Component-Based Software Engineering: Technologies, Quality Assurance Schemes, and Risk Analysis Tools

Cai Xia

Supervisor: Prof. Michael R. Lyu

Markers: Prof. Kam-Fai Wong

Prof. Ada Fu

Abstract

Component-based software development approach is based on the idea to develop software systems by selecting appropriate off-the-shelf components and then to assemble them with a well-defined software architecture. Because the new software development paradigm is much different from the traditional approach, quality assurance (QA) for component-based software development is a new topic in the software engineering community. In this paper, we survey current component-based software technologies, describe their advantages and disadvantages, and discuss the features they inherit. We also address QA issues for component-based software. As a major contribution, we propose a QA model for component-based software development, which covers component requirement analysis, component development, component certification, component customization, and system architecture design, integration, testing, and maintenance.

We also look at the advantages of the Analyzer for Reducing Module Operational Risk (RMOR) tool, and collect some widely adopted Java metrics and tool suites. As our future work we will upgrade ARMOR to windows platformed, off-shelf commercial components based, Java source code oriented risk analysis and evaluation tool.

1. Introduction

Modern software systems become more and more large-scale, complex and uneasily controlled, resulting in high development cost, low productivity, unmanageable software quality and high risk to move to new technology [15]. Consequently, there is a growing demand of searching for a new, efficient, and cost-effective software development paradigm.

One of the most promising solutions today is the component-based software development approach. 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 [12]. 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 Figure 1, where COTS components can be checked out from a component repository, and assembled into a target software system.

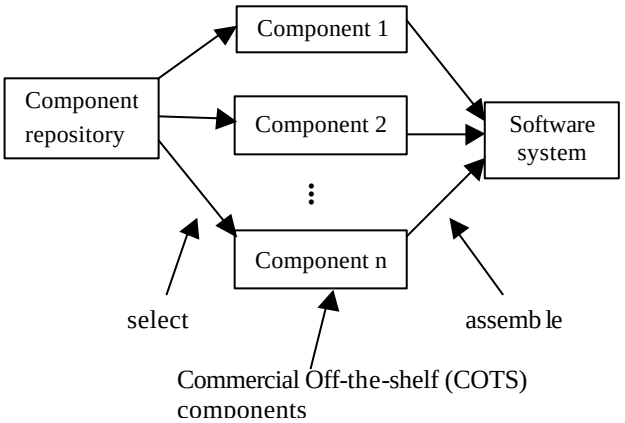

Figure 1. Component-based software development

Component-based software development (CBSD) can significantly reduce development cost and time-to-market, and improve maintainability, reliability and overall quality of software systems [13] [14]. This approach has raised a tremendous amount of interests both in the research community and in the software industry. The life cycle and software engineering model of CBSD is much different from that of the traditional ones. This is what the Component-Based Software Engineering (CBSE) is focused.

Up to now, software component technologies are an emerging technology, which is far

from being matured. There is no existing standards or guidelines in this new area, and we do not even have a unified definition of the key item " component" . In general, however, 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 interfaces [1].

To ensure that a component-based software system can run properly and effectively, the system architecture is the most important factor. According to both research community [2] and industry practice [5], the system architecture of component-based software systems should be a layered and modular architecture. This architecture can be seen in Figure 2. The top application layer is the application systems supporting a

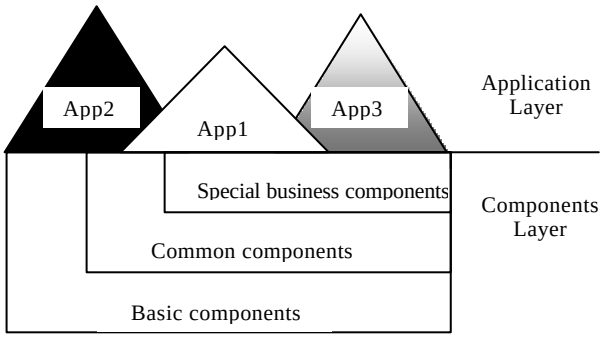

Figure 2. System architecture of component-based software systems

business. The second layer consists of components engaged in only a specific business or application domain, including components usable in more than a single application. The third layer is cross-business middleware components consisting of common software and interfaces to other established entities. Finally, the lowest layer of system software components includes basic components that interface with the underlying operating systems and hardware.

Current component technologies have been used to implement different software systems, such as object-oriented distributed component software [23] and Web-based enterprise application [13]. There are also some commercial players involved in the software component revolution, such as BEA, Microsoft, IBM and Sun [7]. An outstanding example is the IBM SanFrancisco project. It provides a reusable distributed object infrastructure and an abundant set of application components to application developers [5].

2. Current Component Technologies

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 [1]. Among the component infrastructure technologies that have been developed, three have become somewhat standardized: OMG's CORBA, Microsoft's Component Object Model (COM) and Distributed COM (DCOM), and Sun's JavaBeans and Enterprise JavaBeans [7].

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 vendor and object technology user companies [11]. Simply stated, CORBA manages details of component interoperability, and allows applications to communicate with one another despite of 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 location is completely transparent. The ORB is responsible for intercepting a call and finding an object that can implement the request, pass its parameters, invoke its method, and return 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 applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

CORBA is widely used in Object-Oriented distributed systems [23] including component-based software systems because it offers a consistent distributed programming and run-time environment over common programming languages,

operating systems, and distributed networks.

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

Introduced in 1993, Component Object Model (COM) is a general architecture for component software [9]. 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 of 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 [20].

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 machines, DCOM simply replaces the local interprocess communication with a network protocol. Neither the client nor the component is aware the changes of the physical connections.

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 applications of multiple platforms, as well as reusable, client-side and server-side components [19].

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 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 Current Component Technologies

Comparison among current component technologies can be found in [Brow98], [Pour99a] and [Szyp98]. Here we simply summarize these different features in Table 1.

	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

Table 1. Comparison of current component technologies

3. Quality Assurance for Component-Based Software Systems

3.1 The Life Cycle of Component-Based Software Systems

Component-based 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 software systems is different from that of the traditional software systems. The life cycle of component-based software systems can be summarized as follows [12]: 1) Requirements analysis; 2) Software architecture selection, construction, analysis, and evaluation; 3) Component identification and customization; 4) System integration; 4) System testing; 5) Software maintenance.

The architecture of software defines a 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 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 software systems. Integration is to make key decisions on how to provide communication and coordination among various components of a target software system.

Quality assurance for component-based software systems should address the life cycle and its key activities to analyze the components and achieve high quality component-based software systems. QA technologies for component-based 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 [21] [22] and component-based approach to Software Engineering [10] have been studied, there is still no clear and well-defined standards or guidelines for component-based software systems. The identification of the QA characteristics, along with the models, tools and metrics, are all under urgent needs.

3.2 Quality Characteristics of Components

As much work is yet to be done for component-based software development, QA technologies for component-based software development has to address the two inseparable parts: 1) How to certify quality of a Component ? 2) How to certify quality of software systems based on components? To answer the 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 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 [16] [17]. Such metrics often used to classify components include [6]:

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 have 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 [8].

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

Because component-based 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 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, Hong Kong productivity Council has developed the HKSQA model to localize the general SQA models [4]. In this section, we propose a framework of quality assurance model for the

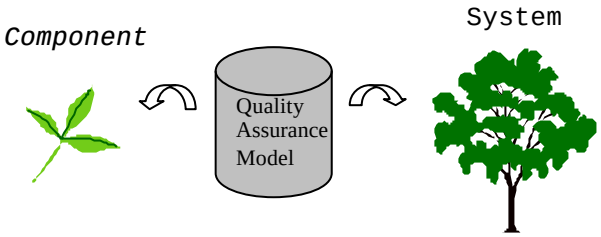

Figure 3. Quality assurance model for both components and systems

component-based software development paradigm. The main practices relating to components and 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 platform and the interfaces related to the component.

The component requirement process overview diagram is as shown in Figure 4. Initiated by the request of users or customers for new development or changes on old system, 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 components satisfying the requirements with correct and expected results, well-defined behaviors, and flexible interfaces.

The component development process overview diagram is as shown in Figure 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.

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 to the requirement for both functionality and reliability; and 3) *component testing*: confirm 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 environment or a simulated environment. The component certification process overview diagram is as shown in Figure 6. The input to this phase should be component development document, and the output should be testing documentation for system maintenance.

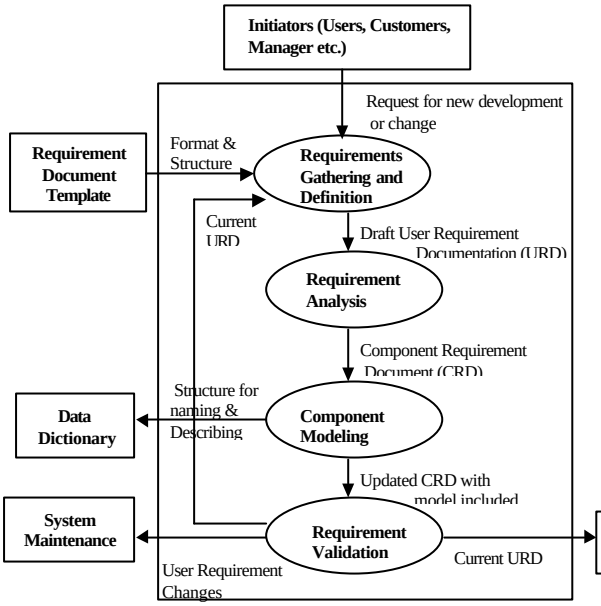

Figure 4. Component requirement analysis process overview

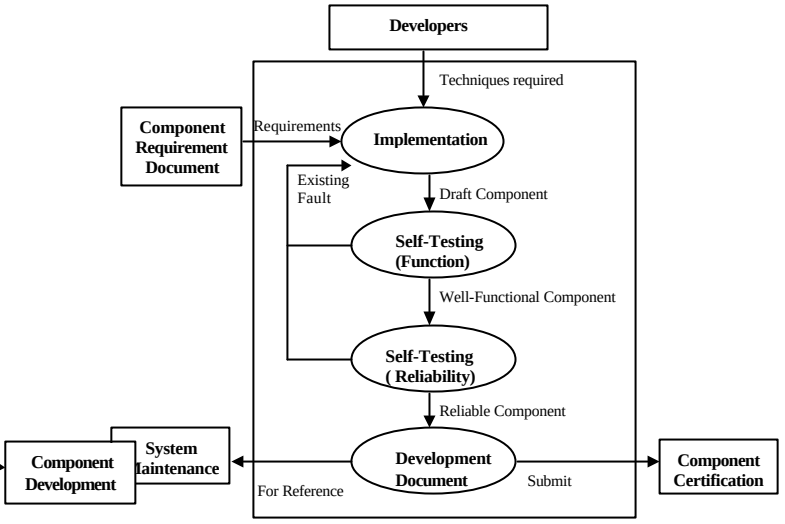

Figure 5. Component development process overview

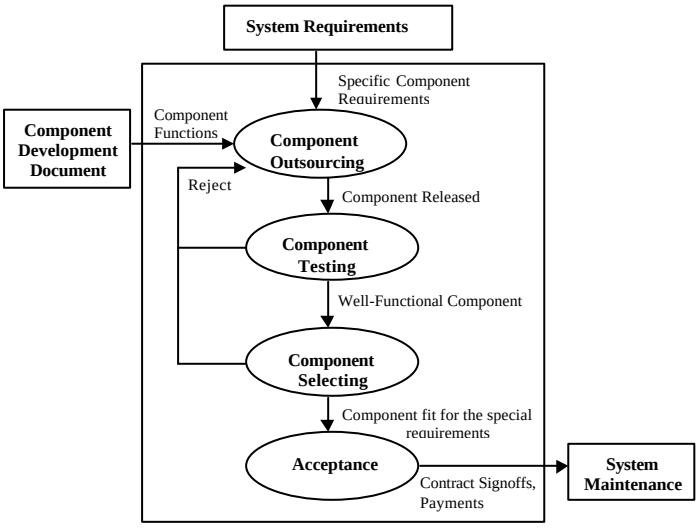

Figure 6. component certification process overview

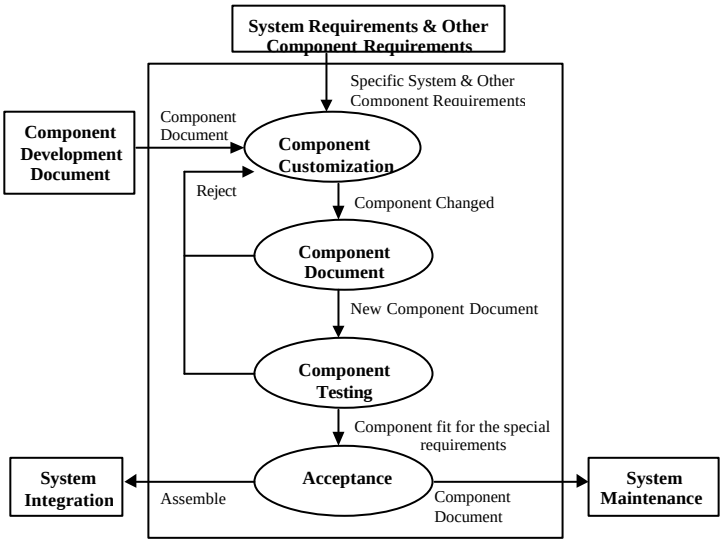

Figure 7. Component customization process overview

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 special platform; 3) upgrading the specific component to get a better performance or a higher quality.

The objectives of component customization are to make necessary changes for a developed component so that it can be used in a specific 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 as shown in Figure 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 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 architecture from other architectures. The process overview diagram is as shown in Figure 8. This phase consists of system requirement gathering, analysis, system architecture design, and system specification. The output of this phase should be the system specification document for integration, and system requirement for the system testing phase and system maintenance phase.

4.6 System Integration

System integration is the process of assembling components selected into a whole system under the designed system architecture.

The objective of system integration is the final system composed by the selected components. The process overview diagram is as shown in Figure 9. The input is the 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 system ready for the system testing phase, and the document for the system maintenance phase.

4.7 System Testing

System testing is the process of evaluating a system to: 1) confirm that the system satisfies the specified requirements; 2) identify and correct defects in the system implementation.

The objective of system testing is the final system integrated by components selected in accordance to the system requirements. System testing should contain function testing and reliability testing. The process overview diagram is as shown in Figure 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.

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 system to a changed environment.

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 as shown in Figure 11. According to the outputs from all previous phases as well as request and problem reports from users,

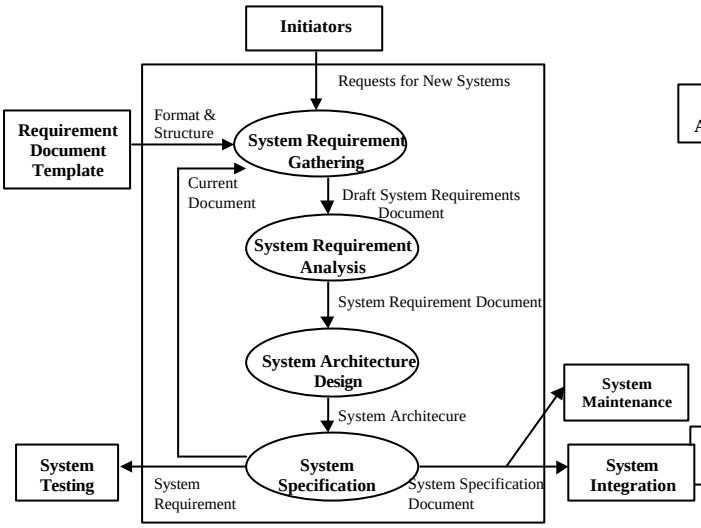

Figure 8. System architecture design process overview

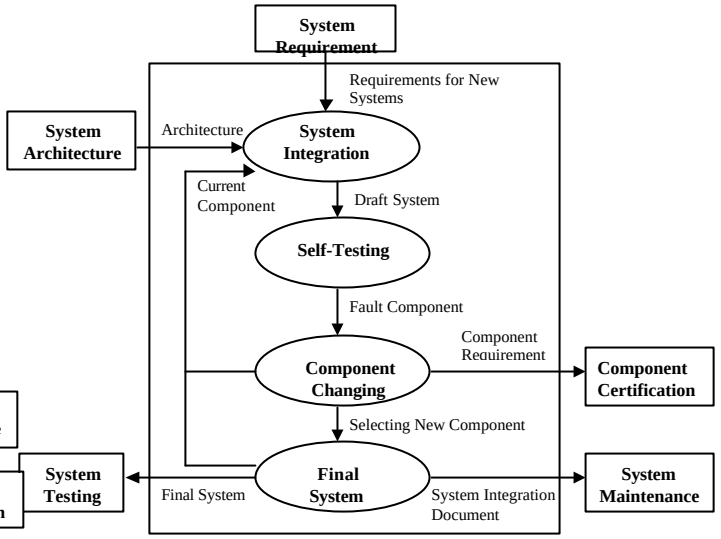

Figure 9. System integration process overview

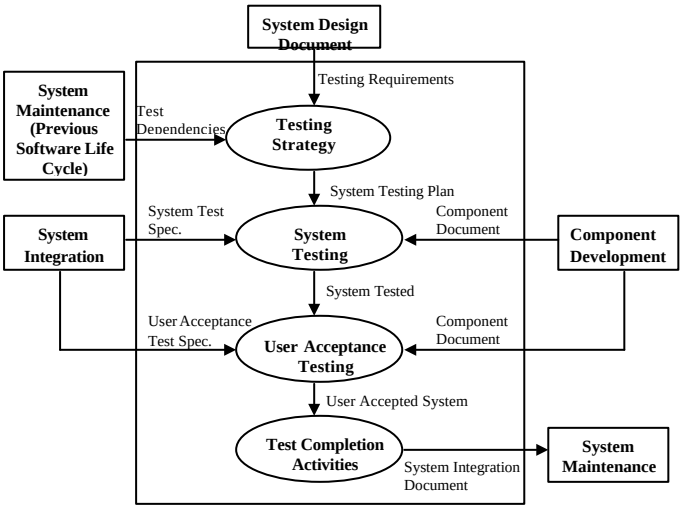

Figure 10. System testing process overview

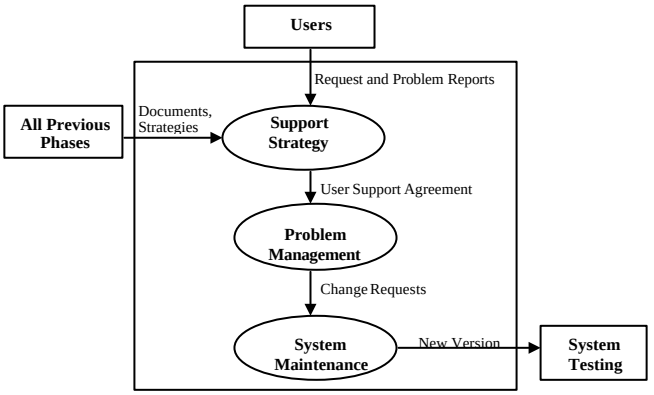

Figure 11. System maintenance process overview

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.

5. ARMOR: A Software Risk Analysis Tool

As we have mentioned before, there are a lot of metrics and tools to measure and test the quality of a software system. But little of them can integrate the various metrics together and compare the different results of these metrics, so that they can predict the quality as well as the risk of the software.

5.1 Purpose of ARMOR

ARMOR(Analyzer of Reducing Module Operational Risk) is such a tool that is developed by Bell Lab in 1995 [24]. ARMOR can automatically identify the operational risks of software program modules. It takes data directly from project database, failure database, and program development database, establishes risk models according to several risk analysis schemes, determines the risks of software programs, and display various statistical quantities for project management and engineering decisions. The tool can perform the following tasks during project development, testing, and operation: 1) to establish promising risk models for the project under evaluation; 2) to measure the risks of software programs within the project; 3) to identify the source of risks and indicates how to improve software programs to reduce their risk levels; and 4) to determine the validity of risk models from field data.

ARMOR is designed for automating the procedure for the collection of software metrics, the selection of risk models, and the validation of established models. It provided the missing link of both performing sophisticated risk modeling and validate risk models against software failure data by various statistical techniques.

5.2 Objective and Overview of ARMOR

The Objectives of ARMOR are summarized as follows:

1) *To access and compute software data deemed pertinent to software characteristics.* ARMOR access three major databases: project source code directory (for product-related metrics), program development history (for process-related metrics), and the Modification Request (MR) database (a failure report system at Bellcore).

2) *To compute product metrics automatically whenever possible.* By measuring the project source files, ARMOR directly computes software code metrics related to the software product.

3) *To evaluate software metrics systematically.* A preliminary analysis of the effectiveness of the computed and collected metrics is obtained by studying the correlation of these metrics to the software failure data in the MR database. This study provide information about the candidate metrics for the establishment of risk models.

4) *To perform risk modeling in a user-friendly and user-flexible fashion.* Metrics are selected with appropriate weighting to establish risk measures (i.e.,risk scores) of each software module. Several modeling schemes are provided in ARMOR. Risk models could be defined, removed, and executed easily at the user' s discretion.

5) *To display risks of software modules.* Once computed, risk scores computed the risk models could be used to highlight each software module by different colors. Risk distribution can be demonstrated in various forms.

6) *To validate risk models against actual failure data and compare model performance.* Using several validation criteria, the risk models are compared with actual failure data to determine their predictive accuracy. Model validation results are provided in a summary table. Validated models could be saved for a later reusage.

7) *To identify risky modules and to indicate ways for reducing software risks.* Once a valid model is established, the risk score computed for each module can be compared with the risk score contributed by the individual metric components. This process is iterated to identify the dominanting metrics which need to be addressed for the reduction of module operations risk.

5.2 Architecture of ARMOR

Table 3. shows the high-level architecture for ARMOR.

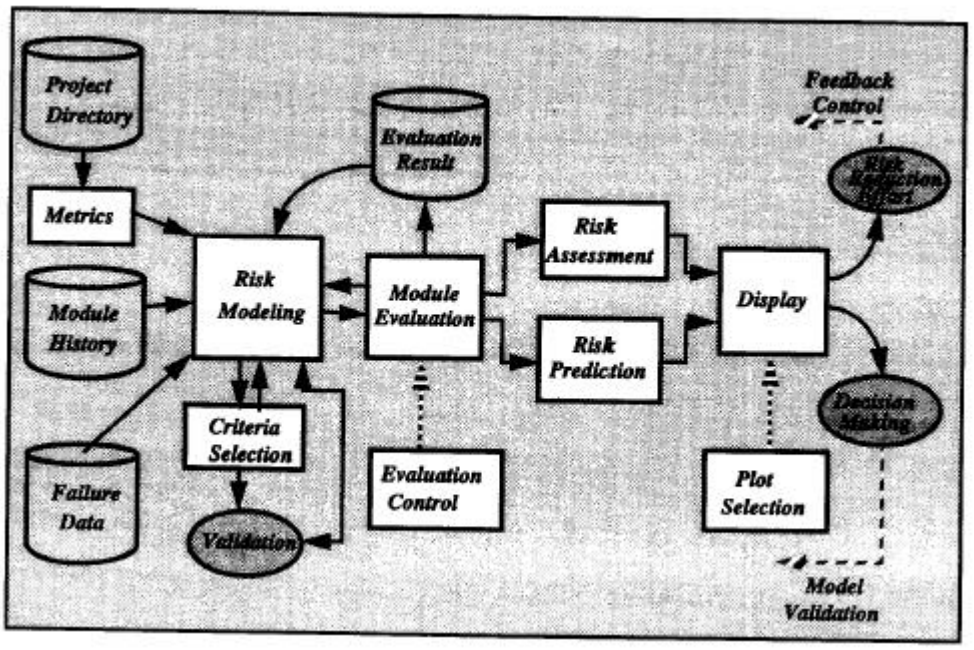

Table 3. High-level architecture for ARMOR

5.3 Context of ARMOR

ARMOR is composed of seven major functional areas:

1) File Operations (" File" menu)
2) Selecting Scope (" Scope" menu)
3) Computing and Selecting Metrics (" Metrics" menu)
4) Model Definition and Execution (" Models" menu)
5) Risk Evaluation (" Evaluation" menu)
6) Model Validation (" Validation" menu)
7) Help System (" Help" menu)

The on-screen appearances of the six main functions above are showed in the following figures. It can be seen that the application of risk modeling and analysis to software modules is a straightforward process. Users are also given a considerable amount of choices in constructing and applying risk models.

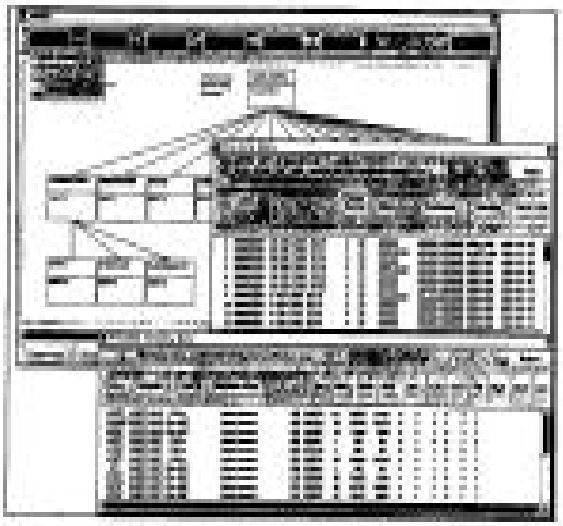

Figure 3: Open Project, Process, and Failure Database

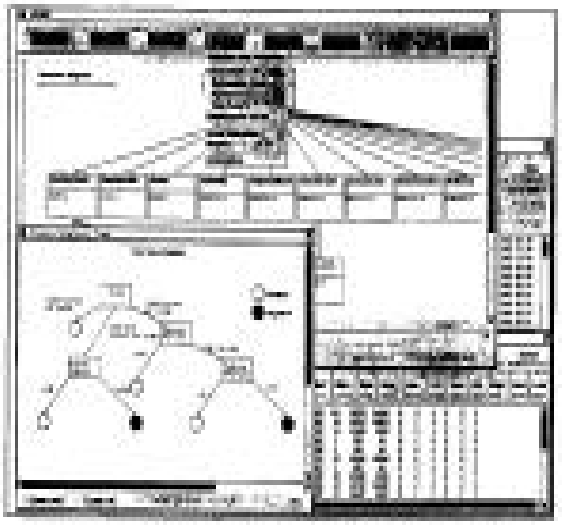

Figure 4: Construct Risk Models by a Classification Tree

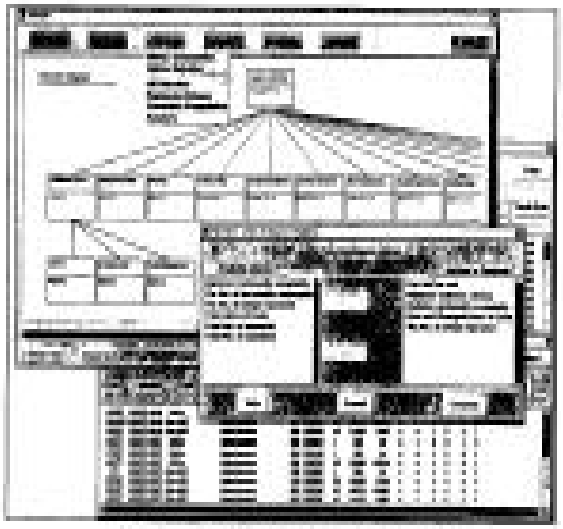

Figure 5: Select and Compute Metrics

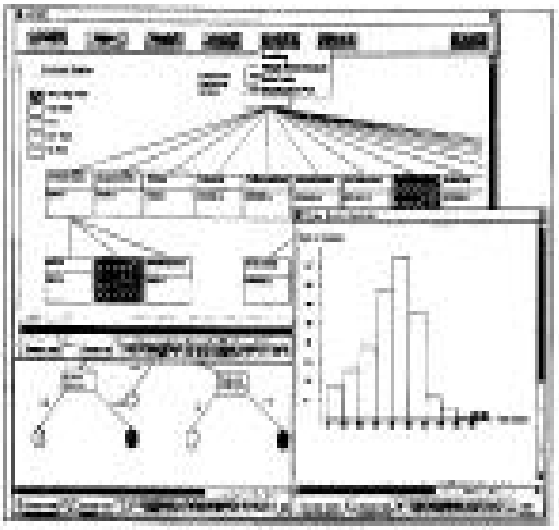

Figure 6: Highlight Risky Modules and Risk Distribution

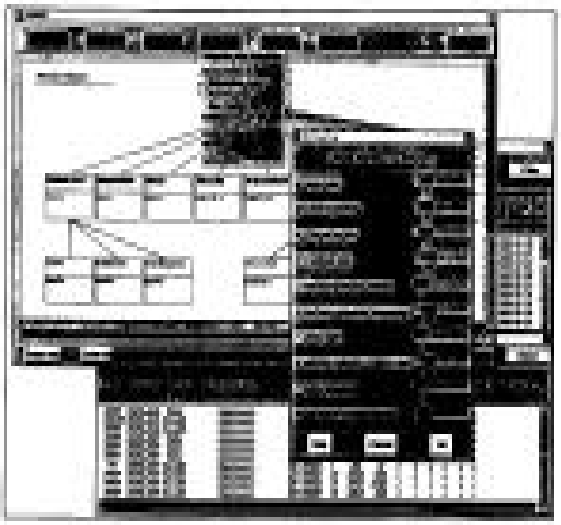

Figure 8: Select Metrics and Weighting Criteria for Models

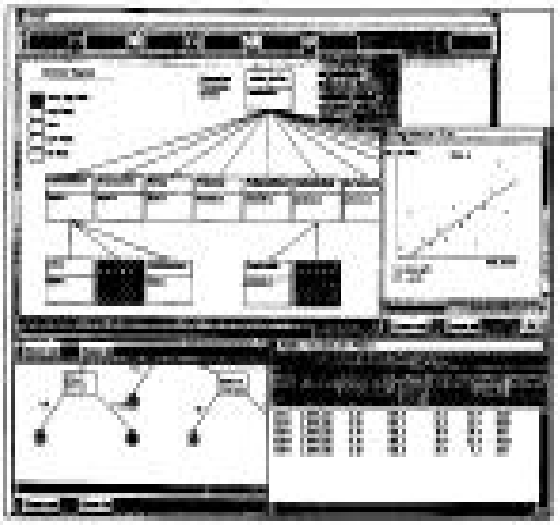

Figure 7: Display Regression Analysis and Model Validation

6. Risk Analysis and Evaluation Tool for CBSD

The complete functionality of ARMOR was not implemented, and it is only a prototype of the risk analysis tool. ARMOR was implemented in a UNIX X-windows environment, using Extended Tcl/Tk as its interface builder.

Based on the special features and process of component-based software development, and the idea of ARMOR, we propose to upgrade ARMOR to the windows-based application that aims at the evaluation and risk analysis of the component-based software system. Because most of the off-commercial components now use Java as their programming language, such features related to Java should be addressed in our new evaluation tool.

6.1 Java Features Addressed in the Metrics

According to [25], the main three metrics categories related to object-oriented programming such as Java are: 1) Inheritance metrics, such as the Depth of Inheritance Tree (DIT) and the Number of Children (NOC); 2) Communication metrics, such as Response For a Class (RFC), Coupling Between Objects (CBO), and Lack of Cohesion in Methods (LCOM); 3)Complexity metrics, such as the Cyclomatic Complexity (CC) and the Weighted Methods per Class (WMC). Inheritance metrics are used to examine the inheritance hierarch of object-oriented programs; the communication metrics estimate the internal and external communication of software components; and the complexity metrics measure the logical structure complexity of selected components.

In another recent research that extends software quality assessment techniques to Java systems, the metrics that the researchers choose can be divided into five main categories: [26]

1) *Java Classes:* This analysis aims to evaluate the complexity of the system classes. Class complexity can be characterized by its length, by the number and the type of methods it declares.

2) *Program Coupling:* This analysis provides indicators of program coupling. Program coupling is defined by dependencies between parts of the system.

3) *Java Methods:* This analysis evaluates the complexity of the Java methods,

including the length, number of parameter and number of variables of a method.

4) *Hierarchical Structure:* This analysis looks at the hierarchical structure of the program in order to evaluate the maintainability of the program.

5) *Clone Detection:* This is to identify whether code duplications exists in components. Code duplications can reduce software maintenance costs and improve quality.

6.2 Some Widely-used Metrics in Current Components Market

Based on the features of Java programming language and the widely used off-shelf commercial components, there are some metric suites to address this field. We have collected some well-adopted metrics and testing tools on component marketing today. They are Metamata Metrics and JProbe Metrics.

Metamata Metrics

Metamata Metrics calculates global complexity and quality metrics statically from Java source code, helps organize code in a more structured manner and facilitates the QA process [27]. It has the following features:

- ?? Most standard object oriented metrics such as object coupling and object cohesion
- ?? Traditional software metrics such as cyclomatic complexity and lines of code
- ?? Can be used on incomplete Java programs or programs with errors - and consequently, can be used from day one of the development cycle
- ?? Obtain metrics at any level of granularity (methods, classes...)
- ?? Performs statistical aggregations (mean, median...)
- ?? Works with both JDK 1.1 and JDK 1.2

This is the examples of Metamata Metrics:

Metric	Measures	Description
Cyclomatic Complexity	Complexity	The amount of decision logic in the code
Lines of Code	Understandability, maintainability	The length of the code; related metrics measure lines of comments, effective lines of code, etc.
Weighted Methods per Class	Complexity, understandability, reusability	The number of methods in a class
Response for a Class	Design, usability, testability	The number of methods that can be invoked from a class through messages
Coupling Between Objects	Design, reusability, maintainability	The number of other classes to which a class is coupled
Depth of Inheritance Tree	Reusability, testability	The depth of a class within the inheritance hierarchy
Number of Attributes	Complexity, maintainability	The amount of state a class maintains as represented by the number of fields declared in the class

Table 4. Examples of Metamata Metrics

JProbe Metrics

The JProbe from KL Group has different suites of metrics/tools for different purpose of use [28]. They are designed to help developers build robust, reliable, high-speed business applications in Java. Here is what the JProbe Developer Suite includes:

?? *JProbe Profiler and Memory Debugger* - eliminates performance bottlenecks and memory leaks in your Java code

?? *JProbe Threadalyzer* - detects deadlocks, stalls and race conditions

?? *JProbe Coverage* - locates and measures untested Java code.

JProbe Developer Suite paints an intuitive, graphical picture of everything from memory usage to calling relationships, helping you navigate to the root of the problem

quickly and easily.

Metamata metrics and Jprobe suites are both used in the QA Lab of Flashline, a industry leader in providing software component products, services and resources that facilitate the rapid development of software systems for business. We have collected them to try to use the result of such metrics, or integrate them into our risk analysis and evaluation tool that is based on the idea of ARMOR.

7. Conclusion and Future Work

In this paper, we survey current component-based software technologies and the features they inherit. We propose a QA model for component-based software development, which covers both the component QA and the system QA as well as their interactions.

We also look at the advantages of the ARMOR tool, and collect some widely adopted Java metrics and tool suites. As our future work we will upgrade ARMOR to windows platformed , off-shelf commercial components based, Java source code oriented risk analysis and evaluation tool.

As our future work, we will use the results of Metamata Metrics and JProbe Developer Suites in the new ARMOR tool, and may try to integrate them into the new tool. The new ARMOR tool targets to evaluate and analyze the quality and the risk of the components, as well as the component-based software systems.

References

[1] A.W.Brown, K.C. Wallnau, " The Current State of CBSE, " IEEE Software, Volume: 15 5, Sept.-Oct. 1998, pp. 37 – 46.

[2] M. L. Griss, " Software Reuse Architecture, Process, and Organization for Business Success, " Proceedings of the Eighth Israeli Conference on Computer Systems and Software Engineering, 1997, pp. 86-98.

[3] P.Herzum, O.Slims, " Business Component Factory - A Comprehensive Overview of Component-Based Development for the Enterprise," OMG Press, 2000.

[4] Hong Kong Productivity Council, http://www.hkpc.org/itd/servic11.htm, April, 2000.

[5] IBM: http://www4.ibm.com/software/ad/sanfrancisco, Mar, 2000.

[6] I.Jacobson, M. Christerson, P.Jonsson, G. Overgaard, " Object-Oriented Software Engineering: A Use Case Driven Approach," Addison-Wesley Publishing Company, 1992.

[7] W. Kozaczynski, G. Booch, " Component-Based Software Engineering," IEEE Software Volume: 155, Sept.-Oct. 1998,

pp. 34–36.

[8] M.R.Lyu (ed.), Handbook of Software Reliability Engineering, McGraw-Hill, New York, 1996.

[9] Microsoft:http://www.microsoft.com/isapi, Mar, 2000.

[10] J.Q. Ning, K. Miriyala, W. Kozaczynski,, " An Architecture-Driven, Business-Specific, and Component-Based Approach to Software Engineering," Proceedings Third International Conference on Software Reuse: Advances in Software Reusability, 1994, pp. 84 -93.

[11] OMG: http://www.omg.org/corba/whatiscorba.html, Mar, 2000.

[12] G. Pour, " Component-Based Software Development Approach: New Opportunities and Challenges," Proceedings Technology of Object-Oriented Languages, 1998. TOOLS 26., pp. 375-383.

[13] G. Pour, " Enterprise JavaBeans, JavaBeans & XML Expanding the Possibilities for Web-Based Enterprise Application Development," Proceedings Technology of Object-Oriented Languages and Systems, 1999, TOOLS 31, pp.282-291.

[14] G.Pour, M. Griss, J. Favaro, " Making the Transition to Component-Based Enterprise Software Development: Overcoming the Obstacles – Patterns for Success," Proceedings of Technology of Object-Oriented Languages and systems, 1999, pp.419 – 419.

[15] G.Pour, " Software Component Technologies: JavaBeans and ActiveX," Proceedings of Technology of Object-Oriented Languages and systems, 1999, pp. 398 – 398.

[16] C. Rajaraman, M.R. Lyu," Reliability and Maintainability Related Software Coupling Metrics in C++ Programs," Proceedings 3rd IEEE International Symposium on Software Reliability Engineering (ISSRE'92), 1992, pp. 303-311.

[17] C. Rajaraman, M.R. Lyu, " Some Coupling Measures for C++ Programs," Proceedings

TOOLS USA 92 Conference, August 1992, pp. 225-234.

[18] C.Szyperski, " Component Software: Beyond Object-Oriented Programming," Addison-Wesley, New York, 1998.

[19] SUN http://developer.java.sun.com/developer,Mar. 2000

[20] Y.M.Wang, O.P.Damani, W.J. Lee, " Reliability and Availability Issues in Distributed Component Ojbect Model (DCOM)," Fourth International Workshop on Community Networking Proceedings, 1997, pp. 59 –63.

[21] S.M. Yacoub, B. Cukic, H.H. Ammar, " A Component-Based Approach to Reliability Analysis of Distributed Systems," Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems, 1999, pp. 158 –167.

[22] S.M.Yacoub, B. Cukic, H.H.Ammar, " A Scenario-Based Reliability Analysis of Component-Based Software," Proceedings 10th International Symposium on Software Reliability Engineering, 1999, pp. 22 –31.

[23] S.S.Yau, B. Xia, " Object-Oriented Distributed Component Software Development based on CORBA," Proceedings of COMPSAC' 98. The Twenty-Second Annual International, 1998, pp. 246-251.

[24] M.R.Lyu, J.S.Yu, E. Keramidas, S.R. Dalal, " ARMOR: Analyzer for Reducing Module Operational Risk," , Proceedings of FTCS'25. Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995. pp. 137-142.

[25] T. Systa, Yu Ping, H. Muller, " Analyzing Java Software by Combining Metrics and Program Visualization," Proceedings of the Fourth European Software Maintenance and Reengineering, 2000. pp.199 –208.

[26] J.-F. Patenaude, E. Merlo, M. Dagenais, B. Lague, " Extending software quality assessment techniques to Java systems," Proceedings of Seventh International Workshop on Program Comprehension, 1999.pp. 49–56.

[27] Metamata http://www.metamata.com/metrics.html, Dec., 2000.

[28] JProbe http://www.sitraka.com/software/jprobe/, Dec., 2000.