Chapter 3: Composition

Software exists as nearly independent components and their compositions thereof. It is understood through reductionism of components, their functions; and synthesized through composition keeping in mind a set of stated constraints. The nature of components is understood by how they work together in realizing the function. As stated in chapter 2, this thesis considers software architecture as a static structural composition of components; and software architecting as the process of defining and composing components.

As part of the architecting process [27], as shown in figure 3.1, an architect must segment or decompose the problem [27][38] so that at some level in the decomposed problem tree, the decomposed sub-problems become addressable. This decomposition depends on prior exposure to relevant problem domains, technologies that bear on matters of scale of solutions and on engineering techniques that address matters of variety in solutions. Whist decomposition of the problem is crucial to devising an architecture and leverages past experience and relevant expertise, what is equally important, and not always addressed in sufficient detail, is the composition of components, as shown in figure 3.2, so discerned into a whole: the Divide and Conquer strategy is never complete without focused attention on Conquer. In the systems engineering world, the use of objectives tree and product breakdown structures
often serve as a useful mechanism for this purpose. However, in the software world, relatively little attention has been paid to composing components into systems and it is expected that the glue (which is usually code or some form of script), as shown in figure 3.3, would facilitate it. Such a glue is external to the components that are composed and contains constructs that enable external coordination (usually implemented as communication protocols) between components. So far, there is no theory worked out on composition and most of the solutions have been found to be useful under specific situations. Therefore, understanding Composition is crucial to reasoning about architecture.

![Figure 3.2: Solution Composition (Conquer Strategy)](image)

Going further, the first step is to arrive at a definition of composition that clarifies the author’s position on what composition is. Before that, a literature survey of the existing definitions of composition is taken up so that the proposed definition can be seen in the context of past work.

### 3.1 Review of Literature on definition of composition

According to Kiniry [68-69], Composition encapsulates the general notion of putting two or more constructs that are of the same type together in some way. It is a constructive operation that results in a new thing that has some of the properties of the constituent pieces. The semantics of the composition operator used to effect the composition decides the properties of the new construct. The very centre of composition is the bringing together of parts to form a whole. This involves carrying out a simple set of
operations on parts and interconnections by utilizing the composition operator, which is simplified by their representations. Since the outcome of the composition is a “whole”, it is designed top-down. The key to understanding the composition depends upon the logical structure of combination of different parts. As a process, Composition deals with composing constructs to form new constructs and utilizing these new constructs as part of a larger structure. Even though Kiniry’s notion of composition is generic and is more aligned with the notion of synthesis and creativity, he has not considered the competing/conflicting factors in the environment that influence the composition of parts to form a whole.

In UML [31], Composition is a relationship found in class and object diagram. It is currently named as Composition association relationship (UML 1.4) and is a binary structural relationship between the whole and its parts. It is a variant of the “has a” association relationship and exhibits a strong life cycle dependency between instances of the container class and instances of the contained class(es). It represents the whole-part relationship and specifies that the lifetime of the part is dependent on the lifetime of the whole. In other words, the composition relationship indicates that the whole and its parts are strongly dependent on each other for its existence that the contained object ceases to exist when the contained object ceases to exist. Even though in UML, Composition is defined as a relationship between whole and parts and is frequently utilized in developing UML models, UML however does not define how, when and the specific order in which parts of the composite are created. There is also no clarity on the nature of the relationship as each part can either belong to the whole or each whole can be considered as a grouping of parts. Simply stated, Composition is ill-defined in UML. While the stereotype extensibility mechanism can be utilized to define the meaning of composition in UML diagrams, this is subject to interpretation by the model users and could be potentially mis-understood and mis-represented.

In SysML [32], Composition is defined as a form of aggregation which requires that a part instance be included in at most one composite at a time. Composition is recursive in the sense that a composite may be considered as a part of another composite. It is responsibility of the composite for the creation and destruction of the parts. In SysML, composition is a relationship found in block diagrams and is defined that the component blocks can exist only in the context of the owner composite block. It is currently named as Composition association relationship and is a binary structural relationship between the whole and its parts (or composite and its parts). It is also considered as a relationship that represents the interactions between parts. Even though SysML discusses about the notion of composition of components and Composition relationship, Composition is ill-defined in SysML either as a relationship or as an organizational (structural) phenomenon.
According to Sifakis et al [71-73], Composition can be considered as gluing a set of components (which could be simple or composite) together to build larger composite Components. They propose using families of composition operators to facilitate this gluing so as to support heterogeneity of components and their composites. The different Composition operators build composite Components by composing separately the corresponding layers of its arguments and then juxtaposing the different layers together. They consider a component to be a tuple represented as C = (B [K], IM [K]), where B is the behaviour model and IM is the interaction model of a set of interacting Components symbolised as K. Accordingly, the composition of two Components C[K1] and C[K2] would then be C[K1 U K2] = (B[K1 U K2], IM [K1 U K2]) = (B [K1] X B[K2], IM [K1] U IM [K2] U IM [K1, K2]) where X is a binary associative behaviour Composition operator. Even though glue mechanisms and operators define how associated Components interact with each other, Sifakis et al base Composition of components on a component framework and an associated Architecture. In other words, the target Architecture of the resulting Software is fixed and the different Composition mechanisms are chosen to support this target Architecture.

According to Attiogbe et al [74], a composition is a well formed Component assembly which is encapsulated within a component. This is achieved by defining a compose operator that builds a new Component by combining one or several Components. Additionally, a component provides services and consumes services and has interfaces to support provision and consumption of services. The parameters of the compose operator are the composite (outer Component) with its interface and a well formed assembly which is a set of {components, Connectors, Interfaces}. The interfaces establish the link between the provided and required services of the components. The result of the composition is a new Component which contains every provided/required service of the contained Components and provides/requires the promoted services from other Components that are outside its boundary. Even though the definition of Attigobe et al, of composition is meaningful in the context of software architecture, their Composition approach works more at the abstract specification level for a specific Component model/framework that they have put together rather than for any Architecture. Their approach is also tightly bound to Component based services development as they utilize interface connection Architecture as the basis of their Composition mechanism.

According to Petty and Weisel [75], Composition is the ability to select and assemble components in various combinations into valid systems that satisfies specific user requirements. The defining characteristic is that different systems can be composed at configuration time in a variety of ways, each suited to some distinct purpose. Composition is more than just putting together components to form
Systems/super components; it is the mechanism that provides ability to combine and recombine, to configure and reconfigure a set of components from those available into different Systems that meet different needs. Composition can be either syntactic or semantic. Syntactic Composition requires that the components are constructed so that their implementation is compatible for all possible configurations that might be composed. Semantic Composition is a question of whether the components are meaningfully composed and whether the resultant computation is valid. In other words, semantic Composition is to enable formal reasoning about the characteristics of the composed Components. Even though some approaches to Composition have been put forth by Petty and Weisel, they are more in the realms of computability theory and mathematical logic rather than in the realms of software architecting. Their general outlook is that a set of valid Components is composable if and only if their composite is valid (this term is left undefined by Petty and Weisel).

According to Allen [44-45], an important class of composition in software architecture is active interaction between Components based on discrete actions. Accordingly, Components carry out some part of the total computation and interact to combine their behaviours resulting in a new behaviour for the System as a whole. These interactions are either sequential (such as in a batch model) or a network of ordered interactions. According to Allen [44-45], Components are composed into a functioning System using the same kinds of interactions repeatedly. The goal then is to identify patterns of configurations (architectural style) which can then be used to characterize software architecture and/or create Compositional rules. Allen proposes the WRIGHT Architecture description language [44-45] which is built around the architectural abstractions: Components (independent computational entity), connectors (Composition pattern between two components) and configurations (composite of components and connectors). Allen uses the notion of Glue to indicate how the behaviours of the two components that are connected together combine to form a complete interaction. Even though the term composition is used by Allen, it actually refers to interaction between independent computations and not composition of components to form composites.

According to Sametinger (who reflects on Nierstrasz and Dami) [76-78], Composition is the process of constructing applications by interconnecting Software Components through their plugs. In addition, Sametinger states that Plugs are well-defined ways to interact and communicate with the components. He believes that parameterization is the key for Component Composition and customization. Sametinger distinguishes between different forms of components Composition: Internal Composition when Components are included in a software system (they are integrated into the System and are an inherent
part of it); External Composition when Components run on their own and they communicate with other Components through some means; Textual Composition when reusable Components are inserted at different locations according to parameters; Functional Composition when simple functions are combined to build more complicated ones with the result of one function passed on as input to another function; modular Composition facilitate data encapsulation, information hiding and minimize the interdependencies of components; object oriented Composition where in Components can be extended, encapsulated, inherited, dynamically bound and designed to support polymorphism. Even though Sametinger’s introduces different classification of component composition, Sametinger’s definition of composition is predominantly a Software engineer’s view of assembling Components and their associated mechanisms.

According to Prieto-Diaz et al [79], Composition is based on the definition/use of bindings between Components. Accordingly, each Component defines/provides a set of facilities that are available to other Components and uses/requires facilities provided by other Components. The purpose of the glue that composes multiple Components together is to resolve the definition/use relationships by indicating for each use of a facility. Even though Diaz’s approach is good for describing implementation relationships between the components of a System, it is not well suited to describe the interaction relations that are central concerns of architecting. As a result, it becomes difficult to reason about the System or its Composition.

According to Nierstrasz et al [78-79], Software composition is the construction of software applications from components that implement abstractions pertaining to a particular problem domain. These abstractions could be in the form of wrappers, packaging (gluing Components together), services, agents, adaptors, namespaces and operators. Additionally Nierstrasz et al [78-79] are of the view that, software architecture is a description of the way in which a specific System is composed from its Components (in other words, the composition description of a System is its software architecture). This is because, a flexible application can be achieved if its Architecture allows Components to be added, removed, replaced and reconfigured without disturbing other parts of the software system. They further characterize Software composition into the framework level, Composition level and the instance levels. Even though Nierstrasz et al, gave a practical definition of composition and characterised it into 3 different levels, their expressions were predominantly proposals to the research community. Their proposal discusses about the need for a comprehensive model of software composition and an appropriate Composition language to facilitate designing, specifying and reasoning about Component frameworks.
According to Vicki de Mey [80], Composition is defined as communication between Components through their Composition interfaces. Vicki de Mey defines Components to be made up of Behavior and a Presentation. While the Behaviour is responsible for the component’s Composition interface and the work the component is designed to perform, the Presentation is the representation of the component (display). He further defines the notion of Ports and Links to enable Component communication. Vicki de Mey is also of the view that composition enables a set of components to be assembled together and function as a single Component. Vicki de Mey proposes the use of a composition model which comprises of a set of rules for Component Composition in a particular domain. Decoupling the rules for Composition from components allows a variety of software composition operations. Even though Vicki de Mey has put together a definition of composition, his perspective is predominantly based on Visual Composition of components wherein applications are constructed by the direct manipulation and interconnection of visually presented Components.

According to Councill et al [81], Composition is the combination of two or more Software Components to yield new Software Component behaviour at a different level of abstraction. The characteristics of the new Component behaviour are determined by the components that are being combined and by how they are combined. Composition also includes definitions of the necessary interfaces and rules for combining Components; how Components can be composed to create a large structure and how one Component can be replaced by another that exists within that large structure. This large structure is termed as the component infrastructure (or) Component framework. The components within a component framework interact with each other through method (or) service invocations. Composition is usually specified using glue languages or through visual notations. In case of component frameworks, Composition is made possible by creating Components that satisfy the specific interaction standards defined by the component infrastructure. It is evident that Counciell et al [81] propose the use of component models which define standards for naming, meta-data, Component behaviour specification, Component implementation, interoperability, customization, Composition and deployment. The implementation of the component models provide a run-time environment and basic services necessary for the components to work together. From this, it is clear that the target Architecture of the software system is fixed and it corresponds to the architecture of the component model. A new Architecture is possible if a new Component model that corresponds to this Architecture along with the necessary run-time environment and basic services is made available.
In CORBA [82], Composition denotes both the set of artefacts that constitute the unit of component implementation and the definition of this aggregate entity. It is also defined as a named scope that contains elements that constitute the composition. A composition definition specifies the component home type (implicitly identifies the component type for which the composition provides the implementation), Abstract storage home binding (implicitly identifies the storage type that incarnates the component), Home Executor (describes the relationship between the home executor and the other elements of the composition), Component executor (programming artefact generated as the skeleton of the component executor), Delegation specification (specification of home operation delegation) and Proxy Home (Provides implementations of home operations without contacting the container). A composition binds all the above mentioned entities together and requires that the relationships between the bound entities define a constituent whole. Even though there is a definition of composition in CORBA, this notion of composition is relative to its Component model and the underlying Component framework as a result of which the target software architecture is fixed.

3.2 Proposed definition of composition

Many definitions have been given about Composition by practitioners and theorists. These definitions have evolved by the observation of design principles that architects adopt. Some consider composition as the organization of the whole out of its parts or in other words the conception of single elements, the interrelating of these elements and the relating of them to the total form. Others look at it as a process and suggest that the process of composition results in the creation of an ordered expression of an architectural form from its raw materials like space, mass and flows. Others consider composition as a means of synthesis by joining different parts into a compound. Some others consider composition as the act or mechanism to combine simple entities to build ones that are more complicated. 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 composition is put forth.

Generally, Composition is considered as an amalgamation of two or more elements in order to derive a new form that exhibits emergent characteristics. This thesis considers composition as both an activity (process of combining elements) as well as the schema of the composed composite (structure comprising multiple elements). As an activity, Composition is “The act of creating a representation of an unknown and original composite and its properties from a collection of constituent elements”. Composition can also be viewed as the act of creating relations while taking into account other relations. The objective of this activity is the creation of the composite that has the desired properties and not the accumulation of
components in some form. The properties to be predicted include technical aspects, formal structure and spatial structure. This activity involves generation of the composite’s form, predicting its schema and predicting its emergent properties before its embodiment using constituent elements. It involves understanding how the components necessary for creating this form is organized formally, semantically and how it is represented and how these representations can be acted to produce effective transformations.

As a schema, Composition is “The structure of a composite Component in terms of its constituent components, their interrelationships and the emergent properties”. Composition means creating relations between Components taking into account other relations. It is the way in which Components are put together to create a “composite”. It is semantically motivated and is understood through reductionism. A composition comprises of a set of components and the logical/formal structure by which the components are interconnected. It establishes the interdependence between the various components and is a description of how Components compose structurally and operate dynamically. The fundamental characteristic of a composition is that every time it is changed, say by addition or subtraction of a relationship or component, then the properties of the whole composition changes. The emergent property in such a case is the more developed behaviour of the composite which was not realizable without the composition. In order to address Composition wherein a formal or logical structure that relates Components with other Components is possible, these Components need to be of the same type. That is, the internal structure of all the “Components” should be the same which implies that the internal structure of the “Components” is same as the internal structure of the “composite”.

Towards understanding this particular viewpoint, a simple example of an Adobe PDF to Microsoft Word document converter Composite (APMWD) Component is illustrated in figure 3.3. Assuming that there are two higher level Components that has the ability to manipulate the corresponding document formats, the APMWD composite is a composition of the Adobe PDF Component, which serves as the source, the Microsoft Word Component, which serves as the sink, and an intermediate translator Component that serves as the bridge between the two different document formats (between the source and the sink). Individually, the Adobe PDF Component and the Microsoft Word Component are useful only for manipulating the specific document formats, while the composition provides the ability to translate document in one format to another. While the individual Components would be rich in their respective manipulative capabilities, collectively they serve the purpose of enabling the transformation
from one document format to another thereby satisfying the purpose for which they are composed. This is the emergent behaviour of the composition.

![Adobe PDF to MS Word Converter Composite](image)

**Figure 3.3: Illustration of Composition**

### 3.3 Review of Composition Mechanisms

There exist a variety of component models and composition mechanisms that are promoted by the industry and the academia. Each of these models and mechanisms are designed for a specific purpose and have been found to be useful for software developers in that space. So far, a single component model and an associated composition mechanism that is applicable across different architecture instances has not been identified. Additionally, most of these mechanisms are tackled predominantly at the implementation level rather than providing an architectural abstraction for architects. Almost in all cases, there exist programming language constructs that support component models and composition mechanisms. While Microsoft Visual basic adopts scripting as the basis for composition, Java adopts strong abstract types as the basis for supporting composition, and Microsoft Visual C++ adopts abstract base classes as the basis for enabling components composition. At the operating system level, APIs have been the preferred mechanism for facilitating composition. Scripting languages like Python, Perl, PHP, Java Script, and VB Script have resorted to gluing code as the basis for enabling composition.

Component technologies that enable dynamic loading of components (composition orchestrators) like Microsoft COM [9], and CORBA [82] have promoted the use of interface definitions, APIs and programming language constructs to facilitate composition of these dynamically loaded components. In the case of Service Oriented Architectures [18], the abstract specification of the services that needs to be composed is defined at compile-time and at run-time, from the list of available services in the service repository, that matches the abstract service specification, a service is identified and orchestrated. Additionally, the abstract specification defines the interfaces between the components that provides the
services and the components that consumes these services. Composition systems like PICCOLA [83] adopt component interface definitions and interface definition languages as the basis for enabling composition. These systems are predominantly modelled on the compiler architectures and utilize constructs like composition linkers, composition scripts, composition compilers and component definitions as part of their component based development environment.

3.4 Review of Literature on Specifying Composition

The objective of specifying composition is to assist the understanding of the Software’s Composition and key properties pertaining to its constituent elements. Traditionally, an architecture description language based on a formal, abstract model of system behaviour should provide a practical means of describing composition. However, this is true only if the architecture description language supports the notion of composition. Currently, there is no universal agreement on what architectural description languages should represent with regard to software composition and how to represent it. The general view seems to be that software composition is represented by the instance of a configuration (which is at par with a sub-System) of components and connectors. In this section, the various approaches to specify components composition is discussed.

According to Allen [44-45], in the WRIGHT Architecture description language, a configuration is a collection of component instances combined via connectors and is completed by describing the attachments which define the topology of the configuration (by showing which Components participate in which interactions). Accordingly, the attachment declarations bring together each of the constituent elements of the composition. The component carries out a Computation, part of which is a particular interaction, specified by a Port. That port is attached to a Role, which indicates what rules the port must follow in order to be a legal participant in the interaction specified by the connector. If each of the components, as represented by their respective ports, obeys the rules imposed by the roles, then the connector Glue defines how the Computations are composed to form a single, larger computation. In order to define a hierarchy of composite components, the nested Composition description has an associated set of bindings which define how the ports on the inside are associated with the ports on the composite. Similarly, for connectors, the roles on the inside are identified with the roles on the outside. It is evident that Allen facilitates specification of computation (behaviour) Composition in the WRIGHT Architecture description language and other forms of composition like function Composition, object Composition, features Composition, state Composition are not supported in WRIGHT.
According to Garlan [34], in the ACME Architecture description language, a System (and also a sub-System) is defined as a configuration of components and connectors. While the component represents the computational elements and data stores, the connectors represent the interactions among Components. Components’ interfaces are defined by a set of ports which identifies a point of interaction between the component and its environment. Computationally, connectors mediate the communication and coordination among Components thereby providing the compositional ‘glue’. Connectors also have interfaces that are defined by a set of roles which defines a participant of the interaction represented by the connector. The correspondence between the sub-System and its external interface is defined by its rep-map (representation map). Using these constructs, Garlan defines the compositional structure of a System as a hierarchical graph of components and connectors. It is evident that Garlan enables specification of computation (behaviour) Composition in the ACME Architecture description language. However the formal notion of composition is not defined in ACME and as in WRIGHT, other forms of composition like function Composition, object Composition, features Composition, state Composition are not supported in ACME.

According to Bruneton et al [84-85], a FRACTAL Component is a run-time entity that is encapsulated and has a distinct identity. In their FRACTAL Architecture Description Language, a composite Component is defined by specifying the interface it provides, the interface it requires, the sub-Components that it contains and the bindings between these sub-Components and the composite Component itself. A primitive component is defined by specifying the interfaces it provides, the interfaces it requires and the class that implements this component. Using these two component definitions, component architectures can be defined for the underlying FRACTAL component model. It is evident that FRACTAL is a programming language agnostic component model that can be used to design, implement and deploy different kinds of software systems. However, the formal notion of composition is not defined in FRACTAL while the underlying component model and supporting component framework supports implementation of component composition.

UML [31] and SysML [32], provide out-of-the-box modelling constructs for creating architecture models and views in the form of diagrams. The expectation is to use one or more of these diagrams to express the composition. While there is no out-of-the-box diagram to create a composition view of the components, it is indeed possible to use package diagrams to show the structure of the components at the level of packages (each package is a composition of a set of components). A typical package diagram is made up of the modelling constructs: package, packageable element, dependency, element import,
package import, and package merge. It is also possible to use appropriate UML [31] and SysML [32] stereotypes, to define new composition diagrams and views by extending existing constructs in the package diagrams or by deriving from package diagrams with new properties and constraints. A typical extension of the package diagram modelling constructs could be: component, composite, component import, composite import, component merge, and composite merge.

In CORBA [82], Composition is specified as a named scope that contains elements that constitute the composition. The elements of the composition specification are: specification of the life cycle category; an identifier that names the composition in the enclosing module scope; Composition body comprising of the mandatory home executor definition and an optional proxy home definition. A minimalistic schema of composition in CORBA is expressed in the form:

```plaintext
Composition <category> <composition_name> {
    home executor <home_executor_name> {
        implements <home_type>; 
        manages <executor_name>; 
    };
};
```

Where <composition_name> is the name of the composition, <category> identifies the life cycle category supported by the composition, <home_executor_name> is the name assigned to the generated home executor skeleton, <home_type> is the name of a component home type, and <executor_name> is the name assigned to the generated Component executor skeleton. Even though CORBA introduces a schema for composition, it is evident that this notion of composition is an implementation view and is relative to the CORBA component model and the underlying Component framework as a result of which the target software architecture is fixed.

### 3.5 Summary

This thesis considers Software architecture as the high level structures of software that is defined by a structural composition of components. This thesis considers a composition to comprise of a set of components and the logical/formal structure by which the components are interconnected. Further this thesis suggests that a composition specification can be built by application of compositional semantics on a set of component specifications to arrive at a new composite component specification.