Challenges in Component Based Software Engineering
as the Technology of the Modern Era

Usman Ali Khan¹, I. A. Al-Bidewi² & Kunal Gupta³

¹&²Department of IS, FCIT, King Abdul Aziz University, Jeddah, Kingdom of Saudi Arabia
³Amity Institute of Information Technology, Amity University, Lucknow, Uttar Pradesh, India

Abstract - Over a period of time, Component Based Software Engineering has been thought of as the answers to the Software Crisis, and have also provided for many of the shortcomings through the concept of Reusability, Substitutability and Extensibility. The underlining principles of CBSE have revolutionized the thought process of software developers; but, not without shortcomings in the rapid changing scenario of changing technologies. In this paper, we investigate the strength of the Component approach, and then the shortcomings, and difficulties related to CBSE with respect to real time, embedded, and large scale system.

I. INTRODUCTION

The concept of software reuse has been around for the past three decades as a way to drastically reduce the cost of developing software. Software reuse is the process of creating software systems from existing software, instead of constructing software from scratch [1], and it has been extensively addressed by researchers and practitioners, however a more systematic approach applying reuse in all life-cycle phases and iterations has to be part of the process [2]. The need for software reuse has become urgent as the size and complexity of software have started to escalate steeply and as the product development cycle has been compressed. Enterprise software, such as database systems, ERP systems, CRM systems, SCM systems, data warehousing systems, etc. each consist of millions of lines of code, and require expensive technical consulting for proper use. Even software embedded in consumer electronics products such as cell phones and digital television now consist of millions of lines of code. The competitive pressures have forced vendors to reduce product development cycles. One of the challenges for software development organizations that try to apply software reuse programs is to make the specification, persistence and easy access to the component repository feasible, mainly considering the elaboration phase, but also addressing the construction phase of the software product.

II. SOFTWARE REUSE

Software reuse is the process of creating software systems from existing software, instead of constructing software from scratch [3], and it has been extensively addressed by researchers and practitioners, however a more systematic approach applying reuse in all life-cycle phases and iterations has to be part of the process [4]. Reuse initiatives have the main objective of using intermediate or final products conceived in other projects that have already been successfully tested (certified) and implemented before, aiming to reduce time-to-market and getting better general quality, testability and debugging procedures. Even considering that reuse of classes and components libraries are useful, it does not improve fundamentally the software development process, so concepts and tools of a higher level are necessary to leverage the development process [5].

Even when there is a component repository available to developers, it is very important to consider that the components have to be easily discoverable, otherwise it could be easier to develop a new component rather than spending much effort trying to reuse it. Up to now, the activity of “finding reusable components remains a significant barrier for exploiting systematic software reuse” [6].

Another aspect to consider when implementing reuse initiatives is software architecture, which is also considered as the basis for achieving reuse, and when combined with component-based software development, the result is the notion of software product lines [7].

Clements et al., in [8], consider that architecture is a reusable model that can become the basis for an entire family of systems, built using common assets. Also states that “Software architecture is an asset an
organization creates at considerable expense. This expense can and should be reused”.

Fig. 1: Approaches to Software Reuse

The ability to successfully implement a reuse program, which implies a careful, strategic balancing of reuse risks and costs, can result in the following qualitative benefits:

- Increased dependability through reused software that has been tried and tested in working systems
- Reduced process risks resulting from less uncertainty in the costs of reused software vs. new development software
- Reduced product risks by reusing proven components (performance, quality, reliability) and through development consistency
- Effective use of specialists who develop reusable software that captures their knowledge for all projects, not just specific projects
- Compliance to standards, e.g., user interface standards that can be implemented as a set of standard reusable components
- Accelerated development that allows systems/products to be brought to market as early as possible, leveraging reductions from decreased development and validation time through strategic reuse
- Economies of scale in product-line reuse become achievable through domain models, reusable components and common environments

III. ANXIETY IN THE USAGE OF COMPONENT BASED METHODOLOGY:

The basic unit of CBSE is a component. One widely accepted definition, due to Szyperski[9], is: “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”

According to a recent survey [10], a software component model defines what components are (Semantics), how components are defined, constructed, and represented (Syntax), and how components are composed, assembled, or deployed (Composition). A wide variety of different component models have been proposed and developed in the software engineering domain.

Two concepts are central to component-oriented development: the component; and, separately, the component model. There are various interpretations of these concepts; cf. e.g. [11] or [15]. The acceptation given to those concepts, and their realization in practice, largely influence the goodness of the approach, the fulfillment of all the application needs of interest, and ultimately, the opportunities for software reuse. An example of a Component Model is depicted in Figure 2.

Fig. 2 : CBSE Model for Software Development

Using component-based software engineering (CBSE), applications are composed of reusable components with well defined interfaces and behavior. It has become a commonly used development paradigm. Using CBSE, software systems are built by composition of reusable building blocks called components. In addition to the services provided by a component, also the services required from other components or the environment are specified.

Broadly speaking, there are two types of component-based reuse: 1) The component based software systems that are built with assembling components already developed and prepared for the integration with no change, and 2) Using components already developed with changes to customize the need of the client. Reuse without change means simply selecting a component from a software component database, and dropping it into new software being
developed. The cost of developing the component anew is zero! There are, in general, three types of component-based reuse in wide use.

One is reuse of most of existing software when developing the next version of the software. Typically, some 60-80 percent of the existing software gets to be reused in this situation. Another is “reuse” of third-party software, such as a sorting package, a database loader, etc. on the market or on the Internet as open source code. A third type of reuse is common functions available in programming language libraries, such as the math functions in the C Programming Library. There are a large variety of technical reasons and risks that make reusing existing components without change disadvantageous.

3.1 Functional differences

This is possibly the most serious reason an existing component cannot be reused without change. It is rarely the case that an existing component and a component to be newly developed match precisely in functions. The new component to be developed may require some changes to corresponding functions in the existing component, or may require additional functions. The existing component may be excessively rich in functions and, for say, performance reasons, most of the excess functions may need to be eliminated. It is entirely possible, especially in this age of digital convergence where such devices as cell phones and MP3 players are continually taking on additional capabilities, software developed to provide one particular major capability for one product line may be reused in its entirety for another product line; for example, software for digital cameras may be reused on cell phones. However, it is unlikely that smaller granele components may be reused without change across product lines (i.e., business units).

3.2 Programming language difference

An existing component developed in one language may not always be the most subtle choice for integration with components of another language, if no changes are desired in the existing component. If it has been decided that new software must be developed in, say, Java, an existing component written in, say C, cannot be reused without change, even if it satisfies all other requirements, such as the functionality, operating environment, system architecture, etc.

A component model has always a corresponding domain-specific language able to fully express it. Language X can express a superset of the necessary concepts, and it is able to fully express the component model. That is not true, for language Y, which lacks the expressive power for all the concepts of interest. This situation is illustrated in Figure 3.

3.3 Target environment differences

Target environments include platforms (chipset for embedded systems, and operating system), computer system architecture (single CPU, parallel computer, hub and spoke distributed architecture, peer to peer distributed architecture, networking protocol, etc.), computing environment (disk, CD ROM, USB, flash memory, etc.) and operating constraints (main memory size, secondary storage size, power consumption, screen size, paper size for a printer, etc.). Software components developed for one particular target environment often cannot be reused without change for another target environment.

3.4 Operating environment differences

Operating environments include the number of simultaneous users, read-only or read-write, volumes of data to manage, data input and output rate, etc. Software developed for a single user cannot in general be reused to support a multiple simultaneous-user environment. Software designed to only read data cannot in general be reused to support a read-write environment. Software designed for slow input and output of data cannot in general be reused to support an environment where the input and output rate is very high.

3.5 Industry Standard Differences

There are many industry-wide standards on a very wide range of aspects of computing and communications. Examples include Web document standards (HTML, XML, etc.), communications standards (CDMA, GSM, 3G, Bluetooth, WiFi, etc.), cable television standards (OCAP, ACAP), connectivity standards (DLNA, Marlin, OSGi, UPnP, etc.), multimedia data encoding standards (MPEG, JPEG, HDD vs. BluRay, etc.), national language encoding standards (Unicode, etc.), metadata management standards (MOF, XML, CWM, Dublin Core), external database access standards (ODBC, JDBC), etc. Software components that explicitly reflect one standard cannot be reused in software supporting another standard.
a. **Unclear and ambiguous requirements**

In general, requirements management is an important part of the development process, its main objective being to define consistent and complete component requirements. Reusable components are, by definition, to be used in different applications, some of which may yet be unknown and the requirements of which cannot be predicted. This applies to both functional and non-functional requirements. In case the requirement changes with time, it becomes essential for the components to behave accordingly, thus requiring change at the component level.

b. **Sensitivity to changes**

As components and applications have separate lifecycles and different kinds of requirements, there is some risk that a component will not completely satisfy the application requirements or that it may include concealed characteristics not known to application developers. When introducing changes on the application level, changes such as updating of operating system, updating of other components, changes in the application, etc., there is a risk that the change introduced will cause system failure.

### 3.6 Data format differences

A large volume of data is managed either by a file system or a database management system. Software components that interface with file systems are significantly different from those that interface with database management systems. Different countries use different formats to store such data as date, time, currency, etc., and use different measurement units. Some software encode data in ASCII, while others use EBCDIC. Software components that explicitly deal with such data formats and encoding schemes cannot be reused without change in software that use different formats or encoding scheme.

### 3.7 Algorithm and data structure differences

A wide variety of algorithms implement key functions or capabilities supported in software systems. Examples include sorting, searching, data replication, database locking, database logging for recovery, security and encryption, message routing on a network, etc. For each such function or capability, typically there is more than one algorithm or technique, and accompanying data structures (e.g., linked list, hashing, binary tree, B+-tree, R*-tree, heap), to implement it, each with different tradeoff considerations. One technique may be simple and quick to implement, while it may result in low performance or low level of reliability or low level of security. One technique may be good to support the management of a small amount of data, while it may be totally inappropriate for a large volume of data or a very high input data rate. One technique may be good to support the routing of a large number of small messages, while it may be unacceptable for routing a mixture of a small number of large messages and a large number of small messages. Software components that implement algorithms and techniques, and accompanying data structures, under certain tradeoff considerations cannot in general be reused without change under different tradeoff considerations.

### IV. PERFORMANCE CHALLENGES FOR CBSE

CBSE posed two new major challenges for software performance engineering:

a. The development process is distributed among multiple parties (at a minimum, the component developer and component consumer). For performance analysis, this distribution must be reflected in the design of prediction models.

b. Component performance specifications must be parameterized according to their usage and deployment.

These challenges have to be addressed efficiently by CBSE.

### V. COMPONENT–ORIENTED DEVELOPMENT APPROACH : AN EVOLVING PROCESS IN THE DEVELOPMENT OF REAL TIME, EMBEDDED AND LARGE SCALE SOFTWARE SYSTEMS

CBSE has evolved over the past three decades. It was for a long time believed that CBSE is inefficient to address several aspects of the embedded system, Real Time Systems and Large Scale Software Systems. With time, CBSE evolved to answer these requirements. It is still believed, we further need to evolve to design better systems using CBSE, several lessons have been learned in the past which we need to provide answers for:

#### 5.1 Creation of Domain Specific Component Model:

The creation of a component model generically targeting real-time embedded systems is not sufficient. That component model would in fact qualify as "domain neutral". "Domain specific" in this context denotes a component model a specialized for a given real-time embedded application domain. Support for "domain-specific" concerns entails the support for all the concerns carried by that specific application domain. Without that support, the adoption of the component model as industrial baseline may become unattractive, as the extent of modification required to fully adapt the domainneutral part to domain-specific needs may be
Challenges in Component Based Software Engineering as the Technology of the Modern Era

5.2 The Component Model should provide for a design flow:

The notional purpose of a component model is to permit the creation of a whole software system as a composition of software components. In actual fact, however, the component model does more than that. Not only does it prescribe the syntactic rules to create design entities and the way to relate them to one another, but – whether intentionally or implicitly, in any case inevitably – it also establishes a defined design flow. The design flow comprises a series of steps that must be followed to create components, assemble them and ultimately produce the software system. It may also determine a set of precedence relations between those steps.

5.3 Development direction should never be forced on CBSE:

Software design proceeds in one of three possible directions: top-down; bottom-up; some combination of them. A general evaluation of the virtues and limitations of those directions is not very useful, as the goodness of one’s choice is highly dependent on the specific context of application. We can however reason on when those directions are more likely to be chosen: top-down development is likely to occur when components are specified for the first time (as top-down reasoning helps to master the design complexity); in contrast, bottom-up development is likely to prevail when reuse of components enters the picture. Similarly to the design flow, a component model may well promote a given development direction, perhaps even actively. Problems arise however when active promotion of one direction causes active opposition to another.

5.4 The Right Level of Abstraction should be sought:

One of the most interesting and qualifying aspects of a component model is the level of abstraction of its design entities. It is important to understand that the user space of the component model is above all the realm of the software architect. The primary goal of the software architect is to direct the software design in accord with the methodological principles of the chosen software architecture. Hence the component model shall provide a view of the software (and partly of the system) at a level of abstraction that corresponds with that goal. This implies that the level of specification required to the component model is most definitely not the implementation level: the component model shall not pollute the design view with implementation artifacts like threads, semaphores, and such like.

5.5 Stateful Components Should be Provided for:

Some component models only allow the creation of strictly stateless components. A component model of that kind does not permit to specify at design level the attributes (i.e., typed parameters) of the component, which collectively form the state of the component. The state of the component is thus relegated to the algorithmic code of the component implementation. In that manner however the component model is unable to represent that information in the design specification.

5.6 Consider extra-functional concerns and analysis from the outset:

The component model is not completely independent of extra-functional and analysis concerns. Designing a component-oriented approach without considering those aspects from the outset can lead to inconsistencies, infeasible implementation and costly corrections to the approach.

VI. CONCLUSION

In this paper we have established some of the challenges associated with component-based development. We consider component technologies as an excellent tool for building robust software in a fast pace of changing environment. Therefore we analyzed in general CBSE challenges which have to be addressed in order to put in place of sound CBSE practices. But those practices have different relevance for different application areas. The main purpose of the analysis performed within each specific application area is to identify, driven by the set of identified challenges. Meeting these needs is the main challenge faced when provisioning enterprise-scale solutions in the Internet age.

REFERENCES:


