Architectural Connectors

Ph. D. Seminar Report

by

Arvind W. Kiwelekar
Roll No: 04405301

under the guidance of

Dr. R. K. Joshi

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Acknowledgements

I would like to thank my guide, Dr. R. K. Joshi for the consistent directions he has fed into my work.

Arvind W. Kiwelekar

Contents

Abstract

Software Architecture is an offshoot of software engineering discipline. Software Architecture provides a conceptual framework for gaining control over the complexity of ever increasing size of a software system. The main objective of this report is to understand various abstractions provided by the software architecture and to create a basis for understanding semantics of architectural connectors.

The report proceeds by defining software architecture and placing the software architecture in the context of software engineering. *Architectural Description Languages* (ADL), *Architectural Styles*, *Unit operations*, *Components* and *Connectors* are the examples of different abstractions that frequently occurs in the field of software architecture. These abstractions are studied in depth and reported.

Further, the report discusses semantics issues for architectural connectors. Semantics of architectural connectors like *Client-Server Connectors*, *Shared Variable*, and *Pipes* is given in three different formalisms i.e. First Order Logic, Process Algebra and Category Theory. Finally, the report compares these techniques based on the type of reasoning supported by them.

Chapter 1

Introduction

1.1 Introduction to Software Architecture

Software Architecture is an offshoot of software engineering discipline. As it happens with every nascent disciplines, software architecture is defined in numerous ways. Each one of this definition highlights certain aspect of the software architecture.

One of the widely accepted definition of the software architecture is due to Bass and Kazman[3]. They define architecture of a software system as a structure of software depicted in terms of components, externally visible properties of components and the relationship among the components. This definition highlights the structural aspect of a software system.

Another, equally accepted definition [14] of software architecture enumerates the design issues for specifying overall system structure that one must address at this level of abstraction. These issues are gross organization, global control structure, protocols for communication and synchronization, assignment of functionality to design elements, physical distribution of design elements, composition of design elements, scaling and performance, and selection among design alternatives.

Our view of Software architecture is that of a discipline providing mechanisms to deal with ever increasing size and complexity of a software system. Software architecture manages the complexity of a software system by separating the computations performed by components from the interactions in which they involve with other components. Hence, *architecture of a software system is depicted by specifying the constituent components and their*

interactions with each other. Although, this definition misses other structural and design aspects of the software system it gives equal importance to computation and interaction.

To get better insight, software architecture is often compared with other architectural disciplines. In this respect, software architecture finds many similarities with building architecture. Like building architecture, software architecture is described by multiple views. Like building architecture, software architecture exhibits different architectural styles. Like building architecture, we find a close relationship between architectural style and engineering. Unlike building architecture, for software architecture boundaries between architecture, design, and implementation are not clearly defined.

So, to have a better understanding of what activities comes under software architecture, here, we are placing software architecture in the context of software engineering process. Normally, development cycle of a software project involves phases like requirement analysis, specifying architecture, specifying design, and implementing the software. Details of the activities that comes under each one of these phases are -

- **Requirement Analysis** is concerned with determination of the information, processing and characterizing of that information.
- **Architecture** is concerned with selection of architectural elements, their interactions and specifying the constraints on those elements and their interactions. Software Architecture provides a framework that satisfies the requirement and acts as a basis for the design.
- **Design** is concerned with the modularization, specifying detailed interfaces of the design elements, their algorithms and procedures, and the data types needed to support the architecture. Design must satisfy properties specified in the architecture and the requirement analysis.
- **Implementation** is concerned with the representations of the algorithms and data types. Implementation must satisfy the design, architecture, and requirements.

1.1.1 Elements of Software Architecture

The architecture of a software system is modelled using following design level entities.

- **Components** Components represent the primary computational elements and data stores of a system. Typical examples of component include such things as clients, servers, filters, objects, blackboards and databases. Components may have multiple interfaces, each interface defining a point of interaction between a component and its environment. Components are classified based upon how they are packaged. Packaging of a component also determines their mode of interactions. Following table summarizes common packaging techniques and their mode of interaction.

Type of a Component	**Interaction Supported**
Module	Procedure Call, Data Sharing
Object	Method Invocation
Filter	Data Flow
Process	Message Passing, RPC, Communication Protocol, Synchronization.
Data File	Read, Write
Database	Schema, Query Language

Table 1.1: Types of Components and Interactions Supported by them

- **Connectors** Connectors represent interaction among components. They provide the glue for architectural designs. From the run time perspective, connectors mediate the communication and coordination activities among components. Examples include simple forms of interaction, such as pipes, procedure call, and event broadcast. Connectors may also represent complex interactions, such as client-server protocol, or a SQL Link between a database and an application. Connectors have interfaces that define the roles played by the participants in the interaction. Connectors are further described in detail in the Chapter 3.

- **Systems** System represents graphs of components and connectors. A particular arrangement of components and connectors are defined as a system configuration. In general, systems may be hierarchical. Components and connectors may represent subsystems that have their own internal architecture.

- **Architectural Style** Architectural styles describe the families of system that use the same types of components, types of interactions, structural constraints, and analysis. System built within a single style can be expected to be more compatible than those that mix styles: it may be easier to make them interoperate, and it may be easier to reuse parts within the family. Architectural styles are further described in detail int the chapter 4.

- **Application Oriented Properties** These properties describe the states of a data structure that are of significance to the processing elements manipulating that structure. They can be used for such things as controlling the order of processing, helping to define the effects of a processing element on a data structure and even helping to define operations needed by the processing elements to achieve those effects.

Formulating the process of deriving an architecture for a software system from the requirement is a difficult task. But certain set of operations help an architect to simplify the task of architectural design. These operations are described in the Chapter 4. Normally, valuable time of a Software Architect is spent for giving a concrete form to these conceptual notions. To document these concepts and notions software architect uses Architectural Description Languages (ADL). ADLs are used to describe the architecture of a software system. The ADL document acts as an input for a design process. Architectural description languages are further described in detail in the Chapter 2.

Next section illustrates the elements of software architecture by giving an example.

1.2 Example of a Software Architecture

This example describes a compiler from architectural viewpoint. While describing the architecture of a compiler we are making a distinction between *processing element* and *data element*. Processing element and data element are types of components. We are giving architecture of a compiler in two different architectural styles.

A typical compiler have five phases: lexical analysis, syntactic analysis, semantic analysis, optimization and code generation. Optimization phase

is considered as preferred, but not necessary aspect. So, the architectural elements used in compiler are given in Table 1.2.

Processing Elements
Lexer, Parser, Semantor, Optimizer , Code generator
Data Elements
Characters, Tokens, Phrases, Correlated Phrases,Annotated Phrases, and Object Code
Application level properties
has-all-tokens, has-all-phrases, has-all-correlated-phrases, has-all-optimization-annotations

Table 1.2: Architectural Elements of a Compiler

Normally, a compiler is constructed in *sequential architecture*, when one phase completes its operation it calls the next phase. In this architectural style, components forms the programme module and connectors are procedure calls. As we have seen in the introductory section, architecture can be described in multiple views. Here, we are describing the architecture in two different views. One is the processing view as shown in the Figure 1.1. Second view, called *data view* is given in Figure 1.2.

Sometimes, we require to improve on the performance of a compiler i.e. specifically if compiler is running in a multiprocessor environment. In such an environment compiler is built using parallel architectural style. For a compiler built using parallel architectural style, shared variables are connectors. A Parallel architecture for a compiler is described in the Figure 1.3. This example highlights two facts

1. A given software can be constructed in more than one architectural style.

2. Multiple views are associated with a given software architecture.

1.2.1 Roles of Software Architecture

The benefits that software architecture are

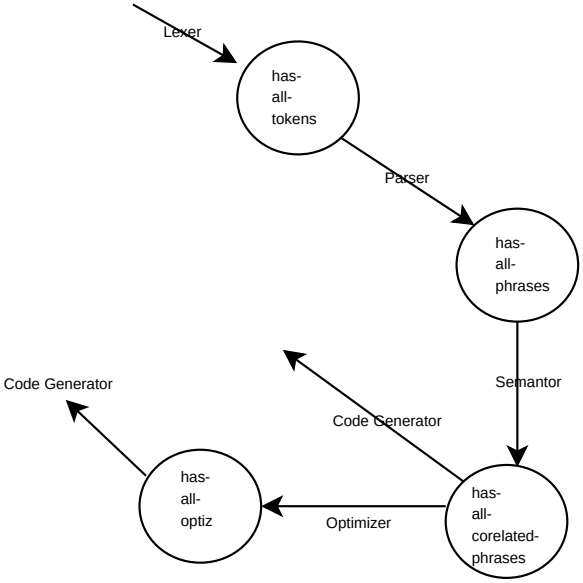

Figure 1.1: Data View of Sequential Compiler Architecture

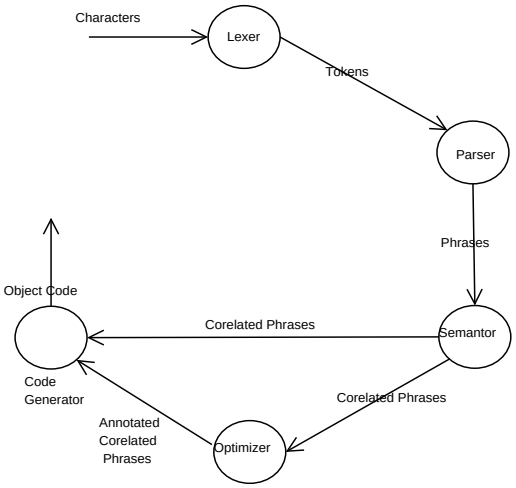

Figure 1.2: Processing View of Sequential Compiler Architecture

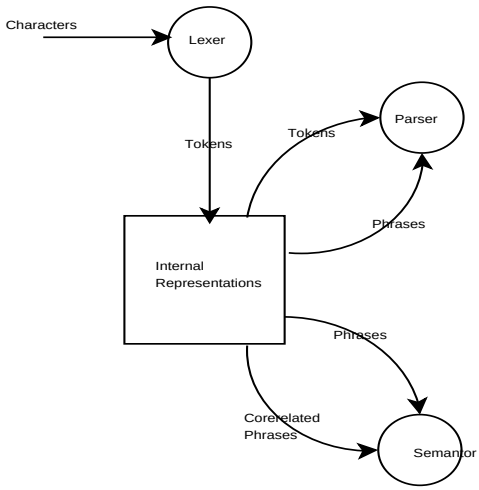

Figure 1.3: Shared Data Structure Compiler Architecture

- **Understanding** Software architecture simplifies our ability to comprehend large systems by presenting them at a level of abstraction at which system's design can be easily understood.
- **Reuse** Architectural design can support reuse in several ways. Architectural styles, Frameworks, Design Patterns, Domain Specific Software Architectures (DSSA) are some mechanism that achieve reuse at architectural level.
- **Construction** An architectural description provides a partial blueprint for development and dependencies between them.
- **Analysis** Architectural descriptions provide new opportunities for analysis, including system consistency checking, conformance to constraints, dependency analysis etc.
- **Communication** An architectural description often serves as a vehicle for communication among stakeholder.

Chapter 2

Architectural Description Languages

Box-line diagrams are normally used to describe the structure of a software system. System modelling using box-line diagram is the simplest way to identify important elements of a software system and their relationship. However, the problem with the approach is that boxes and lines are overloaded with the meaning assigned to them. For example, a line in the diagram represents either a link between two software component, or an interdependence of a software component, or a procedure call. Earlier, Module Interconnection Languages (MIL) are used to describe the inter-relationship among the modules of a software. Also, Interface Description Languages (IDL) are used for the same purpose in object and component oriented programming. MIL and IDL captures relationship between the source code but fails to capture the dynamic relationship among the components of a software. Hence, a different set of tools and techniques are required to define the structure of a software system. **Architectural Description Languages** (ADL) describe the structure of a software system at a level of abstraction that is more closest to the intuition of a system designer. In this chapter, First we are enumerating the characteristics of a generic ADL. Then, we are going to describe how these characteristics are realized in some sample ADLs.

2.1 Elements of an Architectural Description Languages

An ideal Architectural Description language must provide notations to describe architectural components and their interactions at a higher level of design for a sufficiently large-scale software system. Shaw and Garlan [19] propose a framework for characteristics of languages describing the architecture of a software system. Such a language is required to possess following characteristics

- **Composition** An ideal ADL should be able to describe a system as a composition of independent components and connections. Composition capabilities allow us to combine independent architectural elements into a larger system. With this capability- we can divide a complex system hierarchically into smaller manageable subsystems, we can understand components and connectors in isolation from the system, and we separate architecture level concerns from design and implementation level concern.

- **Abstraction** An abstraction suppresses unnecessary details while bringing out important properties. Use of abstraction is a widely used technique for managing complexity. In programming languages, we use records, modules, procedures; as an abstraction for data and set of operations. At architectural level, we need a separate set of abstractions to address architectural design issues. An ideal ADL should have at the minimum *components*, *connectors*, and *system* as a set of abstractions to describe real word components and interactions of components within a system.

- **Reusability** Reusability allows us to develop a new system using existing or predefined components. Mechanisms to realize reusability is provided at all level of software evolution i.e. from coding to design. In an ideal ADL, it should be possible to reuse components, connectors, and architectural patterns predefined in different architectural setting.

- **Configuration** Architectural descriptions should localize the description of system structure, independently of the elements being structured. They should also support dynamic reconfiguration. This will

permit us to understand and change the architectural structure without having to examine each of the systems individual components.

- **Heterogeneity** An ideal ADL should able to inter-operate with other ADLs.

Table 2.1 gives examples of few ADLs along with their distinguished characteristics.

ADL	**Purpose**
Wright[2]	Specification and analysis of interaction between architectural component.
Aesop[8]	Supports use of Architectural style.
Adage [5]	Supports the description of architectural frameworks for Avionics applications.
C2[16]	Supports the description of user interface systems using a message-based style.
Rapide[10]	Allows architectural designs to be simulated
SADL[9]	provides a formal basis for architectural refinement.

Table 2.1: **Examples of ADL**

Next section describes few examples of ADLs that are widely used for describing the architecture of a software system. The objective of reviewing these ADLs is to know how above mentioned characteristics are supported by these different ADLs

2.2 C2SADL : A domain Specific Architectural Description Language

The C2SADL is an example of a domain specific architectural description language. The C2SADL allows us to model the architecture of a software

system that uses *C2 architectural style*. Most of the GUI and distributed applications are structured around C2 style.

Here, we are going to describe essential ingredients of C2 style [16]. In a C2-style architecture, software connectors transmit messages between components, while components maintain state, perform operations and exchange messages with other components via two interfaces. These two interfaces are named as top and bottom. Each interface consists of a set of messages that may be sent and set of messages that may be received. A component interface may be attached to at most one connector. A connector may be attached to any number of other components and connectors. 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. Request messages may only be sent "upward" through the architecture, and notification messages only be sent "downward."

The C2 style further demands that components communicate with each other only through message passing, never-though shared memory. Also, C2 style requires that notifications sent from a component correspond to its operations, rather than the needs of any components that receive those notifications. This constraint on notifications helps substrate independence, which is ability to reuse a C2 component. The C2 style does not make any assumptions about the languages in which components or connectors are implemented.

2.2.1 Component Specification

C2SADL allows to define two separate interfaces for a component. One interface is called *top* and another one is called *bottom*. A component will have two main sections i.e. *Interface* and *Behavior*. Interface section is divided further into two sections and i.e. *Out* and *In*. A example of a component definition is given in the Figure 2.1

2.2.2 Connector Specification

Connectors bind components together into a C2 architecture. They may be connected to any number of components as well as other connectors. A connectors primary responsibility is the routing and broadcast of messages. A secondary responsibility is message filtering.

Component `MeetingInitiator` **is**
Interface top_domain is
out
`GetPrefSet();`
`GetExclSet();`
`RemoveExclSet();`

in
`PrefSet();`
`ExclSet();`
`EquipReqts();`

behavior
received_messages `PrefSet` **may_generate** `RemoveExclSet`

Figure 2.1: **Component Definition**

Connectors may provide a number of filtering and broadcast policies for messages, such as -

- **No Filtering**: Each message is sent to all connected component on the relevant side of the connector i.e. bottom for notification and top for requests.
- **Notification Filtering** Each notification is sent to only those components that registered for it.
- **Prioritized** The connector defines a priority ranking over its components, based on a set of evaluation criteria specified by the software designer during the construction of the architecture. This connector then sends a notification to each component in order of priority until a termination condition has been met.
- **Message Sink** The connector ignores each message sent to it. This is useful for isolating subsystems of an architecture as well as incrementally adding components to an existing architecture.

A connector has an upper and lower domain, defined by the components and connectors attached to it. The syntax for connector specification is described in the Figure 2.2.4.

Architecture `MeetingSchedular` is
Conceptual_Components
`Attendee, ImportantAttendee, MeetingInitiator`
Connectors
`MainConn` is **message_filter no_filtering**;
`AttConn` is **message_filter no_filtering**;

Architectural_topology
Connector `AttConn` **connections**
top_ports `Attendee`;
bottom_ports `MainConn`;
Connector `MainConn` **connections**
top_ports `AttConn; ImportantAttconn`;
bottom_ports `MeetingInitiator`;

System `MeetingSchedular_1` **is**
architecture `MeetingSchedular` **with**
Attendee instance Att_1, Att_2;

Figure 2.2: **Architecture and System Definition in C2SADL**

2.2.3 Architecture Specification

Architecture specification in C2SADL means specifying component connector topology. It identifies components, connectors and their interconnection. The syntax for architecture specification is described on the fig 2.2.4.

2.2.4 System Specification

An instance of the architecture is specified by instantiating the components. The syntax for defining system in C2SADL is given in Figure 2.2.4

2.3 Wright: General Purpose Architectural Description Language

Wright is a general purpose ADL supporting specification and Analysis of interaction between architectural components. Unlike C2, Wright does not enforce the rules of a particular style, but is applicable to multiple styles. However, it still places certain topological constraints on architecture. For example, as in C2, two components cannot be directly connected, but must communicate through a connector. On the other hand, unlike C2, Wright disallows two connectors from being directly attached to one another. Here, we are explaining the syntax of Wright by giving an example of a architectural description in Wright.

Figure 2.3 shows how a simple client-server system would be described using Wright[19]. An architecture in Wright is described in three parts[11]:

- Component and Connector Types.
- Component and Connector Instances.
- Configuration of Component and Connector Instances.

The First part of the description defines component and connector types. A component type is described as a set of ports and component-spec that specifies its function. Each port defines a logical point of interaction between the component and its environment. Thus port allows to define multiple interfaces for a component.

A connector type is defined by a set of roles and glue specification. The roles describe the expected local behavior of each of the interacting parties. Roles are interfaces of a connectors. The glue specification describes how the activities of the client and server roles are coordinated.

The second part of the system definition is a set of component and connector instances. These specify the actual entities that will appear in the configuration.

In the third part of the system definition, component and connector instances are combined by prescribing which component ports are attached as which connector roles.

System `SimpleExample`
Component `Server` =
Port `provide`;
Component `Client` =
Port `request`;
Connector `C-S-Connector` =
Role `client`;
Role `server`;

Instances
`s:Server, c:Clinet, cs:C-S-Connector;`

Attachments
s.provide **as** cs.server;
c.request **as** cs.client;
end `SimpleExample`

Figure 2.3: **Architecture Definition in Wright**

2.4 Acme: As a Architectural Description Interchange Language

Acme is an architectural description language supporting interchange of architectural description written in different ADL. Acme supports seven basic entities using which we can built a architectural description of a software system. These seven entities are components, connectors, systems, ports, roles, representations, and rep-maps [9].

- **Components and Ports:** *Components* represent the primary computational elements and data stores of a system. They correspond to the boxes in box-line descriptions of a software system. Examples of a components are clients, server, filters, objects, and databases.

  Components' interfaces are defined by a set of *ports*. Each *port* identifies a point of interaction between the component and its environment. A component may provide multiple interfaces by using different types of ports. A port can represent an interface as simple as a single procedure

System *simple_cs* = {
Component client = {Port send-request }
Component server = {Port receive-request }
Connector rpc = {Roles { caller, callee } }
Attachments: { client.send-request to rpc.caller;
server.receive-request to rpc.callee }
}

Figure 2.4: **Simple Client-Server System in Acme**

signature, or more complex interfaces, such as collection of procedure calls that must be invoked in certain specified orders.

- **Connectors and Roles**: Connectors represent interactions among components. Computationally speaking, connectors mediate the communication and coordination activities among components. Informally they provide the glue for architectural designs, they correspond to lines in box-line diagrams. Examples are pipes, procedure call, method invocation, client-server protocol, and SQL link between database and application.

  Connectors also have interfaces that are defined by a set of roles. Each role of connector defines a participant of the interaction represented by the connector. Binary connectors have two roles such as caller, and callee roles of an RPC connector, reading and writing role of a pipe. Some connectors may have more than two roles. For example, event broadcaster. Figure 2.4 gives and example to explain syntax for writing and architectural description in Acem using these primitives.

- **System**: System represents configurations of components and connectors.

- **Representation and rep-maps** Acme supports the hierarchical description of architectures. Specifically, any component or connector can be represented by one or more detailed, low-level descriptions. Each such description is termed as representation in Acme.

  When a component or connector has an architectural representation there must be some way to indicate correspondence between the inter-

nal system representation and the external interface of a component or connector that is being represented. A rep-map defines this correspondence.

Finally to summarize we are mapping the characteristics of an ideal ADL to the mechanisms used in Acme to realize those characteristics. Table 2.2 provides this mapping.

ADL Characteristics	**Mechanism in Acme**
Composition	Representation and Rep-Maps
Abstraction	System, Representations, Component, Connectors, Ports and Roles
Reusability	Templates
Configuration	System and Attachments
Heterogeneity	Properties
Analysis	Open Semantic Framework

Table 2.2: **Examples of ADL**

2.5 xArch: Extending Existing Language to Define a New ADL

ADLs that have been discussed so far are the ADLs that have designed from the scratch. Another approach of defining a new ADL is to extend existing language. XArch[6] is an extension of XML. xArch is basically a set of XML schemas that allows us to define architecture of a system. xArch makes an explicit distinction between the *run-time architecture* of a system and *design time architecture* of system. At design time, precise information about components and connectors are not known. At design time, we are interested in approximate behavior of components that can be well described in text. At run time, we are interested in knowing the state of a component i.e. whether a component is started, running or blocked. Hence, xArch defines two sets of schemas- one called *Structure and Type schema* and another one is called *Instances schema*. Also, xADL provides a placeholder in Structure and Type schema to provide an *implementation mapping*.

2.6 Some other Approaches

ArchJava [1] is another example of ADL that is based on the principle of extending existing languages. ArchJava is an extension of Java programming language. ArchJava aims at preserving *communication integrity* of components i.e. at run time, components should engage in only those interactions that has been specified in the architecture of a system.

Also, we can use design specification languages like UML [11] to model software architecture.

2.7 Comparison of ADLs from Connector point of View

Table 2.7 gives the comparison of ADLs that we have discussed in this chapter on the basis of how connectors are supported in those language.

Table 2.3: **Comparison of ADLs from Connector Point View**

Type of ADL	**Support for Connectors**
C2SADL	C2SADL being an domain specific ADL supports only message passing connectors. No other connector types are supported.
Wright	Wright is a general purpose ADL. Supports to define connectors in terms of roles and glue specification. Type checking in terms of definition/use is supported. Composition of connectors are not allowed . No support of in-built connectors.
Acme	Acme is an interchange language. Connectors are defined in terms of glue/role specification. Definition/use type checking is supported. Connector composition and in-built connectors are not supported. To facilitate interchange of architectural specification among multiple languages, Acme annotates connectors with application specific properties.
xArch	Supports to define connectors in terms of roles and glue specification.

Chapter 3

Architectural Connectors

Software Architecture treats connectors as a first class entities. Earlier techniques for specifying software structure like Module Interconnection Languages and Interface Definition languages (IDL) emphasizes components as a focal point of system development. Techniques like Module Interconnection Languages and IDL does not differentiate between *implementation* relationship and *interaction* relationship. This chapter first gives reasons for why one should treat connectors on an equal footing as that of components. Second section, of this chapter describes types of connectors. Third sections describes an example of ADL that supports in-built connectors to facilitate architectural specification. The main objective of this chapter is create the basis for understanding the semantics of architectural connectors to be discussed in the Chapter 5

3.1 Introduction

In software architecture, components are the primary computational entity. Components are realized at programming level by different mechanisms like-objects, databases, files, processes etc. Software architecture is basically intended to model the interactions among different components. Software architecture derives the behavior of a software system from the behavior of individual components and how they interact with each other. Interaction is the central focus of software architecture. Connector is an abstraction used for mediating and regulating interaction among the component. At programming level, connectors may be realized by simple procedure call. Sometimes,

to realize a connector at programming level more elaborate support from operating system is required. Operating system provides mechanisms like pipes, events, streams, and communication protocols to realize a complex connectors. There are many similarities between the working of a component and connector. Like a component connectors to can be composed. Like components, connectors is also an abstraction that must realized differently at programming level. Next section gives some of the reasons behind treating connectors as a first class entity.

3.1.1 Why Connectors are First Class entities?

In order to understand rationale behind treating connectors as a first class entity, one must appreciate the distinction between implementation and interaction relationship.

First point of distinction lies in the mechanisms used to realize *implementation* and *interaction* relationship at programming level. Normally, implementation relationship is realized by procedure call. More elaborate mechanisms like pipes, protocols of communications, streams and events are required to realize interaction.

Second point of distinction is the *consistency check* performed on them. Consistency check performed on implementation relationship is of the type that whether use of a procedure call is consistent with its definition. For a connector, consistency check performed is of the type that whether the realization is guaranteing the specified protocol of communication. For example, whether a server is initialized first in a client-server type of interaction or not.

Completeness criteria for implementation and interaction is also quite different. Completeness criteria for implementation relationship means that checking of every facility required by a module is provided by some other module or not. Similarly, with respect to interaction relationships, the completeness of a system is concerned with whether the assumptions of each component about the rest of the system have been met and whether all participants are present in the specified interaction.

Hence, Architectural Definition Languages (ADL) treats connectors and components alike. Additionally, following are some more reasons for having a separate mechanism for describing connectors in ADL.

- Connectors may be quite sophisticated, requiring elaborate and com-

plex specifications.

- Definition of connectors should be localized.
- Connectors are potentially abstract. They may be parametrizable.
- Connectors may require distributed system support. To provide reuse of connectors.

3.1.2 Types of Architectural Connectors

Mehta and Medvidovic [12] has observed that architectural connectors are required to realize four types of basic services i.e. communication, conversion, coordination and facilitation. They classified architectural connectors into several categories depending upon how the basic services are realized by connectors.

Types of architectural connectors identified by them are- Procedure Call, Event, Stream, Distributor, Data Access, Arbitrator, Adaptor, and Linkage. Now, we are briefly characterizing these connectors.

- **Procedure Call:** Procedure call connector model the flow of control through various invocation techniques (coordination), and perform transfer of data among the interacting components through the use of parameters (communication). Examples of PC connectors include functions, procedure, object oriented methods, callback invocation, operating system call. Higher order connectors such as RPC can also be composed "on top of" a procedure call by adding *facilitation* and *conversion* services.
- **Event:** Event connectors are similar to procedure call connectors in that they model the flow of control among components (communication). In this case, the flow is precipitated with an event. Messages containing a description of the event can be generated upon the occurrence of a single event or a specific pattern of events. The contents of an event message can be structured to contain information about the event and other application specification information (communication). An example of this connector type are GUI events. Some events like page fault are caused by hardware.

- **Stream:** Streams are used perform transfers of large amount of data between autonomous processes (communication). Streams are also used in client-server systems with data transfer protocols to deliver results of computations. Streams may provide unidirectional or bidirectional data transfer. Examples of stream connectors are Unix Pipes, TCP/UDP communications.

- **Distributor:** Distributed systems require identification of component locations and interaction paths to them based on symbolic names. Distributor connector perform the identification of interaction paths and subsequent routing of communication and coordination information among components along these paths (facilitation) Examples are Domain Name Service.

- **Data Access:** Data Access connectors allow components to access data maintained by a data store component. (communication) Examples of persistent data access include database query mechanisms , such as SQL, and file I/O.

- **Arbitrator** When component are aware of the presence of other components but can not make assumptions about their needs and state, arbitrators streamline operation and resolve any conflicts (facilitation), and redirect the flow of control(coordination. For example, multithreaded systems that require shared memory access use concurrency control to guarantee consistency and atomicity of operations.

- **Adaptor** Adaptor connectors provide facilities to support interaction between components that have not been designed to inter-operate. Adaptors involve matching communication policies and interaction protocols among components (conversion) These connectors are necessary for inter-operation of components in heterogeneous enviornments.bv

- **Linkage**: Linkage connectors are used to die the system components together and hold them in such a state during their operation and interaction. Linkage connectors enable the establishment of ducts, identified as primitives of connectors that form the channels for communication and coordination which are then used by more functional connectors to enforce interaction semantics. Examples of linkage connectors are C export mechanism and Java dynamic class loader.

3.2 UniCon: Language for Universal Connector

UniCon [18] is the Architectural Description Language that supports built-in connector types. Like all other ADLs, UniCon has a mechanism to define components at the same time it supports some predefined components. In UniCon, components are composable. Composition of connectors is not allowed at present.

In UniCon, connectors define the protocols and mechanics of interaction together with any additional mechanisms required to carry out the interaction: auxiliary data structures, initialization, initialization routines, and so on. The connector definition is also the location for specifications of required behavior such as interchange representations and the internal manifestation of the connector in the code of a component. At present all connectors are primitives.

The protocol defines the allowable interactions among a collection of components and provides guarantees about those interactions. To do this it defines roles or the responsibilities of various parties that set requirements for the players of components whose interactions are to be governed by the connector. The author of the component is responsible for ensuring that these responsibilities are satisfied by the implementation. The protocol must include:

- The connector type.
- Assertions that constrain the entire connector; these are the commitments about the interaction that the protocol supports.
- The roles that participate in the protocol; each consist of a name and the type and optional attributes like signature, functional specifications or constraints on their use.

A connector type expresses the designers intention about the general class of connection to be provided by the connector, it restricts the numbers, types and specifications of the Roles provided by the connector. In particular, some roles may require players, some may be optional but constrained if present and some may be restricted to match certain player types.

The roles are the visible semantic units through which the connector mediates the interaction among components. Their types are primitive typing

units. They are used to identify the players that must cooperate in a successful interaction. Roles identify the kinds of interactions a connector can establish- the kinds of components it can work with and the player types it can handle. When a role appear in a protocol, it must specify a name and role type and may optionally specify other attributes; some of these attributes may be required in particular instances.

At present, only primitive implementations of connectors are supported. The in-built connectors supported are Pipe, FileIO, ProcedureCall, DataAccess, PLBundler, RemoteProcedureCall, and RTScheduler. These connectors are summerized in the Table 3.2

Connector Type	**Roles Types and the Players Supported**
Pipe	Source (accepts StreamOut of Filter, ReadNext of SeqFile) Sink(accepts StreamIn of filter, WriteNext of SeqFile)
FileIO	Reader(accepts ReadFile of Module) Readee(accepts ReadNext of SeqFile) Write(accepts WriteFile of Module) Writee(accepts WriteFile of SeqFile)
ProcedureCall	Definer (accepts RoutineDef of Computation or Module) Caller (accepts RoutineCall of Computation or Module)
DataAccess	Definer(accepts RoutineDef of Computation or Module) User(accepts GlobalDataUse of SharedData, Computation or Module)
PLBundler	Participant(accepts PLBundle, RoutineDEf, RoutineCall, GlobalDataUse, GlobalDataDef of Computation, Module or SharedData)
RemoteProcCall	Definer(accepts RPCDef of Process or SchedProcess) Caller (accepts RPCCall of PRocess or SchedProcess)

Table 3.1: In-built Connectors Supported in UniCon

3.3 Higher Order Connectors

The types of connectors that we have seen so far are the most primitive types of connectors. A more complex type of connector can be constructed from predefined primitive connectors. Higher order connectors are those connectors which takes primitive connectors as there argument and returns another connector as a result. Garlan []has identified the different ways to construct a higher order connectors. *Bundling*, *Monitoring*, *Confirmation*, *Security*, and *Compression* are some of the methods that are commonly used to form higher order connectors. Component Adaptation is widely used technique in component based software development. Higher order connectors is giving an alternative solution to address the mismatch of component i.e. connector adaptation.

Chapter 4

Architectural Design

Designing an architecture for a software system requires ingenuity. But having a knowledge of *Architectural styles* and set of *Unit Operations* helps ones to make it more simpler. Architectural style gives us an insight regarding type of components and interactions that will be involved in the system to be designed. Unit Operations help us to gain control over the complexity of the system. First Section of this chapter discusses architectural styles. Second Section elaborates on how to apply unit operations.

4.1 Architectural Styles

An architectural style found in a software system is defined as a pattern of components and connectors that occur repeatedly. Usually an architectural style has similar type of components and connectors. They exhibit a particular way of interaction among the components. They are not a complete software system/subsyetm. Architectural styles are intentionally ambiguous on the number of components and connectors present in it. Architectural styles puts some constraints on components and connectors. Most of the ADL supports a mechanism to define architectural styles.

Shaw and Garlan [19] has catalogued such architectural styles that frequently occur in software system design. Although, this is not a complete catalogue it essentially defines what an architectural style is and how it should be used in a system design. Following sections explains some of these architectural styles.

4.1.1 Pipes and Filters

In this type of architectural style, filters are components. Filters have a set of input and set of outputs. The output of a filter is produced by doing certain kind of transformation on the input data. The pipes are connectors, they forms the conduits for data from one filter to another. Pipes transfers the data without doing any kind of transformation.

The main invariants of this style are- 1. filters do not share any state with each other or they are completely independent entities, 2. they do not know the identity of their upstream and downstream filters.

The best known example of this style is programs written in Unix shell.

The advantages of this style are- first, they allow the designer to understand the overall input/output behavior of a system as a simple composition of the behaviors of the individual filters; second, they support reuse; third systems are easy to maintain. Disadvantage is pipe and filter style often lead to batch organization of processing.

4.1.2 Object Invocation

This style is based on data abstraction and object-oriented organization. Data representation and primitive operations are encapsulated in an abstract data type. Here, objects are components and method invocations on objects are connectors. Two main invariants of this style are that object is responsible for preserving the integrity of its representation and representation of object is hidden from other object. The main disadvantage of this style is that in order to interact with another object, first object is required to know the identity of second one.

4.1.3 Event based or Implicit Invocation

In this style of invocation, rather than invoking a method explicitly, a method gets invoked upon occurrence of a particular event. This style of invocation is called implicit invocation or reactive invocation. The components in this style are the set of procedures. Connectors are the events. The main invariant of this style is that announcer of the event does not know which component is going to be affected by the occurrence of the event.

Architectural Styles	**Type of connectors used**
Pipes and Filters	Pipes
Object Invocation	Method Invocation, Procedure Call.
Event Based	Procedure Call, Method Invocation
Layered Systems	Procedure Call, Data Access, Method Invocation
Repositories	Data Access

Table 4.1: Comparison of Architectural Styles from Connector Point of View

4.1.4 Layered Systems

A layered system is organized hierarchically, each providing service to the layer above it. In some layered systems inner layers are hidden from all except the adjacent outer layer. Thus in these systems components implement a virtual machine at some layer in the hierarchy. The connectors are defined by the protocols that determine how the layers will interact.

4.1.5 Repositories

In a repository style, there are two quite distinct kinds of components: a central data structure represents the current state and a collection of independent components operate on the central data store. Interactions between the repository and its external components vary significantly among systems.

The choice of a control discipline leads to two major subcategories. If the types of transactions in an input stream trigger selection of processes to execute, the repository can be traditional databases. On the other hand, if the current state of the central data structure is the main trigger for selecting processes to execute, the repository can be a blackboard.

Blackboard systems have traditionally been used for applications requiring complex interpretations of signal processing.

Following table summarizes architectural styles from connector point of view. 4.1

4.2 Unit Operations

A software system exhibits two types of quality attributes. One set of characteristics are observable during system execution time like response time, bandwidth, and throughput. These are known as *functional properties*. Second set of quality attributes are not observable during system execution time like modifiability, portability etc. These are known as *non-functional properties*. Unit operations [3] are the set of operations that guides us on how to organize the software when particular non-functional attribute is a design goal.

4.2.1 Separation

Separation places a distinct piece of functionality into a distinct component that has a well defined interface to the rest of the world. It is the most primitive and most common tool of a software architect. Separation isolates a portion of a system's functionality. The motivation for determining what portion of a systems' functionality to isolate comes from a desire to achieve a set of quality attributes. For example, one might separate functionality for performance or ease of creation.

Separation may also be used to ensure that changes to the external environment do not affect a component, and changes to the component do not affect the environment, as long as interface is unchanged. Thus, the operation of separation aids both modifiability.

Examples of separation are found in data-flow architectures, compilers, and user management systems.

4.2.2 Uniform Decomposition

Decomposition is the operation of separating a large system component into two or more smaller ones. Uniform decomposition is a restriction of this operation, limiting the composition mechanisms to a small, uniform set. Having uniform mechanism eases integration of components and scaling of the system as a whole. Two commonly used decomposition mechanism are Part-whole and Is-a relationship. Examples are commonly found in the object oriented system.

4.2.3 Replication

Replication is the operation of duplicating a component within an architecture. This technique is used to enhance reliability and performance. This unit operation is used in hardware as well as in software. When components are replicated it requires the simultaneous failure of more than one component to make the system as whole fail. As the amount of replication in a system increases, the available work can be spread among more of the systems' components, thus increasing throughput.

4.2.4 Abstraction

Abstraction is the operation of creating a virtual machine. A virtual machine is a component whose function is to hide its underlying implementation. Virtual machines are often complex piece of software to create, but once created they can be adopted and reused by other software components, thus simplifying their creation and maintainability. Java Virtual Machine is the example of use of this unit operation.

Separation and abstraction are related but are not the same concept. There are many examples of separation for reasons other than to create a set of abstract services, such as load balancing, parallelizing operations, and dividing work among development teams.

4.2.5 Compression

Compression is the operation of removing layers or interfaces that separate system functions and so it is the opposite of separation. These layers may be either software or hardware. When one compresses software, one takes distinct functions and places them together. Compression serves three main purposes- 1. To improve system performance. 2. To circumvent layering when it does not provide needed services. 3 To speed up development.

4.2.6 Resource Sharing

Resource sharing is an operation that encapsulates either data or services and shares them among multiple independent consumers. Typically there is a resource manager that provide the sole access to the resource. Shared resources are costly to build, but ones built they provide numerous advantages

like integrability, modifiability, and portability. The X-Windowing server, for example, is a shared resource that provides an abstraction of the underlying graphics hardware.

Chapter 5

Specifying Semantics of Architectural Connectors

5.1 Introduction

The main intention of formalizing the semantics of architectural connections is to define the notion of connectors more precisely. Various formal techniques are used to give semantics of connectors. We are restricting ourselves to three widely used techniques i.e. 1. Process Algebra, 2. First Order Logic and 3. Category Theory. This chapter discusses how to use of process algebra and First order Logic. The Chapter 6 will discuss use of category theory in formalizing semantics of architectural connector.

5.2 Requirement for specifying Semantics of Architectural Connectors

Techniques used to provide the semantics of connectors must satisfy two different types of requirements. First, such techniques should be able to provide *expressive notations* to specify connectors. Second, such techniques should provide *analytical capabilities* to reason about connectors [2]. Here, is the list of desired properties that are expected from such techniques-

1. **Expressive Requirement**
    - Allow us to specify common cases of architectural connections. For example, pipes, client server interaction etc.

- Allow to specify complex dynamic interactions among components. For example, server must be initialized first before client in client-server interaction.
- Should allow to make fine grained distinction between a given architectural connector. For example, a shared variable which must be initialized before its first use by the initializer, shared variable that must be first initialized by any process before its first use, shared variable which must be initialized first but it does not matter who initializes it, all these are the type of more general shared variable type of connector which can be read and written only.

2. **Analytical Requirement**

   - Should able to understand the behavior independently of specific context in which connectors are used.
   - Should able reason about whether the use of connectors is compatible with its definition.

5.3 Connector Semantics using Process Algebra

Robert Allen and David Garlan [2] first proposed to use Process Algebra to provide the semantics for architectural connectors. Process Algebra is based on notion of Communicating Sequential Processes (CSP). CSP [17] [15] is the widely used technique to model communication protocols. CSP provides a rich set of constructs for describing communicating entities. A subset of CSP is used to model architectural connections. Notations of CSP that are used in providing semantics of connectors are described in the next section.

Definition 5.1 ***Process Notations***

- *Processes and Events : A process describes an entity that can engage in communication events. The simplest process is STOP process that engages in no events. The set of events in which a process can engage is denoted by* αP.

  Events may be primitives or they can have associated data. For example $e?x$ *represents an in input of data* x *by an event* e, *and* $e!x$ *represents*

an output operation of x over channel e The most primary event is $\surd$ used to represent success.

- *Prefixing: A process that engages in event e and then becomes process P is denoted by $e \rightarrow P$*
- *External choice: A process that can behave like P or Q where the choice is made by the environment, is denoted by $P \Box Q$*
- *Internal Choice A process that can behave like P or Q, where the choice is made by the process itself is denoted by $P \sqcap Q$*
- *Named Process: Process names can be associated with a process expression.*
- *Processes can be composed using $\|$ operator. Parallel processes may interact by jointly engaging in events that lie within the intersection of their alphabets.*

Now, we are giving few examples to demonstrate how these notations are used to define architectural connections.

Example 5.1 ***C-S-connector*** $=$
role $Client = (request!x \rightarrow result?y \rightarrow Client) \sqcap \$$
role $Server = (invoke?x \rightarrow return!y \rightarrow Server) \Box \$$
glue $= (Client.request!x \rightarrow Server.invoke?x \rightarrow Server.return!y \rightarrow Client.result!y \rightarrow glue) \Box \$$

The Example 5.1 gives the specification for client-server connector in process algebra. This example identifies tow roles for c-s-connector i.e. Client and Server. Two roles are modelled in terms of process. The behavior of client is modelled by process definition. Client engages in two events i.e. result and request. Similarly process Server models role server, it engages in two events invoke and return. The glue process coordinates the interaction between client and server. This example describes how to model a connector in process algebra.

Example 5.2 $connectorShared\_Data_1 =$
role $User_1 = \ set \rightarrow User_1 \sqcap get \rightarrow User_1 \sqcap \surd$
role $User_2 = \ set \rightarrow User_2 \sqcap get \rightarrow User_2 \sqcap \surd$

glue $= User_1.set \rightarrow glue \Box User_2.set \rightarrow glue \Box User_1.get \rightarrow glue \Box User_2.get \rightarrow glue$

The examples 5.2 and 5.3 demonstrates how CSP notations are used to provide a specification for fine grained connectors. Connector described in Example 5.2 is the most general type of shared variable connector which models a shared variable with two users and they can access it without any initialization. Connector described in Example 5.3 is the special case of Example 5.2 in which a special process called initializer is used to initialize a connector. After initialization user can access data.

Example 5.3 $connectorShared\_Data_2 =$
role $User = \ set \rightarrow User \sqcap get \rightarrow User \sqcap \surd$
role $Initializer =$
$let\ A = set \rightarrow A \sqcap get \rightarrow A \sqcap \surd$
$in\ set \rightarrow A$
glue $= let\ Continue = Initializer.set \rightarrow Continue$
$\Box User.set \rightarrow Continue$
$\Box User.get \rightarrow Continue$
$\Box Initializer.get \rightarrow Continue \Box \surd$
$in Initializer.set \rightarrow Continue$
$\Box User.set \rightarrow Continue \Box \surd$

5.4 Proving Correctness of Software Architecture using First Order Logic

This section defines architectures at two different levels. First level is called the *abstract level* which is more closer to the designer's intuition. The second level is called *concrete level* architecture and it is more closer to the implementation. Concrete level architecture is the realization of the architecture

defined at abstract level. Correctness of software architecture is intended to prove the *completeness assumption*. Completeness assumption means that if an architectural fact is not explicit in the architecture, or deducible from the architecture, then the fact is not intended to be true of the architecture. Completeness assumption is proved in two steps. First by proving for type-level properties that are achieved only once for each pair of architectural style. In the Second step, instance level properties that is proved for every architecture. The steps involved in carrying out correctness proof are enumerated below [13]-

1. Define the theory of architectural style for abstract and concrete architecture in First order logic.

2. Provide following mappings from abstract architecture to concrete architecture.

    - Name Mapping
    - Style Mapping
    - Interpretation Mapping

We are going to explain these steps by taking an example. We are going to define the architecture of compiler at abstract level by using dataflow style and at concrete level by using shared memory style.

5.4.1 Defining Theory of Architectural Style

Definition of a theory of architectural style involves identifying vocabulary associated with a particular style and expressing them in the form of a first order predicates.

For example, Dataflow style vocabulary contains predicates for describing functional components, ports, values associated with ports, dataflow channels, values associated with dataflow channels, and connections of channels to ports. Hence, dataflow style is described using predicates like Function, OutPort, Supplies, InPort, Accepts, Carries, Connects. These predicates are defined in the table 5.4.1

The shared-memory style uses the reading and writing of a variable for intercommunication. Shared-variable communication is modelled using a call site as an interface between a function and the shared variable. A call site

Functions	Predicates
1. **OutPort** : $oport \times function \rightarrow bool$ 2. **Supplies** : $oport \times val \rightarrow bool$ 3. **InPort** : $iport \times function \rightarrow bool$ 4. **Accepts** : $iport \times val \rightarrow bool$ 5. **Carries** : $channel \times val \rightarrow bool$ 6. **Connects** : $channel \times oport \times iport \rightarrow bool$	1. Function(parser, analyzer) 2. OutPort(oast,parser) 3. $\forall v[Supplies(oast, v) \supset ast(v)]$ 4. InPort(iast,analyzer) 5. $\forall v[ast(v) \supset Accepts(iast, v)]$ 6. Channel(ast_channel) 7. $\forall[ast(v) \supset Carries(ast\_channel, v)]$ 8. Connects(ast_channel,oast,iast)

Table 5.1: **Architectural Specification for Compiler using Data Flow Style in First Order logic**

serves as the same purpose as a port in the dataflow style. The name of every different call site must be unique. The shared memory style has the following style specific sorts, variable, Holds, CallSite, Writes, Puts, Reads and Gets. Its definition and use is explained in the table 5.4.1

5.4.2 Defining Name Mapping

An Name Mapping (I_N)associates the objects declared in an abstract architecture with objects declared in a concrete architecture. The name for Dataflow style to Shared Memory style is defined as-

Functions	Predicates
1. **Holds** : $variable \times val \rightarrow bool$ 2. **CallSite** : $site \times function \rightarrow bool$ 3. **Writes** : $wsite \times variable \rightarrow bool$ 4. **Puts** : $wsite \times val \rightarrow bool$ 5. **Reads** : $rsite \times variable \rightarrow bool$ 6. **Gets** : $rsite \times val \rightarrow bool$	1. Function(parser, analyzer) 2. Variable(tree) 3. $\forall v[ast(v) \supset Holds(tree, v)]$ 4. $CallSite(site_1, parser)$ 5. $\forall v[Puts(site_1, v) \supset ast(v)]$ 6. $Writes(parser, tree)$ 7. $CallSite(site_2, analyzer)$ 8. $\forall[ast(v) \supset Gets(site_2, v)]$ 9. Reads(analyzer,tree)

Table 5.2: **Architectural Specification for Compiler using Shared Memory Style in First Order logic**

$$oast \mapsto site_1$$
$$iast \mapsto site_2$$
$$ast\_channel \mapsto tree$$

5.4.3 Defining Style Mapping

A style mapping says how the constructs of the abstract-level style can be implemented in terms of the constructs of the concrete-level style. More specifically, it maps all atomic formulas of the abstract-level theory to formulas of the concrete-level theory. Let I_S denote the style mapping which is

defined as-

$Function(t_1) \rightarrow Function(t_2)$
$Outport(t_1, t_2) \rightarrow CallSite(t_1, t_2) \wedge \exists v Puts(t1, v)$
$Supplies(t_1, t_2) \rightarrow Puts(t_1, t_2)$
$InPort(t_1, t_2) \rightarrow CallSite(t_1, t_2) \wedge \exists v Gets(t_1, v)$
$Accepts(t_1, t_2) \rightarrow Gets(t_1, v)$
$Channel(t_1) \rightarrow Variable(t_1)$
$Carries(t_1, t_2) \rightarrow Holds(t_1, t_2)$
$Connects(t_1, t_2, t_3) \rightarrow Writes(t_2, t_1) \wedge Reads(t_3, t_1)$

5.4.4 Interpretation Mapping

An interpretation mapping is determined from a name mapping I_N and a style I_S as follows: for every predicate P, all terms t_1, t_2,.....t_1, every variable x, and all formulas F and G of the abstract language,

1. $I(P(t_1, t_2, t_3...t_n)) = I_S(P(I_N(t_1),$
   $I_N(t_2), ...I_N(t_n)))$
2. $I(\neg F) = \neg(I(F))$
3. $I(F \wedge G) = I(F) \wedge I(G)$
4. $I(F \vee G) = I(F) \vee I(G)$
5. $I(F \supset G) = I(F) \supset I(G)$
6. $I(\forall x F) = \forall x I(F)$
7. $I(\exists x F) = \exists x I(F)$

Let I_M^D denote the interpretation mapping from theory Θ_D, to theory Θ_M. Both the ground facts and general axioms in Θ_D must be mapped. For example-

$I_M^D(Connects(ast\_channel, oast, iast))$

$= I_S(Connects(I_N(ast\_channel), I_N(oast), I_N(iast)))$

$= I_S(Connects(tree, site_1, site_2))$

$= Writes(site_1, tree) \wedge Reads(site_2, tree)$
which is intended implementation.

Chapter 6

Mathematical Semantics of Architectural Connectors

6.1 Introduction

In this chapter, we are taking a system level perspective of softwares. We are looking softwares as a collection of interconnecting components. In a modular approach to software development, software system is built around existing components. Existing components are augmented whenever necessary while preserving their properties. Here, our main objective is to define collected behavior of a software from the behavior of individual components. To do this we are making use of categorical techniques.

Earlier, categorical techniques are effectively used to provide semantics for various notions that are found in General System Theory [7]. Fiadeiro et.al. adapts those techniques to software system. Our discussion in this chapter is primarily based upon the framework proposed by Fiadeio et. al. in [7]. We have selected three different notions i.e.- *configuration of a software system*, *interconnection of components*, and *architectural connectors*, to emphasize the role of categorical techniques in software system design. Category Theory is all about mathematical structures and mapping between those structures. Hence, Second section of this chapter gives necessary categorical background required to define semantics for *configuration*, *interconnection*, and *connectors*. Component is an essential ingredient of any system. So, in the Third section, we are going to specify design of a component in CommUnity. CommUnity is a design specification language that suits our

requirement. Unlike, other commercial design specification languages (IDL and UML), CommUnity is a very simple design specification language. CommUnity offers a minimal set of features for writing component specification. Hence, essentials of CommUnity language are discussed in the Third section. Here, we must clarify that providing semantics for interconnection, configuration and connectors in the category theory is not dependent on any a particular language for component specification. As said earlier, category theory is all about mathematical structures and mapping between those structures. In the Fourth section, we objectifying components in two different mathematical structures called *signature* and *design*. Fourth section, also defines mapping between these two structures. Finally, Fifth section develops the semantics for configuration, interconnection and design using ideas defined in the earlier sections.

6.2 Category Theory

6.2.1 Basics of Category Theory

Categories originally arose in mathematics out of the need of a formalism to describe properties of different mathematical structures in a unified way. A category itself is a mathematical structure. It is a generalization of mathematical structures like ordered sets, poset, groups, and monoids. A category theory is an abstract structure. A collection of objects, together with a collection of arrows between them. For example, the objects could be geometric figures and arrows could be ways of transforming one into another. The notion of function is one of the most fundamental in mathematics. Category theory is the algebra of function. The principal operation on function is taken to be composition.

An essential to computer programming is the ability to abstract from the real world problem to a machine based representation with which solution is to be computed. During abstraction procedure, our prime concern is not with internal representation involved with the operations to be carried out and how they combine.

Category theory provides just an abstraction, studying objects and arrows between them and the properties and constructs which may be defined in terms of arrows and their composition.

Fundamental to success of such an abstraction is the wealth of information

about the object is embodied in the arrows between them. A category has objects and morphism. We have no immediate access to internal structure of objects. Thus all properties must be expressed in terms of morphisms.

Category theory has role in program specification. Indeed many familiar programming tasks can be described in categorical terms.

Category theory is used as a mathematical tool in the semantics of programming languages.

Here, we are defining categories, and two other universal constructions found in category theory i.e. coproduct and colimit. We are using universal constructions co-product and co-limit in formulating architectural semantics.

Definition 6.1 (Categories) *A category A consists of a set of objects called* $obj(A)$ *and a set of morphisms or arrows called* $Arr(A)$*. The objects are denoted-*

$$A, B, C, ...X, Y, Z$$

and morphism are denoted by

$$f, g, h.....$$

Further

- *Each morphism has a designated domain and codomain in* $Obj(A)$ *when the domain of f is A and the codomain of f is B we write* $f : A \rightarrow B$
- *Given morphism* $f : A \rightarrow B$, $g : B \rightarrow C$ *there is designated composite morphism* $gof : A \rightarrow C$
- *Given any object A there is a designated identity morphism* $1_A = A \rightarrow$
- *The data above is required to satisfy the following.*
  - ***Identity Law*** *If* $f : A \rightarrow B$ *then* $1_B of = f$ *and* $fo1_A = f$
  - ***Associative Law*** *If* $f : A \rightarrow B$, $g : B \rightarrow C$ *and* $h : C \rightarrow D$ *then* $(hog)of = ho(gof) = A \rightarrow D$

Definition 6.2 (Coproduct) *A coproduct of two objects A and B is an object* $A + B$ *together with two arrows* $i_1 : A \rightarrow A + B$ *and* $i_2 : B \rightarrow A + B$*, such that for any object C and a pair of arrows* $f : A \rightarrow C$, $f : B \rightarrow C$,

there is exactly one arrow $[f, g] : A + B \rightarrow C$ *making the following diagram commute.*

In the category SET, the disjoint union $x \bigoplus y$ *is the co-product of* x *and* y*.*

Definition 6.3 (Pushout) *Let* C *be a category and* $f : x \rightarrow y$, $g : x \rightarrow z$ *are the morphisms of* C*. A pushout of* f *and* g *consists of two morphisms* $f' : y \rightarrow w$ *and* $g' : z \rightarrow w$ *such that-*

- *f;f' = g;g'*
- *for any other two morphisms* $f'' : y \rightarrow v$ *and* $g'' : z \rightarrow v$ *such that* $f; f'' = g; g''$*, there is a unique morphism* $k : w \rightarrow v$ *in* C *such that* $f'; j = f''$ *and* $g'k = g''$

In SET, pushouts perform

6.3 Specifying Design of a Component

Fiadeiro et.al. uses CommUnity as a language to specify design of components. Interface Definition Language(IDL) and Unified Modelling Language(UML) are other examples of languages that are used in practice to specify design of components. The choice of CommUnity as a design specification language is guided by its simplicity. CommUnity provides a minimal set of features that are required for component specification. Using the features offered by CommUnity, we are able to capture all kinds of interactions of component to outside world. IDL/UML also allows us to specify internal design details of components. Here, we are giving more importance to interaction of component to outside world than internal details. CommUnity supports separation of interaction from computation. CommUnity allows us to specify component design at two different levels i.e. signature level and another one at design level. In CommUnity methods or functions are called as actions. At signature level, actions are described by number and type of parameters it takes as input and output. At design level, functionalities of actions is specified using sentences in First-Order logic. Following section discusses elements of CommUnity language.

6.3.1 Elements of CommUnity

The two basic elements of CommUnity are Actions and Variables. The generalized structure of component specification in CommUnity is given below-

Component P *is*
out $out(V)$
in $in(V)$
prv $prv(V)$
do
$g \in sh(\Gamma) \quad g[D(g)] : L(g), U(g) \rightarrow R(g)$
$g \in prv(\Gamma) \quad g[D(g)] : L(g), U(g) \rightarrow R(g)$

1. **Variables(V)**: V is the set of variables. Variables can be declared as input, output, or private. Input variables are read from the environment and cannot be modified by the component. Output and private variable are local to the component. loc(V) is used to denote union of output and private variables.

2. **Actions (Γ)** The named action can be declared as shared or private. Private action represent internal computation. Shared action represent possible interaction between the components and enviornment. For each action name ,g, the following attributes are defined

   - Write Frame (D(g)) : $D(g) \subseteq ofloc(V)$ The subset of local variables that gets affected or written by action g. Given a variable $v \in V$ we also denote $D(v) \subseteq \Gamma$ to denote set methods that modifies variable v.
   - Precondition $L(g)$ and $U(g)$ Lower and upper bounds on enabling condition of g.
   - Postcondition $R(g)$ is a post condition of command g.

Example 6.1 (Design Specification of a Buffer in CommUnity)

Component `buffer [t: sort bound: t]` *is*
out $i : t$
in $o : t$
prv $rd :\ bool,\ b :\ list(t)$
do

$sh \ [] \ put : \quad (—b— \ ¡ \ bound) \rightarrow b := b.i$
$prv \ [] \ next : (|b| > 0 \wedge \ \sim rd) \rightarrow o := head(b) \ ||b := tail(b)||rd := true$
$sh \ [] \ get : \ rd \rightarrow rd := false$

This example models a buffer with limited capacity working on FIFO order. We frequently come across this kind of buffer in applications like printer drivers, networking protocols etc. t is a type variable. So, design of component buffer is parametric one. Here; i, o, and {rd, b} are output, input, and private variables respectively. Actions supported by the buffer are put, next, and get. put is a shared action for which ($|b| < bound$) is an enabling condition with b as its write frame. Action next is a private one with ($|b| > 0 \wedge \ \sim rd$) as enabling condition with $\{o, b, rd\}$ in its write frame. For shared action get ($rd == True$) is the enabling condition and $\{rd\}$ as a write frame.

6.4 Design Objects and Morphisms

As seen in the second section, categories are collection of objects and arrows. Normally in any category, objects are mathematical structures and arrows are structure preserving maps. We are using categories to model software as a system. So, software components takes the form of objects in the resultant category. We are assigning different meaning to arrows depending on the context. To model software component as a object in the category, software component should be given some kind of mathematical structure. Here, we are representing software component using two different mathematical structures i.e. *Signatures* and *Designs*. Following sections formally defines these structures-

Definition 6.4 (Signature) *A signature is a tuple* $< V, \Gamma, tv, ta, D >$ *where*

- V *is an S-indexed family of mutually disjoint finite sets.*
- Γ *is a finite set.*
- $tv : V \rightarrow \{out, in, prv\}$ *is a total function.*
- $ta : \Gamma \rightarrow \{sh, prv\}$

- $D : \Gamma \rightarrow 2^{loc(V)}$ *is a Total function.*

Following example illustrate how to represent the component buffer that has been defined in example 3.1

Example 6.2 *The signature of component Buffer is a tuple* $< V, \Gamma, tv, ta, D >$ *with V,* Γ*, tv, ta, and D defined as follow-*

- $V = \{i,\ o,\ rd,\ b\ \}$
- $\Gamma = \{\ put,\ next,\ get\ \}$
- $tv : V \rightarrow \{out, in, prv\}$
  $i \mapsto out$
  $o \mapsto in$
  $rd \mapsto prv$
  $b \mapsto prv$
- $ta : \Gamma \rightarrow \{sh, prv\}$
  $put \mapsto sh$
  $next \mapsto prv$
  $get \mapsto sh$
- $D : \Gamma \rightarrow 2^{loc(V)}$
  $put \mapsto \{b\}$
  $next \mapsto \{o, b, rd\}$
  $get \mapsto \{rd\}$

Definition 6.5 (Design) *A design is a pair* $< \Theta, \Delta >$*, where* $\Theta =< V, \Gamma, tv, ta, D >$ *is a signature and* Δ*, the body of the design is a tuple* $< R, L, U >$ *where:*

- *R assigns to every action* $g \in \Gamma$*, a proposition over* $V \cup D(g)'$
- *L and U assign a proposition over V to every action* $g \in \Gamma$

Following example illustrates the design structure for Component Buffer specified in the example 3.1-

Example 6.3 *The design for Component buffer is a pair* $< \Theta, \Delta >$ *where* $\Theta =< V, \Gamma, tv, ta, D >$ *is same as that of defined in the example 3.3. Here we are defining* Δ

- ***Postconditions*** $R : \Gamma \rightarrow P(V \ \cup \ D(g'))$

  $put \mapsto \{b = b.i\}$
  $next \mapsto \{o = o.head, b = tail, rd = True\}$
  $get \mapsto \{rd = False\}$

- ***Preconditions*** *(L and U):*
  $put \mapsto \{|b| < bound\}$
  $next \mapsto \{|b| > 0 \wedge \sim rd\}$
  $get \mapsto \{rd = True\}$

From these examples, we can note that, signature structure defines the interface of a component. Design structure assigns precondition and post-condition to the actions in the component. This is nothing but specifying components contract. (Design by Contract.) This also defines the object in the category. So, the category that we are using to model the software system will have to different types of objects one is **signature** and another one is **design**.

Now, we are in a position to define arrows interconnecting these objects. We will have two different types of arrows- 1. **Signature Morphism:** Interconnecting signature objects and another 2. **Design Morphism** interconnecting design objets. Informally, purpose of these morphisms is to check whether the contract specified by the component is satisfied in the resultant system or not. Contract checking is performed in two different steps. First, signature morphism allows to identify the corresponding variable and synchronizing actions in the system. Second, design morphism allows to perform a check on preconditions and post-conditions. These morphisms put constraints on mapping so that only valid assignments be defined by the morphisms.

Definition 6.6 (Signature Morphism) *A morphism* $\sigma : \theta_1 \rightarrow \theta_2$ *between signatures* $\theta_1 =< V_1, \Gamma_1, tv_1, ta_1, D_1 > \theta_2 =< V_2, \Gamma_2, tv_2, ta_2, D_1 >$ *is a pair* $< \sigma_{var}, \sigma_{ac} >$ *where-*

1. $\sigma_{var} : V_1 \rightarrow V_2$ *is a total functions satisfying:*

(a) $sort_2(\sigma_{var}(v)) = sort_1(v)$ *for every* $v \in V_1$

(b) $\sigma_{var}(o) \in out(V_2)$ *for every* $o \in out(V_1)$

(c) $\sigma_{var}(i) \in out(V_2) \cup in(V_2)$ *for every* $i \in in(V_1)$

(d) $\sigma_{var}(p) \in prv(V_2)$ *for every* $p \in prv(V_1)$

2. $\sigma_{ac} : \Gamma_2 \rightarrow \Gamma_1$ *is a partial mapping satisfying for every* $g \in \Gamma_2$. $\sigma_{ac}(g)$*is defined:*

(a) *if* $g \in sh(\Gamma_2)$ *then* $\sigma_{ac}(g) \in sh(\Gamma_1)$

(b) *if* $g \in prv(\Gamma_2)$ *then* $\sigma_{ac}(g) \in prv(\Gamma_1)$

(c) $\sigma_{var}(D_1(\sigma_{ac}(g))) \subseteq D_2(g)$

(d) $\sigma_{ac}(D_2(\sigma_v ar(v))) \subseteq D_1(v)$ *for every* $v \in loc(v_1)$

As defined in the above section, signature morphism is a pair of two functions σ_{var} and σ_{ac}.

σ_{var} is a total function from V_1 to V_2. i.e. $Component \rightarrow System$. σ_{var} identifies a variable in the system for each component variable. While performing these assignments rules from 1.a to 1.d be satisfied. $ Rule 1.a enforces type-checking and says that type of the system variable be matched with the type of component variable. $ Rule 1.b says that output variable of component should become the output variable of a system. $ Rule 1.c is performing an assignment of input variable of a component with a system variable which may be either an input or output variable. $ Rule 1.d is performing an assignment of a private variable of a component with private variable of a system. In short, *these rules are telling us how to keep a component in a container.*

σ_{ac} is a partial function from V_2 to V_1 i.e. from a $System \rightarrow Component$. This function has a opposite direction as that of σ_{var}. From the action point of view, system plays a role of synchronizing actions of individual components. A unique action from the set of action of a component is associated for a synchronizing set of a system. But reverse is not true, a component method may participate in more than one synchronizing set of a system. To satisfy this functional property, mapping goes from system to component. The mapping is partial in the sense that out of total actions supported by the system only few will be realized by a component. $Rule 2.a and $ Rule 2.b assigns private methods of a system to private methods of a component and shared methods of a system to shared methods of a component. The

next two rules says that actions of the system in which a component is not involved can not have local variables of the components in its write frame.

Definition 6.7 (Design Morphism) *A morphism* $\sigma : P_1 \rightarrow P_2$ *of designs* $P_1 =< \Theta_1, \Delta_1 >$ *and* $P_2 =< \Theta_2, \Delta_2 >$ *and, consists of a signature morphism* $\sigma : \Theta_1 \rightarrow \Theta_2$ *such that for every* $g \in \Gamma_2$, σ_{ac} *is defined:*

1. $R_2(g) \supset \sigma(R_1(\sigma_{ac}(g)))$
2. $L_2(g) \supset \sigma(L_1(\sigma_{ac}(g)))$
3. $R_2(g) \supset \sigma(U_1(\sigma_{ac}(g)))$

Design morphism helps us to check whether the preconditions and post-condition specified for every action in the component is maintained by the system or not. It puts additional constraints for valid action mapping. Condition 1 reflects the fact that the effects of the actions of the components can only be preserved or made more deterministic in the system. This is because the other components in the system cannot interfere with the transformations that the actions of a given component make on its state. Condition 2 and 3 allow the bounds that the component design specifies for enabling action to be strengthened but not weakened.

6.5 Semantics for Interconnection and Configuration

As seen in the section 3.3 names of the variables are local to the components. If certain named variable appears in more than one component, it does-not mean that the variable's value is shared by both component. If we require such kind of sharing then that has to be made explicit. *Interconnection* is a mechanism that allows us to make such *name bindings* explicit in the design.

Configuration of a system allows us understand the collected behavior of the system from the individual components depending upon how the components are placed within the system. Within a system, components may be placed in two different ways. In the first arrangement, a system is just a collection of components having no interaction between them. In the second kind of arrangement, components are interconnected and has interaction among themselves. Our objective is to derive the collected behavior of the

Component P_b `[t: sort]` *is*
out $a : t$
prv $b : t$
do
sh [] f : $(true) \rightarrow a := \psi(a, b)$
prv [] $g : (true) \rightarrow b := \psi(a, b)$

system in both situations using the categorical constructions. First, We are defining design of two components that we are extensively using in the running example to illustrate applying of categorical constructions.

Example 6.4 (Component P_r)
Component P_r `[t: sort]` *is*
in $x : t$
prv $d : bool, a : t$
do
sh [] r : $(\sim d \wedge (x \neq a)) \rightarrow a := x$
prv [] $t : (\sim d \wedge (x = a)) \rightarrow d := true$
Component P_r reads an input variable **`x`** *continuously from the environment through the action* **`r`** *, until it reads the same value twice upon which method* **`t`** *sets the boolean variable* **`d`***. From this specification we can construct two objects called SIGNATURE(P_r) and DESIGN(P_r)*

Example 6.5 (Component P_b) *From this specification we can construct two objects called SIGNATURE(P_b) and DESIGN(P_b)*

6.5.1 Semantics for Interconnection

The model of interaction between components in CommUnity is based upon on action synchronization and the interconnection of input variables of a component with output variables of other component. This procedure is also called name binding.

In the above example, we would like to establish a communication channel between component P_r and P_b. Purpose of this interconnection is that P_r should read the value provided by P_b. To achieve this, we interconnecting output variable of P_b with input variable of P_r and synchronizing the execution of method f with method r.

So, we are defining a third object called ChC as shown in the figure. Componet Ch C will have one input variable c and a shared method s. Also, we are defining two morphisms-

1. σ_a i.e. $ChC \rightarrow P_b$. This morphis, will define mappings like $\{ c \mapsto a, s \mapsto f \}$

2. σ_a i.e. $ChC \rightarrow P_b$. This morphism, will define mappings like $\{ c \mapsto x, s \mapsto r \}$

What we have done above to establish interconnection between two component is equvivalent to writing a glue code in programming language. The object that we have defined along with its morphism has mathematical structure. It means that the objects will take the form either signature or design structure and morphisms will take the form of either signature or design morphisms.

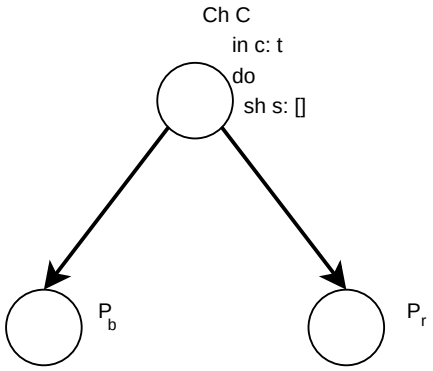

Figure 6.1: Interconnection Semantics

6.5.2 Semantics for Configuration

Configuration allows us to derive the collected behavior of a system from its components. Let us consider a simple case in which a system is composed of two components having no interactions between them. Two components that we are talking about are P_r and P_b. By looking at the signature of two given components, we can infer that the system which is composed of these

components should have four methods in it i.e. r, t, f, and g. The category construction that will give such amalgamated sum is the coproduct. By taking the coproduct of P_r and P_b we will have the component that will exhibit the behavior of P_r and P_b.

The coproduct of P_r and P_b is an object denoted by $P_r||P_b$ and it has two morphisms from $i_b : P_b \rightarrow P_r||P_b$ and $i_r : P_r \rightarrow P_r||P_b$ Component design for the object $P_r||P_b$ and mappings defined by the arrows i_r and i_b is given below:

Component $P_r||P_b$ `[t: sort]` *is*
in $x : t$
out $a : t$
prv $d : bool, a0, b : t$
do
sh [] r : $(\sim d \wedge (x \neq a0)) \rightarrow a0 := x$
prv [] $t : (\sim d \wedge (\mathrm{x} = \mathrm{a0})) \rightarrow d := true$
sh [] f : $(true) \rightarrow a := \psi(a, b)$
prv [] $g : (true) \rightarrow b := \psi(a, b)$

$i_b : P_b \rightarrow P_r||P_b$
$a \mapsto a$
$b \mapsto b$
$f \mapsto f$
$g \mapsto g$

$i_r : P_r \rightarrow P_r||P_b$
$x \mapsto x$
$a \mapsto a0$
$d \mapsto d$
$r \mapsto r$
$t \mapsto t$

Notice that the attribute a of P_r was renamed. This is because coproduct models disjoint sum so any thing that has same name must be renamed.

But, very few systems are there, in which component have no interaction among themselves. Most of the systems are constructed by interacting components. *Pushout* is the categorical construction used to derive the the collected behavior of the system having two interacting components. Following example illustrates this mecahnism.

Pushout of two objects P_r and P_b is a third object denoted by $P_r||_c P_b$ alongwith two morphisms μ_r and μ_b. The definition of $P_r||_c P_b$ in CommUnity,

and mapping provided by μ_r and μ_b is given below-

Component $P_b||_cP_r$ `[t: sort]` *is*
in $a : t$
out $a0 : t$
prv $d :\ bool, b : t$
do
$sh\ \ [] \ \ fr : \quad (\sim d \wedge (a0 \neq a)) \rightarrow a := \psi(a, b)||a0 := a$
$prv\ \ [] \ \ t : (\sim d \wedge (\mathrm{a0 = a})) \rightarrow d := true$
$prv\ \ [] \ \ g : (true) \rightarrow b := \psi(a, b)$

$\mu_b : P_b \rightarrow P_b||_cP_r$
$a \mapsto a$
$b \mapsto b$
$f \mapsto fr$
$g \mapsto g$

$\mu_r : P_r \rightarrow P_b||_cP_r$
$x \mapsto a$
$a \mapsto a0$
$d \mapsto d$
$r \mapsto fr$
$t \mapsto t$

6.6 Semantics of Architectural Connectors

Architectural connectors are defined as a collection of roles whose behavior is governed by the glue specification. Hence, architectural connection consists of-

- Two designs G and R, called the glue and the role of the connection, respectively.
- A signature θ and two morphisms $\sigma : design(\theta) \rightarrow G$;$\mu : design(\theta) \rightarrow R$ connecting the glue and the role.
- A connector is a finite set of connections with the same glue that, together, constitute a well formed configuration
- The semantics of a connector is the colimit of the diagram formed by its connections.

Chapter 7

Conclusions

This report presents an overview of software architecture field and develops basis for formalizing semantics of architectural connectors. The report concludes with-

- Architecture of a software system is the highest possible abstraction to describe the structure of a software system. Software Architecture describes a software system in terms of constituent components and interactions among those components. This type of description offers enormous benefits and challenges. Benefits are in terms of increased level of understanding, reuse, and better communication.

- Challenges offered by the software architecture are devising better ways to describe the software system. Most of the earlier research was focussed to solve the problem of how to describe a software architecture. As a result, a set of Architectural Description languages have been emerged. These languages either extends an existing language or a new ADL was created. Adding new features to ADLs is still a open area of research.

- Extracting architectural specification from requirement is a difficult task but use of architectural styles and unit operations simplifies the task of designing an architecture.

- Then report presents semantics of architectural connectors in three different formalisms i.e. First Order Logic, Process Algebra and Category theory. Each one of this formalism supports a different kind of

reasoning. Use of the first order logic allows us to answer the question- whether a concrete architecture realizes an abstract architecture or not. By using Process Algebra we can answer the question- whether the use of connectors is consistent with its definition or not. Category theory allows us a set of operators. These operators derives the collected behavior of a system from the specification of components and their interactions.

7.1 Recent Trends

The current research in the field of software architecture is addressing following problems-

- Linguistic researchers are devising new mechanisms for enforcing architectural specification at run time [1].
- Researchers engaged in the field of formalizing software specification are trying to give semantics to higher order connectors, and defining a category theory based algebra for connector composition.[7]
- Analytical models for estimating functional and nonfunctional properties from architectural specification is also one of major area of research currently pursued by researchers. [4]

7.2 Proposed Future Work

As a continuation of the study of architectural connectors, it has been proposed to undertake the architectural modelling of Linux operating system. The proposed work includes

1. To extract architectural information from the existing source code.
2. To identify architectural connectors used in the Linux.
3. To model Linux architecture using ADLs discussed in the report.

Bibliography

[1] Jonathan Aldrich. Using types to enforce architectural structure. *Unplished*.

[2] R. Allen and D. Gallen. A formal basis for architectural connection. *ACM TOSEM*, 6(3):213–249, May 1997.

[3] L. Bass. *Software Architecture Practice*. Addision-Wesley, 1998.

[4] Len Bass. Achieving usability through architectural styles. *Conference on Human Factors in Computer Systems*, pages 171–172, 2000.

[5] L. Coglianese. Dssa- adage: An open enviornment for architecture based avionics development. In *In Proceedings of AGARD'93*, May 1993.

[6] Eric Dashofy. An infrastructure for the rapid development of xml-based architecture description language.

[7] Jose Luiz Fiadeiro. A mathematical semantics for architectural connectors.

[8] D. Garlan. Exploiting style in architectural design environments. In *In Proceedings of SIGSOFT'94 The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering,*. ACM Press, December 1994 179-185.

[9] David Garlan. Acme: An architecture description interchage language. In *Proceedings of CASCON'97*, November 1997.

[10] David Luckham. Specification and analysis of system architecture using rapde. In *IEEE TRansactions on Software Engineering Special issue on Software Architecture*, pages 336–355, April 1995.

[11] Nenad Medvidovic. Modeling software architectures in the unified modeling language. *ACM Transactions on Software Engineeirng and Methodology*, 11(1):2–57, January 2002.

[12] Nikunj R Mehta Nenad Medvidovic. Understanding software connector compatibilities using a coonector taxonomy. December 2002.

[13] Mark Moriconi and Xiaolei Quian. Correctness and composition of software architectures. 1994.

[14] Dewayne E. Perry. Foundations for the study of software architecture. *Software Engineering Notes*, 17(4):40, October 1992.

[15] Hoare C. A. R. *Communicating Sequential Processses*. Prentice Hall, Englewood Cliffs N. J.

[16] Nenad Medvidovic Richard N. Taylor. A component and message based architectural style for gui software. *IEEE Transactions on Software Engineering,*, 22(1996):390–406, June 1996.

[17] A. W. Roscoe S. D. Brookes, C. A. R. Hoare. A theory of communicating sequential processes. *Journal of the Association for Computing Machinery*, 31 No. 3:560–599, July 1984.

[18] Mary Shaw. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):314–335, 1995.

[19] Mary Shaw. *Software Architecture Perspective on an Emerging Discipline*. Prentice Hall India, first edition, 2000.