References

[1] L. Chen and A. Avizienis. N-Version programming: A fault-tolerance approach to reliability of software operation. In *Eighth Annual International Conference on Fault-Tolerant Computing*, pages 3–9, Toulouse, France, June 1978.

[2] S. Conte, H. Dunsmore, and V. Shen. *Software Engineering Metrics and Models.* Benjamin/Cummings Publishing, Menlo Park: CA, 1986.

[3] B. J. Cox. *Object-oriented Programming : An Evolutionary Approach.* Addison-Wesley, 1986.

[4] B. Daniels, editor. *Achieving Safety and Reliability with Computers.* Elsevier, Essex, England, 1987.

[5] S. P. Fiedler. Object-oriented unit testing. *Hewlett-Packard Journal*, 40:69–74, April 1989.

[6] B. Hailpern and H. Ossher. Extending objects to support multiple interfaces and access control. Technical Report RC 14016, IBM Research Division, February 1990.

[7] W. Hseush and G. E. Kaiser. Modeling concurrency in parallel debugging. In *2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 11–20, Seattle WA, March 1990. Special issue of SIGPLAN Notices, 25(3), March 1990.

[8] Y. S. Maarek and G. E. Kaiser. Change management for very large software systems. In *7th Annual International Phoenix Conference on Computers and Communications*, pages 280–285, Scottsdale AZ, March 1988.

[9] N. H. Minsky. Controlling the evolution of large scale software systems. In *Conference on Software Maintenance-1985*, pages 50–58. IEEE Computer Society Press, November 1985.

[10] N. H. Minsky and D. Rozenshtein. A software development environment for law-governed systems. In P. Henderson, editor, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 65–75, Boston MA, November 1988. ACM Press. Special issue of SIGPLAN Notices, 24(2), February 1989 and of Software Engineering Notes, 13(5), November 1988.

[11] D. E. Perry and G. E. Kaiser. Adequate testing and object-oriented programming. *Journal of Object-Oriented Programming*, 2(5):13–19, Jan/Feb 1990.

[12] F. Redmill, editor. *Dependability of Critical Computer Systems 2.* Elsevier, Essex, England, 1989.

[13] M. Shapiro, P. Gautron, and L. Mosseri. Persistence and migration for C++ objects. In S. Cook, editor, *3rd European Conference on Object-Oriented Programming*, British Computer Society Workshop Series, pages 191–204, Cambridge, UK, July 1989. Cambridge Univ. Pr.

[14] A. Skarra and S. Zdonik. The management of changing types in an object-oriented database. In N. Meyrowitz, editor, *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 483–495, Portland OR, September 1986. ACM. Special issue of SIGPLAN Notices, 21(11), November 1986.

[15] E. J. Weyeker. The evaluation of program-based software test data adequacy criteria. *CACM*, 31(6):668–675, June 1988.

[16] J. M. Wing and M. P. Herlihy. The Avalon/C++ programming language. Technical Report CMU-CS-88-209, Carnegie Mellon University, December 1988.

- "This method fails in testing object-oriented languages that depend on implicit method-calling, such as Avalon/C++'s operation overloading [16]."

  The objection confuses complexity from the point of view of the programmer, and that of the compiler (or tester). The test instrumentation process should be seen as a variation on the compiler; it will be able to "see" anything the compiler can. So the objection is not true, although the complexity of the testing process will unavoidably rise with the complexity of the software under test.

- "This method adds an intolerable level of complexity to the process under test, greatly altering its behavior. The probe effect of the object duplication is unacceptably high."

  This is a serious objection that cannot be answered in this space. At first look, the added costs of testing are a function primarily of the costs of invoking an object's methods. In the worst case, we would only triple this cost.[4] If the cost of method calling *per se* is a large part of the total costs, then the probe effect would be unacceptably large. However, apart from distributed object-oriented languages, we are unfamiliar with any language that has such a disproportionately high cost for method calling.

4 Related Work

The idea of substituting rewritten classes for particular purposes appears in other works, primarily for distributed services or software evolution. Shapiro uses *proxies* as place-holders to obtain distributed services [13]. Skarra and Zdonik have studied type evolution in object-oriented database systems, establishing a system where different versions of the same objects interact compatibly with other objects [14]. Minsky and Rozenshtein have also studied the evolution of object-oriented software [9, 10]. Finally, Hailpern and Ossher implement objects with multiple interfaces with their *views* mechanism [6].

More specifically regarding the testing of object-oriented systems, Perry and Kaiser, following Weyeker, showed that adequate program-based testing of object-oriented languages is precisely as difficult as testing non-object-oriented languages [15, 11]. Note that this is strongly counterintuitive to common assumptions about the advantages of reusable software objects. Fiedler has done preliminary work in testing an application beginning with the classes it uses[5].

5 Conclusions and Future Work

Clearly, the method we outline is not limited to object-oriented testing, because it exploits the encapsulation that is present in any system designed with abstract data types. In the Pughes-Heedless example, the two module interfaces might have been tested similarly in non-object-oriented languages. However, a language-enforced requirement is always a stronger basis for assumptions than an unenforceable programming discipline. A mutual agreement between two programmers to bypass an interface would invalidate testing based on it.

This method is best suited to object-oriented testing because, by definition, object-oriented programs are organized into encapsulated entities, and every access to an object's data may be monitored. In addition, the fine-grained side of object-oriented programs is usefully exploited, since individual versions or instances of objects may be closely examined in testing.

Much remains to be explored from this preliminary basis. In particular, we are concerned about interactions with inheritance, and the degree to which a program's behavior will be altered by this testing, especially with distributed objects.

[4] For example, if all method invocations are a single message, then invoking a method on **Nav_Location** will cost one extra message for logging, and one extra message for the invocation of the same method on the hidden object $\mathcal{E}$.

specifications, an object's interdependency coupling with another object [8], the criticality of code,[2] or the complexity of the code according to some metric [2, 5]. There are a range of fine- and coarse-grain choices regarding objects, but individual objects under test could also report according to local or global criteria, e.g., "Engine_Temp > 150°".

2.3 Information as gathered and used

In the example of Section 1, the **Testing_Logger** object was given as a central data collection point. The exact role of this object will vary with the type of testing. Here, it does not matter which company's staff writes the public logging object, because it passively records information sent to it from the other objects, which are rewritten by each company. In other words, the burden of responsibility for keeping information private lies on each affected company. Any information relating to proprietary information would be sent to private logging objects.

The logged information may be no more than a note that a certain method was invoked. It may also contain the parameters, tests performed on these parameters, results, comparison with expected results, time of invocation, object version numbers, and other relevant information. The logged information may be used as needed, either online or offline. The collected information may be used to verify that certain code was executed with expected results; to make more involved deductions about the tested objects, e.g., the presence of race conditions [7]; or even to institute N-way voting among objects [1]. These examples help emphasize that this testing method is orthogonal to other concerns. For example, different object instances may be instrumented to use N-way voting, but the testing method itself cannot insure the atomicity of the different object's actions.[3] (Testing with N-way voting is a slightly different interpretation of "mutually suspicious parties.")

Finally, no particular role is presumed here for the **Testing_Logger** object, nor should it presumed that there is only one such object. The primary purpose of this instrumentation method is to gather information efficiently without altering the original design of the software under test. Each individual tester may want his or her code instrumented to report to a private logging object, for reasons of efficiency, security, or practicality.

3 Initial Criticisms

In this section, we respond to several *prima facie* objections to this method.

- "This method will not work where there are unredefinable objects. For example, in Objective C, integers are not first-class objects [3]."

  This objection appears to be valid. However, such omissions in a language are likely to be made for reasons of efficiency, and therefore limited in scope. With Objective C, a tester would assume the correctness of operations involving integers, or test them in some other manner.

- "This method fails with systems that can broadcast to all objects. For example, if I can tell every object **garbage_collect yourself** (without using a hierarchy of objects), I cannot hide an object by renaming its class."

  This objection is valid, although such languages have not been studied widely. Because of the absence of encapsulation, such a feature should be considered "illegal" in an object-oriented language, according to some researchers [10].

[2] That is, test the "core" behavior before any "features." Compilers should always compile, if not optimize or produce listings correctly; text formatters always should produce some output, if not form ligatures correctly. Obviously, testing safety-critical code is a problem of a different order. See for example, [4, 12]. We are primarily concerned with approximating correct behavior, not ensuring safe behavior.

[3] In the example of Section 1, imagine that the three objects $\mathcal{C}$, $\mathcal{D}$, and $\mathcal{E}$ are actually three different implementations of the same specification, and the new objects communicate among themselves to deduce hardware or software failure from the direct comparison of results (or their checksums). While this technique is traditionally used for establishing fault-tolerance, it could also be used during testing to detect failures.

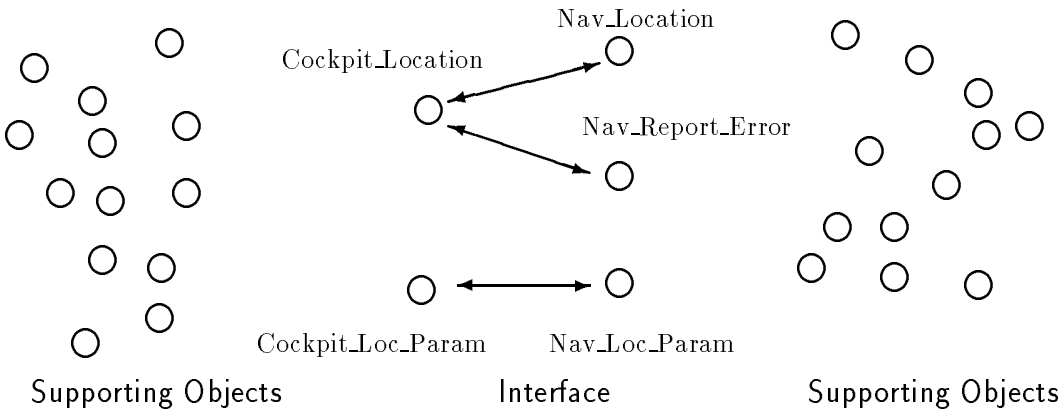

Figure 1: Fragment of Original Design for Pughes-Heedless

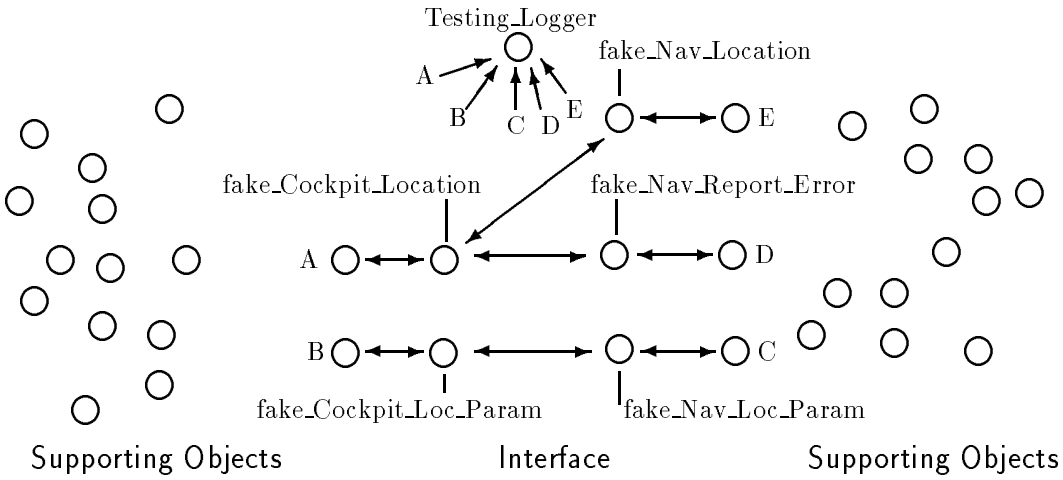

Figure 2: After Instrumentation of Pughes-Heedless example for Testing

2 Testing Programs at the Object Level

We have outlined a method of instrumenting object-oriented programs so that their individual actions are logged efficiently and without affecting the original purpose or design of the software under test, even under rigid conditions of security. The original behavior is altered to some degree (see Section 3). In this section, implications of this technique are discussed in more detail.

2.1 Objects as used in this work

This method depends primarily on the encapsulation properties of object-oriented languages. It only requires that objects be entities with a private collection of data and methods, and that the class of any object in the system under test can be rewritten and duplicated. At this point, there does not seem to be a need to specify details of inheritance, polymorphism, binding time, synchronous/asynchronous method invocation, parallel or distributed objects, or parameterized classes. However, further work is needed in this area.

2.2 Selection of Objects for Testing

Testing at the object level allows a great range of specificity. We may test a single class of objects, all objects, or even a given instance of an object. Depending on the details of the support tools available, we may choose objects by name, by version, by programmer, by path through the code, by module, or for other low-level reasons. We may also choose to test objects on higher-level considerations, such as program

Testing Object-Oriented Programs by Mutually Suspicious Parties

Travis Lee Winfrey*
Columbia University Department of Computer Science
New York City, New York, 10027.
(212) 854-2930
Fax: (212) 666-0140
travis@cs.columbia.edu

1 August 1990

1 Introduction

We motivate our brief treatment of this topic with an example. Let us say that Pughes Aircraft and Heedless Corporation have contracted together: Heedless will supply Pughes with a navigational device for a new aircraft. However, the physical hardware and software for all mechanisms are considered proprietary by Pughes and Heedless, and are closely guarded by each company. Physical barriers may be placed on the devices themselves. However, software developers working at both locations must test the software in simulation. How may testing proceed without examining the proprietary source code?

A simple technique of rewriting and renaming the object classes solves this problem. Each company's software team will only instrument its own software for testing. Proprietary information will remain so. Since the interface will not change, the two components will communicate as previously designed.

Say that Pughes has two relevant objects, **Cockpit_Location and Cockpit_Loc_Parameters** in its cockpit simulator. Heedless has three such objects, **Nav_Location, Nav_Loc_Parameters,** and **Nav_Report_Error** in its device simulator

To test this system, the programmers rename all five objects with unique identifiers, $\mathcal{A}, \mathcal{B}, ..., \mathcal{F}$. They also create five new objects duplicating **Cockpit_Location**, etc. (The methods of the individual objects are not renamed.) These new objects will report to a universally-visible object, which we will call here **Testing_Logger**.[1] For clarity, we will call the duplicate objects **fake_Cockpit_Location**, etc., although it should be understood that in practice they will have the identical names as the objects they replace.

In normal use, **Cockpit_Location** would invoke a method of **Nav_Location**. In the instrumented code, **Cockpit_Location** invokes a method of **fake_Nav_Location**, which will in turn log the desired information to the **Testing_Logger** before passing on the identical arguments to the hidden object, renamed to $\mathcal{C}$. We do not specify here the type of information logged to **Testing_Logger**; Section 2.3 details the possibilities.

In Figure 1, we see a fragment of the original design; Figure 2 shows the modifications after instrumentation for testing. Arrows show the direction of the monitored communications between objects. Where necessary, lines connect names with objects.

*Research in Programming Systems is supported by National Science Foundation grants CDA-8920080, CCR-8858029 and CCR-8802741, by grants from AT&T, BNR, Citicorp, DEC, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research. Copyright © 1990, Travis Lee Winfrey. Tech Report CUCS-041-9.

[1] **Testing_Logger** may need to be duplicated if proprietary and non-proprietary information will be sent to it. See Section 2.3.