Chapter 2: Architecture

Before getting any further, the thesis author would like to state his position on software architecture. In practice, two subjects of architecture are often considered, a) System architecture, and b) Software architecture. System architecture is generic in nature and does not look at specificities of computing or software. It is conceptual in character. It is organized on functional lines, or on resources needed to accomplish the function. It is focused on the problem domain. According to Gharajedaghi [37], and as shown in figure 2.1, System architecture is based on ideas suitable for system design, and expresses its evolution as the needs evolve. It expresses the evolution of the architecture, first on the structural features of resources needed, and then on the function as the system emerges in relation to the evolving needs necessitated by the context or environment of the system. The function is seen as the result of a process that threads through the resources that emerge through structure development driven by the needs for the system.

FIGURE 2.1: GHARAJEDHAGI’S FUNCTION - STRUCTURE - PROCESS – CONTEXT[37]

According to Nori et al [38] and ISO TS 30103 [39], this high level decomposition of the system is further refined by desirable quality characteristics of non-functional requirements as evidenced by the needs. So at a high-level, i.e., at the problem level, this first decomposition or segmentation is “seeing” into problem an acceptable decomposition of the parts that compose to form the system. Now that an acceptable solution is on hand, the next step is designing its software architecture, which takes into
account computing hardware and software platforms, their characteristics, and envisages a static structure of components, reflecting the representation of problem domain ideas in terms of computing and software technologies to be used. In this thesis, the focus is on the latter concerns that are germane to software architecture.

**Figure 2.2: Quality Decomposition [38-39]**

### 2.1 Introduction

Software architecture as a discipline and its work products has been gaining prominence over the last couple of decades. In Software architecture practice, the emphasis on good software architectures has resulted in the need for a systematic approach for architectural design. According to Garlan [28-30][40-45], Good architectures are increasingly based on architectural styles that typically specify the design vocabulary, its usage and semantic assumptions; However, practice has shown that each architectural style is appropriate for certain purposes, but not for others. This thesis considers architecture as high level structures of software that is defined by a structural composition of components. In this endeavour, the first step is to arrive at a definition of software architecture that clarifies the author’s position on software architecture. Before that, a literature survey of the existing definitions of software architecture is taken up so that the proposed definition can be seen in the context of past work.
2.2 Review of Literature on definition of software architecture

According to Bass et al [46-47], software architecture is the structure of the System comprising Software elements, externally visible properties of those elements and the relationships among them. Another view is that software architecture is the structure of a Software comprising components, properties of the components and relationships among them. According to them, software architecture is an abstraction of software and can comprise more than one structure that correlates to the Software elements organization and interactions. They believe that it is not necessary for these structures to be related although they are abstractions of the same System. They also consider behaviour, of the different Software elements, as part of the architecture and it is this behaviour that allows interaction between the various elements. It is evident from their discussions that there is no single, accepted definition of Architecture, but what is clear is that software architecture is focused on reasoning about the structural aspects of the System.

According to ISO, [26-27], Architecture is the fundamental concepts or properties of a System in its environment embodied in its elements, relationships and in the principles of its design and evolution. Another variant of this definition is given in ANSI/IEEE 1471-2000 [40] which is, Architecture is the fundamental organization of a System embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution. The general perception is that Architecture provides a unifying structure for exploration of the problem space and for characterization of the solution space such that better decisions can be made about the System under development. The types of structure that are fundamental depend on the nature of the System as well as the purpose of the architecture. Both these standard bodies generally espouse the view that the architecture of a System is its fundamental structure and it may include principles applying to the structure as well as the specific structures and agree that the purpose of architecting is to identify and describe a fundamental conception of a System that is bound by relations. It is evident that the view of the standard bodies with regard to Architecture predominantly focuses on formulating a basis for harmonizing diverse views of a System.

According to Garlan [28-30][40-45], Architecture is the high level organization of computational elements and interactions between these elements and is a necessary step in raising the level of abstraction at which Software is conceived and developed. Garlan is of the view that the computational elements define the primary points of computation of the System while the interactions serve as the connectors between the computational elements. Garlan suggests creating the architectural form around Component and connector types in which patterns of computation and interactions are repeated. Another
view espoused by Garlan is that the architecture of the System describes its gross structure using one or more views. This structure lists down a set of design decisions like the choice of components, their pathways of interaction, constraints for their interaction and the key properties of these Components. It is evident that the view of Garlan with regard to Architecture predominantly focuses on two fundamental abstractions, the components (an independent unit of computation) and connectors (interaction among Components).

According to Luckham et al [48-49], Architecture is a specification of the components of a System and the communication between them. They assume Architecture as an executable specification of a class of Systems at any level of abstraction comprising interfaces, connections and constraints. According to them, interfaces specify Components of the System, and the connections and constraints define how the components may interact. Based on this understanding, they identify three different classes of Architecture which are: a) Interface connection Architecture that defines all connections between the modules using only interfaces, b) Object connection Architecture that specifies the features that must be provided by modules conforming to the interface, and c) Plug and socket Architectures in which interfaces are allowed to provide services with dual services connected together by a service connection. It is evident that the view of Luckham et al with regard to Architecture predominantly focuses on two fundamental abstractions: the components (an independent unit of computation) and connectors (communication between Components).

According to Garlan and Allen [30][42][44-45], software architecture is an interconnected collection of object instances. This postulation provides a formal basis for reasoning about the properties of the System. Accordingly, software architecture is a hierarchical structure of components, connectors and configurations. To support this formulation, they provided a formal abstract model of System and its behaviour. The vocabulary of this formulation is defined by the basic architectural types: Components (the set of components in the architecture), Connector (set of connectors in the architecture), Port (set of ports of a component), Role (set of roles of a connector), Configuration (set of components and connectors that together form a configuration) and Binding (the glue to the connector). It is evident that the view of Garlan and Allen with regard to Architecture is that of interacting object instances.

According to SWEBOK [50], Architecture is how the Software is decomposed and organized into Components and the interfaces between those Components. In other words, software architecture is a description of the subsystems and Components of a software system and the relationships between them. Thus, software architecture defines the internal structure of software and it describes Components at a
level of detail that enable their construction. The key issues that need to be addressed for creating the architecture are the quality concerns that the Software must address and how to decompose, organize and package Software Components. According to SWEBOK, different views are necessary to describe and document this information and each view represents a partial aspect of a software architecture that shows specific properties of a software system. It is evident that the view of SWEBOK is that software architecture is a multi-faceted artefact produced by the architecting process and is composed of relatively independent and orthogonal views.

According to Maier [51], Architecture is the set of information that defines a Systems value, cost and risk sufficiently for the purposes of the Systems sponsor. This definition is based on the perception about what the architects do to build Systems. Another view of Maier of Architecture is that it is a structure, in terms of components, connections and constraints, of a product, process or element. He believes that both these views are complementary to each other. According to Maier, Architecture strives for fit, balance and compromise among the needs, resources, technology and multiple stakeholder interests. He also suggests heuristic as the guides to solve an architecting problem and accordingly the art of architecting lies in the wisdom of knowing which heuristics (may name or imply specific Components or kits of components) to apply in a specific situation. It is evident that the view taken by Maier is that software architecture is synthesis oriented and operates in domains, with concerns that preclude synthesis and it proceeds through a mixture of heuristic, rational and scientific methods.

According to RAPIDE Design team [52], software architecture is a plan of a distributed object System showing what types of modules are Components of the System, how many Components of each type there are, and how its Components interact. Accordingly, software architecture is a pictorial representation (architectural plan) of how the Software Components are wired up in order to provide the desired behaviour to the software system. Software architectures are developed at different levels each covering different levels of details (from high level involving abstract concepts to low levels involving more detailed concepts). The software architecture at any particular level is consistent with the architecture at other levels (often this correspondence is complicated). It is evident that the view taken by the RAPIDE design team is that software architecture is a multi-layered design artefact that is either object based or interface based.

2.3 Proposed definition of software architecture

Over the last couple of decades, many definitions have been given about Architecture by practitioners and researchers in this field. These definitions have evolved by the observation of design principles that
architects adopt and actions that they take when architecting real Systems. Some define Architecture as a high level design; some define Architecture as a manifestation of the earliest set of design decisions; some define Architecture as the overall structure of the System; some define Architecture as the structure of components, their interrelationships, and the principles and guidelines governing their design and evolution over time; and some define Architecture as Components and connectors. It is evident that there is no common consensus about these definitions, but each of these definitions has been practically found to be appropriate for a specific situation. In this section, a definition of software architecture is put forth.

In buildings, Architecture is considered as a structure of contained spaces and flows wherein spaces are the resources in the broadest sense and flows are the ways in which these resources are accessed as well as transformed. In order to apply this analogy to Software, it is necessary to map spaces and flows to computing elements. In computing, information is the only resource available, and everything in the outside world of interest is represented symbolically by designing representation that maps the entities of interest in the world to symbols. So, software architecture is a static structure of information resources (finitely bound technical spaces) and information flows for the purpose of meeting the requirements. This structure must have elements in it, which are Components. It must have flows to access the resources, which are access methods. It must have flows to transform the resources, which are operations. Software architecture can be functional (specialize contained spaces to realize a specific function) or general purpose (provides for contained spaces to realize more than one function).

Therefore, software architecture is a static structural Composition of components, wherein, Components are analogous to “containers of information and their associated operations”. A component encapsulates information, a set of access methods and operations and may comprise other Components. The represented information can change but the access methods and the operations remain the same over the time period of existence of the component. In essence, a component is a symbolic name for (or reference to) contained spaces (information resources) that a software system manipulates using its flows. It uniquely identifies a set of software elements so that there is no ambiguity when referring the component as part of the software architecture of the System. There could be two kinds of components. One type of component is a contained space and associated flows to access the contained space. The other type of component is transformational in nature. It supports either the transformation of the state of the contained space or transformation of one contained space into another.

Towards understanding this particular viewpoint, a simple example is provided. Minimally, the components of the client server Architecture are the client that serves as the interface to the user; the
server that processes the user provided information according to the architectural requirements and feeds the client with the processed information; a transmission line that serves as the bridge between the client and the server. The transmission line Component transforms the information received in the client, appropriately typed, passes it to the server which processes the information and generates new information that is transmitted back to the client for user interaction. The server Component provides a function to many clients which initiate requests for the specific function. A client requests for a server function by initiating communication sessions with servers. As shown in Figure 2.3, at a higher level, the client server Architecture is composed of three components, the client Component, the server Component and the transmission line (messaging) Component.

All these Components encapsulate a set of access methods and operations which are juxtaposed together, during Composition, to arrive at a set of contained spaces and flows for the composed client server architecture. Going further, the next step is to arrive at an approach to specify software architecture that clarifies the author’s position on software architecture. Before that, a literature survey of the existing approaches of software architecture description is taken up so that the proposed approach can be seen in the context of past work.

**Figure 2.3: Illustration of Software Architecture**

Going further, the next step is to arrive at an approach to specify software architecture that clarifies the author’s position on software architecture. Before that, a literature survey of the existing approaches of software architecture description is taken up so that the proposed approach can be seen in the context of past work.

### 2.4 Review of Literature on Description of Software Architecture

The objective of describing software architectures is to assist the understanding of the Software’s essence and key properties pertaining to its behaviour, composition and evolution. Traditionally, an Architecture Description Language based on a formal, abstract model of system behaviour can provide a
practical means of describing software architectures and architectural styles. Even though architecture
description is considered as a work product of architecting, there is no universal agreement on what
architectural description languages should represent with regard to the software architecture, what an
Architecture description language is and what should be expressed by the architecture description
language. The general agreement is that software architecture is a set of components and the
connections between them and hence components, connectors, configurations and constraints are in
some way or the other common entities across all Architecture description languages. It is evident that
the distinction between Architecture description languages, formal specification of software, module
interconnection languages and programming languages is very blurred. In this section, a review of the
different approaches for describing software architectures is discussed.

According to Allen [44-45], the Wright Architecture specification language represents architectural
structure as a group of components and connectors. Components represent architecturally relevant units
of computation and data storage, and connectors represent the interactions between the components.
The description of a component comprises of the interface and the computation. Each interface consists
of a number of ports which defines an interaction that the component participates. The description of
the component consists of a set of roles and the glue. While each role defines the behaviour of an
interaction participant, the glue defines how the roles interact with each other. In Wright, Components
and connectors are typed. While describing a Software using Wright, one need to declare a set of
component and connector types which collectively form a style; followed by a declaration of a set of
instances of these types and the way in which they are assembled. A style may also contain constraints
on the components and connectors types and are defined as predicates that must be true of any
configuration in that style. It is evident that Wright has been designed to provide a formal basis for
specifying the interactions between architectural Components.

According to Garlan [43] [53], ACME embodies the architectural ontology of components, Connectors,
Systems, Properties, Constraints and Styles and provides a semantically extensible language for
architectural analysis and integration. ACME as an Architecture description language supports definition
of the Structure (System organization), Properties (System information), Constraints (guidelines covering
change over time), and Styles (classes/families of Architecture). ACME defines the architectural structure
using the entities components, connectors, Systems, ports, roles, representations and representation-
maps; defines the architectural properties using annotations; defines the architectural constraints using
first order predicate logic and defines the architectural styles using a type System that encapsulates
recurring structures and relationships. The grammar of the ACME Architecture description language internally establishes consistency relationships between the various semantic elements. While the goal of this initiative is to provide an expressive notation for Software architects to describe their software architectures, according to Garlan, much remains to be done both in practice and research. Even though ACME is a generic ADL and provides a common skeleton for expressing Architectures, it does not provide sufficient basis for architectural description and analysis as it has been designed predominantly as a schema that can be used as an Architecture description interchange language.

According to Shaw et al [54-56], an architectural description language must support System configuration, independence of entities, abstraction and analysis of properties. According to them, Systems are composed from identifiable Components and connectors of various distinct types; and Components interact in identifiable, distinct ways. They define Components as the locus of computation and state (computational capabilities) with each Component having an interface specification that specifies its properties. They define connectors as the locus of definition for relations among Components (mediates interactions between Components) with each connector providing rules for the relations. According to Shaw et al, Components can be primitive or composite; Primitive Components are implemented and Composite Components are configurations of primitive Components and connectors. They also consider connectors to be either primitive or composite; Primitive connectors are implemented while Composite connectors are abstract and derived from these primitive connectors. In order to support Composition of components/connectors, Shaw et al utilize localization, abstraction and encapsulation of information. A subset of these ideas was implemented as part of the UNICON Architecture description language. This language emphasizes on the structural aspects of software architecture and supports the constructs Components and connectors. While composite Components are supported, composite connectors are not supported as part of UNICON. Even though Shaw et al, speak about Component Composition and connector Composition, they predominantly consider them as executable configurations based on black box implementations; they are also constrained by the fixed collection of interaction types. Specifically, their UNICON language and toolset support composing software architecture only from a restricted set of component and connector examples.

According to Medvidovic et al [57-59], Architecture description languages should support Architecture based evolution and specifically evolution of components, Connectors and architectural Configurations. Their Architecture description language, C2SADEL (C2 style based software architecture Description and Evolution Language), is designed by embodying the principles of type theory and defines software
architecture as Component types, Connector types and Topology. A component type is defined by a name
(identifier), a set of interface elements (further defined by a name, set of parameters and a result),
associated behaviour (defined by an invariant and a set of operations) and a possible implementation. A con-
ector type is defined by a name (identifier), a set of connectors (further defined by a name and a set
of parameters) and a possible implementation. The topology defines Component and connector instances
and their interconnections for a given System. Even though, according to Medvidovic et al C2SADEL
supports Component evolution through subtyping and refinement and also supports hierarchical
Composition by which an entire Architecture can be used as a single Component in a different
Architecture, the notion of component/Connector Composition is ambiguous as it is assumed that the
mechanism of extensible types (defined either in terms of communication protocols or as enumerated
types) and type checking would meet the needs
of components and Connectors Composition.

According to Feiler et al [60-61], the architecture Analysis and Design Language (AADL) provides
Components with precise semantics to describe the software architecture. A typical AADL description
appears and functions like a program that is primarily limited to prototype and declarations. AADL’s
semantics include representations for the description of software and hardware parts of a System. An
AADL description is made of components which could be Software abstractions (data, thread, thread
group, subprogram, and process), executable platform abstractions (memory, bus, processor, device,
virtual processor, and virtual bus) and composite/hybrid abstractions (System). Since the thesis’s focus is
on software architecture, only Software abstractions which are data component (representing data
field/type), thread Component (concurrently executing Component), thread group Component
(Composition of threads and data into a single Component), subprogram Component (executable function
or method) and process Component (encapsulation of an executable process in a protected space) are
considered. At the top-level, a System contains all Component instances and each of these Components
can have sub-Components thus supporting hierarchical Components decomposition. The interface of a
component is the component type which provides features and Components communicate with each
other connecting their features. These Components and Component types are then represented textually
or symbolically in accordance with the software architecture of the System of interest. Even though AADL
supports an Architecture-centric, model-based development approach for software systems, the notion
of composition is ambiguous and only certain types of software abstractions are supported (any other
Component needs to be force fitted into the predefined types). These abstractions are not Component
abstractions but are predominantly Software abstractions.
According to Krutchen [62], software architecture deals with abstraction, Composition and decomposition, style and aesthetics and to describe this software architecture, one utilizes a model composed of multiple views or perspectives. Krutchen’s model (4 + 1 view) is made up of five main views which are not fully orthogonal or independent since elements of one view are connected to elements of other views based on certain rules/heuristics. Each view is defined by using a set of elements of the form \{components, containers, connectors\} and is described by a blueprint. These views are: the logical view which represents the object oriented decomposition model, the process view which represents the concurrency and synchronization aspects, the physical view which describes the mappings of the Software onto hardware, the development view which describes the static organization of the Software and the scenarios view which illustrates a few selected use cases of the architectural decisions. The use of views would not solve all the software architecture problems even though use of views makes the architecting problem manageable, it is often offset by errors that creep in due to inconsistency, selection of wrong set of elements and relationships in the various views and fragmentation often resulting in a model that is unmanageable and does not adequately represent the System as a whole.

ISO 42010 [26] defines standard terms and presents a conceptual foundation for expressing, communicating and reviewing Architectures and specifying requirements that apply to Architecture descriptions. Essentially, the standard covers the output of an architecting process, without providing any help on how to construct such a process. ISO 42010 emphasizes that Architecture descriptions are inherently multi-views (as no single view can address all architectural concerns) with each view covering certain concerns of the entire System. This standard proposes a series of guidelines that can be followed for developing the conceptual model of the architecture description. These guidelines are: a) Use Stakeholders and Concerns to identify the most critical problems, b) Use Viewpoints to make sure the architecture products meet stakeholder expectations, c) Consider Quality Reviews, d) Viewpoints provide the descriptions for Architecture content, e) Views and models provide the architectural basis, f) Prioritize stakeholder concerns and models, g) Define an Architecture Framework, h) Tie the architecture Framework into the overall development process, i) Provide for maintenance of the architecture over the System life-cycle. Based on these guidelines, the produced architectural description must have an Architecture design element, Architecture, Architecture description, Architecture description language, Architecture framework, Architecture model, rationale, correspondence, view, view point, concern, correspondence rule, environment, stakeholder, concern, System and System of interest. Essentially, the ISO 42010 standard defines all the possible entities that an Architecture description/Architecture description language should contain at a meta-level.
UML [31] and SysML [32], provide out-of-the-box modelling constructs for creating architecture models and views. They also provide the profile extension mechanism for customizing UML models to the specific modelling needs and domain needs. In addition, they provide the stereotype profile class which enables the extension of a meta-class as part of a profile. UML [31] and SysML [32] stereotypes, define new modelling constructs by extending existing constructs or by deriving from existing constructs with new properties and constraints. Stereotypes are often expressed as annotations, by identifying it appropriately, to enhance the standard interpretation of the model elements. They are applied to the different model elements in conjunction with the meta-class that they extend. In theory, stereotypes and profiles are possible extension mechanisms that can be used to support a wide variety of views and models as long as they do not violate the constraints and conditions imparted on the base model elements. This means that even though it is possible to add new relationships or extend existing relationships, it is not possible to redefine what these relationships are as that would violate the constraints and properties of the basic modelling elements. This is especially true for associations like part of and composed of as they have special interpretations in UML/SysML.

According to Sommerville [63], algebraic specification is an appropriate technique to specify interfaces between software systems and it involves designing the operations on an abstract data type and specifying them in terms of their inter-relationships. Accordingly, a complete algebraic specification comprises a name, an optional generic parameter list (predominantly collections of other types which are instantiated for a specific type specification) and a body. The body of the specification comprises a type name (sort) which corresponds to a set of objects along with an imports declaration for utilizing other types. It also contains a description part where the operations are described informally. The signature part defines the syntax of the interface in terms of operations and their parameters. Operations are usually classified as constructor operators, inspection operators and manipulation operators. The axioms part defines the semantics of the operations by defining a set of axioms that characterize the behaviour of the abstract data type. Even though Ian speaks of a formal specification for software architecture, what has been defined is an informal interface specification expressed as a set of abstract data types. This abstract data type is defined in terms of relationships between the type operations. Composition of interfaces is supported by importing abstract data type definitions of other interfaces and instantiating them as part of the operations.

A formal approach to describe software architectures is to model them as algebras [35-36] [65-67]. In computing, there is only data and computation, therefore components are the architectural
abstractions that are defined as a set of abstract data types and related type operations. The instantiation of these abstract data types leads to the definition of the state of the components. Once these components are agreed and defined, they can be developed independently and then hierarchically composed together to provide the overall software architecture of the System of interest [70]. When using algebras to model architecture, it is necessary to deal with a finite number of components that are partitioned and hierarchically composed together based on the requirements of the System that the architecture should satisfy. Syntactically, a signature (which comprises of the carrier sets referred by its symbolic names, and operations with its parameters and results) together with a set of equations over that signature constitute a specification of a component. In other words, a component specification comprises of the carrier sets, its operation names and the axioms that the operations must satisfy. An architecture specification is built by consecutive application of compositional semantics on component specifications. The compositional semantics of composing component specifications would involve forming the union of the component specifications to arrive at a new algebra (which would be a cartesian product of the constituent algebras) which is then transformed into a single composite component signature (comprising of a symbolic name that refers to the carrier sets and associated operations) and combining the equations of each specification to arrive at new equations on the composite component signature. The structure of a typical architecture specification would be:

<table>
<thead>
<tr>
<th>Sort</th>
<th>&lt;Sort Name&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imports</td>
<td>&lt;List of carrier sets&gt;</td>
</tr>
<tr>
<td>Operations</td>
<td>&lt;Function symbols with their signatures&gt;</td>
</tr>
<tr>
<td>•</td>
<td>&lt;Operation Name&gt; (Sort List) ➔ Return Sort</td>
</tr>
<tr>
<td>Axioms</td>
<td>&lt;Unconditional and conditional equations&gt;</td>
</tr>
</tbody>
</table>

TABLE 2.1: TEMPLATE FOR ALGEBRAIC SPECIFICATION [35-36][65-67]

2.5 Summary

This thesis considers software architecture as the high level structures of software that is defined by a structural composition of components. It takes cognizance of the fact that software architecture is the work product emanating out of software architecting, and considers software architecting as the process of defining and composing components; which are expressed using architecture descriptions [26]. This thesis also takes cognizance of the fact that there are different approaches to express these architecture descriptions.