IJCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Volume-3, Special Issue-1 E-ISSN: 2347-2693

The Intricacies of Software Component Composition

Maushumi Lahon*
Department of CE and Application
Assam Engineering Institute
Guwahati, India
maushumi28@gmail.com

Uzzal Sharma
Department of CS and IT
DBCET, Assam Don Bosco University
Guwahati, India
uzzal.sharma@dbuniversity.ac.in

Abstract - **Components in Component Based Software Engineering (CBSE) are designed for composition and so it is an acceptable fact that, if composition of components becomes more and more error free and easy, the market acceptance and use of component based systems would increase and finally the objective of CBSE could be achieved. As components are prepared by third parties, their composition to build a system according to the requirement specification is a big challenge in the field of CBSE. Many works have been initiated and carried out in this area and many more are yet to be done to bring this area to maturity. Literature review in this area throws light into the various aspects and details that need to be considered in composition. Literature also highlights many approaches to composition and their areas of implementation. The paper attempts to present a review on different aspects of composition by considering the works proposed by various researchers in different areas of composition. Towards the end of the paper an attempt is made to narrow down and concentrate more on the mismatching aspect of composition with a basic concept of the adaptation technique.**

Keywords – Component, Composition, Mismatch, Adaptation

I. INTRODUCTION TO COMPOSITION

As components are independent identities, every component has its own required and provided services. When software systems are designed by assembling the independent components the role of composition becomes very crucial in delivering the required system. According to [1] two entities – components and frameworks go for composition which gives three different classes of interaction: component – component, component – framework and framework – framework. Component – Component composition enables interaction between components and is considered as application level interaction, Component – framework composition enables interaction between the component and the framework and may be considered as system level interaction and Framework – Framework composition enables interaction between different frameworks and their interaction may be considered as interoperability interaction. This enables composition of components in heterogeneous environment. The Technical Report [1] classifies various compositional forms based on the deployment of component and framework. The classifications are -

- Component Deployment (C – F) - The deployment defines the interfaces such that the framework can use the resources of the component.
- Framework Deployment (F_1 – F_2) - The deployment enables one framework to use the resources of the other according to the tiered Frameworks proposed by Szyperski.
- Simple Composition ((C_1 – C_2) / F) - When the interaction is between two components deployed in the same framework.

- Heterogeneous Composition ($C_1/F_1 – C_2/F_2$) - When the interaction is between two components deployed in two different frameworks.
- Framework Extension ((C/F_1) – F_2) - Here frameworks are composed of components and deployed in a different framework which is normally in case of "plug-ins"
- Component (Sub) Assembly ($C_1/F – C_2C_3/ F$) - Here interaction is between components and assembly of components or interaction between the sub assemblies.

Jonge [2] in his work has used the term integration instead of composition and have identified the various situations in which composition takes place. According to the research, integration takes place in different points of time and can be summarised as Development-time, Pre-compile time, Compile time, Distribution-time and Run-time integration(composition). To support this view literature reveals that Lau [3] in his work has emphasised that composition must take place in both the design and deployment phase of a component life cycle and proposed a model in his work using the concept of invocation connectors and composition connectors.

Sametinger in his paper [4] had identified two basic forms of composition – internal composition and external composition. Internal composition denotes the process to compile and to link source code to a system. Textual composition (instantiation of macros and templates) counts also to this form of composition. External composition is where components act independently, i.e., they run on their own. Components may communicate by means of inter process communication (e.g., remote procedure calls).

Literature on composition reveals that component composition depends on the component models, component framework and component composition theory. So different types of components, frameworks and integration gives rise to different composition approaches. Different types of components i.e. black box, white box, grey box and glass box components can be composed using different approaches. Traditional approaches uses black box components for composition, whereas white box components do not have any composition approaches as the component itself can be modified according to the need. For reuse purpose grey box components are suitable as it contains some fixed parts and some modifiable part. They have increased adaptability and are suitable for reuse. Black box composition approaches includes modular composition, object oriented composition, component-based composition and architecture composition which are the traditional approaches [5]. Grey box composition approaches includes aspect oriented programming, composition approaches based on multi-dimensional separation of concerns, feature-oriented composition and invasive software composition.

II. PARADIGMS AND LANGUAGE IN COMPOSITION

Different views are presented in the literature with respect to the symmetric and asymmetric paradigms in composition. In symmetric paradigm the units for composition are considered to be of the same type i.e. the building blocks are identical and so there is no model more basic than the other. In case of asymmetric paradigm the concept of base model is necessary and this paradigm does not support component-component composition. Three kinds of entities are involved in the compositional paradigm which in turn leads to three kinds of symmetry. The entities are composable elements, join points and composition relationship [6]. Composable elements refers to those elements that are to be composed like objects, components etc., join points refers to those points where composition is possible and composition relationship refers to the rules that govern the composition. The entities lead to element, join-point and relationship symmetry. In case of element, symmetric paradigm deals with single kind of composable element whereas asymmetric paradigm deals with two or more types of composable elements. In case of join point entity, a symmetric join point is a point in an artifact or execution, such as a method definition or call, at which composition can occur whereas an asymmetric join point occurs when a method body or execution contains one or more explicit points for composition. In paradigms with both relationship and element asymmetry, relationships are placed only in aspects, and not in components. In paradigms with relationship symmetry, the relationships may be placed anywhere, but they have a scope that ranges across all the concerns being composed. To summarize it can be placed that both the paradigms have their own advantages and disadvantages and research is on to come up with combined paradigms using components and aspects and treating aspects as components.

Composition takes place with the help of some composition language. Traditional languages as well as scripting languages cannot fulfill the requirements of composition and therefore specific language for composition is necessary. IBM T. J. Watson Research Center [7] in their work had identified certain points which are briefly placed below. The points were identified with the knowledge content of the references [8] and [9] which had also discussed on identifying the necessary points to be present in a composition language.

- Composition operations which should be supported by composition language includes :
  - Binding of communication channels - to let components exchange data and invoke behaviour.
  - Creating higher level component aggregates - components are combined to produce higher order functional constructs.
  - Macro expansion of parameterized components - Macro expansion can be used in several ways to compose components.
  - Recursive component composition - Component composition is used to create new components, rather than an application.
- Language must provide a way to address "compositional mismatches" i.e. when direct composition is not possible.
- Language should support component frameworks.
- Learning process for the language must be easy.
- Eliminate the need for supporting tool so that the language can be used with minimum investment.
- Language should support common component models like COM, JavaBeans and CORBA.

III. RELATED WORKS AND DIVERSITIES IN THE AREA OF COMPOSITION

The area of composition includes works of different authors in different directions out of which incremental composition, dynamic composition, automatic composition, pi-calculus (a language for composition), composition based on ADL (Architectural Description Language), Source component composition are a few. A brief look into each of the area gives a feel about the domain of component composition.

Components can interact when all the services requested by one component can be provided by the other component. In cases when several components interact with a single provider the proposed concept of incremental composition [10] eliminates the need for exhaustive combinatorial checking of the inter-leavings. The author introduces simple conditions which must be satisfied by the interacting components for their composition to be incremental and illustrates the concepts using simple examples of interactions. The paper refers to the GenVoca model[11] which is based on simple syntactic expansions, models that resembles higher-order programming [12] and dynamic interconnections of distributed processes [13].

In the work under reference [14] a linking component language Knit is proposed to address the needs of componentized software. The language is especially designed for use with component kits, where standard linking tools provide inadequate support for component configuration. In particular, Knit is developed for use with the OSKit, a large collection of components for building low-level systems. This approach has some similarities with the GenVoca model [11] of software components but has many areas of differences. The contrast is that, Knit promotes the reuse of existing (C) code and enables flexible composition through its separate, unit-based linking language. Pi-calculus is another concept used to develop composition language. In particular Lumpe [15] in his work had proposed a variant of π–calculus which is referred to as πL-calculus as it is much easier to model compositional abstractions than it is possible in the plain π-calculus. Piccola, is a prototype composition language based on the πL-calculus.

Another area of composition is in the field of stream processing because of its diverse applications like audio/video, sensor data analysis etc. The challenge in this area is to provide optimal component composition i.e. to achieve load balancing with respect to multiple function, resource and quality of service. The work as presented in [16] in this area proposes an adaptive composition probing approach which first discovers a number of good candidate component compositions using a limited number of state collection probes and then selects the best component composition that achieves best load balancing from the discovered good candidate compositions. Component composition has been studied under different research context, such as service composition, systems software composition and multimedia application configuration whereas this work focuses on scalable optimal component composition which is important for distributed stream processing.

In the area of dynamic and automatic composition of components the authors in [17] is of the view that architecture analysis and code composition can form the basis for efficient and correct assembly of components. Architecture analysis is implemented in the Charmy tool [18] and code synthesis is implemented in the Synthesis tool [19]. The concept of composition through dynamic adaptation is discussed in the paper where certain definitions of systems based on number of components, connectors and component instances are used as given in [20].

Weakly-Closed System:	A Weakly-Closed System is a system with a fixed number of components.
Closed System:	A Closed System is a system with a fixed number of component instances and fixed connectors.
Weakly-Opened System:	A Weakly-Opened System is a system with variable number of component instances and with fixed connectors.
Opened System:	An Opened System is a system with variable connectors and number of component instances.

The type of system determines whether static or dynamic adaptation is applicable. In the area of dynamic composition Bruneton et.al [21] proposed the Fractal component model where the author claims the model to be a flexible and dynamic model using the concepts of components, interfaces and names. According to the definition a component is formed out of two parts controller and content, interfaces can be either server or client interfaces and names are symbols for denoting in the model.

IV. MISMATCHES IN COMPONENTS

Composition of components can be efficient only when the required services and provided services match. As components are designed and constructed by different parties it is apparent that there are mismatches in the required – provided combination necessary for developing a component based system. Addressing these mismatches in a systematic manner with adequate technique will ensure more versatile systems. In this area many authors have presented many views and techniques to address the issue of mismatch.

According to McIlroy the components may differ in many dimensions which may be grouped into two major sections: performance characteristics and interface details. Though they are not independent of each other, interface details contribute to the majority of differences. Basically interface mismatch can be considered as the area of concern to deal with the diversity and changes between the components. The report on Black Box Composition of mismatched components in [22] lists the following existing practices used to address the issue of interface mismatch -

- Library based development using standard libraries and application domain libraries
- Incorporation of an abstraction layer requiring a lot of anticipation
- Reimplementation in cases where there is a great degree of mismatch
- Non-implementation where developers simply avoid providing some desired functionality hence saving effort in implementation.
- Standardisation i.e. to rule out diversity and changes.
- Information hiding and modular programming where the developers consciously limit what interface details should be visible from other components, in order to restrict the area of interdependency between components. Modular reasoning is concerned with limiting the changes which can invalidate prior reasoning about a composition.
- Porting i.e. port code written for one environment to run in a different environment. This means editing the code to change the details that differ between the two interfaces which is a patch or fork.
- Automated source code transformation which are systematic source level transformations, designed to automate non-localised changes which are laborious and error-prone to perform by hand.

- Glue coding where the aim is to avoid modifying existing code, but instead to implement the desired interface by a thin layer of code which consumes an alternative interface provided by some available component.

Each of these approaches has its benefits and limitations in terms of cost, labour, complexity and maintenance issue. Many alternatives are designed to deal with the drawbacks of the different approaches.

To implement how to deal with mismatches, it is necessary to identify the mismatches. According to [23] there are various levels of mismatches: technical level, signature level, behavioural level, semantic level and service level. Each level of mismatch is addressed using different technique as in case of detecting behavioural mismatch LTS (Labelled Transition System) and STS (Symbolic Transition System)[24] are proposed in related works. There are other works on interfaces which suggests that considering only the functional characteristics of interfaces is not enough but architectural characteristics of interfaces also plays a very important role in the process of integration and thereby minimizing interface mismatch. In the related work [25] the author identifies the architectural characteristics which play a role in component integration. The architectural characteristics could be the joint points that can be addressed to minimize interface mismatches.

To deal with the mismatches literature reveals many techniques. Different authors have proposed different approaches and techniques and contribute to the area of CBSE. Hemer [26] in his work has pointed out that choosing the right component that partially fulfils user requirements and then adapting it to totally fulfill the user requirements is not addressed in previous works and so in his paper he proposes using templates from the CARE language (Hemer & Lindsay 2004, Lindsay & Hemer 1997) to define adaptation strategies for modifying and combining components and how specification matching strategies can be used to semi-automate the adaptation and composition process.

V. ADAPTATION – AN APPROACH TO OVERCOME MISMATCH

As it is already apparent from the literature that while building component based systems there are mismatches which need to be addressed to allow building of the system, it is very crucial to identify the mismatches and deal with them. Component adaptation - a technique which uses adaptors was proposed way back in 1997 by Yellin and Strom. In the paper component behaviour was represented by a Finite State Machine (FSM) as well as a formal introduction to adaptors were proposed which considered an adaptor to be a software which allows operation between two mismatching components. Bracciali et.al in their work [27] had proposed a formal method for behaviour adaptation with three important aspects of component interface, adaptor specification and adaptor derivation. The contribution is towards the definition of a methodology for the automatic development of adaptors that are capable of solving behavioural mismatches between heterogeneous interacting components.

Though there are many ways adaptors being used to address mismatches the authors in [28] argues that the approaches are ad-hoc approaches as they do not consider the other non-functional issues like cost, performance, security etc. and so proposes an engineering approach consisting of methods, best practices and tools to develop trustworthy component based systems. According to the paper under reference [28] software adaptation takes place in different phases of life cycle and they are termed as requirement adaptation, design-time adaptation and run-time adaptation and it also says that mismatches in components occur due to the mismatch in the interfaces of the components and therefore the study of underlying interface models can give a direction in dealing with the mismatches. Existing component object models like copy-paste, inheritance, aggregation and wrapping provide only limited support for component adaptation. These techniques suffer from problems related to reusability, efficiency, implementation overhead or the self problem. To address these problems, a novel black-box adaptation technique named superimposition is proposed [29] that allow the software engineer to adapt a component using a number of predefined adaptation behaviours that can be configured for the specific component. Finally, the software engineer may compose multiple adaptation behaviour types for a single component.

VI. CONCLUSION

It is apparent from the literature that there are many areas and issues that are addressed and need to be addressed in the domain of composition. Researchers are working in this field to make composition more and more acceptable and easy so that component based system gains popularity and in turn moves towards addressing the issue of software crisis. The different areas of composition are presented covering approaches, paradigms of symmetric and non-symmetric composition, composition language, mismatches and adaptation. Different approaches of black box, white box and grey box composition were found and requirements necessary to be incorporated in a composition language could also be found in the literature covered. Importance to deal with the mismatches is addressed in many of the papers and different approaches and techniques are proposed to handle different types of mismatches like behavioural mismatches, signature mismatches etc. that arises in different phases of the life cycle of component based systems. The different approaches presently used are identified and more and more techniques are being proposed in this area. The introduction of adaptors is projected to be a promising solution to deal with interface mismatch and reduce the differences in the services required and services provided by components, which eventually is a reliable step to popularise the concept of component based development and reuse.

REFERENCES

[1] “Technical Concepts of Component-Based Software Engineering”, Vol -II, CMU/SEI-2000-TR-008

[2] Jonge,M , “ To Reuse or To Be Reused, Techniques for Component Composition and Construction” Center for Mathematics and Computer Science (CWI) in Amsterdam under the auspices of the Research school IPA(Institute for Programming research and Algorithmics), January 2003

[3] Lau K K, Ling L and Elizondo P V, “Towards Composing Software Components in Both Design and Deployment Phases”,

[4] Sametinger J,” Software Engineering with Reusable Components”, Springer, 1997

[5] Soni P, Ratti N, “ Analysis of Component Composition Approaches”, International Journal of Computer Science and Communication Engineering, Volume 2 Issue 1, Feb’2013

[6] Harrison W, Ossher H and Tarr P, “Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition”, IBM Research Report, Dec’2002

[7] Curbera F, Weerawarana S and Duftler M J ,”On Component Composition Languages”, IBM T. J. Watson Research Center, April’2000

[8] Nierstrasz O and Meijler T D, “ Requirements for a composition language”, Object-Based Models and Languages for Concurrent Systems, Lecture Notes in Computer Science 924, Berlin, 1995. Springer.

[9] Nierstrasz O and Meijler T D, “ Research directions in software composition” , ACM Computing Surveys, 27(2), June 1995.

[10] Zuberek W.M, “Incremental Composition of Software Components”, Springer-Verlag 2011 (ISBN 978-3 -642-21392-2).

[11] Singhal ,B.D, Thosmas V, Dasari J, Geract S,Sirkin B, “The Gen Voca model of software system generators ” ,IEEE Software, vol.11, n.5,pp.89-94, 1994

[12] Bracha G, Cook W ,” Mixin-based inheritance”, Proc. Joint ACM Conf. on Object Oriented Programming, Systems , Languages and Applications and the European Conf. on Object-Oriented Programming, pp.303-311, 1990

[13] Magee J, Dulay N, Kramer J , “ Specifying distributed software architectures”, Proc. 5th European Software Engineering Conference, Sitges, Spain (Lecture Notes in Computer Science 989), pp.137-153, 1995

[14] Reid A, Flatt M, Stoller L, Lepreau J and Eide E , “ Knit: Component Composition for Systems Software”, Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000), pages 347–360, San Diego, CA, October 23–25, 2000.

[15] Lumpe M, “ A π-Calculus Based Approach for Software Composition”, January’99

[16] Xiaohui Gu, Philip S. Yu and Klara Nahrstedt, “Optimal Component Composition for Scalable Stream Processing ”

[17] Bucchiarone A, Polini A, Pelliccione P, Tivoli M, “Towards an architectural approach for the dynamic and automatic composition of software components”

[18] C. Project. Charmy web site. http://www.di.univaq.it/charmy, February 2004.

[19] Synthesis Project. Synthesis web site. http://www.di.univaq.it/tivoli/SYNTHESIS/synthesis.php, September 2004.

[20] Caporuscio M, Inverardi P, and Pelliccione P, “Formal analysis of architectural patterns” First European Workshop on Software Architecture-EWSA 2004, 21-22 May 2004, St Andrews, Scotland.

[21] Bruneton E , Coupaye T , Stefani J B , “Recursive and Dynamic Software Composition with Sharing”

[22] Kell S, “Black-box composition of mismatched software components”, Technical Report, http://www.cl.cam.ac.uk/, Dec’2013.

[23] Canal C, Poizat P, Salaun G. “Model-Based Adaptation of Behavioral Mismatching Components”, Software Engineering [J].IEEE Transactions on, 2008, 34(4):546-563.

[24] Qureshi M R Z , Alomari E A, “Validation of Novel Approach to Detect Type Mismatch Problem Using Component Based Development”, Modern Education and Computer Science Press, http://www.mecs-press.org/, Aug’13

[25] Alkazemi B Y, “A Precise Characterization of Software Component Interfaces”, Journal Of Software, Vol. 6, No. 3, March’11

[26] Hemer D, “A Formal Approach to Component Adaptation and Composition”, The 28th Australasian Computer Science Conference , 2005

[27] Bracciali A, Brogi A, Canal C , “ A formal approach to component adaptation”, The Journal of Systems and Software 74 (2005) 45–54, 2003

[28] Becker, S., Brogi, A., Gorton, I., Overhage, S., Romanovsky, A., Tivoli, M, “Towards an Engineering Approach to Component Adaptation”, Technical Report Series No. CS-TR-939 ‚January’ 2006

[29] Bosch J, “Superimposition: A Component Adaptation Technique”, University of Karlskrona