Survey of component-based software development

S. Mahmood, R. Lai and Y.S. Kim

Abstract: Because of the extensive uses of components, the Component-Based Software Engineering (CBSE) process is quite different from that of the traditional waterfall approach. CBSE not only requires focus on system specification and development, but also requires additional consideration for overall system context, individual components properties and component acquisition and integration process. The term component-based software development (CBD) can be referred to as the process for building a system using components. CBD life cycle consists of a set of phases, namely, identifying and selecting components based on stakeholder requirements, integrating and assembling the selected components and updating the system as components evolve over time with newer versions. This work presents an indicative literature survey of techniques proposed for different phases of the CBD life cycle. The aim of this survey is to help provide a better understanding of different CBD techniques for each of these areas.

1 Introduction

Over the last decade, the extensive uses of software have placed new demands and expectations on the software industry especially for improving quality and enhancing development productivity. In order to meet these challenges, software development must be able to cope with complexity and to adapt quickly to changes as the result of the increases in demand for the integration of different areas [1]. Software engineers have been of the view that software does not have to be developed from scratch all the times and have thought about assembling previously existing components into large software systems. Software engineers have reused abstractions and processes since the early days of computing, but the early approaches to reuse were ad hoc. As there is an ever-growing need for techniques that could improve the software development process, reduce the time to market and improve the quality of delivered software products, a more organised and focus approach to reuse called component-based software engineering (CBSE) has emerged.

In the context of CBSE, a component is the fundamental building block for a software system. The concept of software component [2–7] has been discussed in detail and several definitions have emerged. As components are the fundamental in the building of component-based software systems, it is important to agree on a definition. In this paper, we adopt Szyperski's *et al.* component definition [8]: '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'. This definition affirms the fact that a component is an independent software package that provides functionality via well-defined interface.

To date, CBSE research addresses many issues ranging from fundamental concepts and definitions to component-based development techniques and process. In the traditional software engineering process, the typical activities involved in building a system are specification, implementation, testing and maintenance. However, because of the extensive uses of existing components, the CBSE activities are somewhat different. CBSE not only requires focus on system specification and development, but also requires additional consideration for overall system context, individual components properties and component acquisition and integration process. Thus, it has brought along a number of challenges to the techniques that have to provide support for it. Software research community has realised these challenges by addressing issues ranging from precise definition of core concepts [2, 4, 5, 8, 9], component evaluation and integration [10–14] to development of methodologies and processes for software component development [15–17].

We shall refer the term component-based software development (CBD) as the process for building a system which consists of the following activities: searching and identifying components based on preliminary assessment; selecting components based on stakeholder requirements; integrating and assembling the selected components; and updating the system as components evolve over time with newer versions. This paper presents an indicative literature survey of the CBD activities for building a system; those areas surveyed are component identification and selection, integration, deployment and evolution. The aim of this paper is to help provide a better understanding of the different CBD techniques for each of these areas.

The paper is organised as follows. In Section 2, we provide an overview of potential benefits, risk and challenges associated with CBD. Section 3 presents an overview of CBD life cycle and structure of our survey. Section 4 presents survey and discussion on component identification and selection techniques. Section 5 presents survey of techniques addressing component integration phase of CBD. CBD deployment and evolution techniques are discussed in Sections 6 and 7, respectively. Finally, the conclusion is presented in a Section 8.

doi:10.1049/iet-sen:20060045

Paper first received 22nd August and in revised form 4th December 2006

S. Mahmood and R. Lai are with the Department of Computer Science and Computer Engineering, La Trobe University, Victoria 3086, Australia

Y.S. Kim is with the College of Software, Kyungwon University Songnam, Kyunggi-Do 461-701, Korea

E-mail: lai@cs.latrobe.edu.au

Table 1: Summary of component-based software development advantages

Advantage	Description
Reduced development time	Less time required to buy a component than to design, code, test, debug and document it
Increased flexibility	Component-based systems are immune to the implementation of the components, thus there is more choice of components from which to choose so that they meet the requirements
Reduced process risk	If a component exists, there is less uncertainty in cost associated with its reuse as compared with new development
Enhanced quality	Components are reused and tested in many different applications. Design and implementation faults are discovered and eliminated in the initial use, thus enhancing the quality of the component
Low maintenance	Easy replacement of obsolete components with new enhanced ones
Standardisation	Some standards can be implemented as a set of standard component development. The use of these standards will improve the overall quality of the components and systems

2 Benefits, risks and challenges

The aim of CBD is to build systems as an assembly of components such that the development of components as reusable entities and the maintenance of the system by customising and replacing such components [1]. The motivation behind the use of CBD is to reduce the development cost, time to market and provide a system that is efficient in meeting the changing customer demands. Some of the potential advantages of CBD [1, 18, 19] are summarised in Table 1.

Although very promising, CBD is a new discipline and there are a number of risks and challenges associated with it. One of the potential risks for the CBD is what if the primary supplier of the component goes out of business, or stops supporting the current version of the component. On the other hand, if the system demands high quality, how we can assure that the component will allow the satisfaction of these requirements. The potential risks of CBD [1, 6, 18, 19] are summarised in Table 2. However, the benefits of CBD are real and these concerns increase the need for careful preparation and planning to address these challenges by defining guidelines, standards and open architecture for CBD.

3 Development process

Because of the extensive uses of the components, the activities involved in each phase of CBD and the relationships between different phases are distant from the traditional waterfall approach [1, 2]. CBD is integration centric with a focus on selecting and assembling existing components to build a software system. Fig. 1 shows an overview of CBD development life cycle. In this paper, we survey current research work proposed for supporting each phase of CBD life cycle based on the following guidelines adopted from [20, 21].

- *Defining answerable question*. Our aim is to get a better understanding of CBD life cycle by analysing techniques proposed for each of its phase.
- *Finding best evidence*. We survey an indicative literature from major software engineering journals and international conferences.
- *Analyse the evidence*. We provide an analysis of the proposed techniques to highlight their advantages and limitation.

4 Component identification and selection

Identification and selection of components is widely recognised as an interrelated process which plays a central role in overall CBD. Software literature [22–26] indicates that CBD success depends on the ability to identify and select suitable components. An inappropriate component selection can lead to adverse affects such as short listing components that barely fulfil the needed functionality or they introduce extra costs in integration and maintenance phases [23]. Fig. 2 illustrates an overview of component identification and selection process.

In CBD, component identification and selection can be categorised as four-stage process, namely, search screen,

Table 2: Summary of component-based software development (CBD) risks

Risk and challenges	Description
Requirement satisfaction	Component selection is an iterative process and the result depends on the classification and retrieval mechanism. The component search is performed on a wide variety of component repositories, even when components are found, they might not perform the specific function or fail to interoperate with one another
Finding suitable components	Software engineers must be reasonably confident of finding suitable components from repositories before they will routinely include a component search as part of their normal development process
Interoperability	CBD poses a considerable challenge of ensuring that component services are provided through standard interfaces to ensure interoperability
Unit and integration testing	Each component must undergo verification and validation testing. However, unlike traditional applications, individual components can be used in a different set of applications which complicates the testing process

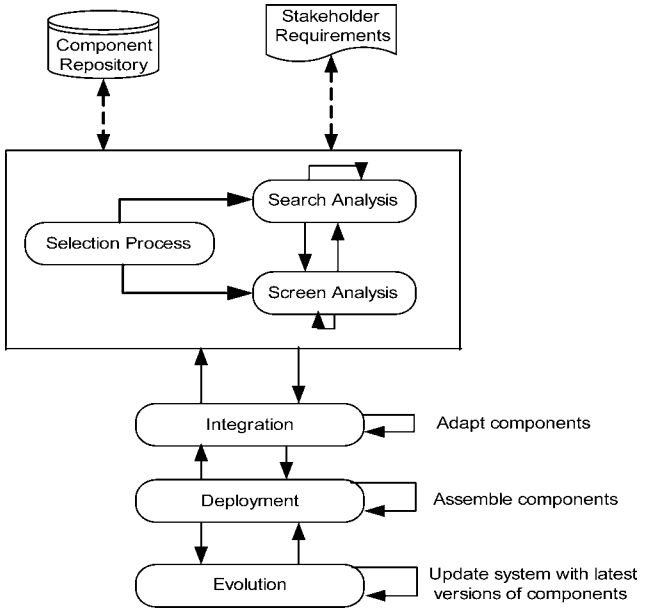

Fig. 1 *Overview of component-based software development (CBD) process*

evaluation and analysis phases. Component identification starts with a search of potential components that match stakeholder requirements, which might themselves be simultaneously evolving. The search stage is usually carried out by dealing with unstructured component information [27] available in a number of formats ranging from marketing brochures to natural language descriptions. Thus, component classification plays an important attribute in overall component selection process. The search phase results in a general list of candidate components which have potential to satisfy stakeholder requirements. As the result of search phase are too general in nature to make final component selection, evaluation and analysis phases are used to systematically compare and rank candidate components based on evaluation criteria.

This section provides an indicative survey of techniques that support component identification and selection in CBD life cycle. First, we discuss component classification techniques because they have direct relationship with the success of searching candidate components. Later, we discuss component identification and selection models and categorise them based on their underlying models. First, we discuss approaches that propose a set of hierarchical steps to identify and select suitable components. The clustering approach category presents discussion on approaches that use a set of predefined rules and heuristics to identify and select components. The iterative category approaches use iterative process of comparing components with stakeholder requirements and use multicriteria decision-making techniques to select suitable components. Finally, approaches that use the concept of knowledge base are discussed. We have summarised the advantages and limitations of reviewed techniques in Table 3.

4.1 Classification

Component classification is one of the key attributes that has direct impact on effectiveness and efficiency of CBD. The aim of classification schemes is to provide a medium for comparing component similarities for better component selection and retrieval. The existing classification approaches can be categorised into four types: simple keyword search, faceted classification, signature matching and behaviour matching. The simple keyword search uses a set of keywords to compare required functionality with component features. The faced classification approaches attempts to classify components based on predefined taxonomies. In signature matching approaches (e.g. [28]), components are represented by their signatures and a hierarchy of matching criteria is defined [29] to help search process. Finally, behaviour matching approaches use formal specifications (e.g. [30]) to describe behaviour of software components and determine whether two components match.

Recently, Cechich *et al.* [27, 31] have categorised component classification approaches into three main groups. First, approaches that use taxonomies and ontologies [32, 33] to provide description of components and uses semantic information for component classification. Second, approaches [34] that uses the concept of learning phase during component identification to classify and support

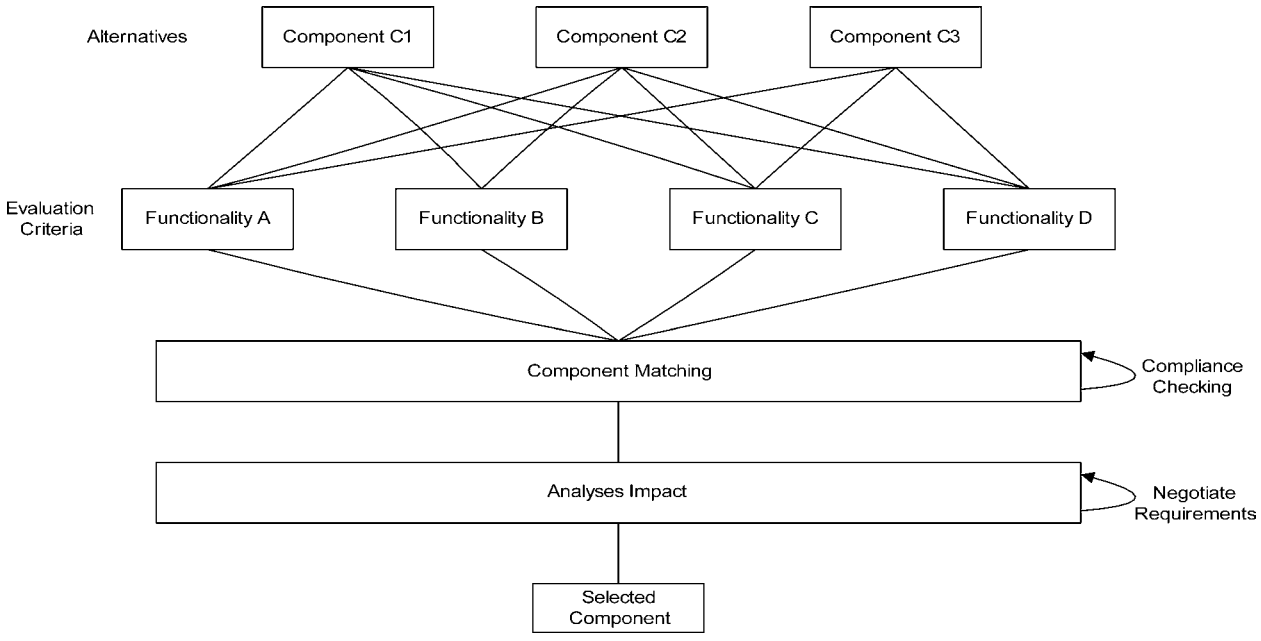

Fig. 2 *Component selection process*

Table 3: Summary of identification and selection approaches

Framework	Strengths	Limitations
Off the shelf option (OTSO)	• Provides systematic component selection process • Addresses the complexity of component evaluation • Analytic hierarchy process provides support for decision-making analysis	• Limitation on how to conduct requirement acquisition process. • Does not include quality aspects directly into the selection process • Inherits the limitation of analytic hierarchy process [22]
COTS-based integration system development (CISD)	• Provides priority list of identified components • Addresses the complexity of component identification • Analytic hierarchy process provides support for identification analysis	• Limitation on how to conduct non-functional requirement acquisition process • Inherits the limitations of analytic hierarchy process
Component development methodology (COMO)	• Supports the component identification through clustering technique • Reduces development time and process risk by undertaking problem domain analysis • Unified modelling language is used to describe the identification process; hence, it helps to integrate with standard practice of development of environment	• Lack of well-defined non-functional requirement acquisition and modelling process • Does not consider quality attributes during the component identification process
Procurement oriented requirement engineering (PORE)	• Provides guideline for designing product evaluation test cases • Provides guideline for requirement modelling • Parallel requirement acquisition and selection process	• Lacks details on how requirements are used in evaluation process • Compliance process between features and requirements is not examined in detail and lack the information on how to perform it • Labour-intensive
COTS-based requirement engineering (CRE)	• Highlights the importance of non-functional requirements • Supports the evaluation of components through systemic criteria that consider quality attributes • Provides guidelines on how to acquire and specify non-functional requirements	• Does not address quality testing issues • Lack of support for the cases when non-functional requirements are not properly satisfied
COTS-aware requirement engineering (CARE)	• Highlights importance to keep requirements flexible as they have to be constrained by the capabilities of available components • Narrows the gap between customer and component requirements by using knowledge base	• Lacks details on how to handle possible mismatching between system and product speciation • Lack of support for the cases when non-functional requirements are not properly satisfied
Component-based web development methodology	• Provides guidelines for requirement modelling for component-based web applications • Provides guidelines on how to specify component-based web applications	Does not address the quality issues
COTS usage risk evaluation (CURE)	• Provides analysis of high impact areas for the use of components • Supports requirements gathering, analysis of data and presentation of results	Requires manual analytical work

component selection process and finally, approaches [35] which propose a concept of semantic distance measure between required and offered functionality.

Zahedi *et al.* [7] proposed a classification scheme to characterise components based on the classification and coding (C&C) scheme which assigns a unique code to each component. The classification of the components can be based on factors such as group of technologies, type of applications and so on. The assigned unique code is later used for the selection of suitable components from component-based knowledge repository. *Component rank* [36] mechanism is based on analysing actual use relations of components and propagating the significance through the use relations. In this model, a collection of software components is represented as a weighted directed graph whose nodes correspond to the components and edges correspond to usage relation. The nodes in the graph are ranked by their weights which are defined as the elements of the eigenvector of an adjacent matrix for the directed graph. These techniques help increase development flexibility; reduce process risk and development time by classifying components based on their architectural and usage behaviour. However, they lack a well-defined requirement modelling process and do not relate well with component selection and retrieval.

Damiani *et al.* [37] presents a hierarchy aware classification schema for object-oriented code, where software components are classified according to their behaviour characteristics, such as services, algorithms and data. These characteristics are constructed from the description of the abstract classes specifying both the framework structure and purpose. The set of characteristics associated with a given component forms its software descriptor. This descriptor base is used to query the system, specifying a set of desired functionalities and getting a ranked set of adaptable candidates. Smith *et al.* [38] proposes multitired classification scheme based on attribute value, faceted and enumerated classification schemes. The attribute value is initially used within the classification for specifying the domain, platform, operating system and development language relating to the component. The functionality of the component is then classified using faceted scheme only using the functional set of faceted scheme. The enumerated scheme is then used to classify and retrieve the component's presentations. Finally, for a detailed survey of existing component classification, please refer to [31].

4.2 Hierarchical approach

The *Off the shelf option* (OTSO) method [15] is a process that directly addresses the issue of component selection. The process is based on hierarchical evaluation, which decomposes the requirements into a hierarchical criteria set. It facilitates a systematic, repeatable and requirement-driven component identification and selection process. OTSO identifies four sub-processes: search criteria, definition of baseline, detailed evaluation definition and weighting of criteria. The main principle of the OTSO method is a well-defined systematic process that covers the whole component selection process. It derives detailed component evaluation criteria from reuse goals, a method of estimating the relative effort or cost benefits of different alternatives and a method for comparing the 'non-functional' aspects of alternatives. The evaluation criteria definition process essentially decomposes the requirements for a component into a hierarchical criteria set and each branch in this hierarchy ends in an evaluation attribute. However, it does not include quality aspects in the selection process. The ability of documenting the baseline in sufficient detail could be an issue that limits the applicability of this model to more complex systems.

Tran et al. [16] proposes COTS-based integration system development model (CISD). Its identification phase includes process of collecting and understanding the overall system requirements, identifying components into product sets and prioritising them for the subsequent evaluation. The major activities during this phase include: (a) requirement analysis, (b) product identification and (c) product prioritisation. The requirement analysis stage encompasses the process of understanding the system requirements and partitioning these requirements into various application and services domains. The product identification stage encompasses the process of collecting information on candidate component products and grouping these products into different product combinations, or sets, for further evaluation. The product prioritisation stage includes the review of all the candidate product sets to generate a prioritised list for further evaluation. The model is still in its early stage and work is required in the areas of handling the changing requirements and architectural mismatches among different COTS. It does not consider non-functional requirements, quality attributes and related costs during the identification stage.

4.3 Clustering approach

Component development methodology, COMO [17] extends unified modelling language (UML) and rational's unified process with semantics related to component development. The model uses clustering techniques for identifying components based on use case and class clustering technique. Use case clustering technique clusters cohesive use cases by considering ⟪*extends*⟫ relationship between use cases and clusters cohesive use cases and classes into component by applying proposed clustering algorithm. A matrix for use case and classes is defined by assigning values according the relationship between use case and class. The relationship between use case and class are defined as '*Create*', '*Delete'*, *'Write'* and '*Read'* and the priority for the relationship types are defined as '*Create*⟩ *Delete*⟩ *Write*⟩ *Read'*. COMO enhances the CBD process by using UML as a standard for component specifications. A limitation is that it only focuses on functional requirements, and does not consider the quality attributes during component identification phase.

Jain *et al.* [39] proposes a approach which assists in identifying and selecting components from an object model representing a business domain. The approach uses a clustering algorithm based on a set of predefined rule and heuristics. The process of component identification begins by grouping related classes of an analysis level domain model. The technique proceeds through a series of successive binary mergers, initially of individual entities (classes) and later of clusters formed during the previous stages. The classes having the highest relationship strengths are grouped first. The process continues until a cut-off point is reached. The component identification approach makes uses of a set of heuristics (automatic and manual) for further refining the initial solution obtained from the clustering algorithm. Further, they propose a tool support for the approach.

Lee *et al.* [40] presents component identification algorithm with the focus on high cohesion and low coupling values. For component identification process, at first, architecture design is performed to identify components by

identifying the architecture layers and subsystems. This division of system into subsystems decreases effort to identify components because subsystem has smaller classes than whole system. In the next step, the subsystem dependencies are determined using sequence diagrams and subsystems must be *re-organised* to make subsystem dependency less complex. This re-organisation of subsystems can be performed by re-arranging classes among subsystems. After subsystems are re-organised, the clustering algorithm is used to identify components.

4.4 Iterative approach

Procurement Oriented Requirement Engineering (PORE) [22] approach is based on interactive process of requirement gathering and product evaluation. It uses a template-based approach to refine the component list until the most suitable component is selected. PORE approach has three main components: (a) a process model that identifies essential goals and prescribes generic processes to achieve each of these goals as well as sequence in which these goals should be achieved; (b) a set of techniques for achieving each of these processes; and (c) a product model that provides semantic and syntax for modelling software products. For each iteration, the software development team *acquires* requirements that discriminate between the components and identifies candidates that are not compliant with these requirements using multicriteria decision-making, rejects component candidates that are not compliant with the requirements and explores the remaining candidates to discover possible new requirements to use to discriminate further between candidates. The approach supports parallel requirement acquisition and selection process. It lacks details on how requirements are used in evaluation process. Similarly, the compliance process between features and requirements is not examined in detail.

Alves *et al.* [24] proposes an approach to evaluate components in terms of how well they match customer requirements and provides conflict management framework to identify the components based on resolution proposals and risk evaluation. COTS-based requirement engineering (CRE) approach highlights the importance of non-functional requirements as decisive criteria to evaluate alternative components. CRE supports the evaluation of candidate components through the definition of systematic criteria that includes a clear description of quality attributes that candidate components have to meet. However, in the cases where there are a large number of COTS alternatives, the decision-making process is complex. Another limitation of the approach is that it lacks support for the cases when non-functional requirements are not properly satisfied.

Clark *et al.* [41] proposes an iterative approach in identifying and selecting components in large repositories. The proposed model enables an integrator to search for components during the stage of requirement articulation. He first specifies his high-level functional requirements, and then decomposes the high-level function specification. A ranking process determines which components among the alternatives are the best according to some secondary characteristics. Finally, multiple selection approaches are evaluated under a global perspective. The technique also allows the integrator to progressively understand the repository structure and the characteristics of the available components. This approach increases the flexibility by allowing integrators to define characteristics as preferences rather than as constraints. The process risk is reduced by making compatibility checks by looking at the communication paths between components which ensures that short-listed components can co-operate to build a system.

4.5 Knowledge base approach

COTS-aware requirement engineering (CARE) [25] approach for component identification has the focus on the utility of knowledge base. The goals and requirements are specified as enterprise goals which are further subspecialised into component goals. CARE points out the importance to keep requirements flexible as they have to be constrained by the capabilities of available components. In this approach, requirements are classified as native requirements acquired from customers and foreign requirements of the COTS components. The method puts emphases on narrowing the gap between customer and component requirements by using knowledge base. The process model describes the activities performed to define the system agents, goals, system requirements, software requirements and architecture. The product model describes the format of the product created using the process. The meta-model describes the knowledge content and structure for the CARE approach. The method highlights the importance of mapping system requirements and product specification; however, it does not support the possible mismatch between both specifications.

Zhuge [42] proposes a problem-oriented component reuse framework by incorporating a problem-solving mechanism with the component repository. The application software system is regarded as a problem, which is solved by the co-operation of a group of agents performing problem description, component retrieval, reasoning and dynamic component checking. The framework is a three-level problem-oriented description approach which enables users to define a problem in top-down refinement form. Domain users can focus on domain problem description and problem solving. The environment evolution of running a composed application is simulated through rule reasoning. The components for composing an application are dynamically checked through simulation before composing the application. It helps avoid complex formal verifications and enhances the correctness of composed application.

Lee *et al.* [43] proposed a component-based methodology for requirement analysis and high-level design for web applications. It uses web pages as the fundamental building block of web applications and introduces pages and component classifications, and a set of tags to implement link semantics. Based on these constraints, it specifies procedures for requirement analysis and design. As the first step, application functions are identified in terms of high-level abstraction and low-level details. In high-level abstraction, the application domain is articulated in terms of its function. For low-level requirement analysis, it introduces a set of symbols for the purpose of articulation. Similarly, three steps are introduced for component specification phase, namely, rendering specification, integration specification and interface specification. The rendering specification considers the role of invisible pages and endogenous components in rendering visible pages in each compendium. The integration specification identifies relationships with other components. Finally, the interface specification summarises the interfaces of the components.

The COTS usage risk evaluation (CURE) [44] is a 'front-end' analysis tool that predicts the areas where the use of COTS products will have the greatest impact on the programme. CURE is designed to find risks relating

to the use of COTS products and report those risks back to the organisation. Ideally, CURE is performed on both the acquiring and the contracting organisation, but this is not necessary. The evaluation consists of four activities: preliminary data gathering, on-site interview, analysis of data and presentation of results. The CURE method has proven to be a useful tool for organisations that acquire or develop COTS-based systems. However, there are several limitations of the current version of CURE ranging from considerable amount of manual analytical work performed by evaluators and training required by the evaluators.

5 Component integration

Integrating components can be illustrated as a mechanical process of wiring components together. It is rarely the case that two components are perfectly matched so the process generally involves more than simply finding two components, which together perform the desired tasks, and then connecting their APIs [45]. The important issue when integrating components is to deal with the mismatches that may occur when putting together pieces developed by different parties, usually unaware of each other [46]. Thus, it is extremely important that component services are provided through a standard, published interface to ensure interoperability [18]. Component integration process composes of adaptation, validation and testing of the selected components. In the following subsections, we survey the techniques proposed for integration.

5.1 Adaptation

Each individual component is developed based on its own requirements and context. It is usually necessary to create some sort of adaptation to reduce the conflict among the selected components. Different component integration mechanisms are presented from an individual component's point of view. Filters are perhaps the oldest component integration mechanism. It only provides access to the data of the components without considering their functionality. The most complete mechanism provided by individual component is an API that allows an external component complete access to all data, functions and events. An internal programming language (IPL) is also used for component integration. It runs from inside the component in the form of macros (e.g. Microsoft Excel and Word). Concept of shared data repository is also discussed as a mechanism of component integration [47]. This method is based on the idea that multiple components share a common data repository, reading and writing the same data object.

Rine *et al.* [12] proposed the use of adapters to integrate components. Adapters are introduced to interconnect components, containing interaction's protocol and manage component interactions. In this technique, each component has an associated adapter. Component request services from each other through their associated adapters. The associated adapter is responsible for solving the syntactic interface mismatch. The individual adapters communicate with each other to fulfil component interaction. Vigder [48] has presented element of architecture for integration and has defined rules that facilities the integration. They have identified a set of component integration components that are required for the component integration. Wrappers isolate the underlying components from other components of the system, *glue* provides the functionality to combine the components and *component tailoring* is the ability to enhance the functionality of a component.

A language concept that facilities the integration of components into applications is used to declare the type of components using the notation of *collaboration interface* [13]. Collaboration interfaces facilitates the bidirectional expression of potential contexts (i.e. *client-from-server contract* and *server-from-client contract*) in which they might be integrated. The decoupling of component implementation from binding via remodularisation allows to mix and match re-modularisation and components on demand. The decoupling of component combined with integration of collaboration interfaces provides reuse in the proposed model. Recently, Dietrich *et al.* [49] have used active rules to design and generate wrappers to adapt components. The wrappers are automatically generated as enterprise java bean components and they act as proxy objects. These proxy objects intercept method calls and provide functionality required by the overall component-based system.

The concepts of aspect-oriented software development have also been incorporated into CBD integration process. For example, Assman [50] presents the concept of invasive composition and uses self-generated glue codes to integrate components. Similarly, Suvee *et al.* [51] presents the JAsCo language for CBD integration. The language is designed to be used with Java Beans component model and introduces concepts of aspect beans and connectors. An aspect bean describes behaviour that interferes with the execution of a component. Further, a connector is used to deploy the aspect beans in a given context.

5.2 Testing and validation

Validating adapted components is a major task for the CBD process. Software component testing and validation techniques focus on the expected behaviour of the component to ensure that the exhibited behaviour is correct [46]. The internal structure of the components are usually unknown, the most appropriate technique for component testing and validation is black box testing [52–55]. The black box testing is for customised parts to uncover the functional and behaviour errors of the new and altered parts in components based on given specification and its strategies include test case generation by equivalence classes, error guessing and random tests. These techniques rely on some notion of the input space and expected functionality of the component being tested [19, 56, 57].

In addition to traditional software testing and validation techniques, CBD introduces a set of new challenges, for example components must fit into the new environment and they often require real-time detection, diagnosis and handling of software faults. Built in self-testing techniques [58–60], methods of producing self-testable components is one possible solution that can help detect faults during the run time [46]. For example, build-in test (BIT) models consists of built-in testing enabled components which implement mandatory interfaces. Testers access the built-in testing capabilities of BIT components through the corresponding interfaces and which contain the test cases. Further, special components called handlers are introduced which do not directly contribute to the testing, but ensure recovery mechanisms in the case of failures.

Wang *et al.* [60] proposed a built-in test model which can operate in two modes, *normal mode* and *maintenance mode*. In the normal mode, the built-in test capabilities are transparent to the component user and the component does not differ from other, non-built-in testing enabled components. In the maintenance mode, the component user can test the component with the help of its built-in testing features.

The component user can invoke the respective methods of the component, which execute the test, evaluate autonomously its results and output a test summary.

Self-testing COTS components [58] strategy proposes to augment a component with functionality of analysis and testing tools thus enabling it to be capable of conducting some or all activities of the component user's testing processes. The idea of this strategy is to allow the component users to test the component with respect to information which is not directly accessible to the component user. The information is either generated by the component itself on demand or is encapsulated in it. In both cases, the information is transparent to the component user and is processed by the component itself for testing purposes without disclosing it.

Gao *et al.* [61] have proposed a component test model to analyse API-based component validation and testing. The test model uses the concepts of component function access graph to represent component access patterns. Further, a set of test API-based criteria is also proposed to test the models. Recently, they have also proposed a component regression test approach [62] to identify component changes and their impacts on CBD. They have also developed a tool called COMPTest which supports automatic identification analysis of API-based component changes and black box test selection.

In CBD, testing individual components is not enough for overall validation of the system. Thus, techniques for overall component-based software systems testing [63] are also fundamental for successful component integration. Test model using component interaction graph (CIG) is presented in [63], which depicts a generic infrastructure of component-based system and suggests key test elements. The interactions and dependence relationships among the components are illustrated by CIG. By utilising CIG, a family of test adequacy criteria which allow optimisation of the balancing among budget, schedule and quality requirements are proposed. This model puts emphasis on the interaction among components in component-based system and is targeted at revealing intercomponent and interoperability faults. For a detailed discussion on CBD testing, please refer to [19, 46].

6 Deployment

The selected components are integrated through well-defined infrastructure and this infrastructure provides the binding that forms a system from the disparate components. The operation of infrastructure consists of three main levels, namely highest abstract level that defines how the different components will interact to carry out the required functionality, lower level that will be used by components to interact and carryout common task; and software component level that will implement the necessary coordination services [2, 3]. The emergence of CBSE has resulted in a range of component models such as Common Object Request Broker Architecture (CORBA), Enterprise Java Beans (EJB) and .NET that define standard forms and interfaces.

Ning [4] identifies that the component deployment stage focus on addressing the two main tasks: (a) packing components so that they can connect at runtime and (b) connecting, disconnecting and reconnecting at the runtime. A tool called packager is proposed which determines how to package components. Packager analyses a component-based architecture specification written in component-based architecture specification language to determine if consistent packages are feasible, select a manufacturing plan for generating the packages based on what is feasible and outputs instructions for executing the plan. Similarly, Yau *et al.* [64] presented a framework for component integration having compatibility checks of component interface properties and enable component coupling at the runtime and Depke *et al.* [65] used the concept of ports that enabled the flexible and consistent integration of software components based on XML documents. These techniques help reduce CBD process risks by considering component specifications and their interface properties, and increase flexibility by providing quantitative analysis of candidate components.

Parrish *et al.* [66] proposed a formal model for identifying conditions under which various component deployment strategies are safe and successful. Framework identifies two types of installation: (a) successful, and (b) safe. Successful installations are the ones in which deployed applications work properly, while safe installations are the ones in which no existing applications are damaged by the installation. It then identifies conditions that are sufficient to guarantee both safe and successful installations. The framework at the current stage is mainly theoretical and needs to be applied to real-life applications in order to provide a set of guidelines for software engineers to follow when building and performing installations.

Xia *et al.* [14] proposed component integration model (CIMO) that facilities in the integration of components by presenting a *component layer* and a *service layer*. *Component layer* contains CIMO components which are based on Microsoft COM objects and service layer consists of CIMO configuration, manager, container and system. It is mainly responsible for supporting the management, communication and configuration of components. CIMO platform provides the support for the set-up and integration of the components in the CIMO application. CIMO platform has one CIMO system to coordinate the whole system and has a number of CIMO containers, which directly contain the CIMO components. To guarantee the uniqueness of the components within the application, each instantiated component is registered into a centralised registry called CIMO component manager. *CIMO* configurator implements the basic software components into a running application. It configures applications, such as process, topology, and link between components, terminates after an application system has been established, sends messages to the application system in the debug mode and shows the application system structure within each node, such as message route within a component and links of the components.

7 Component evolution

A software system goes through continuous evolution during its life cycle for improving performance, correcting faults and the like. A successful component evolution needs components to be syntactic and semantic compatible. Greenwood *et al.* [67] presents a framework approach in evolving component-based systems based on integration of product and process modelling. The evolving process is represented by *Product Tower*, a hierarchy of components which provide views of the product at multiple levels of refinements. The Product Tower provides an explicit structural description of the software as a unifying framework. An important property of this framework is scalability by utilising the product design hierarchy as the reference point for the development of a product. The ability to specialise in the process for any component allows components to be developed using differing methods and indeed different versions of the development tools.

Evolution of a system is managed through the evolution of the design hierarchy.

The concept of version and configuration is also applied in the context of CBD. A configuration model based on component libraries is presented that helps in configuration and evolution of the components in the system. The configuration model keeps track of the different dependences between the components of a component-based system. Metrics together with component documentations is used to determine the impact of a possible change on the component-based system. Development documents play a key role in the different phases of software evolution. Software evolution means change descriptions are changed over time, that is each evolution step implies changes of an appropriate set of development documents [68].

Rausch [69] proposes a well-founded common system model for components with the ability to evolve in a controlled manner. During development, each step in the evolution process implies changes to an appropriate set of development documents. In order to model the dependencies between these documents, the concept of requirement/assurance contracts is introduced. The model includes the concepts of a type and abstract as well as concrete descriptions for types. During system development, a set of these descriptions are created. Software evolution means that these descriptions are changed over time. These contracts show the consequences for the evolution of a component or set of components on the whole system. These contracts can be re-checked whenever the specification of a component evolves, enabling the determination of the impact on the respective evolution step.

8 Conclusions

In this work, we have presented a literature survey of the state-of-the-art research in the area of CBD. The areas surveyed were techniques for component identification and selection, integration, deployment and evolution. In the area of component identification and selection, researchers have proposed a set of systematic approaches to identify and select components by using hierarchical, iterative and knowledge-based analysis techniques. The aim of these techniques is to identify and rank candidate components; and finally select components which best meet stakeholder requirements. However, a majority of the proposed techniques have a selection process based on a strict requirement definition. This implies that either component have to be eliminated because they do not meet the stakeholder requirements or they will need a large change to satisfy such restrictive requirements [24]. For future work, there is a need for a collaborative process in which both stakeholder requirements and candidate components can trade-off between stakeholder requirements and limitations of available components.

Similarly, component integration needs to address the issues of verifying component compatibility and checking the compatibility against open standards. The ability for a system to be able to evolve in a controlled manner is also important and its relationship with the rest of the CBD life cycle needs to be carefully managed. Future research needs be directed to the analysis of risk factors and complexity associated with integration, interoperability and deployment of CBD. More research work should be focused on costs involved in component acquisition and integration, testing and maintainability; complexity issues related to size, interface, semantics and coupling, and overall system quality would also be of good interests to researchers.

There is a need for automatic support for CBD to realise its full potentials. As, objective of CBD is to build a system in a cost-effective way; thus, a set of tools for say, component selection and integration is essential. Other tools for component test and component configuration would be helpful. To conclude, experiences gained in CBD are valuable and need to be reported and documented more frequently. More research effort is necessary to address the risks of CBD; and further empirical research will lead to deriving more accurate models for component-based systems.

9 References

1 Crnkovic, I., and Larsson, M.: 'Challenges of component based development', *J. Syst. Software*, 2002, **61**, (3), pp. 201–212
2 Brown, A.W., and Wallnau, K.C.: 'Engineering of component based systems', *IEEE Comput. Society Press*, 1996, pp. 7–15
3 Brown, A.W., and Wallnau, K.C.: 'The current state of CBSE', *IEEE Software*, 1998, **15**, (5), pp. 37–46
4 Ning, J.Q.: 'Component Based Software Engineering (CBSE)'. Proc. Fifth Int. Symp. on Assessment of Software Tools and Technologies, 1997, pp. 34–43
5 Hopkins, J.: 'Component primer', *Commun. ACM*, 2000, **43**, (10), pp. 27–30
6 Vitharana, P.: 'Risks and challenges of component based software development', *Commun. ACM*, 2003, **46**, (8), pp. 67–72
7 Zahedi, H.J.a.P.V.a.F.M.: 'An assessment model for requirements identification in component based software development', *SIGMIS Database*, 2003, **34**, (4), pp. 48–63
8 Szyperski, C., Gruntz, D., and Murer, S.: 'Component software – beyond object – oriented programming' (Addison-Wesley, 2002, 2nd edn.)
9 Crnkovic, I., Hnich, B., Jonsson, T., and Kiziltan, Z.: 'Specification, implementation, and deployment of components', *Commun. ACM*, 2002, **45**, (10), pp. 35–40
10 Alves, C., and Castro, J.: 'CRE: a systematic method for COTS selection'. Proc. XV Brazilian Symp. on Software Eng., Curitiba, Brazil, 2001
11 Iribarne, L., Troya, J.M., and Vallecillo, A.: 'Selecting software components with multiple interfaces'. Proc. 28th Euromicro Conference, 2002, pp. 26–32
12 Rine, D., Nada, N., and Jaber, K.: 'Using adapters to reduce interaction complexity in reusable component based software development'. Proc. 1999 symp. on Software reusability, 1999, (ACM Press), pp. 37–43
13 Mezini, M., and Ostermann, K.: 'Integrating independent components with on-demand remodularization'. Proc. 17th ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Appl., 2002, (ACM Press), pp. 52–67
14 Xia, Y., Ho, A.T.S., and Zhang, Y.: 'CIMO – component integration model'. Proc. Seventh Asia-Pacific Software Eng. Conf., 2000, pp. 344–348
15 Kontio, J., Chen, S.-F., Limperos, K., Tesoriero, R., Caldiera, G., and Deutsch, M.: 'A COTS selection method and experience of its use'. Proc. 20th Annual Software Engineering Workshop, Greenbelt, Maryland, 1995
16 Tran, V., Liu, D.-B., and Hummel, B.: 'Component based systems development: challenges and lessons learned'. Proc. Eighth IEEE Int. Workshop Software Technol. Eng. Practice, 1997, pp. 452–462
17 Lee, S.D., Yang, Y.J., Cho, F.S., Kim, S.D., and Rhew, S.Y.: 'COMO: a UML-based component development methodology'. Proc. Sixth Asia Pacific Software Eng. Conf., 1999, pp. 54–61
18 Heineman, G.T., and Councill, W.T.: 'Component-based software engineering: putting the pieces together' (Addison-Wesley, 2001)
19 Gao, J.Z., Tsao, H.-S.J., and Wu, Y.: 'Testing and quality assurance for component based software' (Artech House, 2003)
20 Kitchenham, B.A., Dyba, T., and Jorgensen, M.: 'Evidence-based software engineering'. Proc. 26th International Conf. Software Engineering, 2004, pp. 273–281
21 Dyba, T., Kitchenham, B.A., and Jorgensen, M.: 'Evidence-based software engineering for practitioners', *IEEE Softw.*, 2005, **22**, (1), pp. 58–65
22 Maiden, N.A., and Ncube, C.: 'Acquiring COTS software selection requirements', *IEEE Software*, 1998, **15**, (2), pp. 46–56

23 Leung, K.R.P.H., and Leung, H.K.N.: 'On the efficiency of domain based COTS product selection method', *Info. Softw. Technol.*, 2002, **44**, (12), pp. 703–715
24 Alves, C., and Finkelstein, A.: 'Investigating conflicts in COTS decision-making', *Int. J. Softw. Eng. Knowledge Eng.*, 2003, **13**, pp. 1–21
25 Chung, L., and Cooper, K.: 'Knowledge based COTS aware requirements engineering approach'. Proc. 14th Int. Conf. Software Eng. Knowledge Eng., 2002, (ACM Press), pp. 175–182
26 Sommerville, I.: 'Integrated requirements engineering: a tutorial', *IEEE Softw.*, 2005, **22**, (1), pp. 16–23
27 Cechich, A., and Piattini, M.: 'Early detection of COTS component functional suitability', *Info. Softw. Technol.*, 2007, **49**, (2), pp. 108–121
28 Zaremski, A.M., and Wing, J.M.: 'Signature matching: a tool for using software libraries', *ACM Trans. Softw. Eng. Methodol.*, 1995, **4**, (2), pp. 146–170
29 Mili, R., Mili, A., and Mittermeir, R.T.: 'Storing and retrieving software components: a refinement based system', *IEEE Trans. Softw. Eng.*, 1997, **23**, (7), pp. 445–460
30 Zaremski, A.M., and Wing, J.M.: 'Specification matching of software components', *ACM Trans. Softw. Eng. Methodol.*, 1997, **6**, (4), pp. 333–369
31 Cechich, A., Requile-Romanczuk, A., Aguirre, J., and Luzuriaga, J.M.: 'Trends on COTS component identification'. Proc. Fifth Int. Conf. Commercial-Off-the-Shelf (COTS)-Based Software Systems (ICCBSS'06), 2006, IEEE Computer Society
32 Pahl, C.: 'An ontology for software component matching'. Proc. Sixth Int. Conf. Fundamental Approaches to Softw. Eng., LNCS 2624, 2003, pp. 6–21
33 Carvallo, J., Franch, X., Quer, C., and Torchiano, M.: 'Characterization of a taxonomy for business applications and the relationship among them'. Proc. Third Int. Conf. COTS-Based Software Systems (ICCBSS'04), LNCS 2959, 2004, pp. 221–231
34 Bianchi, A., Caivano, D., Conradi, R., Jaccheri, L., Torchiano, M., and Visaggio, G.: 'COTS products characterization: proposal and empirical assessment'. Proc. Empirical Methods and Studies in Software Eng., LNCS 2765, Orlando, Florida, USA, 2003
35 Jilani, L., and Desharnais, J.: 'Defining and applying measure of distance between specifications', *IEEE Trans. Softw. Eng.*, 2001, **27**, (8), pp. 673–703
36 Inoue, K., Yokomori, R., Fujiwara, H., Yamamoto, T., Matsushita, M., and Kusumoto, S.: 'Component rank: relative significance rank for software component search'. Proc. 25th Int. Conf. Softw. Eng., 2003, IEEE Computer Society, pp. 14–24
37 Damiani, E., Fugini, M.G., and Bellettini, C.: 'Corrigenda: a hierarchy-aware approach to faceted classification of object oriented components', *ACM Trans. Softw. Eng. Methodol.*, 1999, **8**, (4), pp. 425–472
38 Smith, E., Al-Yasiri, A., and Merabti, M.: 'A multitiered classification scheme for component retrieval'. Proc. 24th Euromicro Conf., 1998, pp. 882–889
39 Jain, H., Chalimeda, N., Ivaturi, N., and Reddy, B.: 'Business component identification – a formal approach'. Proc. Fifth IEEE Int. Enterprise Distributed Object Computing Conf., EDOC '01, 2001, pp. 183–187
40 Lee, J.K., Jung, S.J., Kim, S.D., Jang, W.H., and Ham, D.H.: 'Component identification method with coupling and cohesion'. Eighth Asia-Pacific Software Eng. Conf., 2001, pp. 79–86
41 Clark, J., Clarke, C., Panfilis, S.D., Granatella, G., Predonzani, P., Sillitti, A., Succi, G., and Vernazza, T.: 'Selecting components in large COTS repositories', *J. Syst. Softw.*, 2004, **73**, pp. 323–331
42 Zhuge, H.: 'A Problem oriented and rule based component repository', *J. Syst. Softw.*, 2000, **50.**, (3), pp. 201–208
43 Lee, S.C., and Shirani, A.I.: 'A component based methodology for web application development', *J. Syst. Softw.*, 2004, **71**, pp. 177–187
44 Carney, D.J., Morris, E.J., and Place, P.R.H.: 'Identifying commercial off-the-shelf (COTS) product risks: the COTS usage risk evaluation'. Carnegie Mellon Software Engineering Institute (SEI), p. CMU/SEI-2003-TR-023, 2003
45 Crnkovic, I., and Larrson, M.: 'Building component-based reliable software systems' (Artech House, 2002)
46 Cechich, A., Piattini, M., and Vallecillo, A.: 'Assessing component based systems. Component based software quality', LNCS 2693, 2003, pp. 1–20
47 Rader, J.A.: 'Mechanisms for integration and enhancement of software components'. Proc. Fifth Int. Symp. on Assessment of Software Tools and Technologies, 1997, pp. 24–31
48 Vigder, M.R., and Dean, J.: 'An architectural approach to building systems from COTS software components'. Proc. 1997 Conf. Centre for Advanced Studies on Collaborative Research, 1997, IBM Press
49 Dietrich, S.W., Patil, R., Sundermier, A., and Urban, S.D.: 'Component adaptation for event-based application integration using active rules', *J. Syst. Softw.*, 2006, **79**, (12), pp. 1725–1734
50 Assman, U.: 'A component model for invasive composition'. Proc. ECOOP 2000 Workshop on Aspects and Dimensions of Concerns, Cannes, France, 2000
51 Suvee, D., Vanderperren, W., and Jonckers, V.: 'JAsCo: an aspect-oriented approach tailored for component based software development'. Proc. Second International Conf. on Aspect-Oriented Software Development, ACM Press, Boston, USA, 2003
52 Goa, J., Gupta, K., and Gupta, S.: 'On building testable software components'. Proc. First Int. Conf. on COTS Based Software Systems, Springer-Verlag, 2002, pp. 108–121
53 Korel, B.: 'Black-box understanding of COTS components'. Proc. Seventh Int. Workshop on Program Comprehension, 1999, IEEE Press
54 Mueller, C., and Korel, B.: 'Automated Black-box evaluation of COTS components with multiple-interfaces'. Proc. Second Int. Workshop on Autom. Program Analysis, Testing and Verification, ICSE, 2001
55 Tahat, L.: 'Requirement based automated Black-box test generation'. Proc. 25th Annual Int. Com. Software and Application Conf., 2001, IEEE Computer Society Press, pp. 489–495
56 Weyuker, E.J.: 'Testing component based software: a cautionary tale', *IEEE Softw.*, 1998, **15**, (5), pp. 54–59
57 Gao, J.: 'Challenges and problems in testing software components'. Proc. Third Int. Workshop on Component Based Software Engineering (ICSE 2000), Limerick, Ireland, 2000
58 Beydeda, S., and Gruhn, V.: 'Merging components and testing tools: the self-testing COTS components (STECC) strategy'. Proc. 29th Euromicro Conf., 2003, pp. 107–114
59 Hornstein, J., and Edler, H.: 'Test reuse in CBSE using built-in tests'. Proc. Workshop on Composing Systems from Components and Component-Based Software Engineering, Lund, 2002
60 Wang, Y., King, G., and Wickburg, H.: 'A method for built-in tests in component based software maintenance'. Proc. Third European Conf. on Software Maintenance and Reengineering, 1999, pp. 186–189
61 Gao, J., Espinoza, R., and He, J.: 'Testing coverage analysis for software component validation'. Proc. 29th Annual Int. Computer Software and Application Conf. (COMPSAC'05), Edinburgh, UK, 2005, pp. 463–470
62 Gao, J., Gopinathan, D., Mai, Q., and He, J.: 'A systematic regression testing method and tool for software components'. 30th Annual Int. Comp. Software and Application Conference (COMPSAC'06), Chicago, USA, 2006, pp. 455–466
63 Wu, Y., Pan, D., and Chen, M.-H.: 'Techniques for testing component-based software'. Proc. Seventh IEEE Int. Conf. on Engineering of Complex Computer Systems, 2001, pp. 186–189
64 Yau, S.S., and Karim, F.: 'Integration of object-oriented software components for distributed application software development'. Proc. Seventh IEEE Workshop on Future Trends of Distributed Computing Systems, 1999, pp. 111–116
65 Depke, R., Engels, G., Thone, S., Langham, M., and Lutkemeier, B.: 'Process-oriented, consistent integration of software components'. Proc. 26th Annual Int. Comp. Software and Applications Conf., COMPSAC 2002, 2002, pp. 13–18
66 Parrish, A., Dixon, B., and Cordes, D.: 'A conceptual foundation for component based software deployment', *J. Syst. Softw.*, 2001, **57**, (3), pp. 193–200
67 Greenwood, R.M., Warboys, B.C., and Sa, J.: 'Co-operating evolving components: a rigorous approach to evolving large software systems'. Proc. 18th Int. Conf. on Software Engineering, IEEE Computer Society, 1996, pp. 428–437
68 Casanova, M., Straeten, R.V.D., and Jonckers, V.: 'Supporting evolution in component based development using component libraries'. Proc. Seventh European Conf. on Software Maintenance and Re-engineering (CSRM), 2003, pp. 123–133
69 Rausch, A.: 'Software evolution in component ware using requirements/assurances contracts'. Proc. 22nd Int. Conf. on Software Eng., ACM Press, 2000, pp. 147–156