©2017 Published in 5th International Symposium on Innovative Technologies in Engineering and Science 29-30 September 2017 (ISITES2017 Baku - Azerbaijan)

# Composition Capability of Component-Oriented Development

*M. Cagri Kaya, Anil Cetinkaya and Ali H. Dogru
Department of Computer Engineering, Middle East Technical University, Ankara, Turkey

## Abstract

This research enhances component-oriented development approaches with the capability to represent the dynamic behavior of the final system through a process model. For an executable system, ordering of the message invocations should also be specified besides the definition of a set of components which only presents a static view. Components, however, are usually server kind software units that respond when a request is made. A central application can be expected to trigger some of the methods while some components will make further requests by themselves, from other components. While providing means for both kinds of communications, the main application should be able to be modelled and executed through the specification of a central process that coordinates the timing and direction of the messages. Once an activation is started from the central process model, a connector assumes further duties in the coordination of the involved interaction. Other capabilities such as adaptation are also incorporated besides the synchronization duties. The considerations in this study have been not to modify existing component models and almost no code writing for integration. Process models can be graphically specified as yet, interpreted leveraging on their capability to invoke external functions. Suggested architecture connects components only to the central process: if interaction is required between two components, that is also managed through the process model.

**Key words:** component, composition, connector, coordination, process model

## 1. Introduction

The age old struggle for providing leverage to software developers continues with a variety of techniques, each seeking extraordinary improvement in the development efficiency. It is possible to generalize most of the modern approaches as “compositional” techniques. Recent emphasis in software architecture, supported with domain specific development and also model driven approaches support the production of code in large scales, with less coding effort. Such advances were necessary considering the ever increasing demand for software that has displayed exponential characteristics. It is impossible to supply this demand through manually produced code.

Some of the approaches in this direction, can be listed as Software Product Lines (SPLs), Service-Oriented Architecture (SOA), Model-Driven Development (MDD) and other software architecture centric techniques such as component-based and component-oriented development. Many projects employ a mixture of the techniques offered by these approaches. This research also exploits the variability management concept as inspired from the SPL approaches, process

*Corresponding author: Address: Department of Computer Engineering, Middle East Technical University, 06800, Ankara TURKEY. E-mail address: mckaya@ceng.metu.edu.tr, Phone: +903122105597

model-based composition as suggested in service-oriented approaches, model-based techniques for the automation of the configuration phase, as well as the component-orientation [1] foundation. This foundation devotes itself completely to development by integration rather than code writing: from the idea formation to executable code, the system is modeled utilizing the component concept – in abstractions in the earlier phases and as code units in the later phases.

System development is now regarded as comprising two major activities: locating components and integrating them. The first activity results in a static view where an analogy can be made to a program having a set of functions to be called, but there is no main program to call them in an order for the correct execution. The second activity, integration is the specification of the order of invocations that is supported with state management. However, since the goal is not writing code but reusing existing components, the integration specification is preferred to be a graphical modeling of a flow specification: there are established tools that enable process modeling that yield a graphical model capable of execution where nodes in the graph corresponding to executable units that can call component methods. Figure 1 depicts the 2-level structure of the proposed reference architecture. Further decomposition of those levels is not prohibited but the main model should be represented mostly in this context: a global flow ordering and a set of executable functions. A former study [2] laid out the foundation for incorporating connectors for integration. However, components were expected to be modified in that work, possibly requiring the establishment of a new component model for serious utilization. In this article, existing components can be used without modification by means of allocatıng all the responsibility in the new connector structures. The central process organizes the necessary coordination using the connectors. Components are only incorporated for service requests.

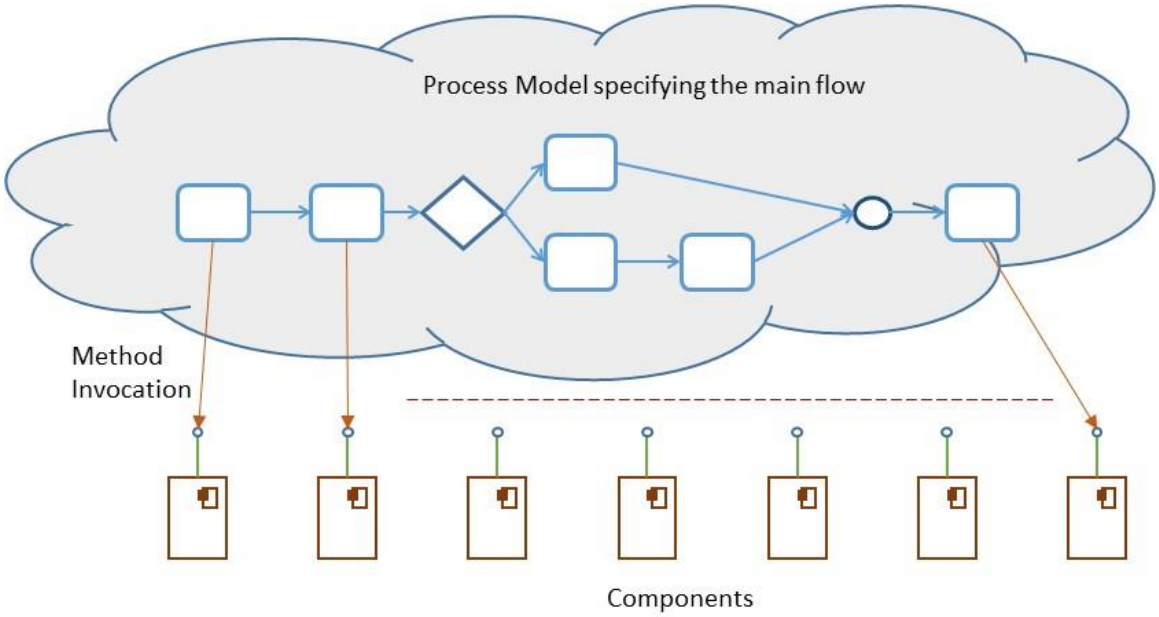


**Figure 1.** Abstract reference architecture for composition

The suggested composition mechanism is based on many existing concepts that are explained in the remaining part of the paper. Among those, connectors are loaded with extra capabilities

considering the usual expectations from them.

In the following sections background information and proposed approach are provided. Then, the proposed approach is described with a case study and some ideas are discussed in the discussion section.

## 2. Background

This section provides some background information about existing concepts. These include structural software units as well as approaches.

### *2.1. Software Components and Connectors*

Components have been the fundamental blocks for software architecture that can be broadly defined as representing a software intensive system by a set of major parts and their connections. Naturally the parts and connections are represented by components and connectors. Components have been offered for efficient reuse, and developed in accordance to a component protocol [3]. They have well defined properties such as the ability to integrate at run-time, besides the platform independence etc. Following their introduction, many component-based development approaches have been studied but as yet, none could become as popular as more general approaches such as object-oriented development. As application domains mature and get populated with reliable components corresponding development techniques should prevail.

Connectors, however have not been specified much. They were only supported as concepts in abstract models, or left to the developers' preferences to be completely defined during implementation. Integrating components hence required some code writing. A recent study [4] categorized the connectors and assigned types based on wide usage. This supported our expectations about reusable connectors to be utilized in the integration of components without code writing. Connectors are defined with abilities to serve such classification with message transition processing capabilities.

### *2.2. Component-Based and Component-Oriented Software Development*

Components are the elementary building blocks to be utilized in "development by integration". Many approaches have surfaced to provide component models for the purpose [5]. Components are represented with their published and sometimes required interfaces. The models help in representing components and their integration; some support late switching of alternative components, and configuration of the system based on sets of components. It is possible to configure the internals of the components if the component and the environment agree.

Component-based development approaches are usually emphasizing the implementation-level issues or they follow established system development techniques that also accommodate components in the composition. The most widely used object-oriented model representation, Unified Modeling Language (UML) [6] for example, is supported by tools that rightfully claim to

be object-oriented, as well as component-based. However, in component-orientation, objects or other structures are not referenced. It is assumed that there is a mature domain where there are sufficient number of tested components and engineering know-how has been established in the industry for composing different applications having component definitions in mind. Inspired by the SPLs we can make an analogy to the domain model, where possible requirements analysis for any new project already exists in parts that can be combined. Therefore, the separation of "what to build" and "how to build" has already been exercised at an earlier time for the project. Now, immediately after the request for the next product, engineers should emphasize mapping quickly arranged requirements to existing code – that is components. As a result, the development starts with modeling that considers components in abstraction and continues modeling with components in their implemented states. The developers are only and completely oriented to components, no other structures.

### *2.3. Variability Management*

Emerged during the SPL related work, this capability works after the organization of the product features in a given application domain, as common and variable features. Variability management starts with representation in the domain model, continues with variability elimination in different phases such as design time or run time etc., and preferred to guide the configuration of the product. Resolution of variability eventually defines a product, that has the determined features as well as the common features for the domain. As in the more general concept of late-binding, the later the resolution, the bigger the power.

The early approaches supported the variability representation as it exists in the starting asset that is the domain feature model. Later it was discovered that it becomes very complex to investigate variability as superimposed on a feature model, that grows to huge complexities in realistic industrial applications. Independent variability models were offered and are being successfully used in software development, such as OVM [7] and Covamof [8].

### *2.4. Service Composition*

Service composition can be described as activities to bring existing services together in order to have a composite service. Two main types of service composition are *orchestration* and *choreography* [9]. While orchestration represents a centralized perspective, choreography corresponds to a distributed viewpoint. Being a centralized approach, orchestration can describe the behavior of the composition, and this results with an executable model. An orchestrator deals with the coordination of interaction among services in the composition. However, composition logic is distributed in participating services in choreography. This is definition of a protocol that all participating partners must conform to and implement in their services. Therefore, in general choreography is not an executable model [10].

## 3. Proposed Approach: Variable Connectors

Integrating components is a capability added to the Component-Oriented Software Engineering (COSE) through a recently developed language - XCOSEML [11]. Later, connectors are enhanced with their internal operations [12] to process a message that should be orchestrated between two ports. In this research, one side of a message communication is always the central process model, behaving as a component and invoking the published methods of a real component. A central control is established in the collaboration logic, controlling and coordinating the message traffic towards executing the system functionalities.

The duties listed in the work for classifying connectors [4] can be interpreted as requirements on connectors. We dispatched such duties to the connectors. This decision was for providing "separation of concerns" where components should only contain their functional duties, the process model should concentrate on the coordination, whereas the issues about connections and adaptations due to incompatibilities among components are assigned to connectors.

As a result, connectors came close to components from the point of view of reusing existing elements. Supported by the variability mechanism that provides configuration so that a connector type can be utilized in many different systems serving the small variety they may require, different per system. Of course different versions of a connector type can be utilized at different points in one system also.

The interesting additional capability a connector carries, is the synchronization of the central process and a component's method. To establish the execution logic, components are triggered by the central process model, whenever their time comes. This is achieved by the central process starting the transition by ordering the connector. The connector invokes the method in the component, before and after providing necessary conversions and returns the result to the executing process model. The current implementation of our solution requires a separate connector between the central process component and any other component for the invocation of any method.

### *3.1. XCOSEML: Component-Oriented Specification with Variability*

This language was developed to support the COSE approach so that further machine processing would be aided, such as verification. Also variability was added, and finally the enhanced connectors are incorporated.

XCOSEML's variability mechanism is inspired both from OVM and Covamof. Variability model and system model are handled separately in XCOSEML. The language is developed as a text-based language unlike its predecessor – COSEML. Six constructs are defined for the language: *package*, *component*, *connector*, *interface*, *configuration interface*, and *composition specification*. Packages represent logical entities in the model and they are implemented by physical components. Functionality of components are shown by their interfaces in terms of methods. The older abstract definition of connectors is extended by adding further capabilities

into them in [12]. Connectors contain the messaging details of the communication: requester and responder interfaces and their corresponding methods. Besides, they can contain some operations on data, such as unit conversion. Configuration interface contains the variability model of the language. It represents variation points, variants, and dependencies between variants, if existing. Variation points and variants are shown in the composition specification by using tags. System configuration in XCOSEML is performed through variant selections in the composition file. When a variation point is bound to a specific variant, corresponding interactions in the composition specification are executed in the final system.

## 4. Case Study

A scenario for withdrawing money from an Automated Teller Machine (ATM) is articulated in this section for demonstrating the coordination duties of connectors. A bank customer requests to withdraw some money from the ATM. First the customer is verified for security purposes, using the bank card and the password. Then the customer's account is checked if there is enough balance to allow the requested withdrawal. Finally, the local money storage is queried to check if there is enough cash in the machine.

Due to business requirements, the operations need to be conducted in the specified order. It can be assumed that the target of all these mentioned queries are implemented as components and the checks are conducted through method calls. First observation is, the components themselves should not determine when to activate the method calls – otherwise there needs to be application specific code to be included in them that disqualifies them from being reusable components, turning them into "modules" of the specific application. This example does not include adaptation functions except for the case where there could be synchronization mismatches between the process and the component, where the connector may have to employ a store and forward mechanism.

The application will execute by the processes invoking the related connector for the user verification request. After processing the returned result, another connector that is connected to the target component for querying the balance will be invoked, again by the central process. The execution will continue in this fashion. Figure 2 represents a Sequence Diagram where the central process is represented as an object as well as the Money Bin, and the Accounts Database, so that a sequence diagram can model the interactions among these components. Actually, these units are assumed to be components comprising the ATM system.

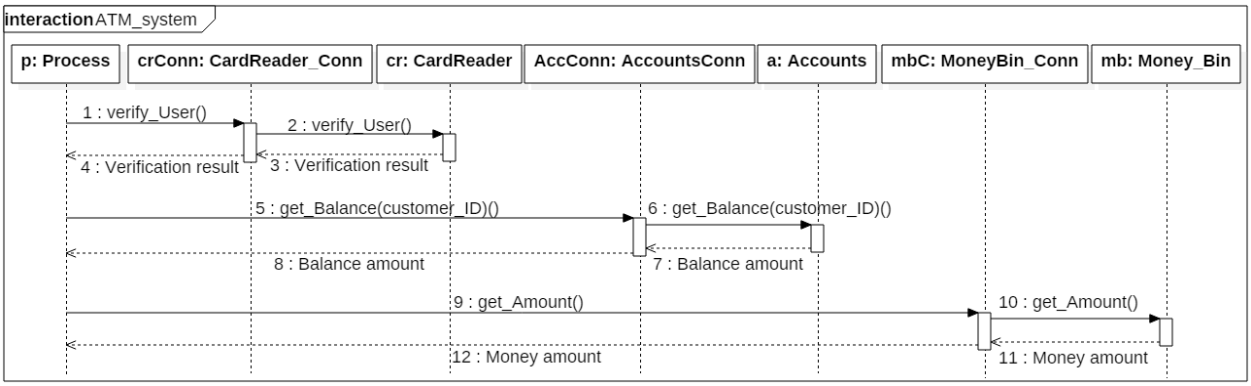


**Figure 2.** Sequence diagram for the ATM system.

In another example a fire extinguisher needs to be started if the temperature is above some value. Here the temperature sensor is a component that provides the result in Fahrenheit and the result needs to be converted to Celsius before the system forwards it to the other component: the extinguisher. In this simple example, the conversion between Fahrenheit and Celsius is assigned to the connector in order to free the other units from adaptation details. Although the two examples demonstrated coordination and adaptation separately, in general, the connectors may need both of the capabilities.

## 4. Discussions

A central control has been established for coordinating the message traffic in an effort to integrate components into an application. The reference architecture emphasizes a two-level hierarchy in the structural units that will be connected for interactions. However, the connection between the central component that is enacting the process for the overall control and other components employs a connector. Any other pairs of components would require a connector and the process is also implemented as a connector. Therefore, a detailed view where further levels in the hierarchy are possible. A component can act proactive and invoke other components methods, or as a request coming from the central process requires, the component trying to respond, may need service from other components. These options are also available. However, an emphasis on the two-level organization is supported by the authors due to the expectation that mature domains will enable this kind of an architecture that will end up with simple and consistent designs.

On the other hand, a pro-active behavior by a component that is targeting a notification to the central process is not compliant at first look. However, besides the option to allow exceptions, this kind of a communication that suggests “interrupt” mode whereas the suggested mode is pole-based, it is possible to accommodate bottom-up requests within the general behavior. In that case, the connector would assume a different kind of synchronization: The center can make its request and the connector will wait for the component’s request. Then response to the center’s request can be processed. In such a scenario, the later arriving request between the center and the components, will start the action. Consequently, this “interrupt” can be served in a poll-based manner, when the center is ready.

The suggested architecture is for general information systems. If there are hard real-time problems, the top-down controlled synchronization and repeating of messages due to the connectors inserted between the communicating parties, this approach may prove inefficient. In such cases, if possible, eliminating the connectors or loading them with different synchronization functions may serve the purpose.

Composition specification of XCOSEML was inspired from XChor – a variability-intensive choreography language for SOA [13]. This work is an effort towards supporting component-oriented development with executable compositions. Therefore, it can be said that the category of the XCOSEML's composition specification has shifted from choreography to orchestration by accepting a centralized and executable approach.

## Conclusions

A process modeling approach has been proposed as the central control unit for the coordination of components, to provide a consistent integration mechanism. Suggested approach will enable development of big applications to a great extent without code writing. However, some requirements may render the approach inefficient. Through investigations on example modeling and development studies it has been observed that integration of components is possible without code writing. The approach is in need for powerful tools. Existing tools include graphical model builders for component-oriented development, text-based tools for specifying compositions, and transforming facilities for verification of the models through existing tools. Although the integration effort is achieved through text-based specifications, it is not difficult to directly use the graphical model for the purpose- that is the most attractive next step in our future work. Different communication models can be addressed in further future work so that the approach is not only for a top-down coordination with a central control unit.

## References

[1] Dogru AH, Tanik MM. A process model for component-oriented software engineering. IEEE software, 20(2):34–41, 2003.
[2] Çetinkaya A, Karamanlıoğlu A, Kaya MÇ, Doğru AH. Eşgüdümlü Bileşenler ile Birleştirme Yaklaşımı, accepted for presentation in Ulusal Yazılım Mühendisliği Sempozyumu, Antlaya, Turkey, October 18-20 2017.
[3] Szyperski C, Dominik G, Stephan M. Component software: Beyond object-oriented programming. Addison-Wesley; 1998.
[4] Mehta NR, Medvidovic N, Phadke S. Towards a taxonomy of software connectors. In: Proceedings of the 22nd international conference on Software engineering 2000; 178-187.
[5] Crnkovic I, Sentilles S, Vulgarakis A, Chaudron MRV. A Classification Framework for Software Component Models. IEEE Transactions on Software Engineering 2011; 37, 593-615.
[6] UML 2.0, Object Management Group. http://www.omg.org/spec/UML/2.0/; last accessed: 6/20/2017.

[7] Pohl K, Böckle G, van Der Linden FJ. Software product line engineering: foundations, principles and techniques. Springer Science and Business Media 2005.
[8] Sinnema M, Deelstra S, Nijhuis J, Bosch J. Covamof: A framework for modeling variability in software product families. In Software product lines, Springer 2004, 197-213.
[9] Peltz C. Web services orchestration and choreography. *Computer* 36.10 (2003): 46-52.
[10] Lemos AL, Daniel F, Benatallah B. Web service composition: a survey of techniques and tools. ACM Computing Surveys (CSUR) 48.3 (2016): 33.
[11] Kaya MC, Suloglu S, Dogru AH. Variability Modeling in Component Oriented System Engineering. In The 19th International Conference on Transformative Science and Engineering, Business and Social Innovation, Kuching, Sarawak, Malaysia 2014.
[12] Cetinkaya A, Kaya MC, Dogru AH. Enhancing XCOSEML with Connector Variability for Component Oriented Development. In Proceedings of the 2016 Society for Design and Process Science, 2016.
[13] Süloğlu S. Model-Driven Variability Management in Choreography Specification. PhD thesis, Middle East Technical University, 2013.