In the database context, filling up relations with different tuples for a given choice of attributes is analogous to filling up a component of the architecture with a group of different functions which provides the same interface to the other components. This does not change the work-flow for high level use-case as does the different way of populating the relations by tuples does not change the query processing algorithms.

We have many different pictorial and textual representational schemes and architecture description languages around. Each of them is effective in a particular problem domain e.g., C2 [37] is suitable for modeling GUI intensive systems. But, most of these schemes are not suitable for variegated problems, i.e., a generalized representation scheme to model any software system does not exist. Today researchers are looking at UML with some modifications as the answer to this problem of architectural representation.

4.1 Pictorial and Textual Representation Schemes

The earliest means of representing an architectural description was textual in nature. We have looked at the decomposition method of representing a software system, and we have seen its limitations. While such a representation is suitable for identifying the components of an architecture, it doesn't go far beyond that. It cannot tell us enough about the type of component, or its features. It tells us nothing about the connectivity of these components, the nature of the connection, or the direction. A good architectural representation scheme should be able to tell us enough about the components, their types and externally visible properties, the connections between components, and the nature of those connections, without becoming too complicated. So the focus is on simplicity of representation by the use of pictorial data, and textual data to a reduced extent. The picture drawn should be able to show the directional control flow of the system.

Traditionally, software architecture has been mostly represented by a simplistic "boxes-and-lines" scheme [7]. This scheme uses boxes for the components and lines for the connectors between components. It is good at representing the components and connectors but doesn't go too far beyond that. It cannot talk about the type and properties of the components and connectors. Moreover, it is at most a semi-formal method of architectural description. It is effective as a preliminary tool but falters when it needs to go beyond that. To make it a useful method, one has to accompany it with detailed textual description of the components and the connectors and that makes it cumbersome to use. Figure 11 shows the boxes and lines representation of the Block Builder system.

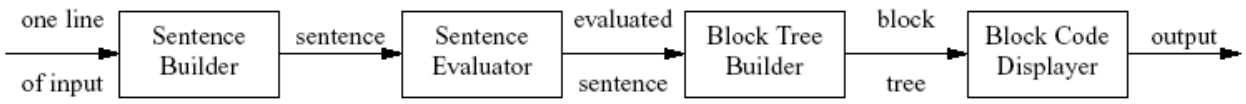


Figure 11: Boxes-and-Lines representation of Block Builder

Using this method, we see what we already know about the Block Builder system from the decomposition model. We know that there are four main components. The improvements in this scheme over the former one are that we can see the connection between components and can say which component connects to whom and what kind of data or control is passed between them. We can also see the direction of communication between two components and so the control/data flow of the software becomes clear. However, we still can't say what the type of a component is or its externally visible properties or what the properties or types of the connectors are.

As an alternative we attempt to use representations analogous to a data flow diagram (DFD) to depict software architecture. A DFD is a process-oriented graphical representation of an application system. In the words of Hoffer, George and Valacich, a DFD "is a picture of the movement of data between external entities and the processes and data stores within a system." The components of a typical dataflow diagram are: the process, the flow, the data store, and the terminator.

The process shows a part of the system that transforms inputs into outputs. The process is represented graphically as a rectangle or a circle. The processes should be numbered in order to conveniently reference them in the DFD. A process is named or described with a single word, phrase, or simple sentence. The process name should describe what the process does. A good name will generally consist of a verb-object phrase such as "Build Token Tree." The flow is represented graphically by an arrow into or out of a process. The flow is used to describe the movement of chunks, or packets of information from one part of the system to another part. The flows represent data in motion. The flows show direction: an arrowhead at either end of the flow or possibly at both ends indicates whether data is moving into or out of a process or both.

The data store is used to model a collection of data packets at rest. The notation for a data store is a box with two parallel lines for the top and bottom. The name chosen to identify the data store is the plural of the name of the packets that are carried by flows into and out of the data store. Data stores are typically implemented as files or databases in a computerized system; but a data store can also be data stored on punched cards, microfilm, or a variety of other electronic forms. A data store might also consist of 3-by-5 inch cards in a card box, or names and addresses in an address book, or several files folders in a file cabinet, or a variety of other non-computerized forms. Data stores are connect by flows to processes. Figure 12 shows a process, a flow and a data store diagrammatically.

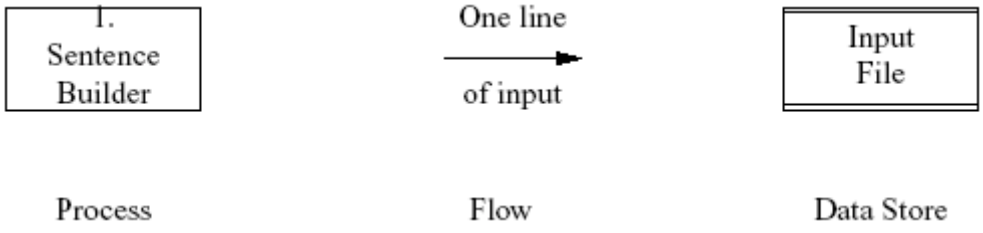


Figure 12: Building Blocks of a DFD

The flows show direction. An arrowhead at either end of the flow or possibly at both ends indicates whether data is moving into or out of a process or both. Similarly data stores have two types of flows, namely a flow from a store and a flow to a store.

Traditionally DFDs have been used for representing data flow and process interaction at a code level. At the architectural level of abstraction though, the scenario is slightly different. Here we need to represent the directional control flow of the system, rather than just the data flow. This is because data flow without control flow associated with it is not informative enough. Only data flow cannot represent the direction of functionality of the system or the start and end of the process. Moreover, data flow is input-centric and not process-centric, i.e., it depends on the input to the system, and can be different for different input sets. So data flow is specific and not

generalized. Let us take a simple example to understand the above problem of input dependency of data flow. Let us consider the simple code snippet shown below written in a procedural language like C.

Table 5: Code segment for DFD example

```
FuncA (int a)
{
        if (a<=3) FuncB(a);
        else FuncC(a);
}

FuncB(int a)
{
        a=a*3.3;
        FuncE(a);
}

FuncC(int a)
{
        a=a/3.3;
        FuncE(a);
}

FuncE(int a)
{
        if ((a>5) FuncF(a);
        else FuncG(a);
}
```

The code segment has four simplistic functions, and each function calls one or more of the other functions. Figure 13 shows the data flow diagram for this code snippet. We can see that the function calls are represented by processes and the flows show the movement of data. Let us now consider the dependency of a DFD on the input data. If the value of 'a' in process 1 is 3, then from FuncA, FuncB is called, i.e., process 2. If, on the other hand, the input value of 'a' is 5, then process 3 gets invoked after process 1. So the actual data flow that occurs at run time depends on the input data. So one Data Flow can actually have multiple control flows at runtime depending on the input for that particular instance. So for the above example, depending on the input, we can have different control flows like 1-2-4-5, or 1-2-4-6, or 1-3-4-5 etc., where the numbers denote the process number (see Figure 13).

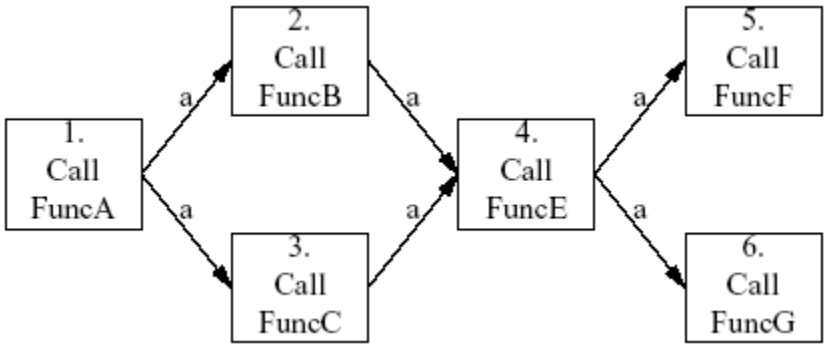


Figure 13: DFD for code segment in Table 5

We can clearly see that depending on the value of ‘a’, process 2 or 3 will be activated. So for any one workflow, for a given input value of ‘a’, either process 2 or process 3 will be functional. This is a fact and is obvious, but there is no way of knowing this from the DFD.

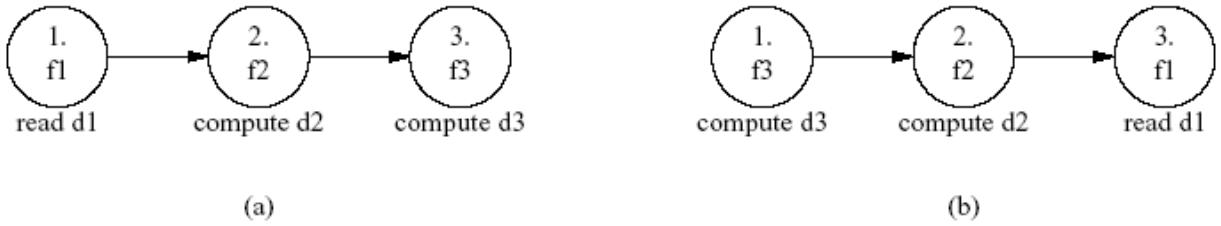


Figure 14: Two Control Flow Diagrams

As another example of multiple possible control flows for a single data flow, let us consider the two control flow diagrams shown in Figure 14. Here we see two control flow graphs. In figure 14 (a), function f1 reads the value of d1, then calls function f2 to compute d2, and then f2 calls function f3 to compute d3. in figure 14(b), function f3 calls function f2. Function f2 in turn calls function f1, and f1 reads the value of d1 and then returns. Now d2 is computed in f2, after which f2 returns. Then d3 is computed in f3. These are two different control flows. But, for both control flows, the data flow (or data dependency) diagram is the same because the value of d3 depends on the value of d2, and the value of d2 depends on the value of d1 in both cases.

The above two examples clearly prove that one data flow can have multiple control flows and the exact data flow path at runtime depends on the input data and can be different for different outputs. So a DFD cannot take care of situations analogous to a logical ‘or’ where only one of n processes may need to be called depending on the calling condition. It also fails to show a situation where a process can be called by only one of n processes depending on the output of these n processes. Moreover, a DFD representation caters to single workflow scenario. At the architectural level, one has to deal with what might happen in different workflows. It is not sufficient to say that any one of these processes may occur. Also, it is good if the representation

can tell which processes must occur together in one workflow, which processes are recursive, and which processes are optional.

4.2 An Improved DFD for Architectural Description: AND-OR DFD

So for the above-mentioned reasons, DFDs are not a good choice for architectural representation. In this work, we suggest some improvements to the traditional DFDs in order to make it more suitable for software architectural representations. The architecture of a software represents its "big" picture, but with a characteristic difference from the dataflow diagrams in that an architecture typically supports multiple high level workflows or use-cases whereas a dataflow diagram can be thought of more like a single high-level use-case. This necessitates different rules for grouping of data and actions at a high conceptual level, although the central role of data in either case remains the same, namely, the interface among the actions. The main concept introduced here is a logical ‘or’ combination of components to support multiple work-flows and the relationships among them, without significant increase in the architectural complexity.

4.2.1 Physical vs. Logical Presence

For each action-component in a architecture, there is a piece of code in the actual software which implements that action. This piece of code corresponds to the physical presence or the conceptual action. On the other hand, multiple uses of that action in different workflows constitute different logical presence of that action. In terms of the software, the logical presence constitutes a function-call for the top level function associated with that action.

4.2.2 How Much to Show

The fact that the architecture represents the big picture means one must remove as much detail as possible and keep only the most significant parts. However, identifying these most

significant parts is non-trivial and requires a considerable amount of skill. Not to include an action only means that it is absorbed into some thing else that has been shown, and not that it is eliminated from the software itself. A similar remark holds for the data. The following is a simple rule for deciding if two actions $A1$ and $A2$ can be merged: If the inputs to $A1$ and $A2$ are not the same, excluding those from $A1$ to $A2$, or vice-versa, and likewise for their outputs, then merge $A1$ and $A2$. Here, the notion of "same" for inputs/outputs is determined in terms of only the major data items.

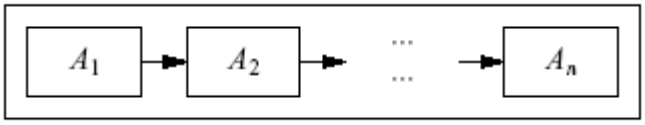
4.2.3 Action Grouping

We call the modified scheme the AND-OR DFD [54]. The two important kinds of grouping of actions are:

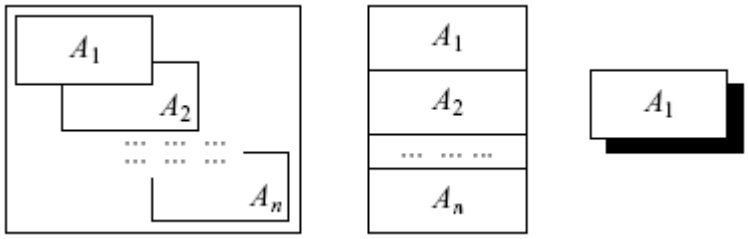
(1) AND-group: Each application of an AND-group of actions means all actions in the group are performed together in some fixed order, but which may or may not be made explicit. (This latter feature makes an AND-group different from a simple Boolean or logical "and".) This is comparable to the notion of a sequential block of code (statements) in a program, except that in the latter the order in which the statements are executed is always the same. Figure 15 (i) represents an AND-group of components. An and-group is shown as an enclosing box around the individual items of an and-group, which may be connected in a chain (say) to indicate the order in which they are applied if that is fixed (as is often the case).

(2) OR-group: This grouping is the other extreme of an and-group in that an application of an OR-group performs exactly one of those actions, with different applications performing perhaps different actions from the OR-group. An OR-group is comparable to an if-then-else statement (or more generally, a case-statement) in a program, except that we are not concerned with the selection or branching condition itself. The specification of the conditions for selecting the

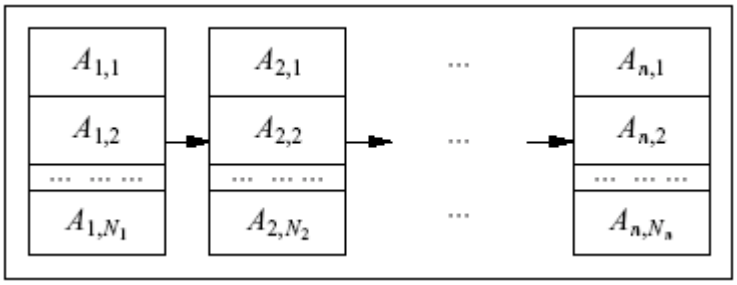
alternative actions is viewed as a lower level design-task and not part of the architecture itself. Figure. 15 (ii) illustrates the graphical notation for an OR-group of components. An or-group of actions is shown as an overlapping set of boxes, with all but one being fully visible, and without any separate enclosing box for the group as a whole. The partially visible boxes emphasize that only one of the actions is used in any instance of that group of actions. If an action is used iteratively, then this is indicated by a bold line for the box.



(i) An and-group of actions $\{A1, A2, \times\times\times, An\}$.



(ii) Three alternative ways of representing an or-group of actions $\{A1, A2, \times\times\times, An\}$.

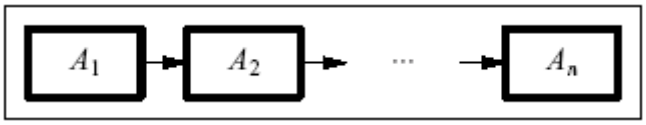


(ii) A possible and-or-combination providing $N1N2\times\times\times Nn$ possible workflows.

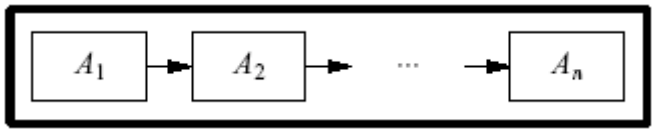
Figure 15: Action Grouping and Representation for AND-OR DFD [54]

Figure 15 (iii) represents a possible combination of AND-OR groups, which will create multiple workflows. Figure 16 shows an and-combination of iterations of several actions (possibly with different number of iterations for the actions), an iteration of the and-combination,

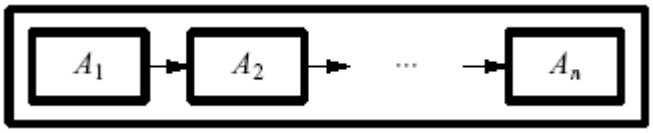
and the combination of both types of iterations that might happen while using this scheme for architectural representation.



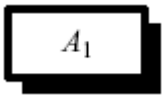
(i) An and-group of iterated actions.



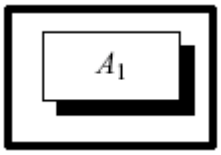
(ii) Iteration of an and-group of actions.



(iii) Iteration of an and-group of iterated actions.



(iv) An or-group of iterated actions.



(v) Iteration of an or-group of actions.

(vi) Iteration of an or-group of iterated actions.

Figure 16: Possible combination of iterative AND-OR groups [54]

In order to demonstrate the capabilities of the AND-OR DFD method of architectural representation, let us consider a software system called C2 Generator. This software system would be written in an object oriented language like JAVA and it attempts to generate a

architectural representation diagram based on the C2 Generator architecture [35]. It takes as input the components of the system to be modeled, the connectors and a list of who notifies whom. The C2 Generator will be discussed in further detail in the next chapter, but it will suffice to say here that C2 Generator is an architecture description language (ADL) that is used to model user interface intensive software systems i.e., applications that have a graphical user interface (GUI) aspect. The architectural style consists of components and connectors. Components and connectors both have a defined top and bottom. The top of a component may be connected to the bottom of a single connector. The bottom of a component may be connected to the top of a single connector. There is no bound on the number of components or connectors that may be attached to a single connector. Components can only communicate via connectors; direct communication is disallowed. When two connectors are attached to each other, it must be from the bottom of one to the top of the other. Both components and connectors have semantically rich interfaces. Components communicate by passing messages: notifications travel down an architecture and requests up. Connectors are responsible for the routing and potential multicast of the messages.

In the most simplistic model of the C2 Generator, we constrain the connector such that one connector can only connect two components. Table 6 shows the decomposition model of the process view of this software. We can talk of two different levels of decomposition for a software system. The logical decomposition model represents the logical components i.e., domain elements. The process decomposition model represents the process components i.e., who-calls-whom and control flow.

We see from Table 6 there are four primary components in this software. The CreateConnection component has five subcomponents, which are the various steps taken to create a connection. First, the component to be connected to first created component is identified

from the connection list. Then new ports are created and attached to both these components. We assume here for simplicity that both components can have unlimited number of ports and so unlimited number of connections. Then the connector is created and the two ports are connected.

It is obvious that the steps in creating a new connection start with reading a component name from the connection list till the connector is attached to the two newly formed ports. This whole process has to be repeated till there are no more entries in the connection list. This iterative property of the system cannot be known from the decomposition model, though it must occur if the system executes correctly. Second, there might be repeated entries in the connection list.

Table 6: Process Decomposition of C2 Generator

Module Name	Submodule(s)
1) ReadInput	
2) CreateComponent	
3) ReadConnectionList	
4) CreateConnection	CreateComponentToBeConnecteed
	CreatePorts
	ConnectPortsToBothComponents
	CreateConnector
	ConnectBothPortsWithConnector

There is no restriction to the number of connections one component can have with other components. For an entry that refers to a component which has already been created, one doesn't need to create it again, but just identify that component and create a new port. Hence, once an entry has been read from the Connection List, one of two things happen depending on the read value. Either the component doesn't exist and needs to be created, or it exists and needs to be identified. Again, there is no way of knowing this from the decomposition model. A traditional

DFD will also not illustrate the two issues discussed above. Let us now consider how the AND-OR DFD tackles these issues.

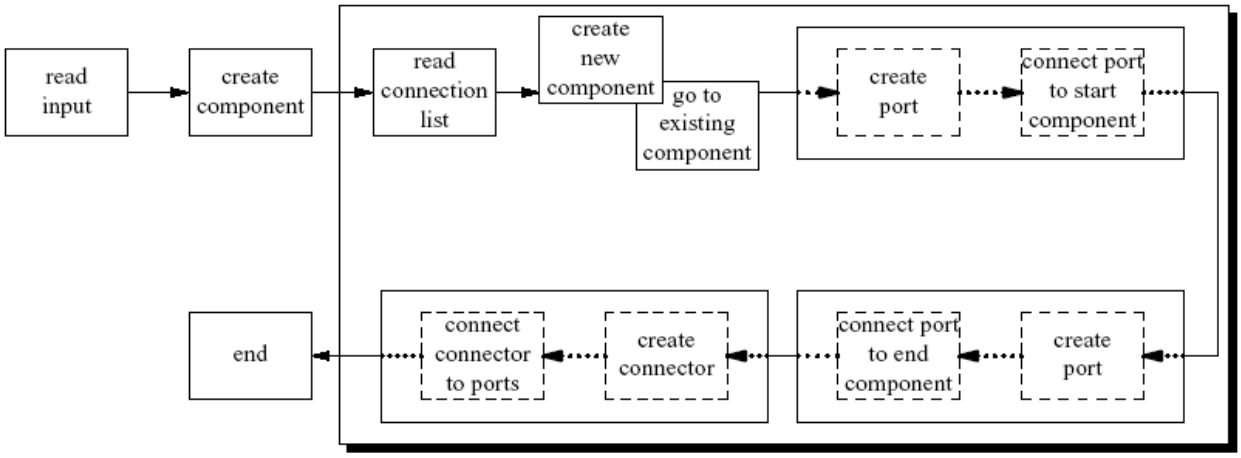
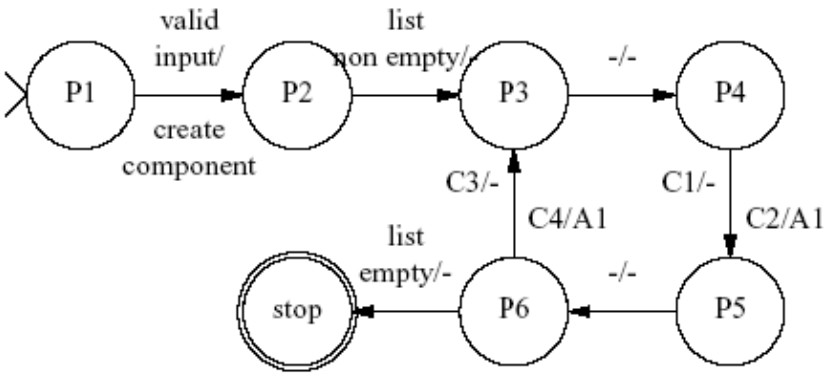


Figure 17: AND-OR DFD representation of C2 Generator (process view)

Figure 17 shows the AND-OR DFD representation of the process view of the C2 Generator. We see that the data flow is represented by this modified DFD, but with two significant differences. Firstly, there is an OR-group of two components that illustrate that once an entry has been read from the connection list, either a new component is created, or control moves to an existing component, depending on the value read from the connection list. Second, the iterative portion of the system has been illustrated by a shaded box. So we now can tell that the steps starting from the reading of the connection list to the connection of the ports by a connector are iterative and are executed for each entry in the connection list. So this representation does improve on the other architectural representation schemes discussed so far, and simplifies the diagram by addressing a few issues that have to do with control-data interaction.

Let us now compare this AND-OR DFD method with a finite state machine (FSM) (or finite state automaton) [22, 32] approach to architectural representation. Traditionally, FSMs have not been used for modeling architecture. They have used to model the control flow of processes, and sequential execution flows of programs. But we are of the opinion the FSMs can be used to

depict the process view of systems. Let us, as an example, take a FSM representation of the C2 Generator's process view, as shown in Figure 18.



P1 --> Read Input
P2 --> Read Connection List
P3 --> Create Port and Attach to Start Component
P4 --> Create Connector and Connect to Port
P5 --> Create Port and Attach to End Component
P6 --> Attach Port to Connector

C1 --> end component exists
C2 --> end component doesn't exist
C3 --> list non empty and end component exists
C4 --> list non empty and end component exists
A1 --> create component

Figure 18: FSM representation of C2 Generator

This representation in essence captures the same picture as the AND-OR DFD. But the problem with FSMs is that these models are focused on action-sequence, not on showing the components and their interactions. Moreover, with FSMs, it isn't suitable to show the inputs and outputs of a component. For example, an action's (component) name may appear in many transitions.

Thus we have seen that the AND-OR DFD and the FSM are suitable for representing the process view of an architecture. The AND-OR DFD can represent iterations and AND-ed and OR-ed processes. It can show multiple workflows and does represent control data interaction. There are still some shortcomings of this method. For example, we cannot say from the AND-OR DFD what is the input value for which a particular choice of component will be made, or

what condition makes a component or a set of components iterate. The FSM can handle multiple workflows but concentrates on action-sequence and not on component and connectors. The decomposition method, of course, is crude because it cannot represent iteration, AND-ed and OR-ed processes. Also, most of the representations, with the exception of the AND-OR DFD and the FSM cannot handle multiple workflows. We shall see in the next chapter that UML can be used to represent different views of an architectural description, and so is an efficient means of representing architecture.

Chapter 5: Architecture Description Languages and UML

Architecture description languages (ADLs) are emerging as the notation for architecture models. ADLs use graphics and text to express architectural information [29]. ADLs are often supported by tools for creation, modification, browsing, simulation, and analysis. ADLs provide constructs to specify architectural abstractions in a descriptive notation. The description of architecture lies in the representation of its basic abstractions, i.e., components and connectors, as we have discussed previously. ADLs have been designed with the aim of providing formalism to architectural descriptions. A useful ADL should be able to provide a set of precise semantics that resolves ambiguity and aids in the detection of inconsistencies and a set of techniques that support reasoning about system properties. Moreover, it should be able to support architectural evolution and reuse of components and connectors. So ADLs attempt to get rid of the shortcomings of architectural representation schemes like boxes-and-lines, DFDs and FSMs, while attempting to formalize architectural representation.

There are a variety of ADLs emerging from various industrial and academic research groups. For example, UniCon has been developed to explore issues of abstractions for architecture and composition of systems [42]. C2 is an ADL that caters to user interface intensive systems [35]. Some ADLs are commercial products like UNAS/SALE which was developed by TRW and marketed by Rational [29]. Other ARPA-sponsored ADLs include LILEANNA [28], Rapide [31], MetaH [29], and ArTek/DADSE [29]. ADLs vary widely in the architecture styles they support and that forms of analyses they permit. Like other tools, there is no one ADL that best fits all possible situations.

A variety of groups use ADLs. Each group has a different perspective of what a good ADL should be. People involved in the acquisition of software specify ADLs for procurements and

evaluate architecture models (encoded in an ADL) that are included in proposals and presented at design reviews.

Each ADL represents a particular approach to the specification and evolution of an architecture. Answering specific evaluation questions demands powerful, specialized modeling and analysis techniques that address specific aspects in depth [35]. However, the emphasis on depth over breadth of the model can make it difficult to integrate these models with other development artifacts, because the rigor of formal methods draws the modeler's attention away from day-to-day development concerns. So different ADLs address different, sometimes overlapping problem domains, but none of them address a relatively large problem set. The use of special purpose modeling languages has made the architecture community fairly fragmented [35]. The need of the hour is an approach to a formal, generalized ADL, and ~~the~~ Unified Modeling Language [UML] [27, 39] has been suggested as an option.

5.1 Architecture Description Languages

ADLs serve the purpose of being an intersection between requirements, programming, and modeling languages, but are certainly different from all of the three. They differ from requirements language in that requirements language describes the problem space while the ADL is focused on the solution space. They differ from programming languages in that programming languages aim to bind specific solutions with the architectural abstractions while ADLs deliberately suppress such binding. ADLS are also different from modeling languages in that modeling languages are more concerned with the behavior of the whole system rather than parts of it while ADLs are more focused on representation of components [7]. Existing ADLs are generally characterized by a graphical syntax accompanied by textual support and formally defined syntax and semantics. They can handle data flow and control flow as interconnection

mechanisms, and have the ability to represent hierarchical levels of detail and multiple instantiations of a template.

A glaring shortcoming of almost all ADLs available today is the inability to address a large problem domain and all types of architectural styles. Let us look at the existing ADLs and how they support the main architectural components, namely components, connectors and configurations.

5.1.1 Components

Components can be computation units or data stores. According to [42], components are loci of computation and state. A component may a single procedure or an entire application and may require its own space for data and execution. The aspects of a component that need to be represented by an ADL are interfaces and types, semantics, constraints, and evolution. The interface enables a component to interact with the external world. They also enable a certain limited degree of reasoning about component semantics [33]. All ADLs provide support for component interfaces. These are called ports in some ADLs like C2 [35], while they are called constituents in other ADLs like Rapide [31]. ADLs can provide software reuse by modeling abstract components as types and instantiating them multiple times as and when needed in an architectural configuration. Almost all available ADLs distinguish component types from instances. Modeling of component semantics enables analysis, constraint enforcement and mapping or architectures across levels of abstraction. Some ADLs do enable modeling of component semantics, while some like Darwin and UniCon only talk of interfaces [15]. Underlying semantic models vary across ADLs. Wright allows specification of component functionality using CSP [22]. One important expectation of an ADL is that it should support

component evolution through subtyping and refinement. As of now, very few ADLs satisfy this criterion, as does Rapide.

5.1.2 Connectors

Connectors serve two purposes. They allow interaction between components and enable the specification of rules that govern these interactions. Connectors, unlike components, may not be compilation units, but rather can be message passing devices, shared variables, table entries, procedure calls, and the like. The aspects of a connector that should be modeled in an ADL are interfaces and types, semantics, constraints and evolution. Some ADLs like C2, Wright and UniCon model connectors as first-class entities. These languages are called explicit configuration languages [15]. Languages like Rapide, on the other hand, are in-line configuration languages [15]. Here connectors are described solely in terms of the components they connect, and cannot be named, subtyped or reused. Designing architectures are easier using the former kind of languages, because in the latter adding new components may require modification of existing connections.

Interfaces of connectors are interaction points between them and the components they attach. Only explicit configuration languages support specification of connector interfaces. Architecture-level communication is often expressed with complex protocols [15]. To make these protocols more abstract and reusable, ADLs may model connector as types. Again, only explicit configuration languages support connector types and distinguish them from instances. Some explicit configuration languages like C2 do not specify connector semantics; while others that are inline configuration languages, like Rapide, do specify connector semantics. ADLs that specify connector semantics generally use a single semantic model for both component and connectors.

5.1.3 Configuration

Architectural configurations are connected graphs of components and connectors that describe architectural structure. Configurations provide answers to questions such as whether appropriate components are connected, if their interfaces match, whether connectors enable proper communication, and whether the combined semantics of the components and connectors result in the desired behavior of the system. ADLs should be able to model the structural information with a simple and understandable syntax. Some ADLs provide both a graphical and textual notation. Graphical specifications provide another way of achieving understandability.

Support for compositionality is very important for architectures because complex behaviors need to be explicitly represented or abstracted away into single components and connectors. Moreover, in some cases, an entire architecture may become a single component for a larger architecture. Some ADLs like Darwin, Rapide and C2 provide explicit features to support hierarchical composition. Another important feature of architecture is the ability to facilitate development of large systems, with components and connectors of varying granularity, implemented by different developers, in different programming languages, and with varying operating system requirements [33]. It is therefore important for ADLs to provide facilities for architectural specification and development with heterogeneous components and connectors.

5.2 The C2 Architecture Description Language

We shall talk about C2 in order to illustrate an ADL. We use the C2 Generator as our working example in this chapter. C2 is an architectural style [35] that is suitable for modeling user interface intensive software systems i.e., applications that have a graphical user interface (GUI) aspect. C2SADEL is an ADL for describing C2-style architectures [35]. In C2-style architecture, connectors transmit messages between components, while components maintain

state, perform operations, and exchange messages with other components via two interfaces which are called top and bottom. Each interface consists of a set of messages that may be sent and a set of messages that may be received. Inter-component messages are either requests for a component to perform an operation, or notifications that a given component has performed an operation or changed state. In the C2 style, components cannot interact directly but can do so using the connectors. Each component interface can be attached to at most one connector. A connector, however, can be attached to any number of other components and connectors. Request messages can only be sent "upward" through the architecture, and notification messages can only be sent "downward."

The C2 style has another requirement that the components communicate with each other only through message passing and never through shared memory. Also, C2 requires that notifications sent from a component correspond to the operations of its internal object, rather than the needs of any components that receive those notifications. This constraint on notifications helps to ensure substrate independence, which is the ability to reuse a C2 component in architectures with differing substrate components (e.g., different window systems). The C2 style explicitly does not make any assumptions about the language(s) in which the components or connectors are implemented, whether or not components execute in their own threads of control, the deployment of components to hosts, or the communication protocol(s) used by connectors.

5.3 Unified Modeling Language (UML)

UML (Unified Modeling Language) is a language to specify, to visualize, to build, and to document the artifact of the software system, as well as to model business and other systems besides software systems [39, 52]. UML is composed of a group of nine diagrams with clearly defined syntax and semantics. These diagrams are: Use case, Classes, Objects, Sequence,

Collaboration, State, Activities, Implantation and Components Diagrams. Each diagram depicts different aspects of a system. Together, they offer the representation of a complete system.

Each view of a system can be described by one or by several diagrams. A diagram can depict aspects of more than one view. Usually, the use case, classes, objects, implantation and components diagrams depict static aspects of a system. The dynamic aspects are represented by the sequence, collaboration, state and activities diagrams [52]. Due to its flexibility, UML has been used with success in the description of architectures and meta-models for software development [23]. However, this flexibility is source of problems such as duplication and inconsistency of information in the global model, as well as poor integration among its several diagrams or among methods that aid in the conversion of the information from a diagram to another. Therefore, to guarantee the conceptual integrity of the description, it is necessary to be able to identify mismatching elements in the model and to integrate its properties. In this section we briefly describe UML, which is currently the leading object-oriented analysis and design model. UML's views are for the most part graphical diagrams. The lead designers of UML were Booch, Jacobson and Rumbaugh [39]. Let us now discuss some of the basic constructs of UML.

- **Use Case** (e.g. Use Case Diagrams): The use case view captures the behavior of a system, subsystem, or class as it appears to an outside user. It partitions the system functionality into transactions meaningful to actors-idealized users of the system. The pieces of interactive functionality are called use cases.
- **Interaction** (e.g. Sequence and Collaboration Diagrams): These are sometimes also referred to as Mini-Uses. A sequence diagram is often used to rigorously define the logic for a use case scenario. Because sequence diagrams look at a use case from a different point of view from which it was originally developed, it is common to use sequence diagrams to validate

use cases. The UML illustrates the behavior of a system by collaboration diagrams. Collaboration models the objects and links that are meaningful within an interaction. A collaboration diagram shows the roles in the interaction as a geometric arrangement.

- **Objects and Classes** (e.g. Class Diagrams, CRC Cards): A class is a descriptor for a set of objects with similar structure and behavior. Class diagrams are the mainstay of object-oriented modeling and are used to show both what the system can do (analysis) and how the diagram will be built (design). Class diagrams show the classes of the system and their interrelationships (including inheritance, aggregation, and associations).
- **Packages** (e.g. Package Diagram): Packages are used to group classes into layers and partitions. As such they show the functional decomposition of a system.
- **State Transition** (e.g. State and Activity Diagrams): This well-known technique is used in UML to describe the states a class can go through. Statechart diagrams are useful when describing a single object, which can have different states during its lifetime. The statechart diagram illustrates how the object changes from one state to another in response to a given event. The attributes define the state of the object at one time, while the methods define the transitions that take place. Statechart diagrams are used to describe complex classes. Relationships between classes are documented with a description of their purpose and an indication of their cardinality (how many objects are involved in the relationship) and their optionality (whether or not an object must be involved in the relationship). An activity diagram is a special form of a state diagram. Transitions in the diagram are mainly triggered by the completion of action by the source state. The diagram focuses on the flow of operations driven by internal processing as opposed to external events. However they do not

make explicit which objects execute which activities. An activity diagram illustrates the dynamic nature of a system by modeling the flow of control from activity to activity.

- **Deployment** (e.g. Deployment Diagrams): It shows the relationship of the components of the system during deployment. As such it presents a physical view of the system. It is, therefore, frequently used to depict the component dependency of the actual implementation.

5.4 Modeling

One might wonder why software engineering, or for that matter, all aspects of software lay so much emphasis on modeling. One of the reasons for the complexity of software is due to the fact that the number of information loaded onto a single person is vastly exceeding the capabilities of the human mind. We are not able to handle thousands of pieces of information at any given time. Instead it seems that the human short term memory is quite limited in that respect. The 7±2 rule is a well-known example. This rule states that the human mind can usually only handle 7 distinct things (plus or minus 2) at the same time [13].Obviously, even the simplest piece of technology consists of more (much more) than 9 pieces! The answer to that is probably the concept of modeling. If we encounter too much information, we consciously or unconsciously group this information in some way that makes it easier for us to recall it later on.

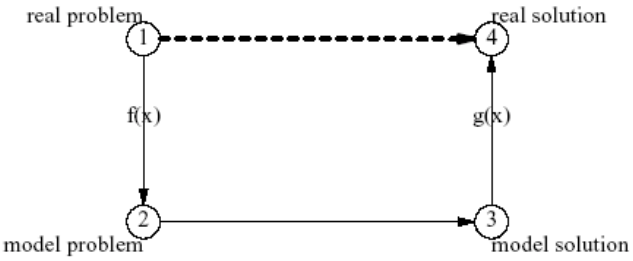


Figure 19: Mathematical Modeling

Figure 19 shows how modeling works in the context of a mathematical problem. For a real problem, we try to create a simplified model problem by applying some transformation f(x) on

the actual problem. This process can be recursive and can go on till we have a model that is simple enough to solve. The model problem then is solved to obtain the model solution. This model solution is then mapped to the actual solution by gradually introducing back the complexities that were removed from the actual problem to create the model problem and solving them.

The same modeling concept can be applied to software. UML is a third generation modeling language. Figure 20 shows how the modeling of a software problem works in the case of software modeling using UML.

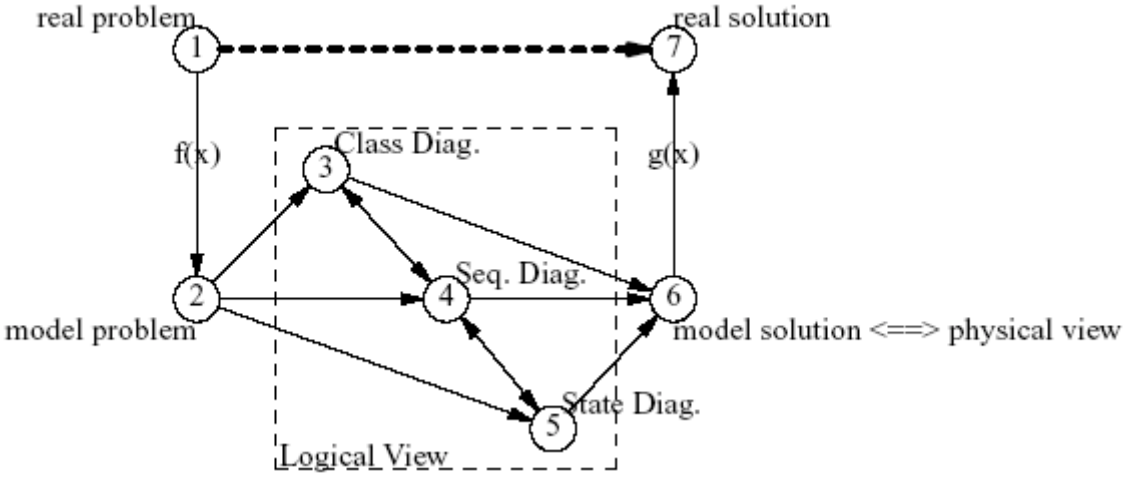


Figure 20: Software Modeling using UML

Again, we try to construct a simplified model of a problem so that we can solve it. The model is now represented by different views. The logical view of the solution of the problem is illustrated by class diagrams, sequence diagrams and state diagrams. Each of these diagrams represents the solution from a different viewpoint, and so are equivalent to each other in some way. An integration of these views will help us to arrive at the diagrammatic representation of the model's solution, which is the physical view. Once the model's solution is obtained we can

easily map it to the actual solution. Of course, the real picture is much more complex since it usually involves multiple levels of abstractions but the basic concept of modeling holds good.

5.5 UML for Software Architecture

In order to represent architecture using UML [1, 6, 12, 13, 21, 24, 27, 37, 38], researchers have separated the architecture into four views: conceptual, module, execution and code [24]. Each of these views addresses different concerns, and separation of these concerns allows the architect the room to make decisions without design trade-offs.

The conceptual view (logical view) describes the architecture in terms of domain elements. Here the architect designs the functional features of the system. For example, one common goal is to organize the architecture so that functional features can be added, removed, or modified. This is important for evolution, for supporting a product line, and for reuse across generations of a product. To represent this view using UML, class diagrams are used to show the static configuration, sequence diagrams or state diagrams are used to show the protocols the ports adhere to, and sequence diagrams are also used to show a particular sequence of interactions between a group of components [1].

The module view describes the decomposition of the software and its organization into layers. An important consideration here is limiting the impact of a change in external software or hardware. Another consideration is the focusing of software engineers' expertise, in order to increase implementation efficiency. The decomposition dependencies, the use-dependencies among layers and the assignment of modules to layers are shown by the use of package diagrams while class diagrams are used to depict class dependencies [1].

The execution view (process view) is the run-time view of the system. It involves the mapping of modules to run-time images, defining the communication among them, and assigning

them to physical resources. Resource usage and performance are key concerns in the execution view. Decisions such as whether to use a link library or a shared library, or whether to use threads or processes are made here, although these decisions may feed back to the module view and require changes there. Again, class diagrams are used to show the static configuration, sequence diagrams to depict the dynamic configuration, and state diagrams or sequence diagrams to show the protocol of a communication path [1].

The code view captures how modules and interfaces in the module view are mapped to source files, and run-time images in the execution view are mapped to executable files. The partitioning of these files and how they are organized into directories affect the buildability of a system [23], and become increasingly important when supporting multiple versions or product lines. Here, tables are used to describe the mapping between elements in the module and execution views and elements in the code view. Component diagrams are used to show the dependencies between source, intermediate, and executable files [1].

5.6 Architecting Using UML: An Example

In order to see how UML can construct the software architecture of a system, let us go back to the example of the C2 Generator. We have already seen the process decomposition of the software. Table 7 shows the logical decomposition of the system. The logical decomposition highlights the main components of the system and their subcomponents if any. It doesn't provide any notion about the connectors, i.e., which modules are interconnected, and what the properties and types of the components and connectors are. But it does serve the purpose of at least highlighting the main components required for the software. We can see from Table 7 that there are four main components, one for generation of a connection and three for the three constituents of a connection, namely component, connector and port.

Table 7: Logical Decomposition of C2 Generator

Module Name	Submodule(s)
1) C2Generator	CreateComponent
	CreateConnector
	UpdateComponentList
	UpdateConnectionList
2) Component	CheckForFreePort
	CreatePort
3) Port	CreateConnector
4) Connection	UpdateConnectingComponent
	UpdateConnectedComponent

We see that the C2Generator component has the task of creating the component(s) and the connector, and updating the component and connection lists. The component module checks for free ports on the component(s) and if there are free ports, then it creates the physical port. The Port component creates the connector in turn, and the connector component joins the two components (called the connecting component and the connected component here) and updates the two components for the connection created.

We shall use the UML constructs defined in [23] for all our diagrams, with some modifications. Figure 21 shows the conceptual component and connector as used in [23]. We can see that the conceptual component contains ports which are the interaction points for the component. Also, a component can be decomposed into component and connectors.

As we can see from figure 21, the component can have one or more ports. There exists a composition relation between the component and the port. It means that a port is part of a component and cannot exist independently. The composition relationship is indicated by the solid diamond below the ‘Component’ component. The asterisk above the port component

indicates that the composition relation is multiple, i.e., a component can have multiple ports and between the component and each of these ports there exists a composition relation. There is also a multiple composition relationship between a connector and itself, meaning that a connector can connect to one or more other connectors. The unfilled diamond in the figure represents an aggregation relation. This indicates that a connector references a port but can exist without it. This is the basic difference between a composition and an aggregation.

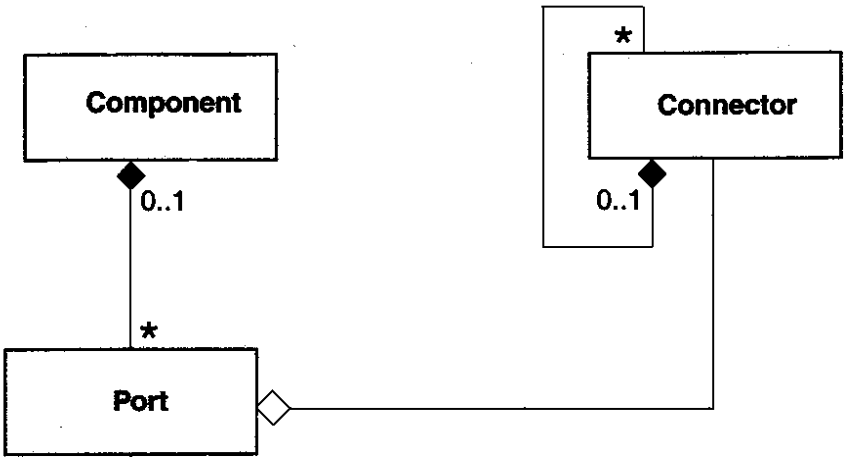


Figure 21: Component and Connector

5.6.1 Conceptual View

We have already seen the conceptual (logical) decomposition of C2 Generator in Table 7. Let us now try to construct the logical architectural view for C2 Generator. Figure 22 shows the conceptual architectural view of the C2 Generator using UML constructs [23].

Figure 22 is a modified UML class diagram and the components, ports and connectors are represented by stereotyped classes. For each of these, a special graphical symbol is used for clarity and ease of understanding. So the components are represented by rectangles, the ports by smaller rectangles with curved corners, and the connectors by elongated hexagons. In the conceptual view, there are three components that interact with the outside world through their ports. The component connects to a port through a connector called PortConn, and the port

connects to the connector through the ConnectorConn. The composition relationships are shown by the solid diamonds.

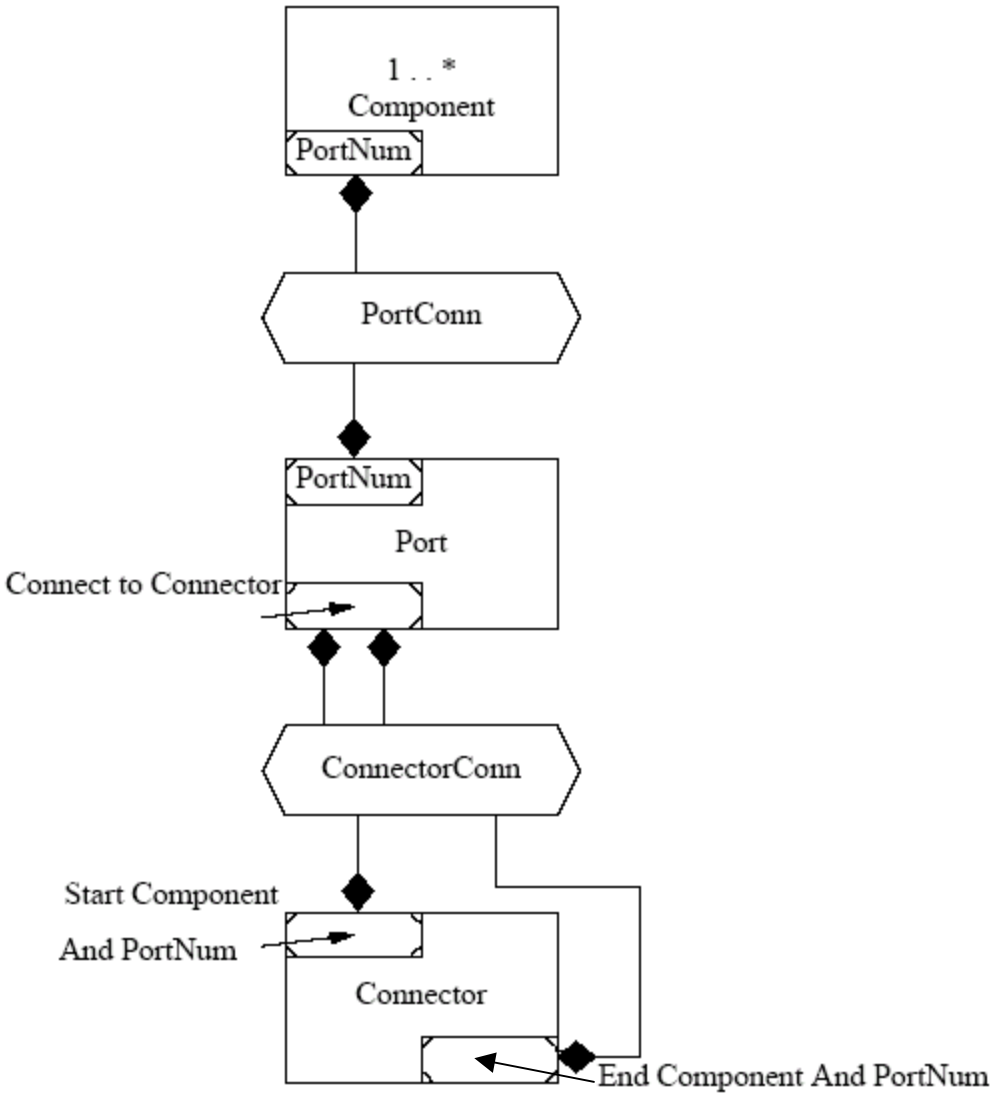


Figure 22: Conceptual View of C2 Generator

The problem with this representation lies in the relationship of the port and the ConnectorConn. The diagram shows a composition relationship between a port component and the ConnectorConn connector. This is obviously not right because the ConnectorConn can exist even if it is separated from one of the ports, i.e., a connector can be broken off from one component and joined to another component. So a composition doesn't hold good here. On the

other hand, even an aggregation doesn't hold good because when the connector is isolated from the ports of both the connecting components, it ceases to exist independently. So here is a situation where there is a composition relationship that involves two components and a connector and there is no notation in UML to denote that.

5.6.2 Execution View: Two Representations, and a Problem

The execution view or process view of C2 Generator will be modeled from the process decomposition model we saw earlier (table 6, chapter 4). The representation of the process view involves class diagrams to show the static configuration, and sequence diagrams to show the dynamic behavior of the configuration, or the transition between configurations. We shall focus on the dynamicity of the configuration. Figure 23 shows one sequence diagram representing the execution configuration of the C2 Generator. We can see the execution flow of the C2 Generator from this figure. The C2Generator first creates the connecting component by calling the CreateComponent() procedure and interacting with the component module. The component module in turn then creates a port and connects the newly created component to it by calling the CreateAndConnectToPort() procedure and communicating with the Port module. The Port module now creates the connector and attaches the port to this connector by calling two functions and talking to the Connector module.

Once this is done and the control is back to the C2 Generator component, it now reads the connection list and checks if the component to be connected exists or not. If it exists, control moves to this existing component and that component is connected via a new port to the already created connector. If the component doesn't exist then it is created before being connected to the connector.

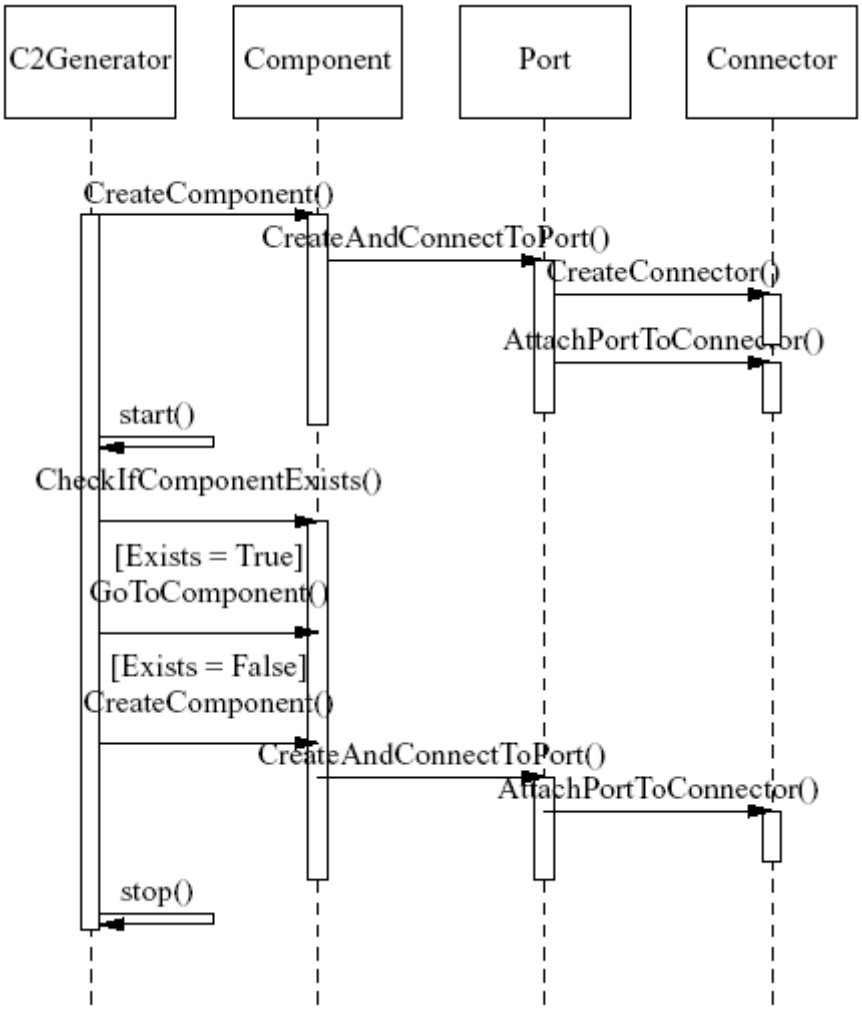


Figure 23: Sequence Diagram for a process view of C2 Generator

Let us now look at another representation of the same process view. Figure 24 is a sequence diagram showing another possible dynamic configuration of the C2 Generator. We can clearly see the difference and figure 24 represents a better way of control flow for the C2 Generator. Here, first both the components taking part in the connector are identified, or created if they do not already exist. Then the control moves to the Component module, and ports are created for both the components by invoking the Port module. Then finally the connector is created and both ports are connected to it.

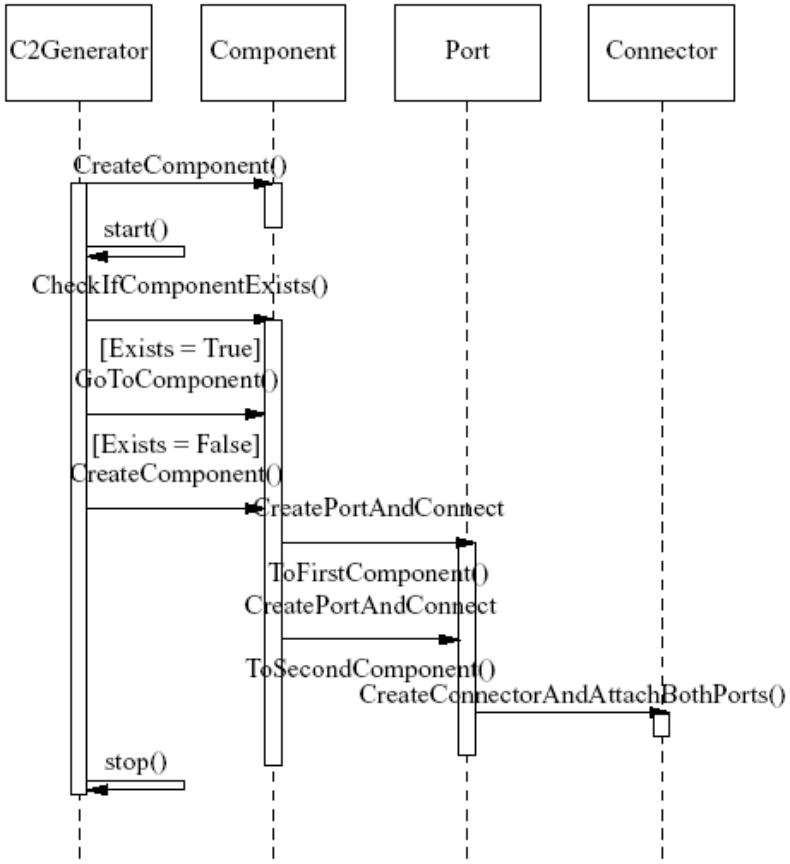


Figure 24: Another Sequence Diagram for a process view of C2 Generator

This implementation is more efficient because the control flow doesn't move back and forth as in the previous implementation. Both the components are ready before the ports are created and both the ports are ready before the connector is created and the connection made. But the point to be noted is that both of these are correct process views for the conceptual view of C2 Generator. There is no way in UML to choose one over the other. So we need other diagrams of the process to accompany the FSM representation in order to make sure that we have a good representation. It would definitely help if we also make AND-OR DFD and FSM diagrams of the process view. This can help in reducing the possibility of having an erroneous representation of the process view. Therefore we can se that while architecting with UML, one might easily

decrease the efficiency of the software by choosing the first implementation rather than the second one.

So we see that UML is rather useful for representing different views of the software architecture of a system [23, 24]. It does reasonably well and represents all the facets of that view clearly. Moreover, UML is good for all the views, and not just the process view which can be adequately represented by the AND-OR DFD or the FSM. We have different constructs that we can use for the different components of a view. Moreover, we can extend UML by constraints, tagged values, stereotypes and profiles [37]. These extension mechanisms can help to add the necessary clarity to a representation or to create a new presentational block.

UML, however, has its own disadvantages. We have already seen some disadvantages of using UML in the example of the C2 Generator. We saw in case of the logical view that the notations were not able to capture the relationship between the port and the connector adequately. We also saw in the case of the process view that one logical flow can have more than one process flow and there is no way to tell whether the one chosen is the best implementation of the software. This implies that while UML can represent the static configuration with ease, it is not so good in depicting the dynamic nature of the configuration. Also, the graphical notation of UML is too complicated for simple mappings like the correspondence between elements in different views. So view integration is very difficult using UML. One more problem with UML is that it cannot represent peer-to-peer communication [24].

Thus, UML has the potential of becoming a generalized ADL due to its wide acceptance, and its features having the capability of representing architectural views. However, UML without any modifications does not quite fit the bill. Research is being done into extending UML capabilities and constructs to make it more compatible for software architectural description [37].

Chapter 6: Summary And Conclusions

6.1 Summary

This work has looked at software architecture, its concepts, definitions, styles and representational schemes. We have looked at two popular definitions of software architecture by Kazman [7] and Wolf [40] and demonstrated their strengths and shortcomings as definitions. First we have seen Kazman‘s definition, which talks about software architecture as a configuration of components, their externally visible properties and the relationships between components. This definition however, says nothing about the type and properties of components, nor does it say anything about the nature or properties of the connections. Next, we have seen how Wolf‘s definition states that there are three kinds of elements, namely processing elements, data elements and connecting elements, and each has different properties and roles. The connecting element acts as a glue that holds the other two kinds of elements. It also states that connecting elements play a fundamental part in distinguishing one type of architecture from another and are significant in distinguishing between architectural styles. It also puts forth the concept of constraints and weighted relationships and the rationale for each component in the architecture. But it does not state any rationale for choosing a particular type of architecture or how a chosen module is suitable. We have taken the example of the architecture of a software system called Code Builder that we have developed in order to explain how much these two definitions tell about software architecture and where they fall short. This software takes as input a C program and identifies the blocks in that program and displays it.

Next, we have looked at architectural views. We have seen how different stakeholders have different interests in the same system and how they view the same system differently from their own perspective. This concept of looking at the same system from a different perspective has

given rise to architectural views. In this light, we have discussed Kruchten's —4+1" view model [30], which is by far the most popular and widely accepted view model.

We have discussed architectural styles, or patterns that have emerged out of experience of software architects in the industry and research on software architecture [43, 46]. We have seen the different styles that are commonly used and accepted, and whether our Code Builder software can be modeled in these styles.

We have taken a more detailed look at the layered architectural style and suggested that this style can be used to address a larger problem domain that the message-passing type of systems it has traditionally been used to model. We have suggested some modifications and rules for this architectural style so that it can model other systems like pipelines, networking systems, parallel processes and procedural processes. By allowing some leniency in the layering model by the introduction of intra-layer communication between subcomponents of a layer and transparency of all lower layers, we can model all procedural systems and parallel processes with a single direction of control.

For processes where the control flow is multi-threaded, we can introduce the concept of bidirectional communication between layers and components in a layer. Of course, pipelines generally conform to the strict layering model of unidirectional control flow and opacity of all lower layers apart from the immediately succeeding layer. We have described the layered model for the Code Builder system and for another software system called the Cost Optimizer that finds the least expensive array of paths for a bunch of messages traveling from one node to another in a network. We have seen how the rules that we have devised can enable these two software systems to be represented using a layering style. Our attempt is to use a layered architectural style as a generalized architectural style for software architectural representation. Our initial

research and the use of layering in our examples have been encouraging. The use of layering in modeling C2 style architecture for GUI intensive software systems [35] and the use of layering in representation of module view of an architecture using UML also indicate the vast potential for the layering style. Further research should be undertaken to model different kinds of systems like concurrent systems and parallel systems using layered architecture. We hope to be able to develop a set of formal rules by which layered architecture can be used as a generalized style for architectural development.

In this context, we have also looked at decomposition as a method of representing software architecture and how decomposition is useful for identifying the components of an architecture. From the decomposition model, one can easily identify the layers of a software architecture and the components in each layer. A simple rule for this transformation is that each component of the decomposition becomes a layer and the subcomponents, if any, become the constituents of that layer. Of course, this method of architectural description is crude and doesn‘t say anything about the type of components, which components interact and how. We have used decomposition for our examples to identify the components and once this is done, it becomes easier to identify the layers of a software system. Later, we have also looked at two different kinds of decomposition of the same software when we have developed the logical view decomposition as well as process view decomposition for the C2 Generator software, which is a software system that we have developed. This software generates C2 style architecture for a software system once the components and a list of ”who-connects-to-whom‘ is specified.

Next, we have looked at different ways of representing software architecture. Representation of software architecture while retaining its clarity, simplicity and representational power has always been a challenge for software architecture researchers. Although there are several popular

and widely used methods for representing software architecture like boxes-and-lines and decomposition, which are informal or at best semi-formal, and ADLs like C2 [35], Rapide [31], and Wright [1] which are formal representation schemes, the search for a generalized architectural representation scheme that can be used for all software systems and across the software industry and research domain is still on.

We have seen the decomposition and the boxes-and-lines schemes and have concluded that they are informal and inefficient. The boxes-and-lines can only identify the components and connections, but can say nothing about the control and data flow. It cannot represent iterative processes, or indicate which components are always executed and which may be chosen depending on the input.

The boxes-and-lines method seems to indicate that the data flow diagram (DFD) may be a more suitable method for representing software architecture of a system because it does indicate data flow through a system and indicates the direction of control flow. So we try the data flow diagram as an alternative method for representing software architecture. We have chosen a simplistic program with a set of function calls to illustrate the use of DFD for representation of software architecture. We have seen from this exercise that the problems of the DFD are similar to the boxes-and-lines method. Moreover, the DFD is data centric, not process centric. One control flow can have multiple data flows for different input value(s). Thus the DFD cannot be a good software architecture representation scheme as it is. In this work, we introduce a modified DFD method called the AND-OR DFD that is more suited to representation of the process view of a system.

The main concept introduced here is an OR combination of components to support multiple workflows and the relationships among them, without significant increase in the architectural

complexity. Two important types of action grouping that are salient features of this scheme are the OR-group and the AND-group. Each application of an AND-group of actions means all actions in the group are performed together in some fixed order, but which may or may not be made explicit. The other extreme is represented by the OR-group, where an application of the OR-group of components means that exactly one of the actions of the group is performed. The specification of the conditions for selecting the alternative actions is viewed as a lower level design-task and not part of the architecture itself.

We have devised a graphical representation scheme for the AND-group and the OR-group. Also, in the AND-OR-DFD scheme, we have devised a way to represent iterative processes by shaded rectangles. We have applied this scheme to represent the architecture of the C2 generator with good results. The process decomposition of the C2 generator is very adequately represented by the AND-OR DFD. We have also compared the AND-OR-DFD representation with a finite state machine (FSM) representation of the process view of the C2 Generator and seen that they are equivalent. The problem with FSM representation is that FSM is focused on action-sequence, and not on showing the components and their interactions, and nor is it suitable to show the inputs and outputs of a component. For example, an action's (component) name may appear in many transitions. In that respect, the AND-OR-DFD is a more useful method for representing software architecture than the FSM.

Next, we have moved forward and studied the more formal architecture representation methods using ADL (architecture description language). There are several ADLs that have been devised by researchers and architects. ADLs are emerging as the notation for architecture models. ADLs use graphics and text to express architectural information [29]. ADLs are often supported by tools for creation, modification, browsing, simulation, and analysis. ADLs provide

constructs to specify architectural abstractions in a descriptive notation. The description of Architecture lies in the representation of its basic abstractions, i.e., components and connectors, as we have seen in the previous section. ADLs have been designed with the aim of providing formalism to architectural descriptions. An useful ADL should be able to provide a set of precise semantics that resolves ambiguity and aids in the detection of inconsistencies, and a set of techniques that support reasoning about system properties. Moreover, it should be able to support architectural evolution and reuse of components and connectors.

We have discussed how existing ADLs support the main architectural components, namely components, connectors and configurations. We have then looked at C2 as an example of an ADL. We have then discussed UML as a modeling language and its basic constructs. We have seen the importance of modeling and how modeling is implemented in the context of software. We have discussed how UML can be used for representing software architecture. For architectural representation using UML, the researchers have classified an architectural representation into four views, namely the conceptual view, the module view, the execution view and code view. We have discussed what each of these four views describes. We have seen the basic UML notation of a component and connector using UML (figure 21).

Next, using the C2 Generator as an example, we have developed the conceptual and process views of this software‘s architecture using UML constructs. The conceptual view (logical view) describes the architecture in terms of domain elements. Here the architect designs the functional features of the system. In the conceptual view of the C2 Generator, we have seen the main components of the software and how they interact. We have seen the relation that exists between the components. We have seen how components interact through their interfaces. We have also identified a problem in this representation. We have inferred that the existing UML notation is

not comprehensive enough to illustrate all the issues involved in the representation of software architecture. We have shown how the connector can exist independently if isolated from one of the two connected components and so it does not share a composition relationship with a component, but it does share a composition relationship with the two connecting components taken together, because a connector has no meaning without its connecting components.

We have modeled the execution view of the C2 Generator using a UML sequence diagram. We have shown the control flow in this diagram, and which component is ”active‘ at which time, and who calls whom in order to do a specific task. We have also put light on an issue that might erode some of UML‘s credibility as a generalized ADL. For the C2 Generator, we have displayed two correct process views, one better in terms of computation cost. But both these process views are correct representations of the conceptual view of the C2 Generator. This again is a shortcoming of UML, because we have no way of saying which representation is better and how. So we suggest that it would help to have more than one representation of the process view, using UML sequence diagrams, and other methods like the AND-OR-DFD. This would minimize the chances of erroneous representations. Thus, we have concluded that while UML is a prospective generalized ADL, we need some changes and modifications to UML to make it more suited to architectural representation.

6.2 Opportunities for Future Work

There are many positives from this work, and further research is needed in a few directions. We should do further research into representation of software systems using layering. We should try to model commercially successful and tested systems using this style in order to determine if layering can be used as a generalized approach to architecture development. In order to achieve

this objective, we need to come up with a protocol that shall lay down the rules by which layering can be used as a generalized architectural style.

As far as representation of software architecture is concerned, further research is needed to determine if the AND-OR-DFD can be used to model the process view of a software without any errors. We also need to do further research into the area of extending and modifying UML in order to developing a generalized ADL that shall be able to formally describe the software architecture of software systems. Research is already being done in this direction [6, 37]. There have been suggestions on how to extend UML [37] in order to take care of the discrepancies that creep in into the software architecture representation of a system using UML when we try to integrate the different views. This is one of the biggest bottlenecks that can deter UML from being a generalized UML. View integration is very difficult using UML and there are view mismatches that need to be resolved [37]. Research is needed in this area.

References

1. Abdurazik, A.: "Suitability of the UML as an Architecture Description Language with Applications to Testing," February 2000, ISE-TR-00-01

2. Abowd, G., Allen, R., Garlan, D.: "Using Style to Understand Descriptions of Software Architectures," Foundations of Software Engineering, Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering Pages: 9 - 20, 1993

3. Allen, R., Garlan, .D: "A Formal Approach to Software Architectures," Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - Information Processing '92, Volume 1, Pages: 134 - 141, 1992

4. Allen, R., Garlan, D.: "A formal basis for Architectural Connection," ACM Transactions on Software Engineering and Methodology (TOSEM) Volume 6 , Issue 3, Pages: 213 - 249, 1997

5. Balek, D., Plasil, F.: "Software Connectors: A Hierarchical Model," Tech. Report No. 2000/2, Dep. of SW Engineering, Charles University, Prague, Revised 11/25/2000, Nov 2000.

6. Balsamo, S., Marzolla, M.: "Simulation Modeling of UML Software Architectures," TR SAH/43, 26 Feb 2003, European Simulation Muticonference 2003, Nottingham, UK, June 2003.

7. Bass, L., Clements, P., Kazman, R.: “Software Architecture in Practice,” Addison-Wesley, 1998, ISBN 0-201-19930-0.

8. Clements, P.: "Formal methods in describing architectures," In Proc. Workshop on Formal Methods in Software Architecture, Monterrey, California, USA, 1995.

9. Clements, C., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: “Documenting Software Architectures: Views and Beyond,” Addison-Wesley, 2003, ISBN 0-201-70372-6.

10. Clements, P., Kazman, R., Klein, M.: “Evaluating Software Architectures: Methods and Case Studies,” Addison-Wesley, 2002, ISBN 0-201-70482-X.

11. Egyed, A. and Medvidovic, N. "Consistent Architectural Refinement and Evolution using the Unified Modeling Language," Proceedings of the 1st Workshop on Describing Software Architecture with UML, co-located with ICSE 2001, Toronto, Canada, May 2001, pp. 83-87.

12. Egyed, A. and Medvidovic, N. "Extending Architectural Representation in UML with View Integration," Proceedings of the 2nd International Conference on the Unified

Modeling Language (UML), Fort Collins, CO, October 1999, pp. 2-16 (44 papers out of 166).

13. Egyed, A. "Integrating Architectural Views in UML," Technical Report USCCSE-99-514, Center for Software Engineering, University of Southern California, Los Angeles, CA 90089-0781, April 1999 (Qualifying Report as part of Doctoral Program).

14. Egyed, A. and Kruchten, P., "Rose/Architect: A Tool to Visualize Architecture," Proceedings of the 32nd Hawaii International Conference on System Sciences (HICSS), Maui, HI, January 1999.

15. Fuxman, A.: "A survey of architecture description languages".

16. Gacek, C., A. Abd-Allah, B.K. Clark, and B.W. Boehm, "On the Definition of Software System Architecture," in Proceedings of the First International Workshop on Architectures for Software Systems - In Cooperation with the 17th International Conference on Software Engineering, D. Garlan (ed.), Seattle, WA, 24-25 April 1995, pp. 85-95.

17. Garlan, D: “Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events,” in Marco Bernando and Paola Inverardi (Eds.) Formal Methods for Software Architectures, Third International School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures, SFM 2003, Bertinoro, Italy, September 2003, Advanced Lectures, pp. 1-24.

18. Garlan, D., Monroe, R., Wile, D.: "Acme: An Architecture Description Interchange Language," Proceedings of CASCON 97, Toronto, Ontario, November 1997, pp. 169-183.

19. Garlan, D., Shaw, M.: "An Introduction to Software Architecture," In V. Ambriola and G. Tortora (ed.), Advances in Software Engineering and Knowledge Engineering, Series on Software Engineering and Knowledge Engineering, Vol 2, World Scientific Publishing Company, Singapore, pp. 1-39, 1993.

20. Garland, J., Anthony, R.: “Large-Scale Software Architecture: A Practical Guide using UML,” John Wiley & Sons, Ltd, 2003, ISBN 0 470 84849 9.

21. Hillard, R.: "Using the UML for Architectural Description," Proceedings of <<UML>> '99, Lecture Notes in Computer Science, volume 1723, Springer.

22. Hoare, C.: “Communicating Sequential processes,” Prentice-Hall International, 1985, ISBN 0-13-153271-5.

23. Hofmeister, C., Nord, R., Soni, D.: “Applied Software Architecture,” Addison-Wesley, 2000, ISBN 0-201-32571-3.

24. Hofmeister, C., Nord, R., Soni, D.: "Describing software architecture with UML," Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1), Pages: 145 - 160, 1999.

25. IEEE 1471/D4.1: "Draft Recommended Practice for Architectural Description".

26. Inverardi, P., Wolf, A.: "Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model," IEEE Transactions on Software Engineering, Volume 21 , Issue 4 (April 1995) Special issue on software architecture, Pages: 373 - 386, 1995.

27. Kandé, M., Strohmeier, A.: "Towards a UML Profile for Software Architecture Descriptions," In A. Evans, S. Kent, and B. Selic, editors, Proceedings of <> 2000 The Unified Modeling Language. 3rd International Conference, pages 513-527, 2000. Springer, Lecture Notes in Computer Science Vol. 1939.

28. Kernighan, B.: "PIC — A Graphics Language for Typesetting User Manual," Computing Science Technical Report No. 116, AT&T Bell Laboratories, Murray Hill, New Jersey, May 1991.

29. Kogut, P., Clements, C.: "Features of Architecture Description Languages," Software Technology Conference, Salt Lake City, April 1995.

30. Kruchten P.: "Architectural Blueprints – the '4+1' View Model of Software Architecture," IEEE Software 12,6 (Nov 1995) pp42- 50.

31. Luckham, D., Kenney, J., Augustin, L., Ver, J., Bryan, D., Mann, W.: "Specification and Analysis of System Architecture Using Rapide," IEEE Transactions on Software Engineering, Volume 21 , Issue 4 (April 1995), Special issue on software architecture, Pages: 336 - 355.

32. Martin, R: "UML Tutorial: Finite State Machines," Engineering Notebook Column, C++ Report, 1998.

33. Medvidovic, N., Taylor, R.: "A framework for classifying and comparing architecture description languages," Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering, Zurich, Switzerland, Pages: 60 - 76, 1997.

34. Medvidovic, N., Rosenblum, D., Taylor, R.: "A Language and Environment for Architecture-Based Software Development and Evolution." In Proceedings of the 21st International Conference on Software Engineering (ICSE'99), pages 44-53, Los Angeles, CA, May 16-22, 1999.

35. Medvidovic, N., Rosenblum, D.: "Assessing the Suitability of a Standard Design Method for Modeling Software Architectures." In Proceedings of the First Working IFIP

Conference on Software Architecture (WICSA1), pages 161-182, San Antonio, TX, February 22-24, 1999.

36. Medvidovic, N., Taylor, R.: "Exploiting Architectural Style to Develop a Family of Applications," IEE Proceedings – Software Engineering, vol. 144, no. 5-6, pages 237-248 (October-December 1997).

37. Medvidovic, N., Rosenblum, D., Robbins, J., Redmiles, D.: "Modeling Software Architectures in the Unified Modeling Language," ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 11 , Issue 1 (January 2002), Pages: 2 - 57, 2002.

38. Medvidovic, N., Egyed, A., Rosenblum, D.: "Round-Trip Software Engineering Using UML: From Architecture to Design and Back," In Proceedings of the Second International Workshop on Object-Oriented Reengineering (WOOR'99), pages 1-8, Toulouse, France, September 6, 1999.

39. "OMG Unified Modeling Language Specification," Version 1.5, March 2003.

40. Perry, D., Wolf, A.: "Foundations for the study of software architecture," ACM SIGSOFT Software Engineering Notes, Volume 17 , Issue 4 (October 1992), Pages: 40 - 52, 1992.

41. Robbins, J., Medvidovic, N., Redmiles, D., Rosenblum, D.: "Integrating Architecture Description Languages with a Standard Design Method," In Proceedings of the 20th International Conference on Software Engineering (ICSE'98), pages 209-218, Kyoto, Japan, April 19-25, 1998.

42. Shaw, M., DeLine, R., Klein, D., Ross, T., Young, D., Zelesnik, G.: "Abstractions for Software Architecture and Tools to Support Them," IEEE Transactions on Software Engineering, 21(4):314-335, April 1995.

43. Shaw, M., Clements, P. :"A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems," Proc. COMPSAC97, 21st Int'l Computer Software and Applications Conference, August 1997, pp. 6-13.

44. Shaw, M., Garlan, D.: "Formulations and Formalisms in Software Architecture," chap. in Computer Science Today, Lecture Notes in Computer Science Vol. 1000, 1995. Berlin: Springer-Verlag.

45. Shaw, M.: "Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status," ICSE Workshop on Studies of Software Design 1993: 17-32.

46. Shaw, M.: "Some Patterns for Software Architecture," In John M. Vlissides, James O. Coplien, & Norman L. Kerth (Eds.), Pattern Languages of Program Design, Vol 2, (pp. 255-269). Reading, MA: Addison-Wesley, 1996.

47. da Silva, L., de Paula, V.: "Comparative Analysis of Architectural Views Based on UML," Electronic Notes in Theoretical Computer Science, Vol. 65 (4) (2002).

48. Sommerville, I.: "Software Engineering," Addison-Wesley Pub. Co., 1996, ISBN 0-201-39815-X.

49. Stafford, J., Wolf, A.: "Architecture-Based Software Engineering," Technical Report CU-CS-891-99, November 1999.

50. Taylor, R., Medvidovic, N., Anderson, K., Whitehead Jr., E., Robbins, J., Nies, K., Oreizy, P., Dubrow, D.: "A Component- and Message-Based Architectural Style for GUI Software", IEEE Transactions on Software Engineering, June 1996.

51. Vestal, S.: "A Cursory Overview and Comparison of Four Architecture Description Languages," Technical Report, Honeywell Technology Center, 1993.

52. http://uml.tutorials.trireme.com/: UML-A Tutorial, MMI-Trireme International Ltd.

53. http://www.sei.cmu.edu/architecture/definitions.html: web resource on definitions of software architecture.

54. Personal communication with Dr. Sukhamay Kundu, Fall 2003.

Vita

Somo Subro Banerjee has been a graduate student in Louisiana State University, Baton Rouge, Louisiana since January 2002. He is currently working towards his master's degree in System Science. He has completed his Bachelor of Engineering degree in Electrical and Computer Engineering from the National Institute of Technology, Rourkela, India, in 1999. Since then, he has been employed for more than two years in Satyam Computer Services Ltd., a leading software player in India in the capacity of Senior Software Engineer. His research interests are software engineering, software architecture, and web technologies. After completing his master's degree, he plans to pursue a doctoral degree in computer science in University of Southern California, Los Angeles.