International Journal of Software Engineering and Knowledge Engineering
Vol. 12, No. 2 (2002) 107–133

COMPONENT-BASED EMBEDDED SOFTWARE ENGINEERING: DEVELOPMENT FRAMEWORK, QUALITY ASSURANCE AND A GENERIC ASSESSMENT ENVIRONMENT

XIA CAI[*,‡], MICHAEL R. LYU[*,§] and KAM-FAI WONG[†,¶]

[*]*Department of Computer Science and Engineering,*
[†]*Department of System Engineering and Engineering Management,*
The Chinese University of Hong Kong, Hong Kong
[‡]*xcai@cse.cuhk.edu.hk*
[§]*lyu@cse.cuhk.edu.hk*
[¶]*kfwong@se.cuhk.edu.hk*

Embedded software is used to control the functions of mechanical and physical devices by dedicated digital signal processor and computers. Nowadays, heterogeneous and collaborative embedded software systems are widely adopted to engage the physical world. To make such software extremely reliable, very efficient and highly flexible, component-based embedded software development can be employed for the complex embedded systems, especially those based on object-oriented (OO) approaches. In this paper, we introduce a component-based embedded software framework and the features it inherits. We propose a quality assurance (QA) model for component-based embedded software development, which covers both the component QA and the system QA as well as their interactions. Furthermore, we propose a generic quality assessment environment for component-based embedded systems: ComPARE. ComPARE can be used to assess real-life off-the-shelf components and to evaluate and validate the models selected for their evaluation. The overall component-based embedded systems can then be composed and analyzed seamlessly.

Keywords: Embedded software; component-based embedded system; quality assurance; CORBA; COM/DCOM; JavaBeans

1. Introduction

Embedded software is used to control the functions of mechanical and physical devices by dedicated digital signal processors and computers. Today, embedded software appears in everything from telephones and pagers, portable MP3 players, television set-top boxes, digital cameras to systems for medical diagnostics, climate control and manufacturing [1]. Once deemed too small and retro, embedded software systems have drawn the interest and attention of researchers recently, because of their wide adoption and the changes of hardware capabilities. The main task of embedded software is to engage the physical world, interacting directly with sensors and actuators. Typically, such software must be extremely reliable, very efficient and compact, and precise in its handling of the rapid and unpredictable timing of

inputs and outputs [2]. Generally, embedded systems have the following features: inherent complexity, hardware/software concurrency, tight cost and performance constraints, and heterogeneous.

To achieve high reliability, efficiency and flexibility, embedded software systems can be built from some existing modules from a third party [3]. Such modules can be regarded as "components", which is the fundamental idea of component-based software development (CBSD). The component-based software development approach is one of the most promising solutions for the emerging high development cost, low productivity, unmanageable software equality and high risk, and move to new technology in software development today [4]. This approach is based on the idea that software systems can be developed by selecting appropriate off-the-shelf components and then assembling them with a well-defined software architecture [5]. This new software development approach is very different from the traditional approach in which software systems can only be implemented from scratch. These commercial off-the-shelf (COTS) components can be developed by different developers using different languages and different platforms. This can be shown in Fig. 1, where COTS components can be checked out from a component repository, and assembled into a target software system.

In general, a component has three main features:

(1) a component is an independent and replaceable part of a system that fulfills a clear function;
(2) a component works in the context of a well-defined architecture; and
(3) a component communicates with other components by its interface [6].

Current component technologies have been used to implement different embedded software systems, such as embedded web servers and embedded databases [3]. As embedded software often encapsulates domain expertise, and becomes more complex, modular, adaptive, and network-aware, the emerging embedded software components business is a consequence of this trend [2].

To ensure that a component-based software system can run properly and effectively, the system architecture is the most important factor. According to both the research community [7] and industry practice [8], the system architecture of component-based embedded software systems should be a layered and modular architecture. This architecture can be seen in Fig. 2. The top application layer is the application systems supporting various customers. The second layer consists of components engaged in only a specific embedded-system or application domain, including components usable in more than a single application. The third layer is cross-system middleware components consisting of common software and interfaces to other established entities. The fourth layer of system software components includes basic components that interface with the underlying operating systems and hosting hardware. Finally, the lowest two layers are the operating system and hardware layers [9].

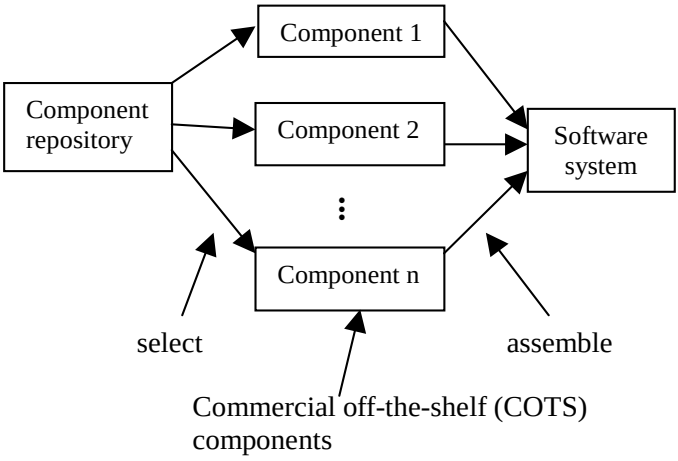

Fig. 1. Component-based embedded software development.

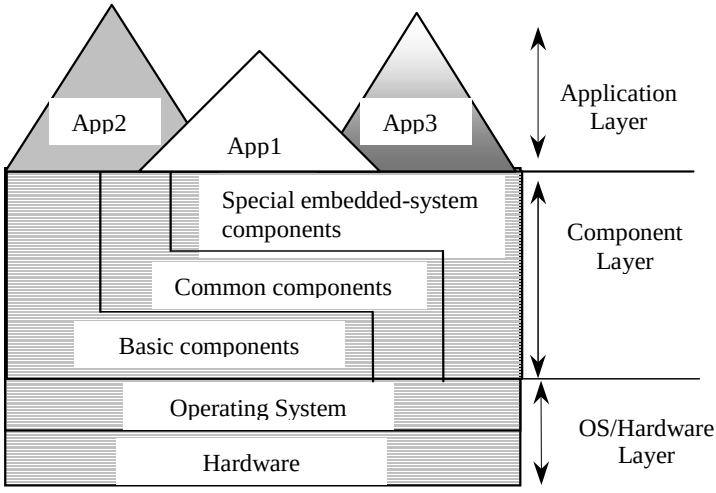

Fig. 2. System architecture of component-based software systems.

Component-based embedded software engineering will introduce component-based software development approach into the embedded software systems. Component-based embedded software development can significantly reduce development cost and time-to-market, and improve maintainability, reliability and overall quality of embedded software systems [10].

Traditional embedded software is often developed using low-level programming languages, such as an assembler or C. However, it is not the case for complex embedded systems because modern embedded systems require software reuse and maintenance, as well as a significant amount of expertise. Also the aggregation and

connection of embedded systems over the Internet needs to cope with the heterogeneous, collaborative systems [10]. As object-oriented programming languages have greatly enhance both software reusability and software quality, they can be used to satisfy complex embedded systems. Object-oriented languages and some new techniques based on them can be applied to embedded systems [11–13]. In this paper, we will address the component-based embedded software engineering based on object-oriented approaches.

The rest of this paper will introduce the current technologies for component-based embedded software, and the quality assurance (QA) issues of such approaches. We formulate a QA model which addresses the quality management in the component-based embedded software development process. We also propose a generic quality assessment environment to simulate the process of selecting qualified components from components repository to build a component-based embedded software, then predict and evaluate the final system based on these components.

2. A Development Framework for Component-Based Embedded Systems

A framework can be defined as a set of constraints on the components and their interactions, and a set of benefits that derive from those constraints [2]. To identify the development framework for component-based embedded software, the framework or infrastructure for components should be identified first, as components are the basic units in the component-based embedded systems.

Some approaches, such as Visual Basic Controls (VBX), ActiveX controls, class libraries, and JavaBeans, make it possible for their related languages, such as Visual Basic, C++, Java, and the supporting tools to share and distribute application pieces. But all of these approaches rely on certain underlying services to provide the communication and coordination necessary for the application. The infrastructure of components (sometimes called a *component model*) acts as the "plumbing" that allows communication among components [6]. Among the component infrastructure technologies that have been developed, three have become standardized: OMG's CORBA, Microsoft's Component Object Model (COM) and Distributed COM (DCOM), and Sun's JavaBeans and Enterprise JavaBeans [14]. All these technologies can be applied to component-based embedded systems [11,12,13].

2.1. *Common Object Request Broker Architecture (CORBA)*

CORBA is an open standard for application interoperability that is defined and supported by the Object Management Group (OMG), an organization of over 400 software vendors and object technology user companies [15]. Simply stated, CORBA manages details of component interoperability, and allows applications to communicate with one another despite their different locations and designers. The interface is the only way that applications or components communicate with each other.

The most important part of a CORBA system is the Object Request Broker (ORB). The ORB is the middleware that establishes the client-server relationships between components. Using an ORB, a client can invoke a method on a server object, whose embedded location is completely transparent. The ORB is responsible for intercepting a call, finding an object that can implement the request, passing its parameters, invoking its method, and returning the results. The client does not need to know where the object is located, its programming language, its operating system, or any other system aspects that are not related to the interface. In this way, the ORB provides interoperability among embedded applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

CORBA is widely used in Object-Oriented distributed systems [16] including component-based embedded software systems because it offers a consistent distributed programming and run-time environment over common programming languages, operating systems, and distributed networks. The OMG has defined two standards for embedded applications: Minimum CORBA and Real-Time CORBA. Minimum CORBA defines a standard, fully interoperable subset (profile) of CORBA functionality that is appropriate for resource-constraint applications, while Real-Time CORBA extends CORBA so that it can be used to build deterministic applications [12].

2.2. *Component Object Model (COM) and Distributed COM (DCOM)*

Introduced in 1993, Component Object Model (COM) is a general architecture for component software [17]. It provides platform-dependent (based on Windows and Windows NT), and language-independent component-based applications.

COM defines how components and their clients interact. This interaction is defined such that the client and the component can connect without the need for any intermediate system component. Specially, COM provides a binary standard that components and their clients must follow to ensure dynamic interoperability. This enables on-line software update and cross-language software reuse [18].

As an extension of the Component Object Model (COM), Distributed COM (DCOM), is a protocol that enables software components to communicate directly over a network in a reliable, secure, and efficient manner. DCOM is designed for use across multiple network transports, including Internet protocols such as HTTP. When a client and its component reside on different embedded machines, DCOM simply replaces the local interprocess communication with a network protocol. Neither the client nor the component is aware of the changes of the physical connections.

More about COM/DCOM, and how to create implementations of COM/DCOM for embedded systems, can be found in [13].

2.3. *Sun Microsystems's JavaBeans and Enterprise JavaBeans*

Sun's Java-based component model consists of two parts: the JavaBeans for client-side component development and the Enterprise JavaBeans (EJB) for the server-side component development. The JavaBeans component architecture supports embedded applications of multiple platforms, as well as reusable, client-side and server-side components [19].

The Java platform offers an efficient solution to the portability and security problems through the use of portable Java bytecodes and the concept of trusted and untrusted Java applets. Java provides a universal integration and enabling technology for embedded enterprise application development, including

(1) interoperating across multivendor servers;
(2) propagating transaction and security contexts;
(3) servicing multilingual clients; and
(4) supporting ActiveX via DCOM/CORBA bridges.

JavaBeans and EJB extend all native strengths of Java including portability and security into the area of component-based development. The portability, security, and reliability of Java are well suited for developing robust server objects independent of operating systems, Web servers and database management servers.

2.4. *Comparison among different architectures*

Comparison among the development technologies for component-based embedded systems can be found in [6], [20] and [21]. We summarize their different features in Table 1.

3. Quality Assurance for Component-Based Embedded Systems

3.1. *The life cycle of component-based embedded software systems*

Component-based embedded software systems are developed by selecting various components and assembling them together rather than programming an overall system from scratch, thus the life cycle of component-based embedded software systems is different from that of traditional embedded software systems. The life cycle of component-based embedded software systems can be summarized as follows [5]:

(1) Requirements analysis;
(2) Embedded software architecture selection, construction, analysis, and evaluation;
(3) Component identification and customization;
(4) Embedded system integration;
(5) Embedded system testing;
(6) Software maintenance.

Table 1. Comparison of development technologies for component-based embedded systems.

	CORBA	EJB	COM/DCOM
Development environment	Underdeveloped	Emerging	Supported by a wide range of strong development environments
Binary interfacing standard	Not binary standards	Based on COM; Java specific	A binary standard for component interaction is the heart of COM
Compatibility & portability	Particularly strong in standardizing language bindings; but not so portable	Portable by Java language specification; but not very compatible.	Not having any concept of source-level standard of standard language binding.
Modification & maintenance	CORBA IDL for defining component interfaces, need extra modification & maintenance	Not involving IDL files, defining interfaces between component and container. Easier modification & maintenance.	Microsoft IDL for defining component interfaces, need extra modification & maintenance
Services provided	A full set of standardized services; lack of implementations	Neither standardized nor implemented	Recently supplemented by a number of key services
Platform dependency	Platform independent	Platform independent	Platform dependent
Language dependency	Language independent	Language dependent	Language independent
Implementation	Strongest for traditional enterprise computing	Strongest on general Web clients.	Strongest on the traditional desktop applications

The architecture of embedded software defines an embedded system in terms of computational components and interactions among the components. The focus is on composing and assembling components that are likely to have been developed separately, and even independently. Component identification, customization and integration is a crucial activity in the life cycle of component-based embedded systems. It includes two main parts: (1) evaluation of each candidate COTS component based on the functional and quality requirements that will be used to assess that component; and (2) customization of those candidate COTS components that should be modified before being integrated into new component-based embedded software systems. Integration is to make key decisions on how to provide communication and coordination among various components of a target embedded software system.

Quality assurance for component-based embedded software systems should address the life cycle and its key activities to analyze the components and achieve high quality component-based embedded software systems. QA technologies for component-based embedded software systems are currently premature, as the specific characteristics of component systems differ from those of traditional systems.

Although some QA techniques such as reliability analysis model for distributed software systems [22, 23] and component-based approach to software engineering [24] have been studied, there is still no clear and well-defined standards or guidelines for component-based embedded software systems. The identification of the QA characteristics, along with the models, tools and metrics, are all under urgent needs. In Sec. 4 we propose an initial attempt to draft a simple QA model for the development of component-based embedded software systems.

3.2. *Quality characteristics of components*

As much work is yet to be done for component-based embedded software development, QA technologies for component-based embedded software development has to address the two inseparable parts: (1) How to certify the quality of a component? (2) How to certify the quality of the whole embedded system based on components? To answer these questions, models should be promoted to define the overall quality control of components and systems; metrics should be found to measure the size, complexity, reusability and reliability of components and systems; and tools should be decided to test the existing components and systems. To evaluate a component, we must determine how to certify the quality of the component. The quality characteristics of components are the foundation to guarantee the quality of the components, and thus the foundation to guarantee the quality of the whole component-based embedded software systems. Here we suggest a list of recommended characteristics for the quality of components: (1) Functionality; (2) Interface; (3) Usability; (4) Testability; (5) Maintainability; (6) Reliability.

Software metrics can be proposed to measure software complexity and assure its quality [25, 26]. Such metrics often used to classify components include [27]:

(1) **Size.** This affects both reuse cost and quality. If it is too small, the benefits will not exceed the cost of managing it. If it is too large, it is hard to achieve high quality.
(2) **Complexity.** This also affects reuse cost and quality. A too-trivial component is not profitable to reuse while a too-complex component is hard to inherit high quality.
(3) **Reuse frequency.** The number of incidences where a component is used is a solid indicator of its usefulness.
(4) **Reliability.** The probability of failure-free operations of a component under certain operational scenarios [28].

4. A Quality Assurance Model for Component-Based Embedded Systems

Because component-based embedded software systems are developed on an underlying process different from that of the traditional software, their quality assurance model should address both the process of components and the process of the overall

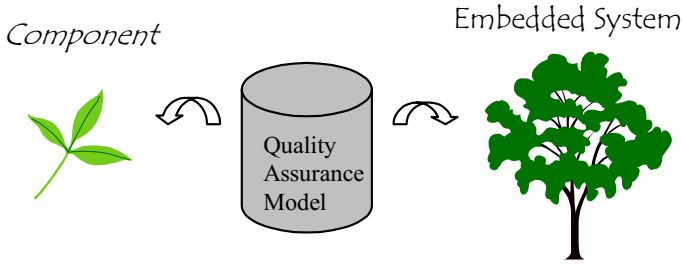

Fig. 3. Quality assurance model for both components and embedded systems.

system. Figure 3 illustrates this view.

Many standards and guidelines are used to control the quality activities of software development process, such as ISO9001 and CMM model. In particular, the Hong Kong Productivity Council has developed the HKSQA model to localize the general SQA models [29]. In this section, we propose a framework of quality assurance model for the component-based embedded software development paradigm.

The main practices relating to components and embedded systems in this model contain the following phases: (1) Component requirement analysis; (2) Component development; (3) Component certification; (4) Component customization; (5) System architecture design; (6) System integration; (7) System testing; and (8) System maintenance. Details of these phases and their activities are described as follows.

4.1. *Component requirement analysis*

Component requirement analysis is the process of discovering, understanding, documenting, validating and managing the requirements for a component. The objectives of component requirement analysis are to produce complete, consistent and relevant requirements that a component should realize, as well as the programming language, the embedded platform and the interfaces related to the component.

The component requirement process overview diagram is shown in Fig. 4. Initiated by the request of users or customers for new development or changes on old embedded systems, component requirement analysis consists of four main steps: requirements gathering and definition, requirement analysis, component modeling, and requirement validation. The output of this phase is the current user requirement documentation, which should be transferred to the next component development phase, and the user requirement changes for the system maintenance phase.

4.2. *Component development*

Component development is the process of implementing the requirements for a well-functional, high quality component with multiple interfaces. The objectives of component development are the final component products, the interfaces, and development documents. Component development should lead to the final compo-

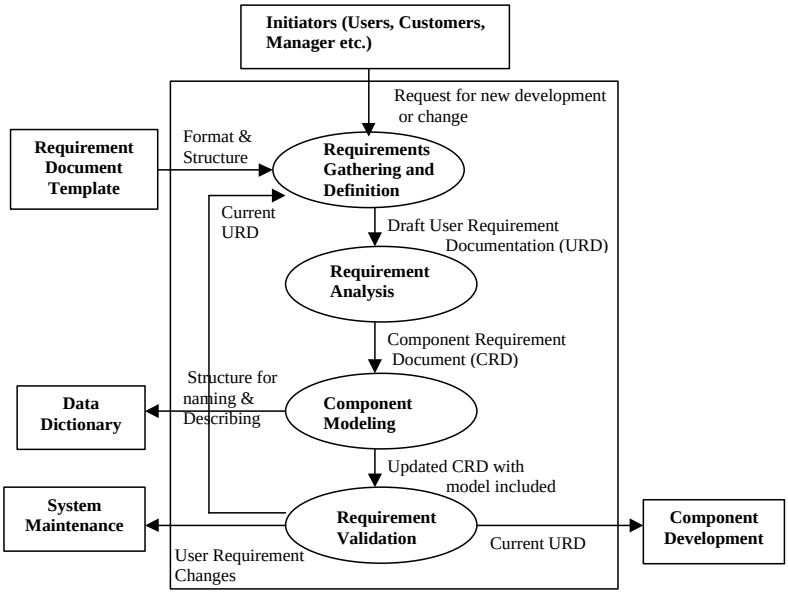

Fig. 4. Component requirement analysis process overview.

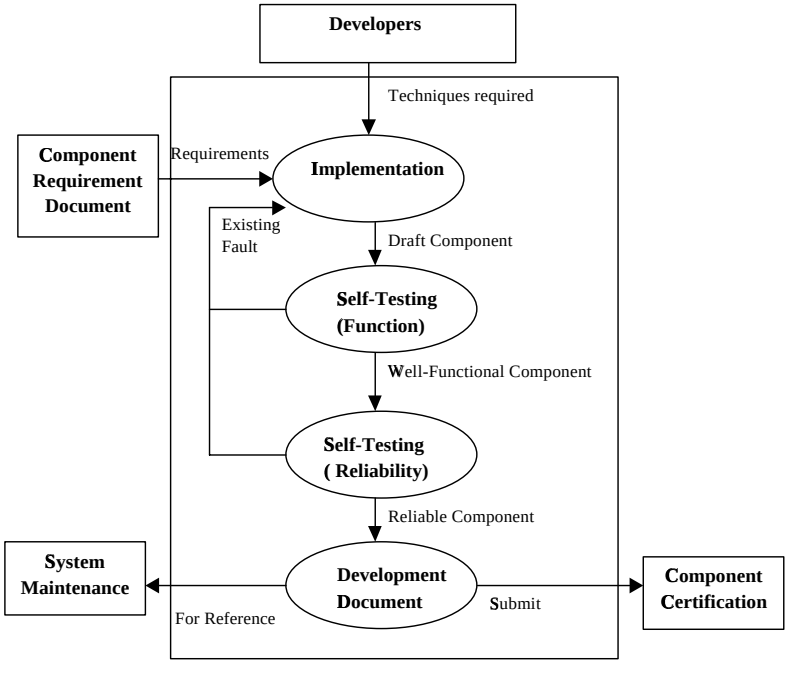

Fig. 5. Component development process.

nents satisfying the requirements with correct and expected results, well-defined behaviors, and flexible interfaces.

The component development process overview diagram is shown in Fig. 5. Component development consists of four procedures: implementation, function testing, reliability testing, and development document. The input to this phase is the component requirement document. The output should be the developed component and its documents, ready for the following phases of component certification and system maintenance, respectively.

The components in embedded systems contain both hardware components (i.e., programmable components such as micro-controllers and Digital Signal Processors [30]) and software components (i.e., basic components including database access and security, communication components and industry-specific components such as phone API and smart card access).

4.3. *Component certification*

Component certification is the process that involves:

(1) *component outsourcing*: managing a component outsourcing contract and auditing the contractor performance;
(2) *component selection*: selecting the right components in accordance with the requirement for both functionality and reliability; and
(3) *component testing*: confirm that the component satisfies the requirement with acceptable quality and reliability.

The objectives of component certification are to outsource, select and test the candidate components and check whether they satisfy the system requirement with high quality and reliability. The governing policies are:

(1) Component outsourcing should be charged by a software contract manager;
(2) All candidate components should be tested to be free from all known defects; and
(3) Testing should be in the target embedded environment or a simulated environment.

The component certification process overview diagram is shown in Fig. 6. The input to this phase should be component development document, and the output should be testing documentation for system maintenance.

4.4. *Component customization*

Component customization is the process that involves (1) modifying the component for the specific requirement; (2) doing necessary changes to run the component on embedded platforms; (3) upgrading the specific component to get better performance or higher quality. The objectives of component customization are to make

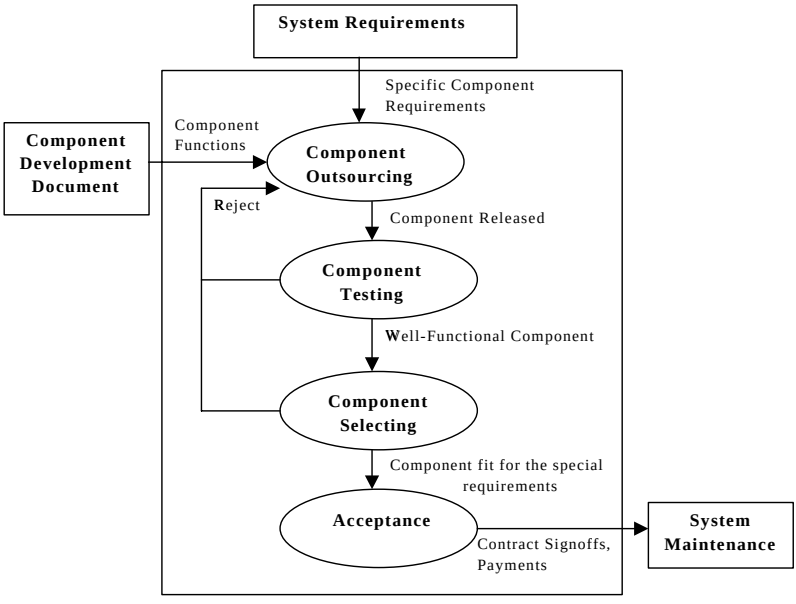

Fig. 6. Component certification process overview.

necessary changes for a developed component so that it can be used in an embedded environment or cooperate with other components well.

All components must be customized according to the operational system requirements or the interface requirements with other components in which the components should work. The component customization process overview diagram is shown in Fig. 7. The input to component customization is the system requirement, the component requirement, and component development document. The output should be the customized component and document for system integration and system maintenance.

4.5. *System architecture design*

System architecture design is the process of evaluating, selecting and creating software architecture of a component-based embedded system. The objectives of system architecture design are to collect the users requirement, identify the system specification, select appropriate system architecture, and determine the implementation details such as platform, programming languages, etc.

System architecture design should address the advantage for selecting a particular embedded system architecture from other architectures. The process overview diagram is shown in Fig. 8. This phase consists of system requirement gathering, analysis, embedded system architecture design, and system specification. The output of this phase should be the embedded system specification document for integration, and the embedded system requirement for the system testing phase

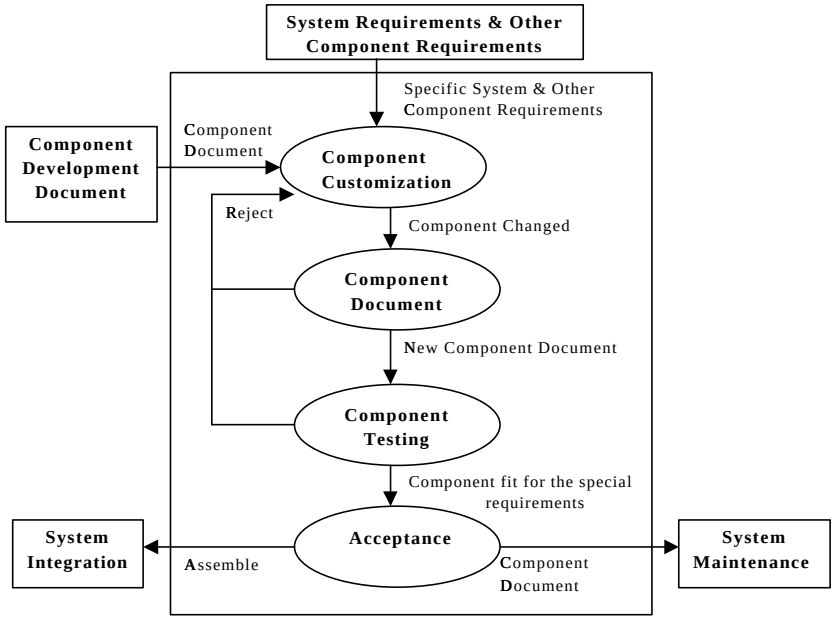

Fig. 7. Component customization process overview.

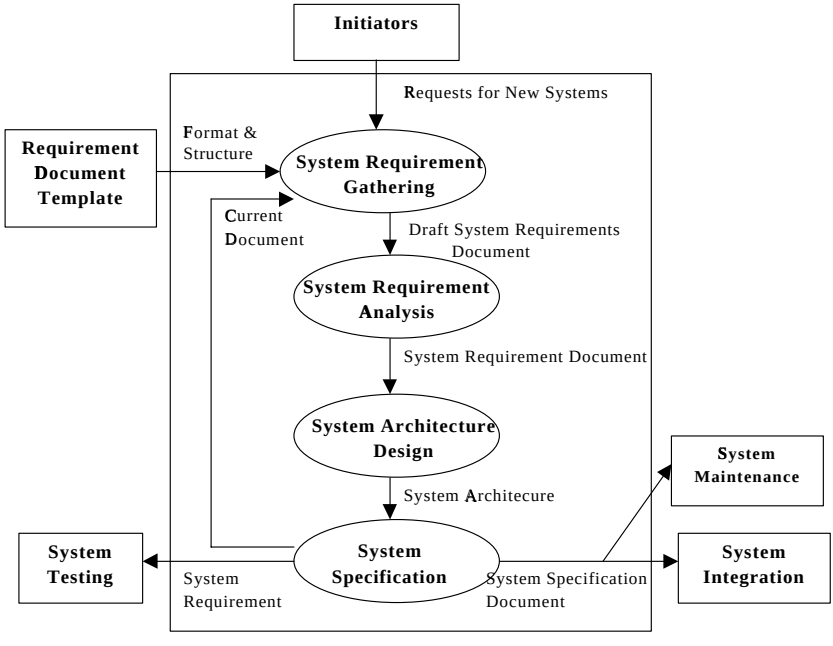

Fig. 8. System architecture design process overview.

and system maintenance phase.

Some of the embedded system design efforts address the hardware-software co-design issue, i.e., the concurrent development of standard hardware components, the selection of programmable components, and the development of the application software that will run on them. Others emphasize the sequence consisting of the initial functional design and its analysis, the mapping of such functional description into architecture, and the consequent performance evaluation and validation [30].

4.6. *System integration*

System integration is the process of assembling the selected components into a complete system under the designed embedded system architecture. The objective of system integration is the final system composed by the selected components. The process overview diagram is shown in Fig. 9. The input is the embedded system requirement documentation and the specific architecture. There are four steps in this phase: integration, testing, changing component and re-integration (if necessary). After exiting this phase, we will get the final embedded system ready for the system testing phase, and the document for the system maintenance phase. It is noted that system integration should emphasize specific properties related to the integration of embedded components, such as timing issues, synchronization issues, etc.

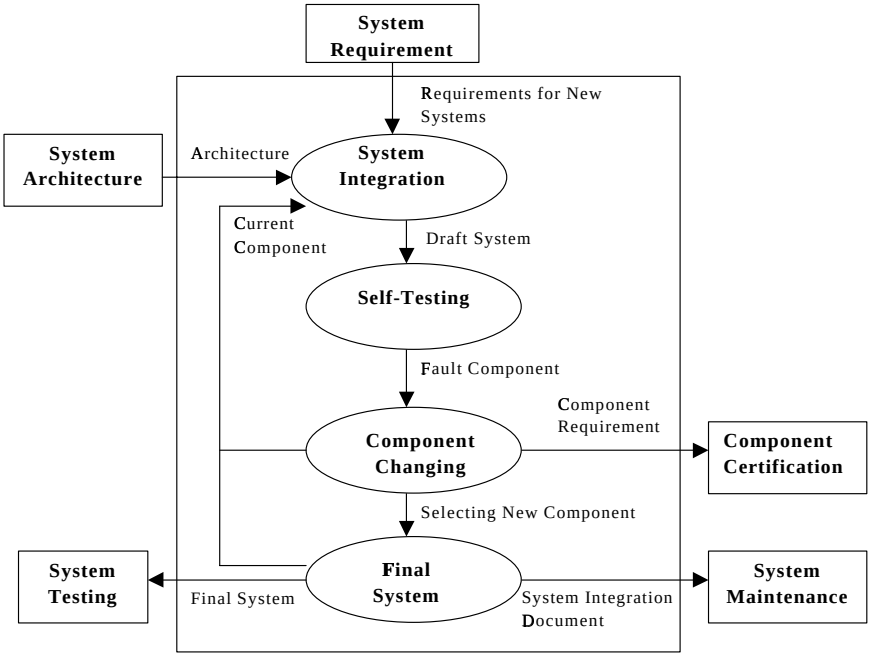

Fig. 9. System integration process overview.

4.7. *System testing*

System testing is the process of evaluating a system to: (1) confirm that the embedded system satisfies the specified requirements; (2) identify and correct defects in the system implementation. The objective of system testing is the final embedded system integrated by components selected in accordance with the system requirements. System testing should contain functional testing and reliability testing. The process overview diagram is shown in Fig. 10. This phase consists of selecting testing strategy, system testing, user acceptance testing, and completion activities. The input should be the documents from component development and system integration phases, and the output should be the testing documentation for system maintenance. Note this procedure should address the interaction testing of embedded components, such as coordination issues, deadlocks, etc.

4.8. *System maintenance*

System maintenance is the process of providing service and maintenance activities needed to use the software effectively after it has been delivered. The objectives of system maintenance are to provide an effective product or service to the end-users while correcting faults, improving software performance or other attributes, and adapting the embedded system to a changed environment.

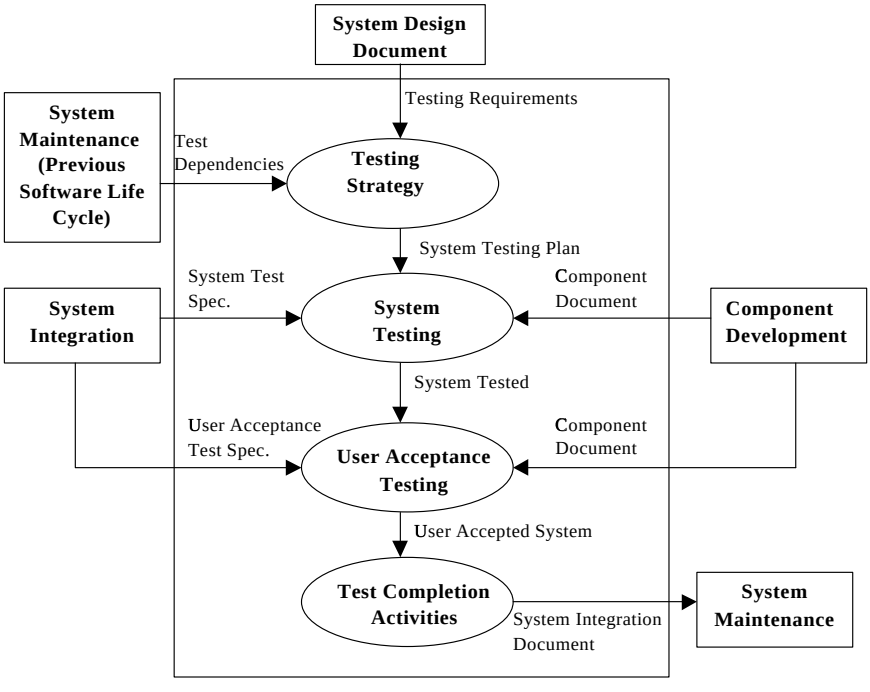

Fig. 10. System testing process overview.

There shall be a maintenance organization for every software product in the operational use. All changes for the delivered system should be reflected in the related documents. The process overview diagram is shown in Fig. 11. According to the outputs from all previous phases as well as requests and problem reports from users, system maintenance should be held for determining support strategy and problem management (e.g., identification and approval). As the output of this phase, a new version can be produced for system testing phase for a new life cycle.

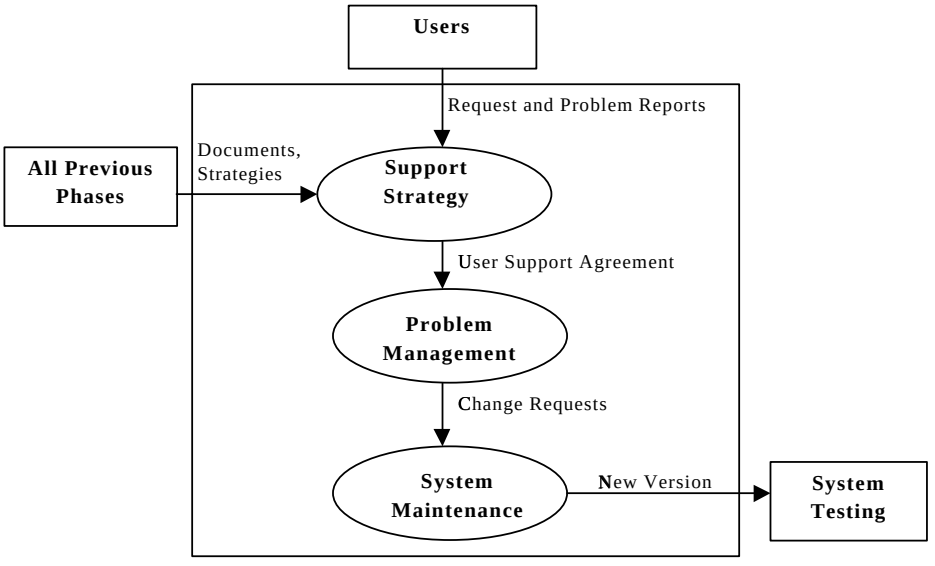

Fig. 11. System maintenance process overview.

5. A Generic Quality Assessment Environment for Component-Based Embedded Systems — ComPARE

We propose a Component-based Program Analysis and Reliability Evaluation (ComPARE) to evaluate the quality of software systems in component-based embedded software development. ComPARE automates the collection of different metrics, the selection of different prediction models, the formulation of user-defined models, and the validation of established models according to faulty data collected during the development process. Different from other existing tools [31], ComPARE takes dynamic metrics into account (such as code coverage and performance metrics), integrates them with process metrics and more static code metrics for object-oriented programs (such as complexity metrics, coupling and cohesion metrics, inheritance metrics), and provides different estimation models for overall system assessment with high accuracy.

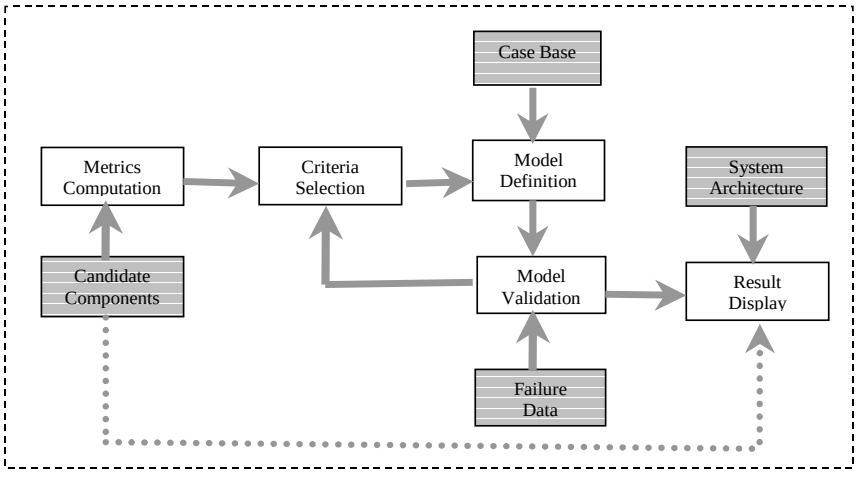

Fig. 12. Architecture of ComPARE.

5.1. *Objective*

A number of commercial tools are available for the measurement of software metrics for object-oriented programs in embedded systems. Also there are off-the-shelf tools for testing or debugging software components. However, few tools can measure the static and dynamic metrics of embedded software systems, perform various quality modeling, and validate such models against actual quality data.

ComPARE aims to provide an environment for quality prediction of software components and assess their reliability in the overall system developed using component-based embedded software development. The overall architecture of ComPARE is shown in Fig. 12. First of all, various metrics are computed for the candidate components, then the users can select and weigh the metrics deemed important to quality assessment. After the models have been constructed and executed (e.g., "case base" is used in BBN model), the users can validate the selected models with failure data in real life. If users are not satisfied with the prediction result, they can go back to the previous step, re-define the criteria and construct a revised model. Finally, the overall quality prediction can be displayed based on the architecture of the candidate system. Results from individual components can also be displayed for sensitivity analysis and system redesign.

The objective of ComPARE can be summarized as follows:

1. *To predict the overall quality by using process metrics, static code metrics as well as dynamic metrics.* In addition to complexity metrics, we use process metrics, cohesion metrics, inheritance metrics and dynamic metrics (such as code coverage and call graph metrics) as the input to the quality prediction models. Thus the prediction is more accurate as it is based on data from every aspect of the candidate software components tailored for embedded systems.

2. *To integrate several quality prediction models into one environment and compare the prediction result of different models.* ComPARE integrates several existing quality models into one environment. In addition to selecting or defining these different models, users can also compare the prediction results of the models on the candidate component and see how good the predictions are if the failure data of the particular component is available.
3. *To define the quality prediction models interactively.* In ComPARE, there are several quality prediction models that users can select to perform their own predictions. Moreover, the users can also define their own models and validate their models by the evaluation procedure.
4. *To display the quality of components by different categories.* Once the metrics are computed and the models are selected, the overall quality of each component can be displayed according to the category it belongs to. Program modules with problems can also be identified.
5. *To validate reliability models defined by user against real failure data* (*e.g., change report*). Using the validation criteria, the result of the selected quality prediction model can be compared with failure data in real life. The users can redefine their models according to the comparison.
6. *To show the source code with potential problems at line-level granularity. ComPARE* can identify the source code with high risk (i.e., the code that is not covered by test cases in the embedded environment) at line-level granularity. This can help the users to locate high risk program modules or portions promptly and conveniently.
7. *To adopt commercial tools in accessing software data related to quality attributes.* We adopt Metamata [32] and Jprobe [33] suites to measure the different metrics for the candidate components. These two tools, including metrics, audits, debugging, as well as code coverage, memory and deadlock detection, are commercially available in the component-based program testing market.

5.2. *Metrics used in ComPARE*

Three different categories of metrics, namely process, static, and dynamic metrics, are computed and collected in CompARE to give an overall quality prediction. We have chosen the most useful metrics, which are widely adopted by previous software quality prediction tools from the software engineering research community. The process metrics selected are listed in Table 2 [34].

As we perceive Object-Oriented (OO) techniques to be essential in the component-based embedded software development approach, we select static code metrics according to the most important features in OO programs: complexity, coupling, inheritance and cohesion. They are listed in Table 3 [32, 35–37]. The dynamic metrics encapsulate measurement of the features of components when they are executed. Table 4 shows the detailed description of the dynamic metrics.

Table 2. Process metrics.

Metric	Description
Time	Time spent from the design to the delivery (months)
Effort	The total human resources used (man*month)
Change Report	Number of faults found in the development

Table 3. Static code metrics.

Abbreviation	Description
Lines of Code (LOC)	Number of lines in the components including the statements, the blank lines of code, the lines of commentary, and the lines consisting only of syntax such as block delimiters.
Cyclomatic Complexity (CC)	A measure of the control flow complexity of a method or constructor. It counts the number of branches in the body of the method, defined by the number of WHILE statements, IF statements, FOR statements, and CASE statements.
Number of Attributes (NA)	Number of fields declared in the class or interface.
Number of Classes (NOC)	Number of classes or interfaces that are declared. This is usually 1, but nested class declarations will increase this number.
Depth of Inheritance Tree (DIT)	Length of inheritance path between the current class and the base class.
Depth of Interface Extension Tree (DIET)	The path between the current interface and the base interface.
Data Abstraction Coupling (DAC)	Number of reference types that are used in the field declarations of the class or interface.
Fan Out (FANOUT)	Number of reference types that are used in field declarations, formal parameters, return types, throws declarations, and local variables.
Coupling between Objects (CO)	Number of reference types that are used in field declarations, formal parameters, return types, throws declarations, local variables and also types from which field and method selections are made.
Method Calls Input/Output (MCI/MCO)	Number of calls to/from a method. It helps to analyze the coupling between methods.
Lack of Cohesion of Methods (LCOM)	For each pair of methods in the class, the set of fields each of them accesses is determined. If they have disjoint sets of field accesses then increase the count P by one. If they share at least one field access then increase Q by one. After considering each pair of methods, $$LCOM = (P > Q)?(P - Q) : 0$$

Table 4. Dynamic metrics.

Metric	Description
Test Case Coverage	The coverage of the source code when executing the given test cases. It may help to design effective test cases.
Call Graph metrics	The relationships between the methods, including method time (the amount of time the method spent in execution), method object count (the number of objects created during the method execution) and number of calls (how many times each method is called in you application).
Heap metrics	Number of live instances of a particular class/package, and the memory used by each live instance.

This set of process, static, and dynamic metrics can be collected from some commercial tools, e.g., Metamata Suite [32] and Jprobe Testing Suite [33]. We will measure and apply these metrics in ComPARE.

5.3. *Models definition*

In order to predict the quality of different software components, several techniques have been developed to classify software components according to their reliability [38]. These techniques include discriminant analysis [39], classification trees [40], pattern recognition [41], Bayesian network [42], case-based reasoning (CBR) [43] and regression tree model [34]. In ComPARE, we integrate five types of models to evaluate the quality of the software components for an overall component-based embedded system evaluation. Users can customize these models and compare the prediction results from different tailor-made models.

5.3.1. *Summation model*

This model gives a prediction by simply adding all the metrics selected and weighted by a user. The user can validate the result by real failure data, and then benchmark the result. Later when new components are included, the user can predict their quality according to their differences from the benchmarks. The concept of the summation model can be formulated in the following:

$$Q = \sum_{i=1}^{n} \alpha_i m_i \tag{1}$$

where m_i is the value of one particular metric, α_i is its corresponding weighting factor, n is the number of metrics, and Q is the overall quality mark.

5.3.2. *Product model*

Similar to the summation model, the product model multiplies all the metrics selected and weighted by the user and the resulting value indicates the level of quality

of a given component. Similarly, the user can validate the result by real failure data, and then determine the benchmark for a later usage. The concept of the product model is shown as the following:

$$Q = \prod_{i=1}^{n} m_i \tag{2}$$

where m_i is the value of one particular metric, n is the number of metrics, and Q is the overall quality mark. Note that m_i's are normalized as a value which is close to 1, so that none of them will dominate the result.

5.3.3. *Classification tree model*

Classification tree model [40] is used to classify the candidate components into different quality categories by constructing a tree structure. All the candidate components are leaves in the tree. Each node of the tree represents a metric (or a composed metric calculated by other metrics) of a certain value. All the children of the left sub tree of the node represent those components whose value of the same metric is smaller than the value of the node, while all the children of the right sub-tree of the node are those components whose value of the same metric is equal to or larger than the value of the node.

In ComPARE, a user can define the metrics and their values at each node from the root to the leaves. Once the tree is constructed, a candidate component can be directly classified by following the threshold of each node in the tree until it reaches a leaf node. Again, the user can validate and evaluate the final tree model after its definition. Below is an example of the outcome of a tree model. At each node of the tree there are metrics and values, and the leaves represent the components with certain number of predicted faults in the classification result.

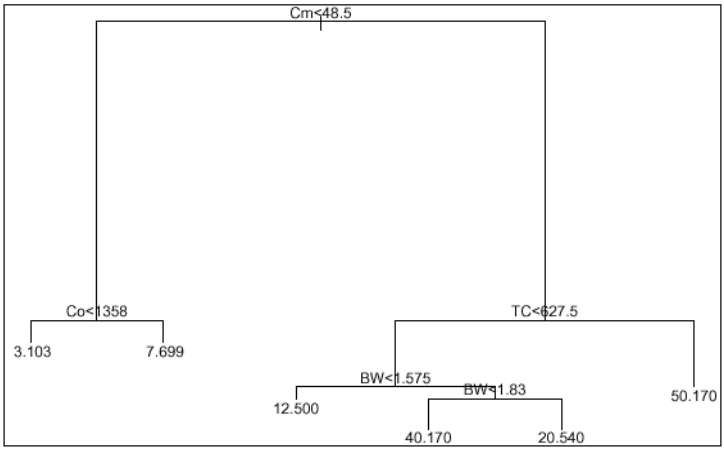

Fig. 13. An example of classification tree model.

5.3.4. *Case-based reasoning model*

Case-based reasoning (CBR) has been proposed for predicting the quality of software components [43]. A CBR classifier uses previous "similar" cases as the basis for the prediction. Previous cases are stored in a case base. Similarity is defined in terms of a set of metrics. The major conjecture behind this model is that the candidate component that has a similar structure to the components in the case base will inherit a similar quality level.

A CBR classifier can be instantiated in different ways by varying its parameters. But according to the previous research, there is no significant difference in prediction validity when using any combination of parameters in CBR. So we adopt the simplest CBR classifier modeling with Euclidean distance, z-score standardization [43], but no weighting scheme. Finally, we select the single, nearest neighbor for the prediction purpose.

5.3.5. *Bayesian network model*

Bayesian networks (also known as Bayesian Belief Networks, BBN) is a graphical network that represents probabilistic relationships among variables [42]. BBNs enable reasoning under uncertainty. Besides, the framework of Bayesian networks offers a compact, intuitive, and efficient graphical representation of dependence relations between entities of a problem domain. The graphical structure reflects properties of the problem domain directly, which provides a tangible visual representation as well as a sound mathematical basis in Bayesian probability [44]. The foundation of Bayesian networks is the following theorem known as Bayes' Theorem:

$$P(H|E,c) = \frac{P(H|c)P(E|H,c)}{P(E|c)} \tag{3}$$

where H, E, c are independent events, and P is the probability of such event under certain circumstances.

With BBNs, it is possible to integrate expert beliefs about the dependencies between different variables and to propagate consistently the impact of evidence on the probabilities of uncertain outcomes, such as "unknown component quality". Details of the BBN model for quality prediction can be found in [42]. Users can also define their own BBN models in ComPARE and compare the results with other models.

5.4. *Operations in ComPARE*

As a generic quality assessment environment for component-based embedded software system, ComPARE implements eight major functional areas: File Operations, Selecting Metrics, Selecting Criteria, Model Selection and Definition, Model Validation, Display Result, Windows Switch, and Help System. The details of some key functions are described in the following sections.

5.4.1. *Selecting metrics*

Users can select the metrics they want to collect for the component-based embedded systems. Three categories of metrics are available: process metrics, static metrics and dynamic metrics. The details of these metrics are shown in Sec. 5.2.

5.4.2. *Selecting and weighing criteria*

After computing the different metrics, users need to select and weigh the criteria on these metrics before using them in the reliability modeling. Each metric can be selected or omitted, and if selected, be marked with the weight between 0 and 100%. Such information will provide input parameters later in the quality prediction models.

5.4.3. *Models selection and definition*

Models operations allow users to select or define the model they would like to perform in the evaluation. The users should give the probability of each item related to the overall quality of the candidate component.

5.4.4. *Model validation*

Model validation allows comparisons between different models and with respect to actual software failure data. It facilitates users to compare different results based on a chosen subset of the software failure data under certain validation criteria. The comparisons between different models in their predictive capability are summarized in a summary table. Model Validation operations are activated only when the software failure data are available.

5.5. *Prototype*

Under the framework that we have described, we prototyped a specific version of ComPARE which targets software components developed by the Java language for embedded systems. Java is one of the most popular languages used in off-the-shelf components development today, and it is a common language binding in the three standard architecture of component-based embedded software development: CORBA, DCOM and Java/RMI.

Figures 14 and 15 show screen dumps for the described ComPARE prototype tool. The computation of various metrics for software components and application of quality prediction models can be seen as a straightforward process. Users also have flexible choices in selecting and defining different models. The combination of simple operations and a variety of quality models makes it easy for the users to identify an appropriate prediction model for a given component-based embedded system with its encapsulated components.

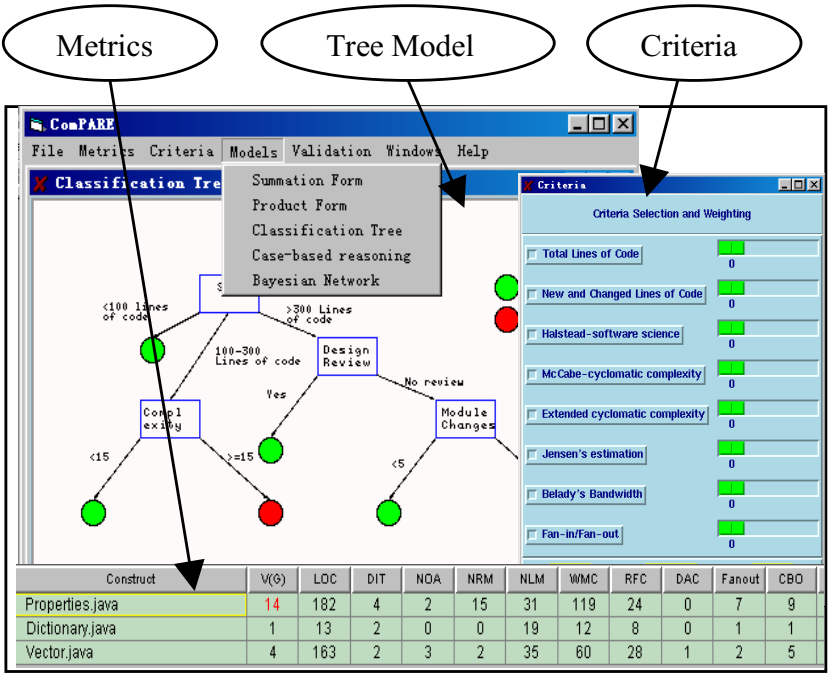

Fig. 14. GUI of ComPARE for metrics, criteria and tree model.

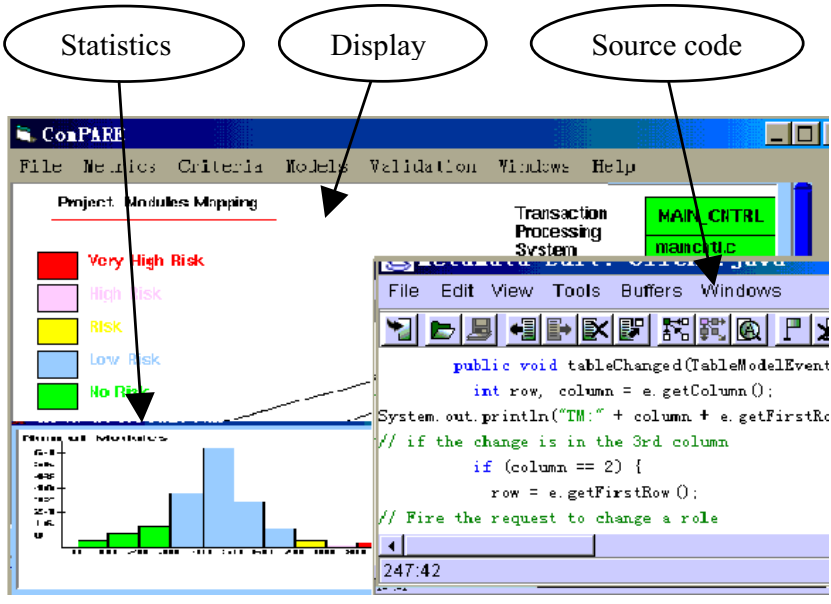

Fig. 15. GUI of ComPARE for prediction display, risky source code and result statistics.

6. Conclusions

In this paper, we introduce a component-based embedded software development framework and the features it inherits. We propose a QA model for component-based embedded software development, which covers both the component QA and the system QA as well as their interactions. As far as we know, this is the first effort to formulate a QA model for developing embedded systems based on component technologies. We further propose a generic quality assessment environment for component-based embedded systems: ComPARE. ComPARE is new in that it collects metrics of more aspects for embedded software systems, including process metrics, static code metrics, and dynamic metrics for software components, integrates reliability assessment models from different techniques used in current quality prediction area, and validates these models against the failure data collected in real life. ComPARE can be used to assess real-life off-the-shelf components and to evaluate and validate the models selected for their evaluation. The overall component-based embedded system can then be composed and analyzed seamlessly. ComPARE can be an effective environment to promote component-based embedded system construction with higher reliability evaluation and proper quality assurance.

Acknowledgements

The work described in this paper was fully supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region (Project No. CUHK4222/01E), and the Open Component Foundation under the Innovation and Technology Fund (Reference No. AF/94/99).

References

1. http://www.utdallas.edu/research/esc/.
2. E. A. Lee, "What's ahead for embedded software?" *Computer*, Sept. 2000, pp. 18–26.
3. A. Rhodes, "Component-based development for embedded systems", *Proc. Embedded of Systems Conference*, No. 313.
4. G. Pour, "Software component technologies: JavaBeans and ActiveX", *Proc. Technology of Object-Oriented Languages and Systems*, 1999, pp. 398–398.
5. G. Pour, "Component-based embedded software development approach: New opportunities and challenges", *Proc. Technology of Object-Oriented Languages*, 1998, TOOLS 26, pp. 375–383.
6. A. W. Brown and K. C. Wallnau, "The current state of CBSE", *IEEE Software* **15** (1998) 37–46.
7. M. L. Griss, "Software reuse architecture, process, and organization for business success", *Proc. Eighth Israeli Conference on Computer Systems and Software Engineering*, 1997, pp. 86–98.
8. IBM: http://www4.ibm.com/software/ad/sanfrancisco.
9. U. Hansmann, L. Merk, M. S. Nicklous and T. Stober, *Pervasive Computing Handbook*, Springer, New York, 2001.
10. J. H. Jahnke and M. Entremont, "Component-based engineering of distributed embedded systems", *Proc. Embedded Systems Conference*, San Francisco, Apr. 2001, No. 371.

11. M. Barr, "Developing embedded software in Java", *Proc. Embedded Systems Conference*, San Francisco, Apr. 2001, No. 371.
12. J. Currey, "Using CORBA to accelerate distributed application development and improve network management", *Proc. Embedded Systems Conference*, San Francisco, Apr. 2001, No. 444.
13. S. Mailet, "Using COM for embedded systems", *Proc. Embedded Systems Conference*, San Francisco, Apr. 2001, No. 504.
14. W. Kozaczynski and G. Booch, "Component-based embedded software engineering", *IEEE Software* **155** (1998) 34–36.
15. OMG: http://www.omg.org/corba/whatiscorba.html.
16. S. S. Yau and B. Xia, "Object-oriented distributed component software development based on CORBA", *Proc. 22nd Ann. Int. Conf. of COMPSAC'98*, 1998, pp. 246–251.
17. Microsoft: http://www.microsoft.com/isapi.
18. Y. M. Wang, O. P. Damani and W. J. Lee, "Reliability and availability issues in Distributed Component Object Model (DCOM)", *Fourth Int. Workshop on Community Networking Proceedings*, 1997, pp. 59–63.
19. SUN http://developer.java.sun.com/developer.
20. G. Pour, M. Griss and J. Favaro, "Making the transition to component-based enterprise software development: Overcoming the obstacles — Patterns for success", *Proc. Technology of Object-Oriented Languages and Systems*, 1999, pp. 419–419.
21. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, New York, 1998.
22. S. M. Yacoub, B. Cukic and H. H. Ammar, "A component-based approach to reliability analysis of distributed systems", *Proc. 18th IEEE Symposium on Reliable Distributed Systems*, 1999, pp. 158–167.
23. S. M. Yacoub, B. Cukic and H. H. Ammar, "A scenario-based reliability analysis of component-based embedded software", *Proc. 10th Int. Symposium on Software Reliability Engineering*, 1999, pp. 22–31.
24. J. Q. Ning, K. Miriyala and W. Kozaczynski, "An architecture-driven, business-specific, and component-based approach to software engineering", *Proc. Third Int. Conf. on Software Reuse: Advances in Software Reusability*, 1994, pp. 84–93.
25. C. Rajaraman and M. R. Lyu, "Reliability and maintainability related software coupling metrics in C++ programs", *Proc. 3rd IEEE Int. Symposium on Software Reliability Engineering* (*ISSRE'92*), 1992, pp. 303–311.
26. C. Rajaraman and M. R. Lyu, "Some coupling measures for C++ programs", *Proc. TOOLS USA 92 Conference*, August 1992, pp. 225–234.
27. I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.
28. M. R. Lyu, "Software reliability theory", *Encyclopedia of Software Engineering*, eds. J. J. Marciniak, Wiley, New York, 2001, pp. 1161–1630.
29. Hong Kong Productivity Council, http://www.hkpc.org/itd/servic11.htm, April, 2000.
30. F. Balarin, *et al.*, *Hardware-Software Co-Design of Embedded Systems*, Kluwer Academic Publishers, Norwell, 1997.
31. M. R. Lyu, J. S. Yu, E. Keramidas and S. R. Dalal, "ARMOR: Analyzer for reducing module operational risk", *Proc. 25th Int. Symposium on Fault-Tolerant Computing* (*FTCS-25*), 1995, pp. 137–142.
32. http://www.metamata.com.
33. http://www.klgroup.com.
34. A. A. Keshlaf and K. Hashim, "A model and prototype tool to manage software risks",

Proc. First Asia-Pacific Conference on Quality Software, 2000, pp. 297–305.
35. M. R. Lyu (ed.), *Handbook of Software Reliability Engineering*, McGraw-Hill, New York, 1996.
36. J. Voas and J. Payne, "Dependability certification of software components", *The Journal of Systems and Software* **52** (2000) 165–172.
37. T. Systa, Y. Ping and H. Muller, "Analyzing Java software by combining metrics and program visualization", *Proc. Fourth European Software Maintenance and Reengineering*, 2000, pp. 199–208.
38. S. S. Gokhale and M. R. Lyu, "Regression tree modeling for the prediction of software quality", *Proc. Third ISSAT Int. Conf. on Reliability and Quality in Design*, Anaheim, California, March 1997.
39. J. Munson and T. Khoshgoftaar, "The detection of fault-prone programs", *IEEE Transactions on Software Engineering* **18**(5) (May 1992).
40. A. A. Porter and R. W. Selby, "Empirically guided software development using metric-based classification trees", *IEEE Software*, Mar. 1990, pp. 46–53.
41. L. C. Briand, V. R. Basili and C. Hetmanski, "Developing interpretable models for optimized set reduction for identifying high-risk software components", *IEEE Transactions on Software Engineering* **19**(11) (1993) 1028–1034.
42. N. E. Fenton and M. Neil, "A critique of software defect prediction models", *IEEE Transactions on Software Engineering* **25**(5) (1999) 675–689.
43. K. E. Emam, S. Benlarbi, N. Goel and S. N. Rai, "Comparing case-based reasoning classifiers for predicting high risk software components", *The Journal of Systems and Software* **55** (2001) 301–320.
44. http://www.hugin.com.