Maintaining Evolving Component-Based Software with UML

Ye Wu and Jeff Offutt
Information and Software Engineering Department
George Mason University
Fairfax, VA 22030, USA
(703) 993-1651
{wuye, ofut}@ise.gmu.edu

Abstract

Component-based software engineering is increasingly being adopted for software development. This approach relies on using reusable components as the building blocks for constructing software. On the one hand, this helps improve software quality and productivity; on the other hand, it necessitates frequent maintenance activities. The cost of maintenance for conventional software can account for as much as two-thirds of the total cost, and it is likely to be more for component-based software.

This paper presents a UML-based technique that attempts to help resolve difficulties introduced by the implementation transparent characteristics of component-based software systems. This technique can also be useful for other maintenance activities. For corrective maintenance activities, the technique starts with UML diagrams that represent changes to a component, and uses them to support regression testing. To accommodate this approach for perfective maintenance activities, more challenges are encountered. We provide a UML-based framework to evaluate the similarities of the old and new components, and corresponding retesting strategies are provided.

Keywords Component-based software, program analysis, software maintenance, Unified Modeling Language (UML)

1. Introduction

Component-based software development facilitates software reuse and promotes quality and productivity. This "building-block" approach has been increasingly adopted for software development, especially for large-scale applications. Much work has been devoted to developing infrastructure for the construction of component-based software [5, 13, 22, 26, 28, 36]. The aims of component-based software development are to achieve multiple quality objectives, including interoperability, reusability, evolvability, buildability, implementation transparency and extensibility. These objectives are intended to facilitate fast-paced delivery of scalable evolving software applications. To this end, a component-based software application often consists of a set of self-contained and loosely coupled components that allow plug-and-play. The components may be implemented by using different programming languages and can execute in various operational platforms distributed across geographic distances. Some components may be developed in-house, while others may be third party off-the-shelf components and the source code may not be available to the developers. When the source code is not available, the component is called *implementation transparent.*

One of the most important objectives for component-based software applications is to improve the maintainability, allowing components to be easily replaced, added, and deleted without adversely affecting the quality of the system. Components can evolve in many different ways. When faults are discovered in a delivered component, corrective maintenance activities need to be performed. Maintaining component-based software is significantly different from maintaining traditional software. Component maintenance is usually carried out by component providers. Component providers can use traditional approaches for corrective maintenance, but component users have difficulties when trying to maintain a component-based application with modified components. This is because implementation

0-7695-1902-4/03 $17.00 © 2003 **IEEE**
IEEE COMPUTER SOCIETY

transparency means the component users only have interface specifications, which may not change. To adequately ensure the quality of component-based software when components are modified, component users need a mechanism to adequately represent providers' modifications, and a methodology to use the information.

This paper presents a mechanism for maintaining component-based software that uses the Unified Modeling Language (UML) [14, 9, 20]. The UML is a language for specifying, constructing, visualizing, and documenting artifacts of software-intensive systems. It can be used to represent key parts of the internal structures of components without relying on the source code.

There are several advantages to adopting the UML. First, the UML provides high level information that characterizes internal behaviors of components, which can be processed efficiently and used effectively during regression testing. Second, the UML has emerged as the industry standard for software modeling notations and various diagrams are available from many component providers. Third, the UML includes a set of models that can provide different levels of capacity and accuracy for component modeling, and thus can be used to satisfy various needs in the real world. In the UML, collaboration diagrams and sequence diagrams are used to represent interactions among different objects in a component. This research uses these diagrams to develop a corresponding interaction graph that is used to evaluate the control flow of a component. Statechart diagrams are used to characterize internal behaviors of objects in a component. The statechart diagrams are used to further refine the dependence relationships among interfaces and operations that are derived from collaboration diagrams.

The literature classifies software maintenance activities into corrective, perfective and adaptive maintenance [33, 35]. *Corrective maintenance* involves modifications on individual classes in a component, and the overall structure of the component remains the same. To adequately perform regression testing for corrective maintenance, we first provide a UML-based mechanism to represent different types of modifications, then based on the UML-based representation, we provide different regression testing strategies with different strengths, which may satisfy different requirements.

Different strategies are needed for perfective and adaptive maintenance activities. The interfaces often remain the same, but the maintenance may change the components internal structures. If the activities introduce new interfaces, new test cases usually need to be generated. For perfective and adaptive maintenance, we first partition the control structure of a component into a different context according to the conditions in the collaboration diagrams. Based on the differences in the contexts, various strategies are provided. In addition to the control structures, we also consider the data dependence relationships that may be affected and thus influence the behavior of the component-based system.

Section 2 of this paper discusses several issues that address the need for a new technique for component-based software maintenance. A new methodology is proposed in Section 3, and Section 4 presents related work on maintaining component-based software.

2. Background

The component-based software literature has introduced a number of new terms, some of which are still used inconsistently. This section of the paper defines terms as used here.

There have been several definitions of software *components*. Szyperski and Pfister [11, 23] suggest the distinctive nature of components from a structural perspective: A component is "a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." Brown [8] defines a component in a broader way: A component is "an independently deliverable piece of functionality providing access to the services through interfaces." For this paper, we will take a generic and simple view of components as an independent piece of software.

Interfaces are access points of components through which a client component can request services declared in the interface and provided by another component. Each interface is identified by an interface name and a unique interface ID. Components often include multiple classes or software modules, and these classes or modules are inter-related with each other, therefore, interface are often inter-related with each other as well.

We define an *event* as an incident that causes an interface to be invoked. We consider only external events in which the responding entity is external to the invoking entity. The incident may be triggered by a different interface, through an exception, or through a user action such as pushing a button. Some exceptions and user actions that require other components to respond may not occur in any interface of a component. To simplify our discussion, we define a *virtual interface* to account for all these possible incidents. Therefore, we define an *event* as an invocation of one interface through another interface.

A component may be a *server* that provides services or a *client* that requests services. Some components

IEEE COMPUTER SOCIETY

may act in both roles. Although components may interact in different ways depending on the underlying infrastructure, components usually only publish their interfaces when acting as a server. An interface itself consists of a service name, a set of parameters, and a set of signatures of the functions that perform the service. To acquire a service, the client first looks up the interfaces published by server components in the system, then makes a request by invoking the identified interface.

3. Methodology

A significant difference between software maintenance for traditional systems and component-based systems lies in the fact that for traditional systems, software implementation and maintenance activities will be carried out by the same company and often even the same people. The company has access to the source code and full control of when and how to perform the maintenance activities. For component-based software, many components will be maintained by third-party component providers and the maintenance activities are invisible to component users (called *implementation transparency*). Even though the modified components may still maintain the same interfaces and events, internal changes may adversely affect the integrated system.

The goal of the methodology in this section is to help component users handle the situation when an internal change to a component has the potential to impact the integrated system. We first establish a UML-based infrastructure to model different types of changes. Then several regression testing strategies are developed to try to validate these changes.

As said before, software maintenance activities are usually classified into corrective, perfective and adaptive. Corrective maintenance is usually performed on individual classes and does not affect a component as a whole, whereas perfective and adaptive maintenance often significantly change the internal structure of the component.

This section first introduces the UML-based infrastructure and corresponding regression testing for corrective maintenance activities. Then, we provide a UML-based framework to evaluate the similarities among old and new components. Based on these similarities and differences, different levels of revalidation will be required.

3.1. Corrective Maintenance Activities

After the component has been delivered to component users, new faults are likely to be identified, requiring corrective maintenance. Component providers have the source code and so can use traditional code-based approaches to define regression tests. But the code is not available to the component users, so in order to create regression testing at the integration and system level, we need a way to represent the changes at a higher level. The UML [14, 9, 20] provides different diagrams for different purposes, for example, class diagrams are used to specify attributes, operations and constraints of classes, and their inheritance relationships with related classes. Collaboration diagrams are used to specify interactions among classes in components and the statechart diagram is used to show behaviors of state-dependent objects or the entire component. Different representations of the software can be derived from different diagrams.

Throughout this section, we will use an ATM example (Figures 1 through 4) to demonstrate our methodology. This ATM component can provide the following services: PIN validation, deposit, withdraw and query.

3.1.1 Representing modifications with UML diagrams

If component providers use UML diagrams to design their components, changes to the source code can easily be depicted in the design documents. We consider class diagrams, collaboration diagrams, and statecharts.

- Class diagrams. Class diagrams provide a class hierarchy within a component and detailed design information of classes in the component. The importance of the class diagram is that when modifications are made to one class, the class hierarchy can be used to determine which other classes can be affected. Those changes will then be represented in other diagrams.
- Collaboration diagrams. Collaboration diagrams illustrate how objects within a component interact. The collaboration diagram in Figure 1 not only shows how objects interact with each other through message description, but also shows the overall control sequence of a component through the numbering mechanism. For instance, the sequence of transitions 1-1.1-1.2-1.3-1.4-2-2.1-2.2-2.3-2.4-2.5-2.6-2.7-2.8 demonstrates a complete PIN validation process. In collaboration diagrams, capital letters demonstrate alternative edges, for example, 1.1 and 1.1A are to be taken under different conditions. Lower case letters show messages

IEEE COMPUTER SOCIETY

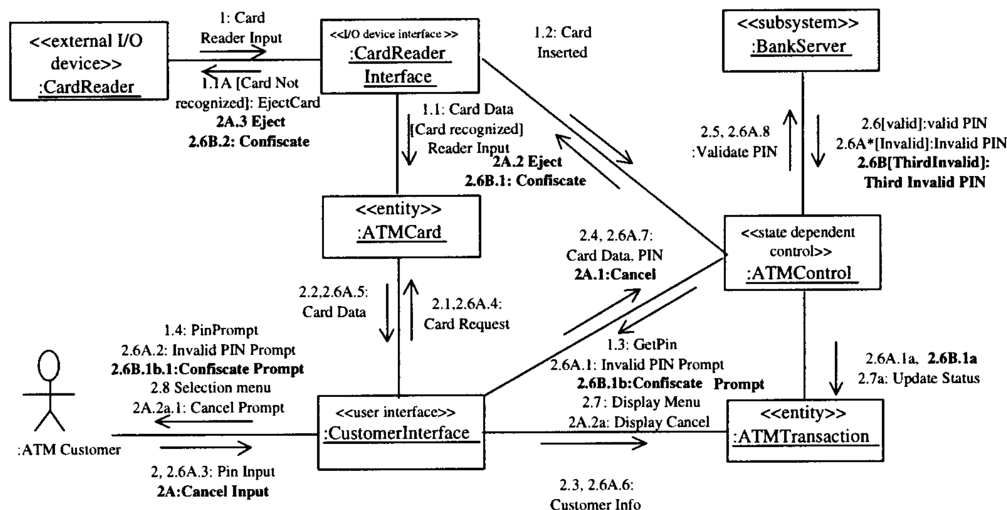

Figure 1. Collaboration Diagram for Validating PINs

that will be passed concurrently. For example, 2.7 and 2.7a will be passed from the ATMControl object to the ATMTransaction object and the CustomerInterface object simultaneously.

Depending on the nature of the modification, different changes that are made to the program can be reflected in collaboration diagrams in the following two ways:

1. Localized changes in a specific member function of a class: In collaboration diagrams, function invocations are represented via messages that are passed among objects. For instance, in Figure 1, message 2.2 (obtain ATMCard data) corresponds to the invocation of the method `getATMCardData()` of an ATMCard object. If the maintenance modifications are to a function that will communicate with other objects, then corresponding messages will be marked as *affected messages*.

2. Changes that might affect interaction sequences: When changes are made in a specific function, the changes could involve adding, removing or changing function calls, which consequently will change the control sequence of the component. The control sequence is illustrated by numbers and can be managed in one of two ways. One, when adding a new function invocation, i.e., adding a new message into the diagram, we need to add a new sequence after the modified function being called in the collaboration diagram. For example, if we need to add a new message between 2.2 and 2.3, the corresponding number for the new sequence is 2.2.1. Two, when removing a function invocation, corresponding messages need to be deleted from the diagram, which will generate a gap in the message sequence and require renumbering all subsequent messages. To avoid the renumbering of subsequent messages, we can create a "dummy" message of a selfloop over the modified function, with a range of numbers that correspond to the messages to be deleted.

- Statechart diagrams. Statechart diagrams are used to depict state changes for a state-dependent control object or component. Statechart diagrams and collaboration diagrams must be consistent with each other. After modifications are depicted on the collaboration diagrams, the impact of the modifications on statechart are derived systematically from collaboration diagrams. The changes in the statechart can add or delete states, or add or delete state transitions. This is illustrated in Figures 1 and 2. Figure 1 shows two changes in bold face, the ability for the user to cancel a transaction (**2A: Cancel Input**) and the ability for the ATM to confiscate a card if the PIN is invalid three consecutive times (**2.6B[ThirdInvalid]**). This is reflected in Figure 2 with the two new states **Ejecting** and **Confiscating** (in bold).

IEEE COMPUTER SOCIETY

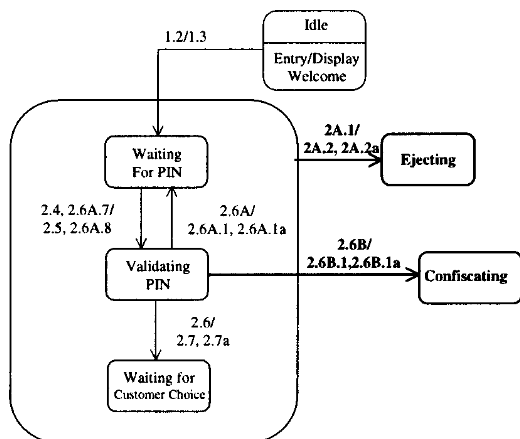

Figure 2. Statechart for ATM Control Diagram

3.1.2 Regression testing for corrective maintenance

The analysis of the UML diagrams in the last section can be used as a basis to develop several regression testing strategies. This method assumes that the UML diagrams were changed correctly. An obvious test requirement is that each changed artifact in the collaboration diagram should be retested at least once. At a further level of detail, all test cases that executed any of these affected artifact need to be rerun. The first approach is cost-effective, but may not result in as thorough a test, while the second approach may be much more costly. A middle approach may be developed that relies on change impact analysis. This has been done at the code level [25], but not at the UML level.

The weakness of the first approach is that only the artifacts that have been directly modified will be tested, thus indirect effects caused by the modifications will be ignored. With an adequate analysis of all possible side effects that might occur, retesting each individual scenario will be more reliable.

Besides the validation of direct modifications, we need to further identify the effects that these changes may impose on other parts of the component. These effects can be classified into the following two categories:

- Impacts of changes on control sequences. In general, artifacts that have been modified in a component can be invoked in different scenarios. For instance, in Figure 1, when we add a control sequence 2A, 2A.1, 2A.2 and 2A.3, this sequence can be invoked in many different scenarios. As shown in the Figure 2, the changed statechart diagram, we need to test that sequence in three different scenarios: (1) cancel while waiting for PIN, (2) cancel while validating PIN, and (3) cancel while waiting for customer choice. Therefore, we not only need to validate the modified artifacts in the collaboration diagram once, we also need to retest all possible affected scenarios that are demonstrated in the statechart diagram. Moreover, if even higher quality requirements are required, we not only need to retest all affected states and state transitions in the statechart diagram, we may need to retest all paths that include affected states and transitions as well.

- Impacts of changes on data dependencies. In addition to the impact of changes on control sequences, changes may also affect data dependence relationships between two control sequences. An invocation of an interface of a component is in fact an invocation of a function implemented by the component. Therefore, when a function declared in an interface v_1 has a data dependence relationship with another function declared in another interface v_2, the order of invocation of v_1 and v_2 could impact the results.

  In a collaboration diagram, messages that flow into an entity object without a corresponding reply message often imply an update of that object. If messages that flow into an object are followed by another message that flows out of that object, it often reflects information retrieval or response after processing from that object. If we change the message that updates an object, we need to retest the control path that retrieves information from the same object. For instance, in Figure 3, message W4 enters the Account object, but no message W5 comes out. Therefore, we assume W4 will update information in the Account object. In addition, messages D2 and D3 go into and come out of the Account object, so they retrieve information. Taken in combination, D2 and D3 depend on W4. If changes have been made to W4, then at least one test case that explored this relationships needs to be retested.

3.2. Perfective and Adaptive Maintenance Activities

For component-based systems, perfective maintenance activities usually retain the existing interface specifications, but they may require the component to be redesigned. Thus, the internal structure of the new component can be different. To ensure the quality of

IEEE COMPUTER SOCIETY

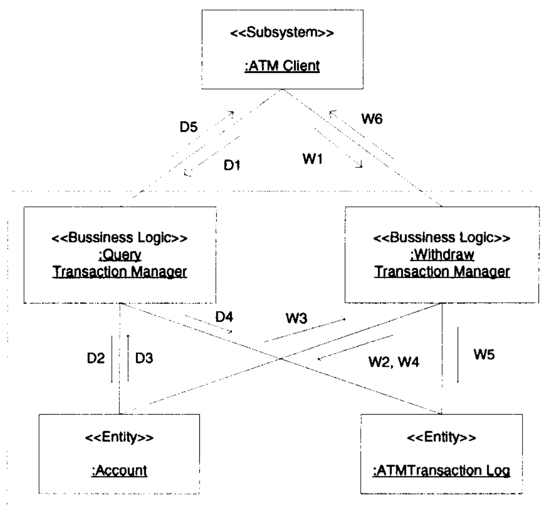

Figure 3. Collaboration Diagram for Withdraw and Query Transaction

the system after a new version of the component is integrated, an intuitive approach is to completely retest that component. Nevertheless, this approach is too costly, particularly when third party components are expected to be frequently updated. Furthermore, it is against the primary objective of maintainability and reducing maintenance cost.

How do we determine what tests can be trusted and reused and what tests need to be modified? Based on the observations from the last section, we know that class diagrams, collaboration diagrams and statechart diagrams can be used to depict the component control structure and data dependence relationships among interfaces. This information may be helpful in perfective maintenance as well. Even though the internal structure of a component can be completely different, including new classes and different sequences of control, some artifacts may still be useful to depict the similarity of the control sequence and data dependence relationships.

3.2.1 Control similarity evaluation

Collaboration diagrams represent control sequences by alternative paths, which are annotated by capital letters on the messages. For instance, Figure 4 shows the three alternate paths W3, W3A and W3B. For each alternative path, a guard constraint is defined on the message to determine when the component will execute that path. Given an alternative path, we can partition the executions of the component into different contexts depending on all possible constraints. A *guard* is a boolean variable or expression that is used to choose alternative paths in a collaboration diagram, while a *context constraint* is a set of guards associated with an execution of an interface of a component. Note that guards are compared for equality based strictly on the string of characters.

For example, Figure 4 shows an execution of the withdraw interface of the old ATM server component is W1-W2-W3-W4-W5. The context constraint is that the execution is constrained by [*Valid account*] and [*Sufficient funds*]. In the new component, the context constraint for the same test case will be [*Within daily limit*], [*Valid account*], and [*Sufficient funds*].

Assume a set of guards $S = s_1, s_2, ..., s_p$ and a set of context constraints $C = c_1, c_2, ..., c_q$ of the old component; a set of guards $S' = s'_1, s'_2, ..., s'_r$ and a set of context constraints $C' = c'_1, c'_2, ..., c'_s$ of the new component. The following list describes different scenarios that may occur and different strategies can be adopted:

- The new component's context constraints remain the same after performing the perfective maintenance activities. That is, $S = S'$ and $C = C'$ (the modified context constraint adds guards to the old context constraint). This means the control structure of the new component remains unchanged. If the new component is adequately tested, we can retest each interface once to make sure the similarity in the control structure will preserve the quality of the component and the component-based system.

- The new component introduces new guards that further partition the execution of the component. That is, $S' \supseteq S$ and for each $c \in C$, there exists a $c' \in C'$, where $c \subseteq c'$. For instance, in Figure 4, the old components have two guards and three different context constraints:

   1. [*Valid account*] and [*Sufficient funds*]
   2. [*Valid account*] and [*Insufficient funds*]
   3. [*Invalid account*]

   The new component introduces one more guard, which is intended to enhance the functionality of the withdraw transaction by introducing a daily access limit. The modification of the collaboration diagram follows the principles described in the last section; the new diagram is shown in Figure 4. Therefore, the new component has four context constraints:

IEEE COMPUTER SOCIETY

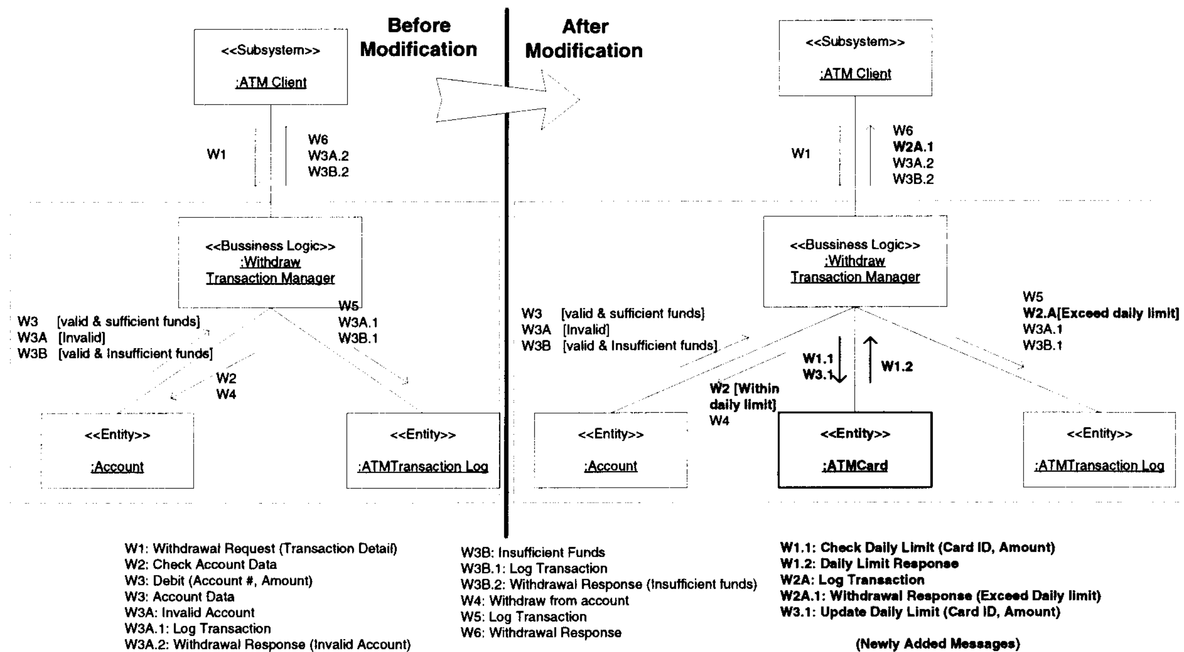

Figure 4. Collaboration Diagram for Withdraw Transaction

1. [*Exceed Daily limit*]
2. [*Within Daily limit*] and [*Valid account*] and [*Sufficient funds*]
3. [*Within Daily limit*] and [*Valid account*] and [*Insufficient funds*]
4. [*Within Daily limit*] and [*Invalid account*]

For this case, new guards such as [*Exceed Daily Limit*] and [*Within Daily Limit*] definitely need to be retested. The combination of other guards all fall into old contexts, thus if the old and new components are properly tested by the same provider, we should have enough confidence that they will be correct. Therefore, in this case, we only need to retest context constraint 1 and one of 2, 3 or 4.

- Removing some guards in the new component will merge some partitions in the old component. That is, $S' \subseteq S$ and for each $c \in C$, there exists a $c' \in C'$, where $c \supseteq c'$. The old component had different values for different guards, thus different outputs are expected. Nevertheless, in the new component, two different guards will be treated as one scenario. So it is necessary to properly test the two different values of a removed guard.

- The new component has guards that have been added, removed or recombined. That is, $S \nsubseteq S'$ and $S \nsupseteq S'$, and there exists $c' \in C'$, where $c' - \{new\ guards\} \nsubseteq$ any $c \in C$ and vise versa. For this case, two different approaches are adopted.

1. Test all new guards, and retest all context constraints c', where $c' - \{new\ guards\} \nsubseteq any\ c \in C$. This is a relatively inexpensive strategy, as it ignores potential side effects of merging partitions.

2. Test all new guards, and test all context constraints c', where $c' - \{new\ guards\} \neq$ any $c \in C$. This is a more strict and safe strategy, as it consider all differences in the partition between the new component and old component, but more test cases will be selected to rerun.

- The order of the guards has some effects. So far, we have assumed that the order of different guards do not adversely affect the integrated system. If that assumption does not hold, the order of guards need to be taken into consideration, which will incur more overhead than the previous approaches.

0-7695-1902-4/03 $17.00 © 2003 **IEEE**
IEEE COMPUTER SOCIETY

3.2.2 Data dependence similarity evaluation

The previous definitions only considered the control sequence. This subsection considers the effects on the data dependences.

Compared with the old components, redesigning the old component may change the data dependence relationships among different interfaces. New data dependence relationships can be added, existing dependence relationships can be removed, or the context that enables the dependence relationships may be changed.

If a data dependence relationship is identified to be new, then either a new test case needs to generated or some existing test cases that explore the dependence relationships must be rerun. If a data dependence relationship is removed, then a test case that was originally designated for that relationship needs to be retested. To determine the changes in data dependence relationships, a context constraint pair (c_1, c_2) needs to be identified for each dependence relationship that defines the scenarios to enable the dependence relationship. To be more specific, under context constraint c_1, an update of an object is issued, and under context constraint c_2, a reference to the same object is issued. To determine whether the scenario changes from the old to the new components, the criteria defined early in this section 3.2.1 can be adopted. Therefore, if either c_1 or c_2 changes, the data dependence relationship needs to be retested under the new scenario.

4. Related Work

Component-based software development synthesizes the advantages of object-oriented and distributed software. Recently, techniques like CORBA [1], COM+ [27], and Enterprise JaveBean [2] have led to component-based approaches being adopted more often. Various methodologies and frameworks have been proposed to support component-based software development [5, 13, 22, 26, 28, 36]. Barrett et al. [5], Cugola et al. [13] and Sullivan et al. [36] proposed event-based infrastructure and methodologies to develop large flexible component-based systems. Mezini and Lieberherr facilitate the construction of complex software by making the collaborations explicit, which results in better reuse. To ensure quality of component-based software a number of studies [12, 15, 37, 38, 40] have been carried out to analyze the characteristics of component-based software. When a component is modified or upgraded, a maintenance activity occurs.

Several researchers have looked at maintenance of traditional and OO software systems. Rothermel [30], Rosenblum [29] and Graves [16] proposed different measurement models to evaluate different techniques. Among different criteria, precision, efficiency and safeness are the three major considerations.

Minimization [19, 24, 34] and coverage methods [6, 7, 18, 15] attempt to select a minimum subset from the test pool to cover portions of the program that need to be retested. Therefore, they are very often efficient and not always safe. However, the implementation transparency nature of some components prevents the minimization and coverage methods from being directly applied to component-based software. For example, the data-flow based approach [18] cannot obtain complete data flow information. On the other hand, some of the methods [15] might be adopted, but they only focus on the interface and event levels instead of the collaboration relationships among them, so it might not be possible to guarantee the quality of the modified software.

Safe maintenance techniques [4, 17, 31, 32, 39, 21, 3, 10] are necessary for life-critical software systems. Safe approaches will guarantee that those modification-revealing test cases [31] will be selected, therefore helping to guarantee the reliability of the modified software. Slicing methods have widely been proposed to support software maintenance, among which, execution and relevance slicing [4, 17] are typical safe slicing methods. Execution slicing methods will retest all test cases once they touch at least one modified statement, block, or function, and relevance slicing approaches use data flow information to determine the tests that will truly affect the output of the program and retest them. Rothermel and Harrold [31, 32] proposed a regression testing approach for both procedure-based and object-oriented software. In their approach, the decision of the test case selection is mainly based on the comparison of the control flow graphs of the original and the modified programs.

Firewall approaches, proposed by Leung and White [39], Kung [21] and Abdullah [3], establish a firewall that separates all modules that may be affected by modifications to others. Then it will choose test cases that exercise at least one of the modules within the firewall. Chen and Rosenblum's method [10] will first identify entities (functions, types, variables and macros) that have been modified or affected. Then a test case is selected if it executes at least one affected entity.

When applying safe approaches to component-based systems, heterogeneity and the distributed characteristics may produce unexpected outputs, because the modifications may introduce incompatibility among different things like operating systems and programming languages. In addition, precision will be sacrificed to accommodate the implementation transparency, re-

IEEE COMPUTER SOCIETY

sulting in larger retest suites and longer maintenance periods.

5 Conclusions and Future Work

We have presented a new UML-based approach for maintaining evolving component-based software. When performing maintenance activities, our approach is designed to overcome the difficulties that are introduced by the fact that component-based software is implementation transparent. The UML is now the industry standard for software modeling notation. Therefore our approach can provide a feasible guideline for component providers to precisely model their behavior and pass through component users. Properly utilized, the information can efficiently and effectively maintain evolving component-based software. For perfective maintenance activities, which has not been fully explored by researchers, we propose a UML-based framework to evaluate the similarity among old and new components, and can be the foundation for adequate perfective and other type of maintenance activities.

Our ongoing research directions on this topic are to develop a tool supporting automation of the technique and to conduct empirical studies to assess the effectiveness of our approach. At the same time, we will further enhance the technique to help resolve problems caused by distributed characteristics such as synchronization.

6 Acknowledgements

We would like to thank Yuqin Ding for helping us to find a number of errors in the text.

References

[1] *CORBA Component Model Joint Revised Submission.* Object Management Group, 1999.

[2] *Enterprise JavaBeans Technology.* Sun Micro, http://java.sun.com/products/ejb, 2002.

[3] K. Abdullah and L. White. A firewall approach for the regression testing of object-oriented software. In *Software Quality Week Conference*, San Francisco, 1997.

[4] H. Agrawal, J. Horgan, E. Krauser, and S. London. Incremental regression testing. In *International Conference on Software Maintenance*, pages 348–357, September 1993.

[5] D. J. Barrett, L. A. Clarke, R. L. Tarr, and A. E. Wise. A framework for event-based software integration. *ACM Transaction on Software Engineering and Methodology*, 5(4):378 – 421, 1996.

[6] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proceedings of the 20th ACM Symposium of Principles of Programming Languages*, January 1993.

[7] D. Binkley. Reducing the cost of regression testing by semantics guided test case selection. In *Proceedings of International Conference on Software Maintenance*, October 1995.

[8] Alan W. Brwan. Background information on CBD. *SIGPC*, 18(1), August 1997.

[9] John Cheesman and John Daniels. *UML components : A simple process for specifying component-based software.* Addison-Wesley, 2001.

[10] Y. F. Chen, D. Rosenblum, and K. P. Vo. Testtube: A system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering*, pages 211–222, May 1994.

[11] Szyperski Clemens. *Component software: Beyond object-oriented programming.* Addison-Wesley, 1998.

[12] J. E. Cook and J. A. Dage. Highly reliable upgrading of components. In *International Conference on Software Engineering*, pages 203 – 212, Los Angeles, 1999.

[13] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *The 20th International Conference on Software Engineering*, Kyoto, Japan, April 1998.

[14] Desmond Francis D'Souza and Alan Cameron Wills. *Objects, components and frameworks with UML.* Addison-Wesley, 1999.

[15] S. Ghosh and A. P. Mathur. Issues in testing distributed component-based systems. In *First International ICSE Workshop on Testing Distributed Component-Based Systems*, Los Angeles, 1999.

[16] T. L. Graves, M. J. Harrold, J. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *Proceedings of the 1998 International Conference on Software Engineering*, pages 188–197, 1998.

[17] R. Gupta, M. J. Harrold, and M. L. Soffa. Program slicing based regression testing techniques. *Journal of Software Testing, Verification and Reliability*, 6(2):83–112, June 1996.

[18] M. J. Harrold and M. L. Soffa. Interprocedural data flow testing. In *Proceedings of the Third Testing, Analysis, and Verification Symposium*, pages 158–167, December 1989.

[19] J. Hartmann and D. J. Robson. Techniques for selective revalidation. *IEEE Software*, 16(1):31–38, January 1990.

[20] George Heineman and William Councill. *Component-based software engineering : Putting the pieces together.* Addison-Wesley, 2001.

[21] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. On regression testing of object-oriented programs. *The Journal of Systems and Software*, 32(1):21–40, January 1996.

[22] G. Larsen. Designing component-based frameworks using patterns in the UML. *Communications of the ACM*, 42(10):38–45, 1999.

[23] Gary Leavens and Murali Sitaraman. *Foundations of component-based systems.* Cambridge, UK, New York:Cambridge University Press, 2000.

[24] J. A. N. Lee and X. He. A methodology for test selection. *The Journal of System and Software*, 13(1):177–185, September 1990.

[25] Michelle Lee, Jeff Offutt, and Roger Alexander. Algorithmic analysis of the impact of changes to object-oriented software. In *34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '00)*, pages 61–70, Santa Barbara, CA, August 2000.

[26] M. Mezini and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 97–116, 1998.

[27] http://www.microsoft.com/com/ Microsoft. *COM+ Component Model.* 2002.

[28] O. Nierstrasz, S. Gibbs, and D. Tsichritzis. Component-oriented software development. *Communications of the ACM*, 35(9):160–165, 1992.

[29] D. S. Rosenblum and E. J. Weyuker. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Transactions on Software Engineering*, 23(3):146–156, 1997.

[30] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[31] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[32] G. Rothermel, M. J. Harrold, and J. Dedhia. Regression test selection for c++ software. Technical Report TR99-60-01, Oregon State University, 1999.

[33] Steve Schach. *Object-Oriented and Classical Software Engineering.* WCB/McGraw-Hill, Boston MA, fifth edition, 2002.

[34] B. Sherlund and B. Korel. Modification oriented software testing. In *Software Quality Week Conference*, pages 1–17, 1991.

[35] IEEE Computer Society. *Standard for Software Maintenance.* Institute of Electrical and Electronic Engineers, New York, 1998. ANSI/IEEE Std 1219-1998.

[36] K. Sullivan and D. Notkin. Reconciling environment integration and component independence. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 22–33, 1990.

[37] J. Voas. Maintaining component-based systems. *IEEE Software*, 15(4):22–27, July/August 1998.

[38] E. J Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59, September/October 1998.

[39] L. White and H. K. N. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Proceedings of International Conference on Software Maintenance*, pages 262–271, 1992.

[40] Y. Wu, D. Pan, and M. Chen. Techniques for maintaining evolving component-based software. In *Proceedings of International Conference on Software Maintenance*, pages 272–283, 2000.

0-7695-1902-4/03 $17.00 © 2003 **IEEE**

IEEE COMPUTER SOCIETY