Improving Current Component Development Techniques for Successful Component-Based Software Development

Sudipto Ghosh

Computer Science Department, Colorado State University
Fort Collins, Colorado, USA
ghosh@cs.colostate.edu

Abstract. Current research on component-based software development (CBSD) emphasizes work on problems faced in composing, certifying and deploying components. Many of these problems are caused by inadequacies in the process of *component development* apart from those that are part of *CBSD* itself. In this paper, it is argued that changes need to be brought to several aspects in the process of development of components in order to make CBSD successful.

1 Introduction

Current trends in the field of software engineering indicate that future software systems will be developed from components. Tools such as C++ Builder can be used to build relatively complex database browsing applications in a matter of minutes. The popularity of Component-based Software Development (CBSD) can be gauged from 1) the number of component technologies available (CORBA, JavaBeans/EJB, DCOM/ActiveX and .Net), 2) the variety of component configuration and deployment tools, and environments available, and 3) the host of components available for use.

Component-based software development has two distinct activities:

1. Development of components for use in the CBSD process.
2. The CBSD process itself which includes component assembly.

It is important to distinguish between the two because more often than not, different organizations (or *independent* entities within the same organization) will be involved with the two aspects.

Current research on CBSD has focused on issues relating to modeling component-based systems (e.g. using UML [3, 4]), component architectures [1, 16], component technologies (the implementation aspects), prediction [9, 14], verification [6] of properties, and testing [5, 7, 12] of component assemblies. Most of these techniques do not focus on the development of the component itself. In most cases, components are assumed to be used black-box possessing limited information. This leads to several problems in the use of components, such as program comprehension, validating component assemblies and maintenance. Such problems may adversely affect the acceptance of CBSD processes in the industry.

We focus on these problems and describe them in Section 2. In Section 3 we demonstrate how these problems may be addressed earlier during the development of the components. Our concluding remarks are presented in Section 4.

2 Problems faced in CBSD

CBSD processes usually require the definition of initial component specifications and designing a component architecture [3]. Component interactions are determined for each system interface operation. Component object architecture constraints are defined as needed. Interfaces are factored as appropriate. Component specifications (realization contracts) and interface specifications (usage contracts) need to be made precise. This process described in [3] is just one of the ways of designing for component-based systems. Since this area is relatively new, work needs to be done in formalizing aspects of the CBSD process. The use of UML and its extensions for components would be one way of describing designs precisely. We will revisit the use of UML later in this paper. The focus of this paper is not on the design of complete systems, however. Instead we are looking at problems related to the use of actual component implementations that meet design specifications of the target application.

The following is a non-exhaustive list of problems that may be faced by component users:

1. ensuring version compatibility
2. understanding components
3. selecting components
4. maintaining component software
5. validating component assemblies
6. performing regression testing on component software

2.1 Ensuring version compatibility

Versions of components may change during the course of evolution of the target application. Usually vendors will provide compatibility with other versions. Nevertheless, the management of version numbers for the applications is more complex because of the availability of multiple component versions. Also, there may be cases where a new component may change the API (e.g. methods move up or down the inheritance hierarchy, or signatures change). Versions may be checked at bind-time if direct interfaces are used [16]. In the case where indirect interfaces are used, there may be a problem coupling arbitrary third parties. One may require immutable interfaces to solve the problem.

2.2 Understanding components

Program understanding is important for software engineering activities such as software reuse and maintenance. Components have important aspects that need to be understood. Components need to be packaged, configured and deployed. UML models can show package diagrams and deployment diagrams. Such models have been used to describe component models. The knowledge about packaging helps in understanding the units that build the component. We also need knowledge to set up the environment where the component is expected to be used, configured and deployed. Component functionality is another aspect that needs to be understood. A component is then treated as a software entity that presents services to other components. Designing and implementing an application requires understanding of how a collection of components will collaborate. UML collaboration diagrams are usually at the level of objects and classes, but can be easily extended for use in components. Specification of service aspects may be facilitated by a contract between the service provider and the consumer. However, there are other aspects (e.g. memory requirements, presence of disk space, location of files) that are also required.

There are several techniques for understanding components. Here are two:

1. *Component implementation model*:
   The component implementation model [15] supports a database to store information about the class structure and inheritance structure of an object-oriented system. This method requires reverse engineering of source code. Since in most cases source code is not available, this method is usually not applicable.
2. *Interface-based models*:
   Several techniques are based on using information from the interfaces of components. Connectors (interfaces) are used as the mechanism by which information and control is transferred between components. Black-box understanding approaches attempt to use the information regarding the interface (e.g. API) to understand the component. Users of components endeavor to write test cases that exercise the component and in the process, get an understanding of the behavior of the component. Korel [8] reports on a black-box method that requires the user to write specifications for the behavior and then automatically generate *test-code* that has the required behavior. A *test case generator* generates test-cases from this test-code and these tests can be used as input to the real component. The author claims that 1) this process removes the guess work, and 2) writing assertions is simpler than writing test-cases. There are several limitations of this approach. Aside from the fact that writing assertions is a non-trivial task, there are practical limitations of the approach which breaks down for most complex types used in real-world systems. The examples used in the proposed approach restrict themselves to scalar types. Complex types present an infinite number of states, and it is much harder to randomly create complex objects that are in valid states than creating integers in valid states. Not handling such types makes the analysis not quite applicable to real-world problems.

In the program comprehension literature, there are several *Cognition models* that have been proposed. The program understanding techniques discussed above are usually part of these cognition models. These models are mostly based on program code, but some may be adapted for black-box components. The models are as follows:

1. *Top-down comprehension model*:
   Brook's top-down comprehension model [2] starts with a global hypothesis and attempts to verify this hypothesis. This is done until a level is reached where the hypothesis can be verified or demonstrated to be false.
2. *Bottom-up comprehension model*:
   Schneiderman's bottom-up comprehension model [13] gathers knowledge from small chunks of code to build the cognitive model to higher levels of abstraction. For most components, we will not have access to code. However, the same model can be applied to black-box components by applying a different method to gather knowledge about individual components.
3. *Systematic comprehension model*:
   Littman et al. [10] describe a model that requires understanding control- and data-flow abstractions in the code. This model is less likely to work for component software.
4. *Integrated comprehension model*:
   von Mayrhauser and Vans [18] developed and integrated the top-down and bottom-up comprehension models. The top-down strategy is advocated for familiar code, and the bottom-up strategy otherwise.

2.3 Selecting components

Based on the functional, non-functional and business requirements of an application, developers may need to select a component from a set of available components. The component needs not only to meet the functional requirements, but also needs to possess the appropriate mix of quality attributes, and have no adverse impact on the quality of the target application [17]. The approach suggested by Voas [17] requires black-box testing to determine quality attributes, and interface perturbation analysis (IPA) and operational system testing to determine the impact of using the component. There are other approaches that may be used to first understand the behavior of a component (e.g. see previous sub-section) and then make a selection.

2.4 Verifying and validating component assemblies

As with any software development process, verification and validation are important activities in CBSD. Since components are mostly black-box, validation of the component assembly cannot make use of code-based test generation techniques. They need to rely only on the specifications, which may be available in various degrees of detail, or as part of the API (Application Programming Interface). Testing approaches attempt to somehow automate the process of test generation and oracle generation from the available specifications. Korel's approach [8] is to write in the form of assertions the requirements of the component and then automatically generate check code from the assertions. This check code could then be used to generate test cases using white-box approaches. These test cases would complement those that are generated using black-box approaches. Edwards' strategy [5] automates black-box testing by automatically generating test-drivers, black-box test data and oracles from a model-based specification of the software component. Model-based specifications can be relatively easily written for *mathematical* Abstract Data Types, but it is not clear how scalable the process would be for specifying arbitrary classes of complex objects. Moreover, in cases where the process is not fully automatable, the effort and understanding required for human intervention may prevent this process from being used altogether. The approach also requires the availability of hooks inside the component that enable the wrapper to obtain state information about the component.

Techniques that do not involve testing are also available. Modular reasoning techniques [19, 20] use local certification to verify properties of the component assembly.

2.5 Performing regression testing on component software

Regression testing is a costly process and often testers use minimization techniques to reduce the amount of re-testing. However, a lack of relevant testing information in components makes it difficult to effectively determine the test cases that should be rerun. Harrold et al. [7] propose to use metadata with components to address the problem of test case selection. The metadata may be based on the information relating test cases to code or specifications. The examples shown in the paper relate to coverage criteria that are based on white-box criteria (statement or branch coverage) or method entry in a component. While it is theoretically possible to apply this technique to any criteria, the technique seems to be not scalable with respect to dataflow-based test criteria. The problems lie in the determination of appropriate meta-data that corresponds to dataflow-based test criteria.

2.6 Maintaining applications using software components

Software maintenance may be necessitated because of changes in components (versions, availability of superior components, etc), or changes in the requirements of the application and a host of other reasons. The process of maintenance may require the development of new wrappers for existing components, or the procurement of new components. This again involves understanding and selection of components. Finally, the new version of the application needs to be validated, and that process will involve regression testing.

3 Addressing problems during component development

All the techniques proposed for various activities require some information about the component. Even if code were available, we would still run into issues mentioned in the section on program understanding. Using code would actually negate the advantages of using components. Ideally a component [16] should be "units with contractually specified interfaces and explicit context dependencies only."

In most cases, the code is not available and the component user needs to adapt the CBSD process according to the amount of information available. As we saw earlier, most of the problems emanate from the fact that component users lack sufficient information about the component. The particular kind of information required depends on the type of activity for which it will be used.

Version control is now well understood and several techniques are available. Determining the compatibility of versions can be done based on information available with a component, or by using component understanding and testing techniques. The problems of selection of appropriate components and software maintenance also boil down to the understanding and/or testing of components. The goal of our work, therefore, is to identify the necessary conditions that will enable 1) easier understanding, and 2) effective verification and validation of components.

Components may be logically-encapsulated code fragments such as programming libraries, class libraries, reusable modules and wrapped legacy systems [16]. The kind of information and the process of providing and gathering the information also depend on the *granularity* of a component. Several techniques are illustrated in the literature by taking as examples components at different levels of abstraction: 1) a function in the API is used in Korel's work [8], 2) a C++ template class implementing an abstract data type in Edwards' work [5], or 3) a Java class in Harrold et al. [7]. The following paragraphs illustrate the issues that arise in each method and possible solutions to the problems. The list is not exhaustive, but does highlight some of the approaches that may need to be adopted for component development.

3.1 Korel's interface probing method

The interface probing method relies on one difficult task: writing assertions about the function to be used. If the developer of the function (API) provides this information in a formal and precise notation, this aspect of understanding the "function" component is simplified. This still does not help in defining other side-effects a function may have because Korel's approach only restricted the technique to function parameters. However, the component developer could also enumerate the side-effects and non-functional aspects of the function.

Since the interface probing method deals with primitive types that are passed as arguments to a method/function, a lot of object-oriented application developers may have trouble using it. In most real programs, arguments are objects. Theoretically, we can model objects in a similar way, as after all, at some point, the object will contain instance variables that are primitive types. However, practically this is not useful. We suggest that using UML class diagrams for describing the argument objects may assist in the description of the function interface. The classes would also provide constraints on the attributes. No "secret" information is being revealed that may jeopardize the technical superiority of that component (or disclose intellectual property). If a component is being developed in-house for reuse, the developers may maintain information about the detailed design of the component.

3.2 Edwards' automatic black box testing of components

This approach makes two important assumptions. One is that a model-based specification is available, or at least an informal specification can be made available to describe the component being used. The other is that component hooks are available that enable the attachment of wrappers. The obvious strategy would be to have the component developer also provide the specifications for the component using a model-based specification language. However, the availability of component hooks may be a problem. Some languages provide support for enabling such features (e.g. reflection in Java). Some component middleware provide higher level abstractions that enable features like introspection (e.g. properties in Java Beans). However, there are cases for which there is no programming language support, nor is there a component middleware that provides the abstraction. It is left to the component developer to provide the hooks. This problem may also arise in the case of Java Beans, where not all properties reveal the entire state of the component. This may break encapsulation and also may not be desired by the developer. In C++, the developer would have to include the hooks as part of the development process itself.

The application of model-based specification languages to complex components is another problem. Complex components would include several cooperating classes. The languages will most likely not scale up well. Moreover, it is not clear whether hooks would need to be provided for all the classes, which gives too much insight on how the component is constructed, or only to the ones interfacing with the outside world. For in-house use of a component, developers may have conditional compilation of the hooks installed in all the classes. Component users would then be able to access all the objects, but still not be required to perform white-box testing which usually stretches the available resources.

3.3 Harrold et al's regression testing method

The regression testing method proposed by Harrold et al. [7] advocates the use of meta-data for components. The meta-data will contain information (or methods to obtain the information) regarding what tests were performed on the components, so as to reduce the amount of re-testing done during maintenance. The papers [7, 11] provide a starting point for describing the kind of data that developers need to provide to enable efficient regression testing. Similar to the approach for providing hooks, developers can use programming language support, component-model support or other means to provide the metadata. The problem of scalability of this method comes into play depending on which test adequacy criterion was used to test the components. Changes made to a component may make it difficult to map the old tests to the new code if, for example, dataflow-based criteria were used in the testing process. However, the key idea is that a format can be selected to provide meta-data information.

Issues like heterogeneity, compatibility with related standards (e.g. CORBA components with DCOM), and concurrency constraints also need to be specified. Developers may provide the same information in different ways using different languages. However, as long as tools are available to analyze the information, this is not a problem. Already there are several standards for describing metadata.

3.4 Discussion

It is clear that component developers need to provide component information. The information is generated during different phases of component development. Regardless of the process used in development, the basic tasks (e.g. analysis, design, implementation, testing and maintenance) remain the same, although they may be performed with less or more rigor, and in different orders. Design models can be maintained by the developer and those aspects of the design that are relevant to future CBSD activities can be packaged along with the component. All tests that are written and any coverage analysis that is performed may be recorded. These additional tasks may cause an extra burden on the already stretched resources of the developers, but will eventually help in producing quality software. This will probably require changes in the software development process used by the developer. The changes will depend on the type of process employed and may be incorporated with varying degrees of difficulty.

4 Conclusion

In this paper we described how various activities in CBSD run into problems that are caused mainly because of lack of information about components. We surveyed the different techniques that are proposed in the literature for solving the problems, and also the different kinds of information each technique requires. Ultimately we discuss the need for improving the process of component development so that the necessary information does become available to the component users. While this paper does not attempt to give a formal description of all kinds of information that will fit all kinds of activities, it is nevertheless a starting point from where researchers and practitioners can take off. Providing information will not solve all the problems faced in CBSD, but it will at least provide a solid foundation upon which sophisticated techniques for CBSD, such as reasoning, verification, validation and certification may be based.

References

1. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Reading, Massachusetts, 1998.
2. R. Brooks. "Towards A Theory of the Comprehension of Computer Programs". *International Journal of Man-Machine Studies*, 18:543–554, 1983.

3. J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software.* Addison-Wesley, New Jersey, 2001.
4. D. D'Souza and A. Wills. *Objects, Components, and Frameworks with UML.* Addison-Wesley, New Jersey, 1998.
5. S. H. Edwards. "A Framework for Practical, Automated Black-box Testing of Component-based Software". *Journal of Software Testing, Verification and Reliability*, 11(2):97–111, June 2001.
6. K. Fisler, S. Krishnamurthy, and D. Batory. "Verifying Component-Based Collaboration Diagrams". In *Proceedings of 4th ICSE Workshop on Component-Based Software Engineering*, pages 84–87, Toronto, Canada, May 2001.
7. M. J. Harrold, A. Orso, D. Rosenblum, G. Rothemel, M. L. Soffa, and H. Do. "Using Component Metadata to Support Regression Testing of Component-based Software". In *Proceedings of the International Conference on Software Maintenance*, Florence, Italy, November 2001.
8. B. Korel. "Black-box Understanding of COTS Components". In *International Workshop on Program Understanding*, pages 226–233, Pittsburgh, Pennsylvania, May 1999.
9. K. Lau. "Component Certification and System Prediction: Is There a Role of Formality". In *Proceedings of 4th ICSE Workshop on Component-Based Software Engineering*, pages 80–83, Toronto, Canada, May 2001.
10. D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. *Mental Models and Software Maintenance.* Soloway and Iyengar, Eds. Empirical Studies of Programmers. Ablex Publishing Corporation, Norwood, NJ, 1986.
11. A. Orso, M. J. Harrold, and D. Rosenblum. "Component Metadata for Software Engineering Tasks". In *Proceedings of the 2nd International Workshop on Engineering Distributed Objects (EDO 2000), LNCS Vol. 1999, Springer Verlag*, Davis, California, November 2000.
12. D. S. Roselblum. "Adequate Testing of Component-based Software". Technical Report TR 97-34, Department of Information and Computer Science, University of California, Irvine, California, August 1997.
13. B. Schneiderman. *Software Psychology: Human factors in Computer and Information Systems.* Winthrop Publishing, Cambridge, MA, 1980.
14. M. Sitaraman. "Compositional Performance Reasoning". In *Proceedings of 4th ICSE Workshop on Component-Based Software Engineering*, pages 93–97, Toronto, Canada, May 2001.
15. H. Sneed and T. Dombovari. "Comprehending a Complex, Distributed Object-Oriented Software System: A Report From the Field". In *International Workshop on Program Understanding, IEEE Computer Society press*, pages 218–225, Pittsburgh, May 1999.
16. Clemens Szyperski. *Component Software Beyond Object-Oriented Programming.* Addison-Wesley Publishing Company, USA, 1998.
17. J. M. Voas. "Certifying Software for High-Assurance Environments". *IEEE Software*, 16(4):48–54, July/August 1999.
18. A. von Mayrhauser and A. Vans. "Program Comprehension During Software Maintenance and Evolution". *IEEE Computer*, 28(8):44–55, August 1995.
19. Bruce W. Weide and Joseph E. Hollingsworth. "Scalability of Reuse Technology to Large Systems Required Local Certifiability". In *Proceedings of the Fifth Annual Workshop in Software Reuse*, pages 133–141, Larry Latour, Steve Philbrick, and Mark Stevens, editors, October 1992.

20. Bruce W. Weide and Joseph E. Hollingsworth. "On Local Certifiability of Software Components". Technical Report OSU-CISRC-1/94-TR04, Dept. of Computer Science, Ohio State University, Columbus, OH, January 1994.