Automated Monitoring of Component Integrity in Distributed Object Systems

David S. Rosenblum

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
dsr@ics.uci.edu
http://www.ics.uci.edu/~dsr/

1 Introduction

The marriage of object-oriented component programming languages such as C++ and Java with distributed object infrastructures such as CORBA has made the dream of component-based software development and assembly a reality. Yet while distributed object computing may ease the *design* and *construction* of software systems, it also introduces significant challenges in ensuring the *integrity* of component collaborations and the overall reliability of system execution. "Integrity" here refers not only to the functional correctness of a distributed object system and its constituent components, but to other attributes as well, such as quality of service and system security. Given the immaturity of distributed object computing, the full scope of this problem has yet to be encountered in development practice or studied by researchers.

In comparison with "traditional" software systems, distributed object systems have several characteristics that make them ill-suited to the use of traditional methods of system validation:

- *Heterogeneity:* The component objects of a distributed object system can execute on different hardware and software platforms and can be developed by multiple, possibly competing vendors. In contrast, traditional software systems typically execute on a single hardware and software platform (although a system may be portable to different platforms) and are developed by one organization or group of departments that has total control over the whole system.
- *Replication:* There may be multiple (heterogeneous) server objects implementing the same interface, providing a dynamic choice as to how to fulfill a request from a client object. This choice could take into account such factors as server security and reliability, network latency, and so on. In contrast, the use of replication in traditional software systems is typically limited to off-line backup components that are brought on-line when the primary copy of the component (or its execution environment) fails.
- *Dynamic evolution:* A distributed object system can be modified and enhanced by replacing, removing and/or adding component objects without disturbing other objects that are executing. In contrast, traditional software systems are typically modified and enhanced by creating a complete new version of the software containing the enhancements. The old version is taken off-line, and then the new version is brought on-line (although in limited cases it is possible to substitute portions of *object code* in a running system).

These characteristics of distributed object systems produce a multitude of technical challenges, for they radically limit the means by which distributed object systems can be validated for correctness and reliability—both at the component level and at the application level—prior to deployment.

The numerous methods that have been defined for verifying and validating software systems can be grouped into two broad classes—*static analysis methods* and *dynamic analysis methods.* The former class of methods attempts to discern system properties through symbolic analysis of the text of a system specification or source code, while the latter class of methods attempts to discover system properties through execution or simulation of the system specification or source code. Given the highly dynamic and evolving nature of distributed object systems as described above, opportunities for static analysis of a distributed object system will be highly limited, since in the most extreme instances it may not be possible to identify "the model" of the system to be validated. Therefore, greater reliance must be placed on dynamic analysis methods for validating the widest range of applications that will be encountered in practice.

The most commonly used dynamic validation method is testing. While traditional methods of unit testing can be applied to individual components prior to deployment, it is well known that unit testing alone is not sufficient for fully exploring application-level behavior, especially for integration testing and system testing of object-oriented programs [4,9]. Yet the fact that a distributed object system can comprise components developed by many different organizations means that no one organization will have complete control over the system, or complete access to its development artifacts, for purposes of integration testing or system testing prior to deployment. Indeed, the whole notion of a "system" as a static, organic, self-contained entity has little meaning in the distributed object domain, where collaborations between components can be established and disestablished seamlessly, transparently and dynamically without disturbing other components. Hence, other kinds of dynamic analysis must be employed to supplement the unit testing of individual components.

2 Permanent Dynamic Integrity Monitoring

To overcome the lack of applicability of static analysis and the limited opportunity for pre-deployment testing in distributed object systems, a *permanent*, *automated*, *embedded*, *dynamic* integrity monitoring capability is needed for ensuring the integrity of post-deployment component collaborations in a distributed object system. Such a capability must be customized to the semantics of the components that are assembled to create an application, and it must transcend traditional validation activities and development processes in the software lifecycle. A number of assertion-oriented technologies have been developed for embedded semantic-based monitoring of traditional software systems, such as the annotation languages ANNA and TSL and their support tools [2,7,8], the assertion processing tool APP [6], and technologies proposed specifically for object oriented languages [1,3,5]. These technologies can be adapted for use in distributed object computing, but a number of additional capabilities will be needed. In particular, supported for embedded semantic-based integrity monitoring will be needed at three levels in distributed object systems:

1. At the *component level*, language constructs are needed for annotating the interface specifications and source code of components with checkable semantic specifications.

2. At the *component services level*, integrity monitoring services are needed that can be mixed in easily with other component functionality.

3. At the *infrastructure level*, specialized support is needed to enable powerful and efficient integrity monitoring throughout a network of collaborating components.

These ideas, and the technical challenges underlying them, can be made more concrete by considering how integrity monitoring could be implemented for CORBA-based systems.

At the component language level of a CORBA-based system, class invariants and pre- and post-conditions on method interfaces are the minimum that are needed for specifying functional aspects of individual component behaviors. Such specifications can be written easily as annotations of IDL interface specifications. However, the definition of the mapping between IDL-level annotations and dynamic checks embedded in component implementations may be non-trivial.

At the component services level, new common object services could be defined in such a way that would allow them to be inherited by components needing an embedded integrity monitoring capability. However, it is not obvious how to define these services in a generic way that allows them to be tailored or parameterized for checking specific predicate logic-like semantic specifications.

At the infrastructure level, the CORBA architecture could be exploited in a variety of ways to enable powerful and efficient integrity monitoring. For instance, the availability of annotated interface specifications located in IDL interface repositories means that the pre-condition of a method could be looked up and checked locally at the sites of the method's clients prior to each invocation of the method. Such a scheme would save network bandwidth and avoid network latency in situations where the client does not satisfy the pre-condition on the method. The availability of multiple implementations of an object interface means that a method invocation that violates the post-condition of the method could trigger a retry of the method invocation on some other component implementing the interface. Furthermore, the unreliability of the first implementation could be noted in the interface repository for future use in directing method invocation requests. However, such enhancements to the basic model of method invocation may require (as yet) non-standard infrastructure facilities such as before/after method triggers in metaclasses, or extensions to ORB communication protocols.

3 Conclusion

These concepts barely scratch the surface of the kinds of automated, dynamic integrity monitoring that will be needed in large-scale distributed object systems. The author has begun a research project studying these and other issues related to validation and integrity monitoring of distributed component-based software systems. A significant challenge for this work will be to find ways of supporting an integrity monitoring capability in a way that minimizes the need for non-standard versions of interface and implementation languages and infrastructure features and protocols. The author is greatly interested in discussing these issues with the other workshop attendees.

References

[1] M.P. Cline and D. Lea, "Using Annotated C++", *Proceedings of C++ at Work*, 1990.

[2] D.C. Luckham and F.W.v. Henke, "An Overview of Anna, a Specification Language for Ada", *IEEE Software*, vol. 2, no. 2, pp. 9–23, 1985.

[3] B. Meyer, *Object-Oriented Software Construction*: Prentice Hall, 1988.

[4] D.E. Perry and G.E. Kaiser, "Adequate Testing and Object-Oriented Programming", *Journal of Object-Oriented Programming*, vol. 2, no. 5, pp. 13–19, 1990.

[5] S. Porat and P. Fertig, "Class Assertions in C++", *Journal of Object-Oriented Programming*, vol. 8, no. 2, pp. 30–37, 1995.

[6] D.S. Rosenblum, "A Practical Approach to Programming with Assertions", *IEEE Transactions on Software Engineering*, vol. 21, no. 1, pp. 19–31, 1995.

[7] D.S. Rosenblum, "Specifying Concurrent Systems with TSL", *IEEE Software*, vol. 8, no. 3, pp. 52–61, 1991.

[8] S. Sankar, D.S. Rosenblum, and R.B. Neff, "An Implementation of Anna", *Proceedings of Ada in Use: The Ada International Conference*, 1985.

[9] E.J. Weyuker, "Axiomatizing Software Test Data Adequacy", *IEEE Transactions on Software Engineering*, vol. SE-12, no. 12, pp. 1128–1138, 1986.