A FRAMEWORK FOR DYNAMIC UPDATING IN COMPONENT-BASED SOFTWARE SYSTEMS

BY

SALEH MOHAMMED ALHAZBI

Thesis submitted in fulfillment of
the requirements for the degree of
Doctor of Philosophy

UNIVERSITI SAINS MALAYSIA

2009

ACKNOWLEDGEMENTS

Although this thesis represents an achievement that bears my name, it would not have been possible without help of others who I would to thank. First, and for most, I thank Allah for all his blessings and guidance.

I would like to express my sincere thanks and deepest gratefulness to my supervisor Dr. Aman Jantan for his supervision, encouragements, guidance, insightful criticism, and for all of his help during my research work and preparation this thesis.

I express my special thanks to College of Computer Science for all facilities and support to achiever this research.

I am so grateful to my parents for being there for me, for their prayers, for their love and care, for teaching me that good things comes with hard work.

I am deeply thankful to my closest friends and my pillars of support, Abdullateef Ghallab and Ghassan Habeeb for their help and support during my thesis preparation.

I am grateful to my family for their time and support to finish this research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CB	Component Builder
CBD	Component-Based Development
CDDG	Component Direct Dependency Graph
CDG	Component Dependency Graph
CDL	Component Description Language
CE	Constraint Evaluator
CLR	Common Language Runtime
CM	Component Manager
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off-The-Shelf
DAS	Dynamically Alterable System
DCUP	Dynamic Component Update
DDL	Dynamic Link Library
DYMOS	Dynamic Modification System
EJB	Enterprise JavaBeans
GRS	Global Recoverable State
IDL	Interface Description Language
IL	Intermediate Language
JMS	Java Message Service
JVM	Java Virtual Machine
MDIL	Microsoft Description Identifier Language
MICS	Message-based Interaction in Component-based Systems
MOM	Message-Oriented Middleware
OOD	Object-Oriented Development

OOP	Object-Oriented Programming
ORB	Object Request Broker
PC	program counter
PDA	Personal Digital Assistant
PODUS	Procedure-Oriented Dynamic Updating System
RAIC	Redundant Array of Independent Component
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RQ	Request Message
RS	Response Message
SEESCOA	Software Engineering for Embedded Systems using a Component-Oriented Approach
SOA	Service-Oriented Architecture
SOC	Service-Oriented Communication
SOFA	SOFtware Appliance

LIST OF APPENDICES

RANGKA KERJA BAGI PENGEMASKINIAN DINAMIK DALAM SISTEM PERISIAN BERASASKAN KOMPONEN

ABSTRAK

Setiap sistem perisian (software) perlu dikemas kini setiap masa bagi pelbagai alasan seperti penetapan pepijat (fixing bugs), peningkatan komponen, atau memudahkan sistem menyesuai terhadap perubahan persekitaran. Secara tradisi, sistem perisian perlu dihentikan bagi melaksanakan pengubahsuaian Kaedah ini tidak sesuai bagi sistem yang beroperasi dalam masa yang panjang seperti sistem perbankan atau telekomunikasi, atau sistem misi-kritikal (mission-critical system) seperti pengawal lalu lintas udara (air-traffic controller). Oleh itu, jenis sistem yang sebegini hendaklah dikemas kini secara dinamik. Walau bagaimanapun, pemgemaskinian dalam talian (online updating) boleh mendatangkan pelbagai risiko, yang boleh merosakkan sistem atau mendatangkan kesan tidak dipercayai.

Tesis ini mengemukakan suatu model dan rangka kerja baru dalam usaha membangunkan sistem perisian berasaskan komponen yang boleh dikemas kini secara dinamik dan selamat. Rangka kerja ini memudahkan integrasi komponen dengan memperkenalkan konsep bas-lembut (soft bus concept). Semua komponen dalam sistem digabungkan pada suatu entiti perisian khusus yang fungsinya menyamai bas perkakasan (hardware bus). Bas-lembut membawa mesej di antara komponen sistem. Rangka kerja yang dicadangkan mampu menerima risiko pengemaskinian dinamik yang mungkin dan membolehkan sistem digulung semula (rolled back) selepas pengemaskinian, sekiranya berlaku kegagalan.

Bagi menunjukkan kedayamajuan penyelesaian kami, rangka kerja ini dilaksanakan menggunakan bahasa pengaturcaraan Java. Eksperimen menunjukkan bahawa penggunaan komunikasi secara tidak langsung antara komponen akan meningkatkan prestasi overhed, dalam purata 30%.

A FRAMEWORK FOR DYNAMIC UPDATING IN COMPONENT-BASED SOFTWARE SYSTEMS

ABSTRACT

Every software system needs 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 systems must be shut down in order to perform the modifications. This is not suitable for long-time running systems such as banking or telecommunications systems, or mission-critical systems such as air-traffic controllers. Therefore, this type of systems should be updated dynamically. However, there are different risks with online updating which would crash the system or affect its reliability

This thesis presents a new model and framework to develop component-based software systems that can be updated dynamically and safely. The framework facilitates components integration by introducing soft bus concept. All components in the system are attached to a special software entity that works similarly as hardware bus. The soft bus routes messages between system's components. The proposed framework addresses the possible risks of dynamic updating and allows system to be rolled back after updating in case of failure.

To show the viability of our solution, the framework was implemented using Java programming language. Experiments show that using indirect communication between components adds overhead performance in average of 30%.

CHAPTER 1
INTRODUCTION

1.1 Problem Overview

Naturally, software needs to be updated over time for various reasons such as fixing bugs, upgrading its components, or adapting the system in response to its environment's changes. Traditionally, software modifications always require shutting down the system, updating, and restarting it. This approach is not suitable for critical systems that require 24 hours/7 days availability such as banking or telecommunications systems, or mission-critical systems such as air-traffic controllers. These systems require dynamic updating so that the system can be updated at running time without service interruption. Generally, the process of updating software systems includes adding, deleting or replacing one or more of its components, which are performed while the whole system is running. Dynamic updating has the same meaning as *online change* (Gupta, 1994), *on-the-fly replacement* (Hauptmann and Wasel, 1996), *live updating* (Preuveneers et al., 2006), *runtime reconfiguration* (Jasminka et al., 2004), and *software hot swapping* (Ao, 2000).

Building dynamically updateable software systems is not a new area of research. In general, there are many approaches that range from redundant hardware to software-based ones.

Hardware-based technique, for example, has been used with critical systems for 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; then the system can be upgraded in the original unit while the backup unit is running. When upgrading is finished, the system running is switched back to the original unit (Hicks, 2001). This architecture might contain more than only two machines to increase the availability. For example, Visa's worldwide processing system (*VisaNet*) uses 21 machines to handle more than 3000 transactions per second generated by visa card users. Although the software used on these machines to handle transactions is updated in average of 20,000 times per year, service availability is about 99.5% of the time. This high availability is the outcome of massive backup machines where the whole center's workload can be shifted to another backup center in case of any failure (Pescovitz, 2000). The main disadvantage of such approach is the cost of the hardware. This is due to special hardware needed to allow running the two redundant versions. The two versions always need to be synchronized and this adds more complexity to the system (Rivka et al., 1992).

In contrast to hardware-based approaches for system dynamic update, many software-based approaches such as (Goullon et al., 1978), (Robert and Insup, 1983) (Segal and Frieder, 1993), (Gupta, 1994), (Plasil et al., 1997), (Andersson and Ritzau, 2000) (Hicks et al., 2001) (Hjálmtýsson and Gray, 1998), and (Ketfi et al., 2002) are developed to support updating software system while it is running.

However, for any system to be dynamically updateable, it must be configured into a set of updateable modules based on certain criteria so it can be updated in a module-by-module fashion. According to Bialek (2006), update units can be first class modules, such as procedures in procedural systems, or classes in object-oriented ones, or they could be modules that are more complex such as components in component-based systems (Bialek, 2006).

High modularity of component-based systems makes them relatively well suited for dynamic updating. The main goal of dynamic updating in component-based software systems is conceptually similar to updating the hardware component while the system is running. However, according on (Gupta and Jalote, 1993) the difference is that the new component may not be the same as the old component in terms of function and performance.

1.2 Research Motivation

This thesis will focus on dynamic updating in component-based software systems which are built by integrating pre-existing components. In contrast to several previous work in this area such as (Plasil et al., 1997), (Feng, 1999), (Ketfi et al., 2002), (Vandewoude and Berbers, 2004) which focus solely on dynamic update, the work reported in this thesis will emphasize on system safety during and after online updating process. We define system safety in the context of dynamic updating as the system's ability to work consistently during and after updating operation.

The significance of this research stems from the importance of the three aspects of the problem: dynamic update, component-based system, and safety. Therefore, the following subsections will explain the aspects in the context of this research.

1.2.1 The need for Dynamic Updating

The demand for on-line software updating is increasing for different reasons: First, software systems are normally updated by loading a new version and restarting the system to reflect the new features. For example, Microsoft Windows requires rebooting for some updates. For a common user, it will be fine but that is not acceptable for systems which need high availability to operate such as telephony systems, financial transaction systems, and air traffic Control systems.

Shutting down financial systems might cause big losses for the company. For example, banks can lose as much as US$2.6 million each hour of downtime (Group, 2002). Results of a survey conducted by Eagle Rock Alliance (Eagle Rock Alliance, 2002) show that:

- 54 % of all participating companies stated that each hour of downtime would cost the company more than $50K.
- 8 % said that each hour would cost over $1M. Of all the companies.
- 4 % estimated that the survival of the company would be at risk if the downtime lasted less than one hour.
- 39 % suspected that downtime lasting up to one day would put the survival of the company at risk.

Second, computing nowadays is no longer limited to computers- PC's and Servers as the diversity in hardware architectures grows drastically. We see more and more different types of devices, such as personal digital assistants (PDA’s), mobile phones, and portable computing devices. These highly dynamic environments require frequent updating for software systems. The challenge is how to update such systems while allowing those systems to continue providing service during upgrades.

Moreover, future computer systems must be able to react to changes in the environment by dynamically adapting themselves to keep functioning with good performance. This also includes reacting to variations in resource availability and adapting their algorithms by replacing their components without the need to stop and restart the system. In addition, significant variations in resource availability should trigger architectural reconfigurations, component replacements (Kon, 2000), (Appavoo et al., 2003).

Third, in distributed environment, a system might be composed of hundreds or thousands of machines. It is required that each node in the system is continuously available to provide service to the rest of the system. Halting the whole system for updating one of its nodes is not acceptable. Therefore, updating such nodes dynamically will allow the system to continue to provide service during upgrades.

Lastly, dynamic update is used to increase system reliability. It can be combined with fault tolerant techniques to keep the system running by masking the fault, rollback and restart dynamically using another fault free part. According to (Deepak and Pankaj, 1997) online change will reduce mean time to repair, which outcomes increasing system availability.

1.2.2 The Advantages of Component-based Paradigm

Rather than addressing the problem of dynamic updating in software systems in general, we have chosen to limit this research to the component-based systems which are built by assembling and integrating pre-made components or purchased Commercial Off-The-Shelf (COTS) software components, according to well-defined software architecture. The work in this thesis is limited to component-based systems because lately Component-Based Development (CBD) is gaining more popularity as a new paradigm for developing complex software systems. It is expected that components and component-based services to be broadly used by non-programmers in building their applications because software development will be shifted from writing code to just integrating existing components. Allen (Allen, 1998) has predicated by the result of his survey that by the year 2003 up to 70% of all new software-intensive systems would rely on component-based software. Tools for building such applications by component assembly will be developed. In addition, automatic updating of components over the Internet, which has already presented in many applications today, will be a standard means of application improvement (Crnkovic and Larsson, 2002).

Building software systems with reusable components brings many advantages, for example (Meijler and Nierstrasz, 1998) have summarized the advantages as the follows:

1. ***Fast time-to-market*.** Building applications by assembling their components can be done more quickly than custom-developed ones, thus brought to market and sold more cheaply.
2. ***Reliability.*** Applications that are built by reusable components will have more reliability than applications built from scratch.
3. ***Division of labor.*** Distributing the work of development to software teams can be done more easily with the development of well-defined interfaces components.
4. ***Adaptability***. One of the important properties of components is that they are replicable. This supports system flexibility, so it would be highly adaptable for any changes in the requirements.
5. ***Easy to build distributed systems***. Nowadays, there is a demand on distributed systems in order to use hardware resources optimally. Developing systems using components will offer natural units for distribution and hides the complexity of distributed programming from application developers.

In general, component-based development overcomes software complexity and increases its maintainability. Dynamic updating in component-based systems aims to modify at least one of it component during runtime. Next section discusses the risk of such updating.

1.2.3 Necessity of Safe Dynamic Updating

The main goal of dynamic updating is to keep the system running without interruption. Therefore, it would not be acceptable that updating a system online leads to erroneous situation where the system might stop or malfunction. The threat with updating a running system is due to lack of test phase to verify any changes to the system. Consequently, online updating process may affect system' correctness.

In general, there are four different sources of risk when updating the system dynamically:

1. ***Interrupting running process***. 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.
2. ***Breaking components' dependencies***. Updating a component of the system should preserve other components' dependencies on the replaced one. For example, if a component *C1* has a 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* which is common source for risk with component interaction (Lutz, 1993), (Cook and Dage, 1999).

3. ***Loosing the state of old version***. When replacing a component online, the new version should start from the point where the old one stopped. Therefore, the state of the old component should be transferred to the replacing one. The problem is how to represent component state in the old and the new version. Also, after loading the new version to the system, the problem is how to map the old state to the new one that might be different type.
4. ***Semantic Errors***. This kind of problem is related to component's logic. Since there is no test phase when updating the system dynamically, the behavior of the new component, especially when it works with the others in the system, cannot be anticipated. Semantic errors are hard, if not impossible, to be identified automatically (Bialek, 2006).

1.3 Scope, Goal, and Objectives of the Thesis

This research work will focus on three directions: *dynamic updating, component-based software systems*, and *software safety* as it is related to dynamic update. Figure 1.1 depicts these three directions in this thesis.

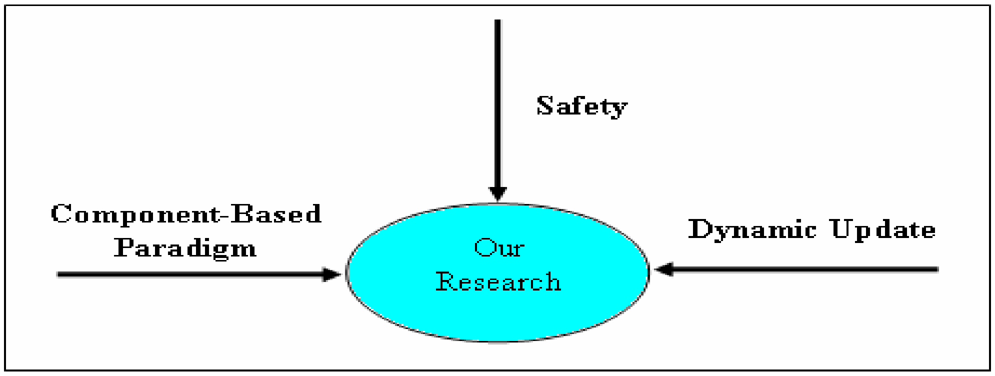

Figure 1.1: Related Areas to this Research

The main goal of this thesis is to define a new framework for developing component-based software systems that can be updated dynamically without risk of crash or malfunction.

To achieve this goal, this thesis particularly will pursue the following objectives:

1. Investigate the problems of dynamic updating in component-based systems with particular emphasis on system correctness and safety during and after online updating, and address the shortcomings of current approaches.
2. Develop a new component-oriented model to develop component-based system that supports the concept of safe dynamic updating. The new model should facilitate exposing component's interface, states and status at runtime. The model should describe components' interfaces that include both services provided by the component and those required from the other ones during execution.
3. Develop a framework that serves as a basis for components integration with support for adding, removing, and updating a component during runtime.
4. The projected framework should support safety in the following aspects:
   a) Before updating is performed, compatibility should be checked to ensure no dependency among components would be broken.
   b) The operation of updating should be isolated to make the system keeps running consistently.
   c) When replacing a component, the new version must start from the point where the old one stopped. Thus the framework should allow transferring component's state straightforwardly.
   d) After updating, in case of an error with the new components, the framework should support roll back mechanism to the old version, therefore the system will not crash.

Thesis Outline

The research and work reported by this thesis will be provided in six chapters organized as follows:

In chapter 1, i.e. this chapter, we have presented the basic concepts, scope, goal and main objectives of this work.

Chapter 2 will present a literature survey of the related work in the three domains of this research: software-based dynamic updating approaches, component-based development methodology, and safety issues regarding online updating. The chapter will focus on the main challenges in building safe dynamically updatable system, how previous research addressed such problems. It will height shortcomings of previous research and will derive the need for this work.

Chapter 3 will present a new component model for developing component-based software systems. The framework, *Message-based Interaction in Component-based Systems (MICS)*, will be explained in detail.

Chapter 4 will analyze how the new model and framework support updating the system dynamically. It will discuss all types of dynamic updating in details and present safety features in the proposed framework.

Chapter 5 will be devoted to enlighten implementation of the framework and build a simple prototypes to figured out implementation difficulties, prove achievability and assess performance overhead of the framework on the system.

Chapter 6 will discuss and summarize the main contributions of this work and presents directions for future work and chapter 7 will conclude this thesis. Figure 1.2 shows the structure of this thesis.

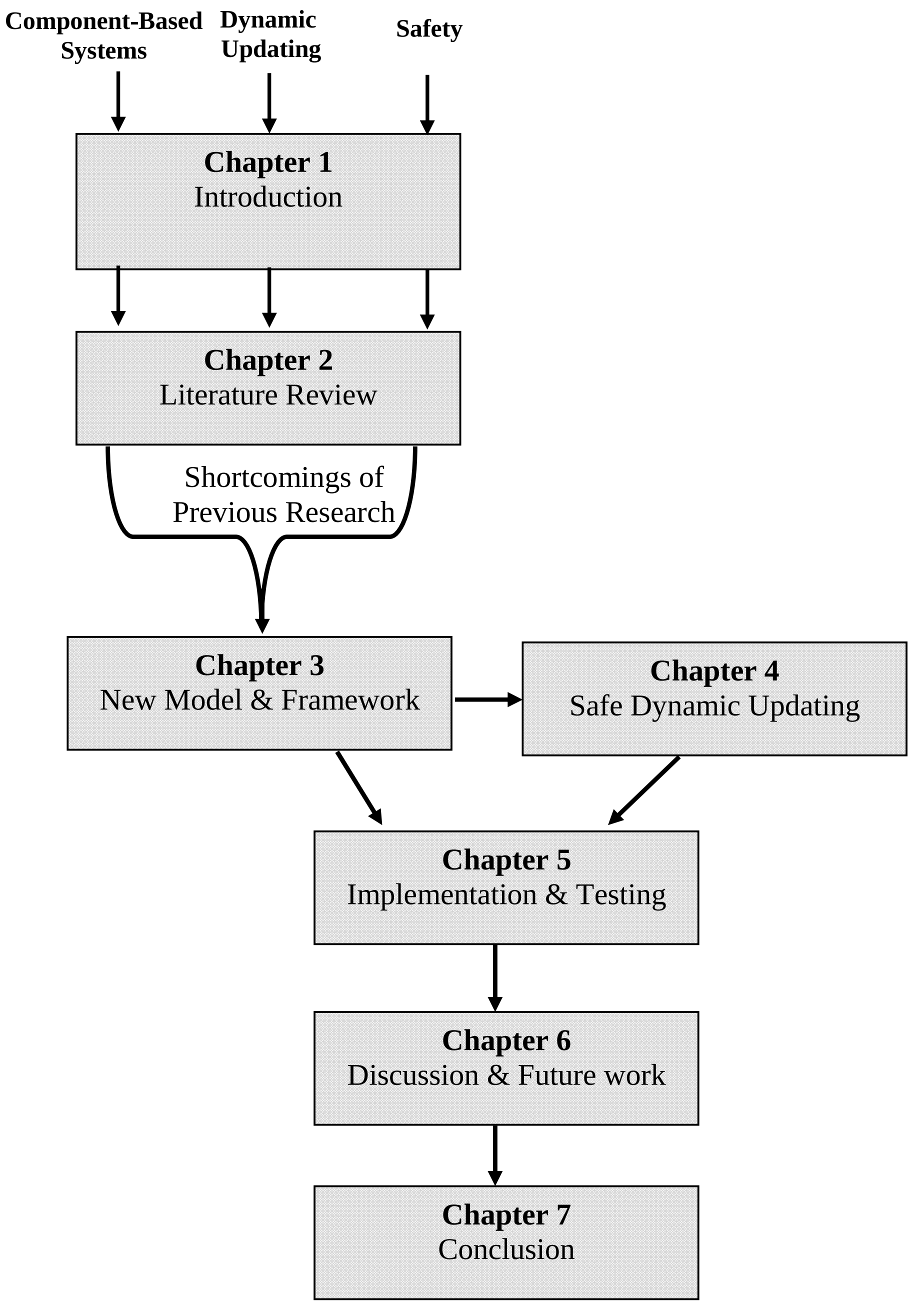

Figure 1.2: Thesis Outline

CHAPTER 2
LITERATURE REVIEW

2.0 Introduction.

This chapter presents a survey of previous works that are related to this research. The literature review, presented in this chapter, covers the three directions of the work. First, section 2.1 clarifies the concept of software component as it is related to component-based paradigm for developing software systems. It explains the relation between objects and components and discusses current technologies that support CBD. Section 2.2, presents literature review of the previous software-based dynamic updating approaches which include approaches for procedure-based, object-oriented systems, and recent approaches for updating component-based systems. Section 2.3 investigates related work on the safety problem when updating component-based system. Figure 2.1 shows the main points of the literature review of this research and the overlapping between the three directions of the research.

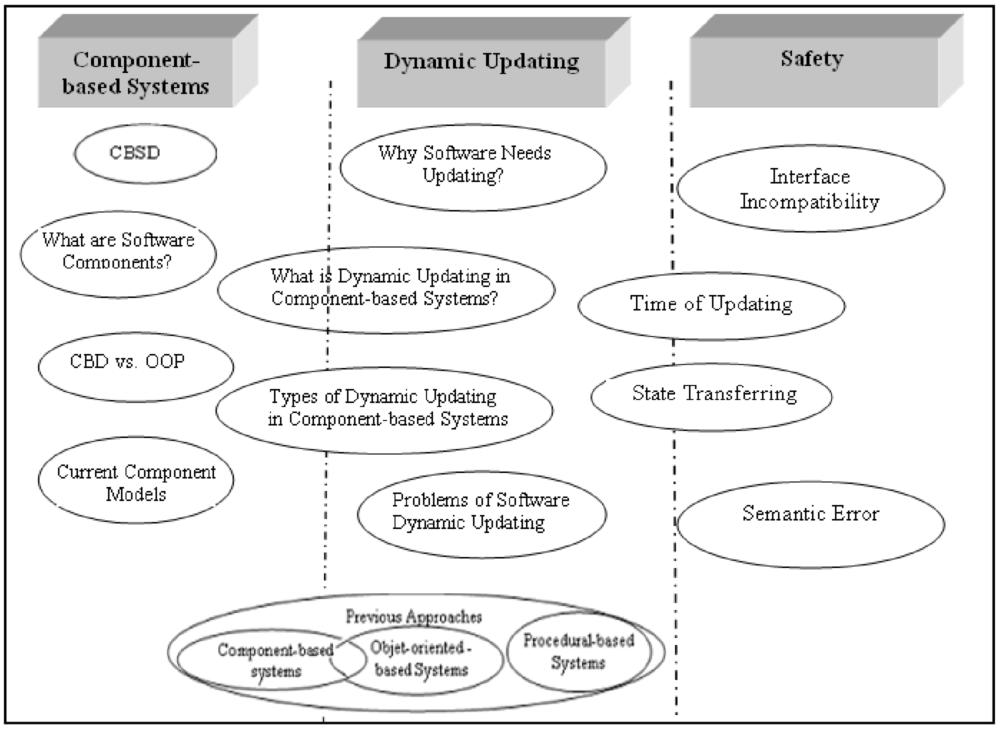
Component-based Systems
Dynamic Updating
Safety
CBSD
What are Software Components?
CBD vs. OOP
Current Component Models
Why Software Needs Updating?
What is Dynamic Updating in Component-based Systems?
Types of Dynamic Updating in Component-based Systems
Problems of Software Dynamic Updating
Interface Incompatibility
Time of Updating
State Transferring
Semantic Error
Previous Approaches
Component-based systems
Objet-oriented - based Systems
Procedural-based Systems

Figure 2.1: Literature Review of the Research

2.1 Software Components and Component-Based Software Development

Software reusability has been considered as the key solution for developing reliable, adaptable, and easy maintainable software systems. Since early structured programming languages in the 1970s, there has always been a stress on reusability concept. The program was divided into modules, and the concept of reusability was applied by using function libraries to implement the system in order to reduce the cost and increase the flexibility of the system.

Next generation of software reusability has appeared with the introduction of object oriented development approach. After its conception in 1980s, the concept of object-oriented development has attracted attention from many researchers and developers. Although object oriented paradigm has offered a lot of support for reusability concept, it could not cope with open systems as it still follows traditional models for software development in which the requirements are assumed to be stable.

Widespread use of software in business and embedded systems has lead to more software complexity. To overcome the complexity, the third generation of reusability in software development was introduced in the late 1990s. Lately, component-based development of software is being increasingly used as a mainstream approach to software development. The major role of a component approach is to manage changes better and make the system more flexible for future changes.

2.1.1 Component-Based Software Development

According to (Sommerville, 2004), the fundamentals of component-based paradigm are:

- Components: they are independent software unit that can be composed with each other.
- Component Model: it defines the standards for component implementation, and deployment so they can interoperate together.
- Middleware: it supports component integration and low-level issues such as resource allocation, security and concurrency.

The main idea of the component-based paradigm is building systems by integrating pre-existing component. In Component-Based Development (CBD), the development is shifted from programming to just integrating pre-built components (Paul, 2001). The development process of the whole system is separated from the process of development of its components and the components should be available in advance and might be bought from a third party, which called Commercial off-the-shelf (COTS). During system development process, much implementation effort is focused on locating and selecting the most suitable components, testing and wiring them together (Crnkovic et al., 2005). Figure 2.2 depicts the process of software development using component-based paradigm.

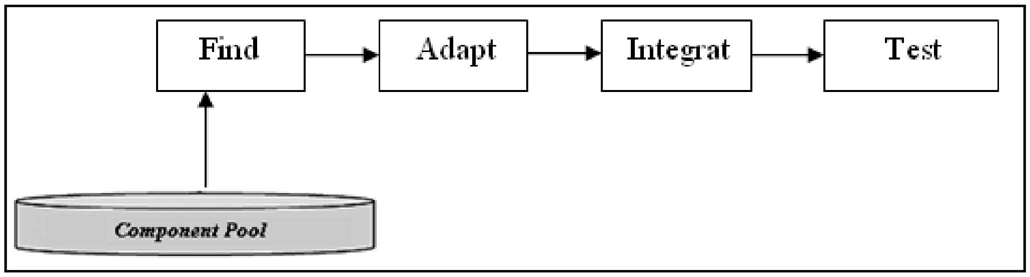

Figure 2.2: Development Process for CBD

Component-based approach provides many benefits, such as reusability, reduced cost of development, and reliability. Moreover, component-based systems have very high modularity thus such systems are more appropriate for adaptability or extensibility. The challenge is to support those features during runtime, so that system can be updated or extended without need to stop the whole system. Such feature seems to be more needed with mobile computing and embedded systems.

2.1.2 What are Components?

The idea behind software components is not new. Software engineers have dreamed for many years of building software systems, like hardware, by combining pre-existing parts. Component software idea can be traced back to a paper published by M.D. McIlroy at the NATO conference at Garmisch in 1968 about the idea of mass-produced software components (McIlroy, 1968).

Nowadays, although component-based development has been widely adopted as a main approach for software development, there is still no universal standard definition for software component. Different definitions of software components were formed. For example, a commonly-used definition is the one stated by Szyperski (2002) who 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*" (Szyperski, 2002). This definition implies the separate development of the components and their ability to be composed through well-defined interfaces.

Another definition is formed by Brown (1997), who defines a component as *"an independently deliverable piece of functionality providing access to the services* *through interfaces"* (Brown, 1997). In this definition, Brown stresses on three aspects of the component as follows:

- Packing: the component is defined as a reusable part that provides the physical packing of model elements.
- Service: the component is defined as a software package, which offers services through its interfaces.
- Integrity: the component is defined as an independently deliverable package of software operations that can be used to build applications or larger components.

Similar to previous definitions, D'Souza and Wills (1997) define a component as "coherent *package of software that can be independently developed and delivered as a unit, and that offers interfaces by which it can be connected, unchanged, with other components to compose a larger system" (D'Souza and Wills, 1997).* From this definition, components seem to be close to the concept of modules, which have been used in programming languages at the 1970s and early 1980s. Furthermore, this definition supports concept of separate development and separate compilation of source code.

In general, software component has the following properties:

- ***A component is independent:*** A component is not bound to a specific system. It is implemented independently from other components and be part of any application.
- ***A component has a well-defined interface***. The interface of a component specifies a flow of dependencies from components that implement services to the components (consumers) that use these services.
- ***A component is replaceable***. Since there is a separation between component interface and its implementation, a component can be replaced by another

version as long as it has the same interface without affecting other components. However, in this thesis, we allow components to be replaced even if they have different interfaces. This affects dependencies with other components and it needs special management in order not to crash the system.

- ***A component is descriptive***. In order to integrate a component with others easily, it must come with clear specification about its services and its requirements.

2.1.3 Differences between OOD and CBD

Usually, the concepts of components and objects are overlapped and sometimes confused. This is because Component Based Development (CBD) uses the same concepts used in Object-Oriented Development (OOD) paradigm. For example, both methodologies hide their internal structure and communicate with outer entities through well-specified interface.

In both methodologies (OOD and CBD), the focus is on utilizing software reusability to build systems with high reliability and low cost. In OOD, it is done through white box reuse where the code is available and inheritance is applied instead of rebuilding everything from scratch. The interactions between system parts are accomplished by sending and receiving messages between the objects of the system. On the other hand, the system in CBD is built by putting pieces of software together where the source code is usually unavailable “black box”. The interaction between components is achieved through well-defined interfaces (Meijler and Nierstrasz, 1998). Interfaces specify the services that components provide or require. Each component can provide or require several interfaces, and each interface can be provided or required by several components. A typical example of an interface is a set of method specifications. An interface is a specification of (part of) the functionality of a component.

Also in term of block's granularity, component is different from object as components exist at different sizes varying from single objects inside a library to whole applications. In most cases, however, components are larger entities and contain several objects.

In CBD, a component is normally built on OOP but with more abstract view of software systems than object-oriented methods. Components written in an object oriented language are implemented as many related classes. This is true even when external access to a component might only be through a single interface. Figure 2.3 depicts an example of a component based on an object-oriented programming. Interface of *Class2* represents the interface of the whole component, *Class2* is subclass of *Class1* and it is associated with *Class3*. However, that does not mean components only include classes. A component could include traditional procedures, or it could be built using assembly language or any other approach.

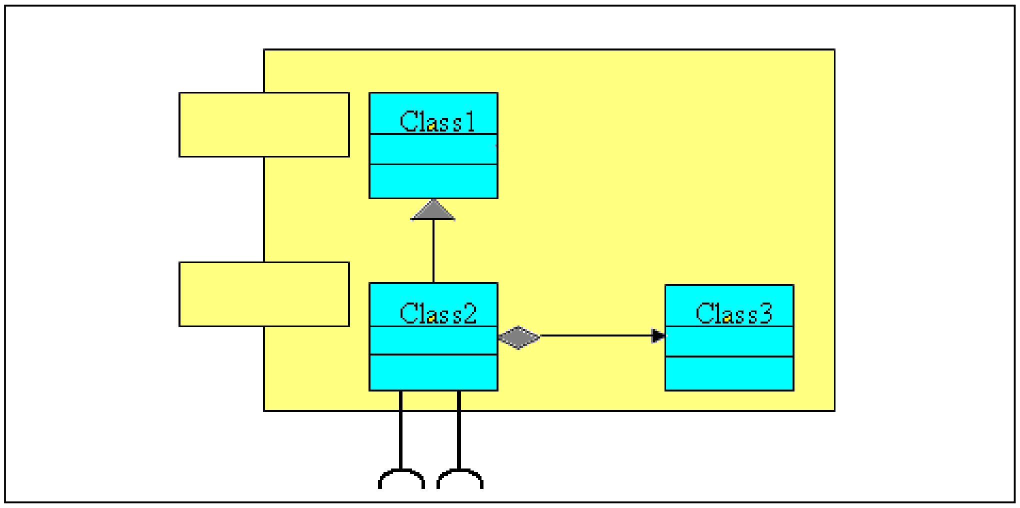

Figure 2.3: Example of a Component based on OOP

Although the main goal of both OOD and CBD is to utilize reusability, they address this goal differently. In object-oriented design, reusability is achieved through class inheritance. When developers create subclass, they should be aware of the implementation of the super class (white box reuse). On the other hand, in component-oriented development, reusability is promoted through binary form (black box reuse). Existing components are used without conscious of its internal details.

Aoyama (Aoyama, 1998) summarizes the difference between OOD and CBD through the following aspects:

1. ***Architecture.*** The architecture of a system in OOD is considered to be monolithic, whereas it is a modular architecture in CBD. Thus the system can be developed partially and incrementally enhanced.
2. ***Process.*** While the OOD follows the conventional process (analysis, design, implement, test), the CBD process consists of two processes: component development and component integration.
3. ***Organization.*** In CBD, there is separation between component development and component integration, which created a new role of component broker. Component broker can sell and distribute software components.

2.1.4 Current Industrial Component Models

Practically, technologies implement component concept differently. The most common component models in industry are JavaBeans, Enterprise JavaBeans, Component Object Model (COM), .Net, and the Common Object Request Broker Architecture (CORBA). Each one of these models has different structures and features as well as different requirements on the environment. Therefore, features defined for a component in a particular technology are not necessarily present in a component specification of another technology. The following subsections present more details on each of these industrial component models.

a. JavaBeans:

A JavaBean is a reusable software component that is written in Java programming language. It can be visually manipulated in builder tools. Javabean is basically a normal java class that follows certain conventions. It extends the Bean class and it is a serializable. Typical unifying features that distinguish a JavaBean are:

- **Support for introspection:** The builder tool can analyze how a bean works
- **Support for customization**: User can customize the appearance and behavior of a bean in the bean builder during integration.
- **Support for events**: Beans interact with each other through events/ listeners model. A bean should register itself with other bean as a listener in order to be notified when an even happens.
- **Support for properties:** Developers can customize and program the bean.
- **Support for persistence**: When a bean is customized, it can save its new state for future use.

Access to all properties is provided by methods, which begin with *get* and *set*. Beans are usually packaged in JAR Files and are identified by their class names. Developers use visual tools, such as bean box, to compose the components and make applications or applets. The applications are distributed to the client as a whole, together with all their constituent beans (Amaratunga, 2000). Because JavaBeans model is based on Java programming language and components in this model are java-based classes, thus it does not support dynamic updating of beans dynamically.

b. Enterprise JavaBeans

Enterprise JavaBeans (EJB) is a component architecture from Sun Microsystems for building server side component in Java. EJB technology enables rapid and simplified development of distributed, transactional, secure and portable applications based on Java technology. An enterprise bean is hosted and managed by an EJB container provided by a J2EE server. There are three types of enterprise beans: *session beans, entity beans,* and *Message-driven* beans.

1. **Session beans:** They implement the business logic of an application. They live only as long as the lifetime of the calling client. Sessions beans can be stateful. In the case that the implemented business process spans multiple requests, the bean needs to retain the state on behalf of the client. In other cases, session beans are stateless.
2. **Entity beans:** They model permanent data. They are long lasting and can serve multiple clients at one time, unlike session beans for which an instance can only be used by one client.
3. **Message-Driven beans:** They model message-related business processes. They are java objects that act as Java Message Service(JMS) listeners that allow J2EE application to process messages asynchronously (Shirah, 2003).

In spite of the rich features of EJB, however, the complexity of its architecture has limited its wide adoption. Moreover, EJB Model also does not support dynamic updating.

c. Component Object Model (COM)

The Component Object model (COM), formerly known as OLE, is introduced by Microsoft in 1993. COM component is a language-independent based on Windows operating system. Therefore a component can be developed by different programming languages and tools like Visual C++, Visual J++, Visual Basic, etc. Each component implements specific interface. While the component can be implemented by different programming language, the interface is described by *Microsoft Description Identifier Language (MDIL)*, which is independent of any programming languages. *COM* objects can be run in the same address space of its client, this type of objects called *Dynamic Link Library (DLL),* or it may run on a separate address space as an executable *(EXE).* In this model, the information about a component is saved in the registry of Windows thus it could be available to other application.

The problem happens when a new application installed and has different version of a component already registered in the system. Installing new version breaks dependencies with applications use the old version, this problem is known as "DLL hell" (Alexander, 2005). Furthermore, although linking *COM* components is postponed until runtime, *COM* model does not support replacing a component while system is running because as soon as the component is linked to the application it can not be modified.

d. Microsoft .NET

Microsoft .NET is a development environment for creating distributed enterprise applications. The main component of .NET is the .NET Framework, which consists of two main parts: the Common Language Runtime (CLR) and the .NET Framework class library. The CLR provides common services for the .NET Framework applications. Programs can be written for the CLR in almost every programming language including C, C++, Microsoft C#™.NET, and Microsoft Visual Basic® as well as some older languages such as Fortran. This is because it translates them into Intermediate Language (IL) like java technique with Java Virtual machine (JVM).

The .NET Framework class library consists of prepackaged sets of functionality that developers can use to extend the capabilities of their software more rapidly. The library includes the following key components(Microsoft, 2005):

- ASP.NET to help build Web applications and Web services.
- Windows Forms to facilitate development of smart client user interface.
- ADO.NET to help connect applications to databases.
- Interoperability support for existing COM applications.
- Improved component versioning and deployment

The .net framework does not support dynamic updating by itself but some frameworks (Rasche and Polze, 2005; Rasche and Schult, 2007) extend .net model to support online reconfiguration.

e. Common Object Request Broker Architecture (CORBA)

Common Object Request Broker Architecture (CORBA) is a standard architecture for distributed object systems. It allows a distributed, heterogeneous collection of objects to interoperate. CORBA defines architecture for distributed objects. The fundamental of CORBA paradigm is requesting services between distributed objects. The services that an object provides are given by its interface. Interfaces are defined in the Object Management Group (OMG) Interface Definition Language (IDL). Distributed objects are identified by object references, which are typed by IDL interfaces. The invocation of the service goes to the local Object Request Broker (ORB) via the IDL stub. The local ORB will route the invocation to the remote ORB, which will pass it on to the object implementation via the skeleton. The location of the object and the details of the routing are completely transparent to the client (SunMicorsystems, 1994). Essentially, CORBA is a component model that can be implemented in different programming languages. Therefore, supporting dynamic updating in CORBA depends on the programming language that is used in implementing the objects (Stiller, 1998).

In summary, none of the above industrial component models support dynamic updating. This is due to the nature of programming languages used to build components in those models. Languages such as java, C++, C# are not sufficient toward runtime class updating because of the following two reasons:

1. Class loading: after the class is loaded to the memory, no changes are allowed therefore any new object is instantiated from one already in the memory (Ebraert et al., 2005).
2. Static safety: such languages ensure program safety by verified compatibility among objects during compile time (Ebraert and Vandewoude, 2005).