Safe Runtime Reconfiguration in Component-Based Software Systems

Saleh Alhazbi¹, and Aman Jantan²
¹Computer Science Dept., Qatar University, Doha, Qatar
²School of Computer Science, Universiti Sains Malaysia, Penang, Malaysia

Abstract - Long life and critical software systems require high availability. Therefore, their reconfiguration should be achieved dynamically during their running time. This type of updating is not safe as static update because it lacks test phase in order to validate the modifications. This paper discusses different sources of risk with runtime reconfiguring, it presents Message-based Interaction in Component-based System (MICS) framework that facilitates updating the system online, and ensures system safety during and after system modification.

Keywords: Component-based, Dynamic Updating, Message-based Interaction, Dependency.

1 Introduction

Traditionally, software reconfiguration requires shutting down the system, applying new modifications, and restarting the system. This approach is not suitable for long-life and high-available systems, such as banking, telecommunication, air traffic controls systems, which require reconfiguration without service interruption. Runtime reconfiguration or dynamic updating implies modifying the system while it is running. In component-based software systems, dynamic update includes adding, deleting, or replacing one of its components while at least one component is running. Several approaches [1-4] have been proposed for developing dynamically component-based systems. The center attention of those techniques was on how to enable dynamic update without much attention to system safety during and after updating. We define system safety in context of dynamic updating as system's ability to work consistently during updating operation and after that. The remaining of this paper is organized as follows. In section 2, we discuss the sources of risk when updating the system dynamically. In section 3, MICS framework is presented for composing component-based systems. Section 4 shows how MICS framework addresses the safety problem with runtime reconfiguring. Conclusion and future work is presented in section 5.

2 Risk of runtime reconfiguration

The main goal of dynamic updating is to keep the system running without interruption. Therefore, it would not be acceptable that updating the system online leads to erroneous situation where system might stop or malfunction. The threat with updating a running system is due to lack test phase in order to verify changes to the system. Consequently, this updating may affect system' correctness. There are four different sources of risk when updating the system dynamically:

2.1 Interrupting running process.

Generally, updating software online cannot be achieved at any arbitrary time because it might interrupt some process that would lead the system to inconsistent state. Well-timing point is needed for any approach to safely update the system online. In previous work, two different approaches are proposed to define suitable time to perform dynamic update:
1) the operation of update is postponed until the target module is inactive[5-8]. Moreover, the target module also should not be disturbed during the time of updating, so the other modules that dependent on the target one should be passivated in order to not to send any request to the target module[9]
2) The update point is predefined in the system itself. This point either be found at runtime such as when new object is created in object-oriented programming[10] or it maybe specified in the code [11, 12] where dynamic updating is only performed when execution reaches a safe updating point that defined when coding the program.

2.2 Breaking components' dependencies.

Updating a component of the system should preserve other components' dependencies on this replaced one. For example, if component C1 has method M1 that is used by other components, replacing C1 with a new one that does not have M1 will cause a problem and might lead the system to crash. This type of errors is identified as interface faults [13, 14] or interface incompatibility [11]. In[15] an algorithm was proposed to validate new component's compatibility with the system. The compatibility is checked by calculating differences between old and new version where the result is one of five possibilities:

1. New component has the same interfaces as the old version which means same methods names and parameters, in this case the new version is compatible and consequently replacing that component is valid.
2. New component has methods as the old one (semantically the same with same parameters) but the
name of methods are different, in this case a glue code is needed to call methods in the new name through a method with the old name.
3. New component has same methods as old one but moreover it has extra functionalities. Those new functionalities are not used by the system. In this case, the component is considered compatible and replacement is valid.
4. New component missed functionalities, but not used by system. In this case also the replacement is valid and component consider compatible.
5. All other cases, the new component is considered not compatible.

Two approaches to handle incompatibility of interfaces: either Using adapters that serve as a translation layer between the new version and its dependents or by updating the dependents before update the target component[16].

2.3 Loosing the state of old version.

When replacing a component online, the new version should start from the point, which the old one stopped at. Therefore, the state of old component should be transferred to the replacing one. The problem here is how to represent component state in old and new version, and how to map old state to the new one that might be different type. Usually, it is assumed there is a relationship between old and new version and because such relation mostly depends on semantic knowledge, transferring the state is usually left to the programmer[17, 18]. For example, in object-oriented languages, the execution state is typically represented by the states of all instance variables and state of the runtime stack, when the object is replaced the values of those variables should be mapped in someway to the variables of the new version[19]. Bialek and Jul [20] defines a state transfer function (STF) to map between old version of component and new one. This function must be attached to update request which is consists of tuples \{ componentName, NewComponentDefinition, STF\}[20]. Mapping the state of old version to the new one could be simple as casting \texttt{int} to \texttt{long} or hard as between different objects. In [21] two techniques are proposed to exchange state information between versions:

1) Direct State Transfer: The implementation of the old version is used directly. It is the responsibility of the new version to interpret and convert the state from the previous version; the state transition function depends on both the old and then new version of the component.

2) Indirect State Transfer: in the approach, an abstract representation of the state is created and all implementation specific program states are mapped to this abstract representation. When state transfer occurs, the old version exports its state in an abstract representation which is later used by the new version. The advantage of this approach is that it does not require knowledge of the implementation of the old component version.

2.4 Semantic errors and system malfunction

This kind of risk is related to component's logic. Since there is no test phase when updating the system dynamically, the behavior of the new component, when works with others in the system, can not be anticipated. Semantic errors are hard, if not impossible, to be identified automatically[11, 12]. Actually, this problem is common with static software upgrading, new version breaks exiting behavior. To solve this problem with static upgrading, Cook and Dage[14] proposed a framework that when upgrading a system it keeps multiple versions of a module running. In their framework, an Arbiter is placed between the external system and the component versions. The arbiter invokes all versions and uses Constraint Evaluator(CE) to select a result according to formal specifications of each version's addressed domain. The arbiter returns the result to the main system and logs the statistics on which version produced the result. Similar to the framework of Cook and Dage presented above, Liu and Richardson [22] proposed a specific architecture called Redundant Array of Independent Component (RAIC) that uses groups of similar or identical components to provide dependable services. The group uses services from one or more components in the group to provide services to the application. From the application view, the RAIC looks as single component. Components can be added or deleted to RAIC dynamically. The RAIC controller is responsible of making judgment about the return values from individual components in the redundant array.

3 MICS framework

In this section, we describe our java-based component model, which is utilized by our Message-based Interaction Component-based System (MICS) framework to develop safe dynamically software systems. In MICS framework, 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. Generally message-oriented pattern of interaction has the following advantages: (1)All dependencies are centralized and no explicit decencies between components which makes component integration easier[23]. (2) It reduces the architecture complexity of the system which means it’s more maintainable and adaptable [24, 25]. (3)Message-based systems are more upgradeable and reconfigurable as new components can be added for satisfying new requirements without changing the basic system architecture [25]. Figure 1 depicts MICS architecture.
3.1 Component Model.

In our model, components represent the essential part of the system. They are the locus of computation and the core providers of system functionalities. They merely services providers and consumers where the communication among them is facilitated by other entities called connectors. We should distinguish between two views of the software component: component type and component instance. The first one is static piece of software that provides specific functionalities, we call it component type, and the second view is the instance that has run-time existence and state. In our model, components only interacts through their interfaces, either provide service to other components or require services from them. Components need connectors to interact with each other, which are defined during composition phase by the integrator. This separation between computations and communications offers loosely architecture. It supports concepts of componentware as the components being easy pluggable and replaceable. Each component has an XML file that describes its interfaces which includes the services provided and required by the component.

3.2 Connectors.

Connectors in our framework are not computation parts of the system. They facilitate components interaction. Each component in MICS communicates with other components in the system through connectors, which hook 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 and call the service from the component. On the other hand, out-port connector represents the gateway for the service required by the component.

3.3 Soft bus

Soft bus in our framework is a special component that is responsible of tracking and identifying all components connected to it, so it routes messages from sender components to the target ones. It simulates the concept of using bus with hardware, so the components can be plugged in or out easily.

3.4 Messages

MICS framework has two types of messages: Request message (RQ), and Response message (RS). Every message contains two parts: a message part (such as service required, service arguments), and a control part (such as message ID, message type). The types of messages as follows:
(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 $<\text{Message type}, \text{Receiver}, \text{Service}, \text{no of arguments}, \text{arguments}, \text{sender}>$
(2) 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 $<\text{Message type}, \text{Receiver}, \text{Result}, \text{arguments}, \text{sender}>$, 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.
3.5 Component Manager.

Component manager is responsible for tracking all component types in the system. When adding a new type to the system, component manager reads its XML description and add it to the list of system components. This description of components service will be used to check compatibility to other components when creating new dependency between this component and others. Component manager also removes the components from the list when it is not used in the system any more.

3.6 Dependency Manager

Updating the system dynamically requires exploring the effects of this modification on the rest of system's components in order not to lead the system to inconsistent state. Dependency Manager is responsible for ensuring the compatibility between components when creating dependency at the beginning of running the system. When replacing a component with a new version, it assures that new version has compatible interface with others. Analyzing component dependencies and how they are implemented are described in our work [26] with more details.

3.7 Update Manager

Update Manager controls all parts of the framework. When there is a request for updating the system on fly, it first checks the status of the target component. If that component is involved in some process, it waits until the component becomes idle. If the new version has the same interface with old ones, Update Manager sends a message to old component to flush its state, disconnect old one from the bus, attach the new one to the bus, and transfer status to the new version. If the new version has different interface from the old one, it uses dependency manager to track all dependents on this component and requires updating them first in order not to break dependencies.

4 Safety in MICS.

The essential goal of runtime reconfiguration is to allow the system running continuously. Therefore, any dynamic updating technique should preserve system safety, so system will not be crashed or malfunctions. Besides facilitating runtime reconfiguration, MICS frameworks supports safety as it is related to dynamic updating. MICS addresses the four sources of risk described in section 2 as follows:
- In MICS, a component during runtime is in one of the following statuses: 1) Idle: a component in this status can provide or required services. 2) Active: in this status, component is engaged in serving a request. 3) Blocked: when a component is blocked, it neither initiates a request nor provides a service. 4) Semi-Blocked: when a component is in semi-blocked status, it only can serve requests from other components but it cannot initiate a request. Any request to update an active component is postponed until the component becomes idle, then update manager sends a message to block the component. Using dependency manager, update manager also sends message to all its dependents to make them semi-blocked so they will not issue any request to the target one and will not disturb updating process.
- MICS framework keeps track of all dependencies among system's components. When a request is issued for updating the system, update manager consults dependency manager to verify compatibility of the new component with others. If it is incompatible, updating is rejected, so the system keeps running.
- As MICS is java-based framework, the class represents component concept and values of its instants fields represent its status. When a request is issued for updating a component, update manager creates a new instance of the new version. Programmer should provide update manager with small program that gets values of old version, and map them to the fields in the new object.
- MICS framework supports the concept of rollback to old version. When a component is updated, the new object is attached to the soft bus and old version is disconnected. This approach of communication among components provides flexibility to rollback to old version if the new version has logical errors. The decision of bringing old version back to the system is made by system administrator.

5 Conclusion and Future work

Runtime reconfiguring is an approach for updating software systems without shutting down the whole system. In this paper, we have presented four possible threats to system safety. We presented our proposed MICS framework, which is a message based interaction approach for composing component-based systems. MICS supports dynamic updating with focus on system safety during and after updating process. Future work in this area can be developing a visual tool that monitors the status of each component in the system and visualizes the interaction of running components in the system.

6 References

Feng, N., S-Module Design for Software Hot Swapping, in System and Computer Engineering Department. 1999, Carleton University.


Bialek, R., Dynamic Updates of Exiting Java Applications, in department of Computer Science. 2006, University of Copenhagen

Hicks, M., Dynamic Software Update, in Department of Computer and Information Science. 2001, University of Pennsylvania.


Vandewoude, Y. and Y. Berbers. Component state mapping for runtime evolution. in International Conference on Programming Languages and Compilers. 2005. Las Vegas, Nevada, USA.


Alhazbi, S., Measuring the complexity of component-based system architecture, in International Conference on Information and Communication Technologies: From Theory to Applications. 2004: Syria.
