A Framework for Dynamic Updating in Component-based Software Systems

Saleh Alhazbi
Computer Science Dept.
Qatar University
Doha -Qatar
salhazbi@qu.edu.qa

Aman Jantan
School Of Computer Science,
Universiti Sains Malaysia
11800, Penang, Malaysia
aman@ cs. usm.my

Abstract

This paper presents DPICS framework for composing component-based software system that can be modified at runtime. It is based on message-pattern interaction between system's components, which facilitates adding, removing, or replacing a component while the whole system is running.

Keywords: *component-based systems, dynamic updating, message-based interaction, protocol.*

1. Introduction

Lately, component-based development (CBD) is being increasingly adopted as a mainstream approach to software systems development. In CBD, software system is build by integrating pre-built, pre-tested components where these components might be developed locally or Commercial-off-the-shelf (COTS).

Component-based software systems, like any software, need to be updated over time for different reasons such as fixing bugs, upgrading its components, or adapting the system in response to its environment's changes. Traditionally, software modifications require shutting down the system, update the system, and restarting it. This approach is not suitable for critical systems that require 24/ 7/365 availability, such as banking or telecommunications systems, or systems that are critical-mission systems such as air-traffic controllers. Therefore, such systems require dynamic updating which means update software at run-time without service interruption. Generally updating software systems includes adding, deleting or replacing one or more of its components while the whole system is running. Dynamic updating has the same meaning with online evolution, on-the-fly adaptation, and software hot swapping. Because the component-based systems seem to have highly modularity, as a result of component-based design, they are relatively well suited for dynamic updating. The main goal of dynamic updating in component-based software systems is similar in concept to updating the hardware component while the system is running. However, the key difference that the new component may not be the same as the old component regarding the functional and performance [1].

Building dynamically updateable software systems is not a new area of research. There are many approaches range from redundant hardware to software-based ones. Hardware-based technique has been used with critical systems since a long time. In hardware-based solution, there is a redundant unit that works as a backup. When there is a need for upgrading the system, the backup unit handles the requests as an alternative to the other one; the system can be upgraded n the original unit while the backup unit is running. When upgrading is finished, the system running is switched back to the original unit. On the other hand, there are also many software-based approaches for building software systems that can be updated at runtime, those approaches can be categorized according to the unit of updating: procedure, class, and component. We presented a review for such approaches in [2]. In this paper, we present our frame work for building dynamic protocol-based interaction component-based system(DPICS) that can be modified at runtime.

2. Background

In this section we briefly clarify the definition of the term component, and discuss different patterns of components interactions.

2.1. components

Generally, a software component is an independent software unit that can be used to build bigger systems and usually it is a black box with a well defined interface. More formally Szyperski [3] defines a software component from a structural perspective as "a unit of composition with contractually specified interfaces and explicitly context dependencies only. A software component can be deployed independently and is subject to composition by third parties." Brown[4] defines a

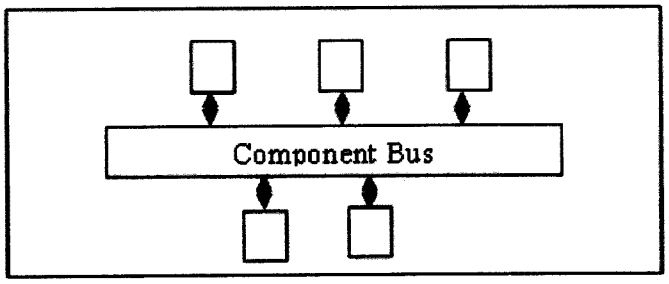

Figure 1

component as "an independently deliverable piece of functionality providing access to the services through interfaces." The definitions of components emphasize on plug and play concept so that software can be composed with components like hardware.

2.2. Component Interactions

In component-based software systems, the functionalities are not performed within one component; it is done by interacting, cooperating between system's components. Usually, a group of components depend on each other to perform a complex functionality of the system [5]. Dependencies between components can be defined as the reliance of a component on other(s) to support a specific functionality or configuration.
Assembling a system composed of reusable components can be achieved through different patterns to specify components communications style. These patterns include consumer-produces, component glue, and message-based through component bus [6]. From architecture prospective, interaction between components can be achieved either implicitly or via connectors [7]. Connectors are architectural building blocks used to model interactions among components and rules that govern those interactions. Unlike components, connectors might not correspond to compilation units in implemented systems [8]. Explicit connectors also make the bindings between components more loosely coupled; as a result, it increases reusability and reduces dependencies among components which supports faster and better component evolution [9].
In message-based interaction style, components communicate with each other by sending and receiving messages. The components of the systems are hooked together to one special component which represents the bus for routing the messages between the components (Figure 1).
However, such a style needs to define an interaction protocol that not only specify interface required for components to interact, but also specifies all rules, formats, and procedures that have been agreed upon between components [10].

3. DPICS Framework

In this section, we present the architecture of our framework DPICS that facilitates updating systems' components during run-time. Our framework is based on message interaction style between components, where this interaction is governed by predefined protocol. Components send/receive messages through a soft bus to provide the functionalities of the system. Additionally, each component is hooked to the soft bus through a connector to facilitate message exchanges. A permanent component in the framework called Update Manager (UM) is responsible for updating a component in the system during run-time. Figure 2 depicts DPICS architecture.

3.1. DPICS Components

In our framework, components are the locus of computation. They are service providers and consumers. They cooperate to provide system's functionalities. Any tow components can only communicate if they are syntactically compatible. Compatibility can be described as the ability of two objects to work properly together if connected, i.e. that all exchanged messages and data between them are understood by each other [11]. In our framework, each component has an xml-based description file that describes its interface; this description includes services provided/required by the component and signature of each service. Moreover, description file contains the attributes of the component that represent its state. Figure 3 illustrates an example of xml file to describe a component *Search_Com* that provides a service *BinarySearch* which returns an integer and has an integer parameter. This component requires a service *sort* that takes an array of integers as argument, and returns an array of integers (sorted).

3.2 Connectors

Connectors in our framework are not computation parts of the system, they just facilitate components interaction. Each component in DPICS communicates with other components in the system

through a connector which hooks the component up to the bus. Each connector represents the gateway between the component and the bus. We have two types of connectors Out-port and In-port, Out-port connector masks the services provided by the component, therefore this connector has the same methods as the component behind it. The task of this type is to interpret incoming messages

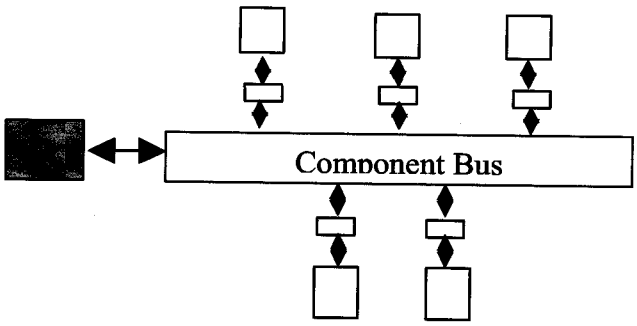

Figure 2

according to the protocol and call the actual service from the component. Those connectors have the ability to buffer incoming messages when the component is busy. On the other hand, out-port connector represents the gateway for the service required by the component, its task is to set-up outgoing messages according to the protocol. Besides those tasks, both types adapt incompatible messages.

3.3 Interaction Protocol

Basically, a protocol is a convention or standard that controls or enables the connection, communication, and data transfer between two computing endpoints. In its simplest form, a protocol can be defined as the rules governing the syntax, semantics, and synchronization of communication [12]. The purpose of DPICS protocol is to specify communication style, format, and rules between system components. Components communicate each other asking for services or providing results. This interaction in DPICS is message-based where we assume there is no duplicated or loses messages.

3.3.1 Messages

Our protocol defines three types of messages: Request message (RQ), Response message (RS), and Failure message (FM). Every message contains two parts: a message part (such as service required, service arguments), and a control part (such as message ID, message type).

1) Request message (RQ): this message is sent from a component to another asking for one of its provided services. The message is six tuple < Message type, Receiver ID, Service, no of arguments, arguments, sender>

```
component>
 <name>Search_Com </name>
 <provide>
   <service>
         <name>BinarySearch</name>
          <return>int</return>
           <arg>int</arg>
     </service>
</provide>
<required>
   <service>
       <name>Sort</name>
       <return>int [ ]  </return>
        <arg>int[ ]</arg>
</required>
<state>
  <data>
     <name> arrayInt</name>
     <type>int []</type>
     <set>setArrayInt</set>
      <get>getArrayInt</get>
  </data>
 <state>
</component>
```

Figure 3

Response Message(RS): this message is sent as a successful response to a previous request, it carries the result back to the sender of the request. This message format is five tuple <Message type, Receiver ID, Result, Sender>, even thought the service might not returns any result, an RS message should send back to the requester component. RS considered as acknowledgment message of finishing the process.

Failure Message (FM): this message is sent as a unsuccessful response to a previous request. Mainly this message is sent back to a component because of runtime error. This message is four tuple <Message type, Receiver, Error, Sender>.

3.3.2 Procedure Rules

The procedure rules of our protocol are described as follows:

- During runtime, each component has three states: idle, busy, and frozen.
- The message interaction in DPICS is synchronous, which means when a component sends a request message, it enters to busy state waiting for RS or FM.
- When a component receives an RM while it is in busy or frozen states, the message is buffered.
- RM messages are buffered as first-in first-out(FIFO).

4. Dynamic Updating in DPICS

In this section, we describe how our framework facilitates updating the system at runtime. This modification includes adding a new component, disconnecting a component, or replacing a component with a new version.

4.1 Adding a New Component

To add a new component to the running system, UM first checks that required services by this new component are available in the system. Then the component is instantiated, assigned a new unique ID, and the component is hooked to the soft bus through connectors. Adding a new component might not be useful for the overall functionality of the system unless there is another component that uses its services which might require replacing another component.

4.2 Removing a Component

Removing a component form a running system is straightforward and only requires UM disconnecting that component from the soft bus. Only idle components can be removed from the system in order to keep system running consistently. When removing a component, it is assumed that we only remove components that are no longer used by any of the components in the system.

4.3 Replacing a Component

When a component C wanted to be updated, Update Manager (UM) first checks the compatibility with the new version. Actually we have two types of component updating: implementation, interface. In implementation update, the interface of the new version does not change and this would not affect other components in the system. In interface updating, the compatibility with other dependant components in the system will be broken which requires a different connector to adapt the new interface. After compatibility affirmation, UM checks the status of the component. If the component is busy, upgrade will not be initiated because that would interrupt some processing which might affect system safety. If the component is in idle states, a message is sent to transfer the component into frozen state. While the component in frozen state, all incoming message to that components are buffered. During that, the new component is connected to the bus with a state as frozen. In order to keep system consistency, state of old component is transferred to the new version, new version starts its task by responding to all buffered messages if there are any where its state changed to busy until it finishes all buffered messages, its state become idle.

4.4 State Transfer

In order to keep the application consistent, the state of old component should be transferred to new one. We mean by state of a component the values of its instance variables. The relationship between the old version and the new one determined at by the programmer as it is not trivial to map instance variables of old class to those in new one. UM uses the xml description file to specify component state variables and how to set/get them. Although there are tools that help to build such function, it can not be fully automated. For example if the old class represents the triangle by three points and the new one represents it as two lines and an angle, one can never expect a software tool to find this relationship fully automatically[13].

5. Related work

In this paper, we presented our framework DPICS for building dynamic updateable component-based system which provides clear separation of concerns between the functionality of the system and the problem of being dynamically updateable. This separation is achieved through the connectors and the soft bus of the system, where the connectors are responsible of interpreting in and out messages of the components which represent the computational part of the system. SOFA/DCUP [14, 15] is related to our work where a component is divided into a permanent part and a replaceable part. The permanent part contains a Component Manager(CM) and wrappers of the component. The replaceable part contains a Component Builder (CB), functional objects, and subcomponents of the component. The application in SOFA/DCUP is a tree-like hierarchy of nested components, therefore if there is an update for a component in the top level, it requires all the application to be redeployed. On the contrary, our framework does not require components to be developed with evolution in mind.

The concept of using soft bus to connect system's components is discussed by Cheng [16] as an important role facilitates hardware reconfiguration and upgradeability and could be used similarly with software. C2 architecture style [17, 18] is similar to our framework, where components in the system are completely unaware of each other as they communicate through asynchronous messages exchanged by connectors. The difference with our framework is that we have one more level of separation by using the soft bus to connect the connectors rather than direct links between connectors. This allows for dynamic component

interface updating which requires connector modifying.

6. Conclusion and Future work.

This paper proposes a framework for developing component-based system that can be modified at runtime (DPICS). The main concept in our framework is to separate the computation part of the system from interaction by using connectors that exchange messages through another component represent the soft bus for the system. The messages interaction is administrated by predefined protocol that specifies message format, types, and rules. The future work in our framework includes investigating performance overhead that might result of using indirect connection (connectors + bus) between components.

References

1. Deepak G. On-line Software Version Change .PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, November 1994.
2. Alhazbi S., and Jantan A. , Hot Swapping in Component-Based Software Systems: State of the Art, Asian Journal of Information Technology (AJIT) 5(7):767-771,2006
3. C.Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley,1999
4. Alan W. Brwan. "Background information on CBD", SIGPC,18(1), August 1997.
5. Vieira M. , et al. "Describing Dependency at Component Access Point", Proc. of Work-shop on Component-based Software Engineering (at ICSE 2001), Toronto, Canada, May 2001.
6. P.Eskelin, "Component Interaction Patterns", on line Proc,6th Annual Conference on the pattern languages of programs(Plop99) 1999.
7. Balek, D., and Plasil, F. Software connectors and their role in component deployment. In Proceedings of the IFIP TC6 / WG6.1 Third International Working Conference on New Developments in Distributed Applications and Interoperable Systems (2001), Kluwer, B.V., pp. 69{84}.
19. 1997 Symposium on Software Reusability (SSR'97), pages 190-198, Boston, MA, May 17-19, 1997
8. Nenad M. and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In M. Jazayeri and H. Schauer, editors, Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97), pages 60–76. Springer–Verlag, 1997.
9. Tansalarak N. and Claypool K., CoCo: Composition Model and Composition Model Implementation In the 7th International Conference on Enterprise Information Systems. (May 24-28, 2005)
10. Holzmann G. , Design and Validation of Computer Protocols, Prentice Hall in November 1990
11. Vallecillo A, Herandez J, and Troya J., Component Interoperability, Tech. Rep. ITI-2000-37, Dept. de lenguajes Ciencias de la computación, University of Málaga, July 2000.
12. http://en.wikipedia.org
13. Yves V. and Yolande B., Component state mapping for runtime evolution, June, 2005, In Proceedings of the 2005 International Conference on Programming Languages and Compilers pg. 230--236 (Las Vegas, Nevada, USA)
14. Plasil, F., Balek, D., Janecek, R, "DCUP:Dynamic Component Updating in Java/CORBA Environment", ech. Report No. 97/10, Dep. Of SW Engineering, Charles University, Prague, 1997.
15. F. Plasil , D. Balek, R. Janecek "SOFA/DCUP: Architecture for Component Trading and Dynamic Updating", in the proceedings of ICCDS'98, Annapolis, Maryland, USA, IEEE CS Press, May 1998.
16. Cheng J., "Soft System Bus as a Future Software Technology," Proc. 8th International Symposium on Future Software Technology, Xi'an, China, SEA, October 2004.
17. Taylor R. , Medvidovic N., Anderson K., E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D.L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. IEEE Transactions on Software Engineering, vol. 22, no. 6, pages 390-406 (June 1996)
18. Medvidovic N. , Oreizy P., and Taylor R., "Reuse of Off-the-Shelf Components in C2-Style Architectures." In Proceedings of the