INFUSE Test Management

Gail E. Kaiser
Columbia University
Department of Computer Science
New York, NY 10027

Dewayne E. Perry
AT&T Bell Laboratories
Computer Systems Research Lab
Murray Hill, NJ 07974

June 1988
(revised December 1988)

CUCS-350-88

Abstract

This technical report consists of the two papers discussing testing technology. *INFUSE: Integration Testing with Crowd Control* describes the test management facilities provided by the INFUSE change management system. INFUSE partially automates the construction of test harnesses and regression test suites at each level of the integration hierarchy from components available from lower levels. *Adequate Testing and Object-Oriented Programming* applies the axions of adequate testing to object-oriented programming languages and examines their implications. Contrary to our original expectations, we discover that in the general case classes must be retested in every context of reuse.

Prof. Kaiser is supported in part by grants from AT&T, IBM, Siemens and Sun, in part by the Center of Advanced Technology and by the Center for Telecommunications Research, and in part by a DEC Faculty Award.

INFUSE: Integration Testing with Crowd Control

Gail E. Kaiser
Columbia University
Department of Computer Science
New York, NY 10027

Dewayne E. Perry
AT&T Bell Laboratories
Computer Systems Research Lab
Murray Hill, NJ 07974

27 January 1988

Abstract

INFUSE is a change management system for large scale software projects. In previous papers, we described its core philosophy of integrating strongly connected modules first and more weakly connected sets of modules later, moving up a hierarchy from singletons to clusters of interdependent modules to merging the change set into the baseline. We have previously applied INFUSE to static consistency analysis of syntactic and semantic properties. In this paper, we extend our work to dynamic consistency analysis, i.e., testing. Unit testing is done for the individual modules at the leaves of the hierarchy, integration testing for the intermediate clusters and acceptance testing at the root. INFUSE supports this by partially automating the construction of test harnesses and regression test suites at each level of the hierarchy from components available from lower levels.

Kaiser is supported in part by grants from AT&T Foundation, IBM, Siemens Research and Technology Laboratories, and New York State Center of Advanced Technology — Computer and Information Systems, and in part by a Digital Equipment Corporation Faculty Award.

keywords: change management, integration, programming in the many, regression testing, software development environment, test management

1. Introduction

INFUSE is a 'city model' software development environment [21], that is, it addresses the special needs of large scale software projects, where the scale is in terms of programming-in-the-many as well as programming-in-the-large. We believe that some seemingly small number of programmers (say, 20) is effectively a 'crowd'. Crowd control inherently makes change management so complex that technological in addition to managerial mechanisms are required to handle the interactions among the programmers. In previous papers [19, 11], we have presented our philosophy and the basic mechanisms for isolating groups of modules[1] into a hierarchy of experimental databases. The goal is to minimize the implications and extent of changes that the programmers must cope with at one time. This paper extends our previous work to support integration testing. INFUSE now supports semi-automatic construction of test harnesses and aids selection of regression test suites, both at each level of the hierarchy.

We propose that any 'city model' change management system should assist the project team with:

- partitioning the changed modules into a hierarchy of sets, for the purpose of isolating groups of modules during integration;
- the time sequence of the integration, that is, the desired ordering of integration with respect to the hierarchy — bottom-up, top-down or sandwich;
- syntactic and/or semantic consistency checking within a set;
- construction of test harnesses (drivers and stubs);
- regression testing for each set of modules at each level of the hierarchy; and
- test management, to keep track of which tests have been passed by which sets of modules.

INFUSE provides all these facilities.

The core of INFUSE is a change management framework for automatically constructing and maintaining a hierarchy of *experimental databases* (EDBs), where each EDB contains a subset of the change set. At each level of the hierarchy, the contents of the EDBs are disjoint. The notion of an EDB was initially introduced in Smile [12], a multiple-user programming environment for C developed as part of the Gandalf project [7]. INFUSE extends this notion to (1) a hierarchy, (2) automatic partitioning of the change set into EDBs, and (3) integration

---

[1] A *module* is any separately compilable syntactic unit, such as an Ada™ package, a Modula-2 module or a C source file.

testing.

We subscribe to the rarely supported ideal of hierarchical integration of large scale software systems. The rationale for this is the widely accepted software engineering rule-of-thumb that interface errors detected early are much less costly to repair than errors detected late [3]. There are two well-known mechanisms for structuring the modules of a system into a hierarchy: managerial and design. The most significant innovation of INFUSE is a new kind of hierarchy, *dependency-order*, where strongly interconnected modules are placed together near the bottom of the hierarchy and more weakly connected modules are placed together closer to the top.

INFUSE generates the hierarchy by applying a clustering algorithm [14] to the change set. Our algorithm uses a non-Euclidean similarity metric based on the dependencies between pairs of modules. The similarity metric between two sets of modules $\{ M_1, ..., M_n \}$ and $\{ M_{n+1}, ..., M_p \}$ is defined by any one of several statistical measures applied to the basic metric. A similarity metric based on dependencies is an approximation to the oracle that would tell us, in advance, exactly how the interfaces of modules will be changed and how this will affect other modules. The intuition is that changes will more likely involve strongly connected modules than weakly connected modules, according to a simplistic proportionality argument.

There are three categories of dependency models [20] that may be employed — unit, syntactic and semantic — as well as several subclasses of these models. For example, the strength of syntactic interdependency between a pair of modules M and N is k, where k is the sum of i, the number of facilities exported by M and imported by N, and j, the number of facilities exported by N and imported by M. This can be refined, as in Tichy's smart recompilation [28], to consider only the external facilities actually used by an importing module. INFUSE does not presume any particular dependency model, but can use any for which a corresponding analysis tool is available.

INFUSE assumes several external facilities:

- programming-in-the-small tools: editors, compilers, linker/loaders, debuggers, formatters, etc.;
- version management of source and object code, such as RCS [27] or Arcadia [26];
- system modelling and configuration management, such as Make [4] or Apollo's Domain Software Engineering Environment [13] (DSEE™);
- a modification request (MR) system [24];

- a consistency analysis tool, either syntactic such as Lint [10], or semantic, such as Inscape [18];
- a program-based and/or a specification-based test coverage analyzer — both kinds of analyzer and their relationship with INFUSE are explained later on.

Although not required for operation, INFUSE can be augmented by an automatic software test driver [17] that provides a standard setup for executing tests and automatic verification of test results. We intentionally leave vague the notation for describing the tests and their required results. We assume for this paper that INFUSE keeps track of which tests are in which test suites, and which of these have been passed, but a human is actually responsible for executing the tests.

In the next section, we give an overview of INFUSE and its test management facilities. The following two sections discuss construction of test harnesses and regression test suites, respectively. We conclude by summarizing the contributions of this paper.

2. Overview of INFUSE

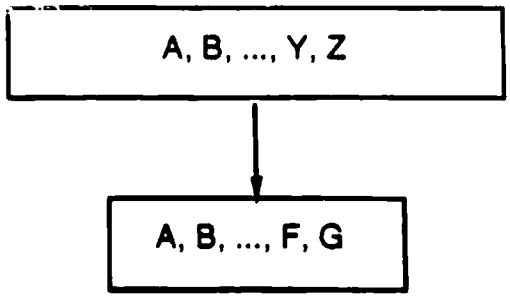

Figure 2-1: Baseline and Change Set

The INFUSE change management system operates as follows for a scheduled change, such as for a new release or a patch to a previous release. The change set is selected manually by a system analyst or automatically by an MR system. INFUSE checks out new revisions of the modules in this set from the version control system as shown in figure 2-1, extracts their dependency matrix and invokes the clustering algorithm to determine a hierarchy according to the strengths of interconnections. The group of modules assigned to the same programmer is treated as a single module for the purposes of clustering. INFUSE then builds the hierarchy of EDBs containing the new revisions of the appropriate modules, as shown in figure 2-2.

Programmers work on their assigned module(s) in the EDBs at the leaves of the hierarchy. For simplicity, we assume a leaf EDB consists of a single module M and the programmer is responsible only for this individual module; thus, we refer to leaf EDBs as *singletons*.

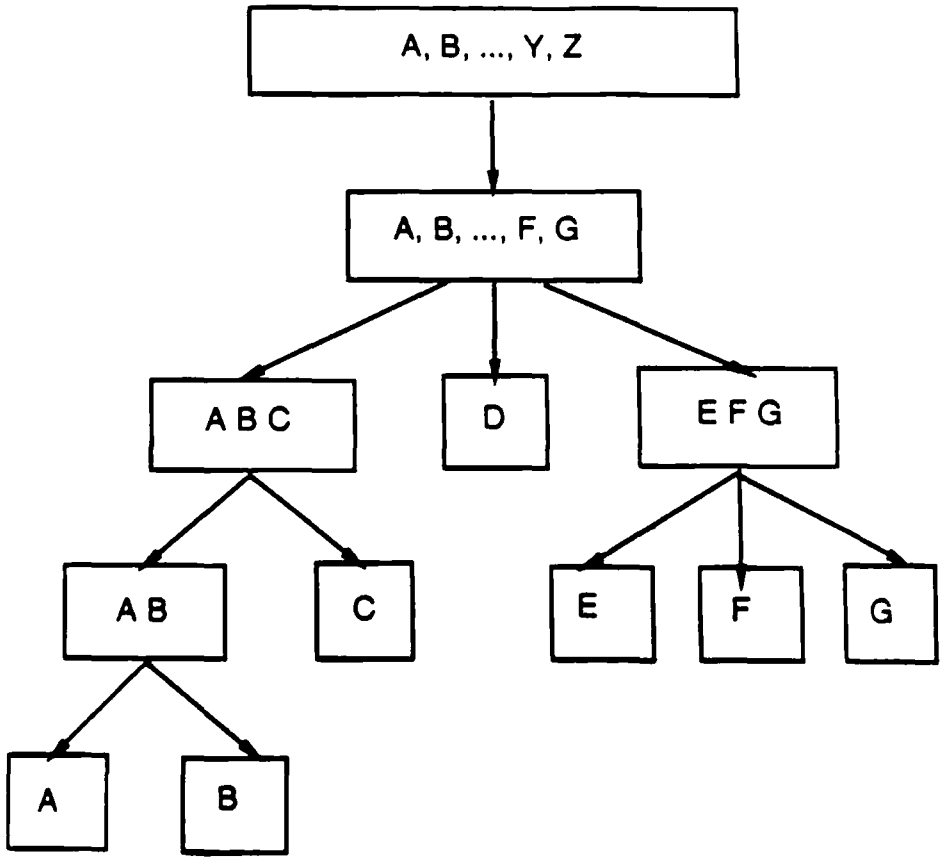

Figure 2-2: Hierarchy of Experimental Databases

When a programmer finishes editing M, he requests the consistency analysis tool, which determines whether or not M is self-consistent. Syntactic consistency requires that every identifier defined in M is used within M in the manner prescribed by the static semantics (i.e., context-sensitive syntax) of the programming language. Each use of an identifier defined externally (i.e., not defined in M) must be consistent with all other uses of the same identifier. In the case of semantic consistency, every identifier must be used correctly with respect to the semantic specification mechanism employed. For simplicity, we assume syntactic consistency analysis throughout the rest of this paper.

Once his module M is consistent, the programmer builds a test harness. The harness consists of a driver D that invokes the module to perform the tests and a set of stubs S that perform, in an abstract sense, the functionality of those external modules referenced by M. In particular, stub $S_{M,N}$ represents all the subroutines and data defined by module N and used in module M. INFUSE compiles and links M together with D and S, and then the programmer proceeds with testing and debugging.

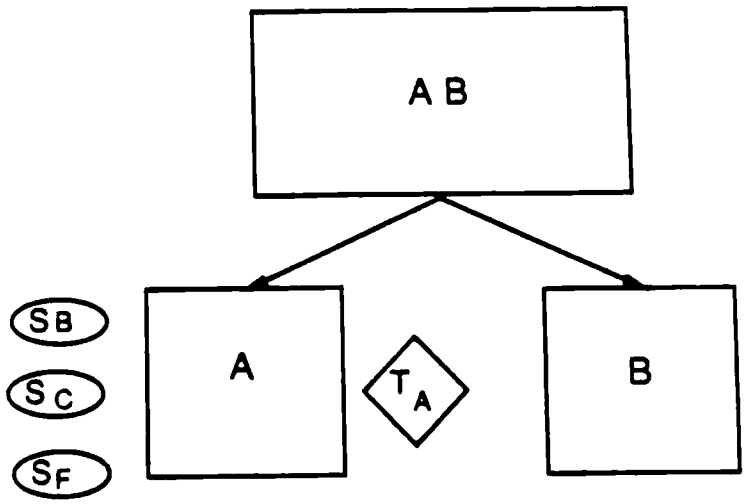

Figure 2-3: Unit Testing Stubs and Test Suite

The programmer devises a set of unit tests that together meet some test data adequacy criteria [29], perhaps with the aid of an adaptive test generation tool [22]. The stubs and unit test suite are associated with a singleton EDB as shown in figure 2-3. After M has passed all these tests, the programmer enters a command to *deposit* it into the parent EDB. Deposit makes the new versions of the modules in the child EDB visible to the other modules in the parent EDB. Before allowing the deposit, INFUSE requires that M is in fact self-consistent in the static sense of the analysis tool and in the dynamic sense of the unit tests. INFUSE associates with M some representation of T, the set of unit tests with their required results, along with D and S.

At some point, all the sibling EDBs have been deposited into their parent EDB, which then contains several modules — typically 2 to 5 — that are very strongly interdependent. INFUSE invokes the static analysis tool to check that these modules are consistent among themselves. If not, it informs the responsible programmers, who negotiate among themselves, agree on further

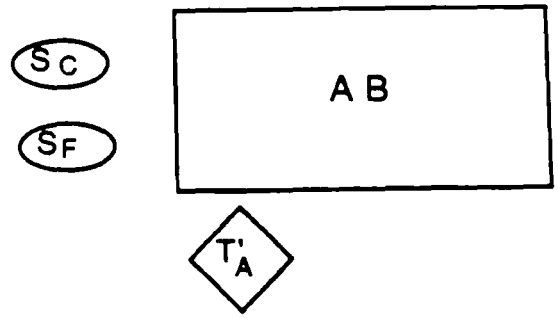

Figure 2-4: Automatically Selected Stubs and Tests

changes, and notify INFUSE of the modules that must again be changed. INFUSE generates singleton EDBs for these modules and the singleton process repeats as necessary.

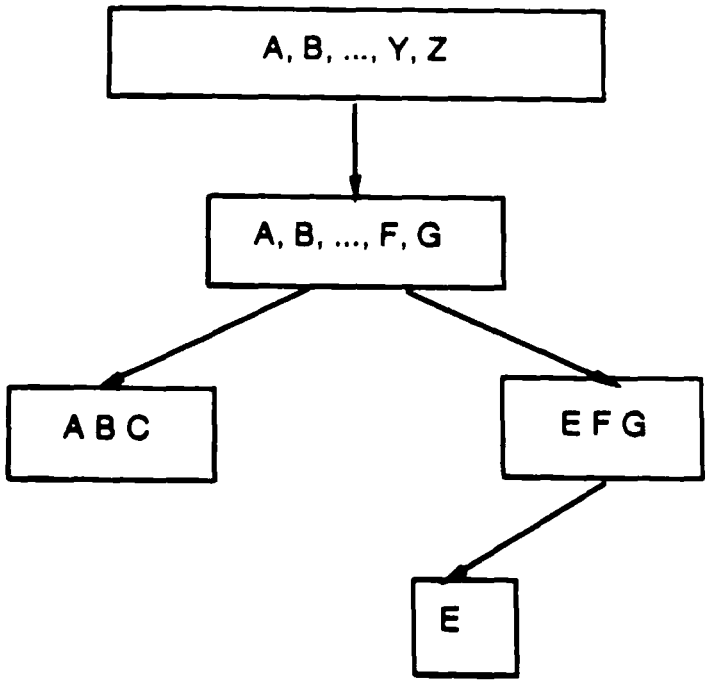

Figure 2-5: Hierarchy After Several Deposits

If the modules are consistent to the extent that can be determined by a static analysis tool, INFUSE constructs a set of stubs S for integration testing from the sets S_M available from unit testing. Usually some programmer must build the new test driver. INFUSE assists the

programmers in selecting the regression test suite T from the unit test suites T_M. The resulting stubs and tests are illustrated in figure 2-4. Then the tests are executed and debugging proceeds. If no errors are detected, the current EDB can be deposited into its parent, and so on, and the hierarchy condenses as shown in figure 2-5. If errors are detected, the programmers negotiate and select a subset of the EDB for further modification. INFUSE locally repartitions this subset into singleton databases, as is done for inconsistencies detected by the analysis tool. A possible result is shown in figure 2-6. After the subset has been modified and redeposited, INFUSE constructs a new regression test suite in the same manner as it constructed the original, failed suite for this EDB.

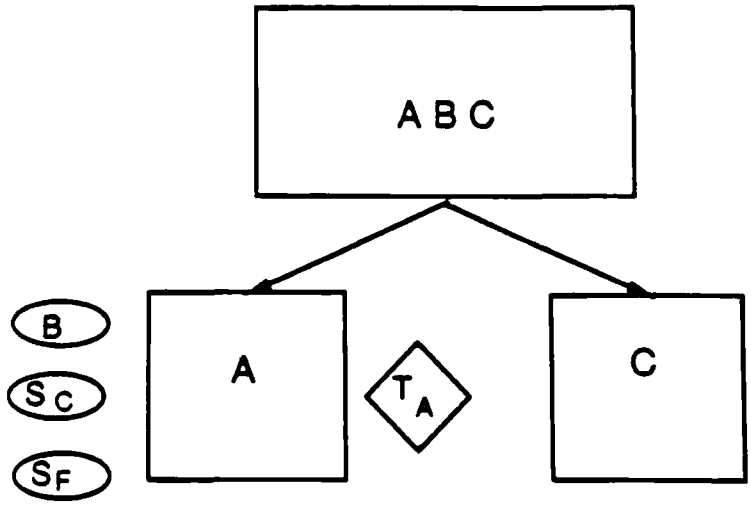

Figure 2-6: Repartitioning for Further Changes

Once all the regression tests have been passed, a programmer can issue the deposit command to move the integrated EDB into its parent. When all the siblings have also been deposited, this process is repeated at each level using as components the modules, stubs and test suites of the previous level in the hierarchy. Any inconsistencies result in repartitioning the subtree below the EDB where the inconsistency was discovered.

At the top-level of the hierarchy, three tasks must be performed. First, the entire change set must be integrated by this mechanism. Then it must be integrated with the unchanged modules in the baseline version of the program. This stage is illustrated in figure 2-7. Finally, after acceptance testing, INFUSE checks in the modules in the top-level database to the version control

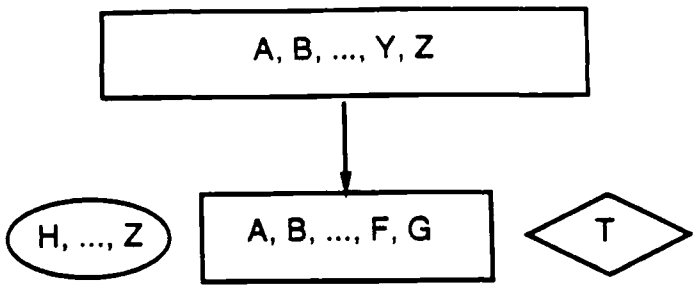

Figure 2-7: Acceptance Testing

system.

One limitation of INFUSE is that it does not provide any support, beyond a standard debugger, for isolating and repairing errors detected by unit, regression and acceptance testing. Several advanced debugging tools have been proposed [25], but none have yet been applied in the context of crowd control. We are in the early stages of applying an existing machine learning algorithm, which integrates explanation-based and similarity-based learning [2], to the problem of fixing bugs 'similar' to previously fixed bugs.

3. Test Harnesses

INFUSE aids the programmers in constructing both the stubs and drivers of test harnesses. First we explain how, for each EDB, INFUSE considers the collection of stubs associated with all its children databases and determines which stubs are replaced by modules, which stubs can continue to be used, and which potentially conflict with other stubs. At the end of this section we sketch how INFUSE determines when a driver from a descendant database can be used in an ancestor database.

3.1. Stubs

Consider an EDB, E, containing the set of modules { M_1, ..., M_k }. E is the parent of a set of child EDBs, each of which contains a disjoint non-null subset of these modules. Each M_i has a set of stubs S_i from its singleton EDB. INFUSE operates according to the algorithm shown in figure 3-1 to construct the set of stubs for E.

INFUSE examines each stub $S_{M,N}$, constructed to represent absent module N as used by module

Let S_E be the set of stubs associated with experimental database E,
M, N and O be modules,
S_M be the set of stubs associated with module M in the current EDB, and
$\bar{S}_{M,N}$ be the stub that represents module N as used by module M in the child EDB containing M
In

$S_E \leftarrow \emptyset$
$\forall M \in E$ *do*
    $S_M \leftarrow \cup_N \{ \bar{S}_{M,N} \}$
    $\forall O \in E$ *st* $O \neq M$ *do*
        if O is complete wrt M then
            $S_M \leftarrow S_M - S_{M,O}$
    $\forall S_{M,N} \in S_M$ *do*
        if $\{ O \in E \mid O \neq M \wedge S_{O,N} \in S_E \} \neq \emptyset$ *then*
            ask user whether to keep, replace or merge
            and modify S_M *accordingly*
    $S_E \leftarrow S_E \cup S_M$

Figure 3-1: Algorithm for Stub Selection

M (i.e., one of the M_i) in one of the child EDBs. All such stubs where N is present in the current EDB (that is, N is one of the M_i distinct from M) are replaced by N, as the first part of the integration. The idea is to use the real module, now that it is available, rather than a stub. This works only when N is *complete* with respect to M. For example, if the design for N calls for it to export facilities f, g and h, but only facilities f and g are currently implemented, then N is incomplete. If M actually uses only f and g, then N is complete with respect to M; if M actually uses f and h, then N is incomplete with respect to M. These two cases are addressed by other tools, such as PIC [30], where the two modules are called "consistent" and "conditionally consistent", respectively. INFUSE uses its analysis tool to detect cases where N does not provide all the facilities simulated by the corresponding set of stubs. In this case, N is treated as as if it were just as candidate stub available from a child EDB.

Other stubs will also remain, since the corresponding modules will not be integrated until higher levels of the hierarchy. Among these, it is likely that many stubs $S_{x,N}$ will be *duplicate*, that is, there is more than one stub representing N in the context of module O another in the context of P, *etc.* This is represented in $S_{x,N}$ by the lowercase variable x. Note that the duplication of a stub do not imply the duplicates are identical, and in fact the content of these

stubs may be markably different, due to the different requirements placed by the context modules. Thus where there is a duplication, it is rarely acceptable to automatically choose one among the supposed 'equivalence class' of stubs to replace all elements of that class with respect to the coming round of compilation, linking, testing and debugging.

INFUSE does not require this kind of conflict — i.e., duplication — to be resolved. Instead, it brings the problem to the attention of each programmer whose module M uses one of the stubs in a particular equivalence class. The programmer can choose to continue using his own stub $S^{-}_{M,N}$ from the previous level of the hierarchy, begin using one of the other stubs $S^{-}_{x,N}$ from the previous level, or create a new stub $S_{M,N}$ from scratch or by merging (using an standard text editor) the contents of some subset of these stubs. The superscript "-" refers to a stub available from a child EDB. If more than one stub remains in the class after all programmers have made their decision, INFUSE does the necessary internal renaming to ensure the decisions are reflected in the executable image generated by normal compilation and linking.

3.2. Drivers

The set of drivers for an EDB is of course closely tied to its test suite. For each EDB, we can divide the members of the test suite into two classes:

1. tests that originate at this EDB and check the functionality, performance, *etc.* of the corresponding subsystem; and
2. tests that originated at a descendant EDB that are reapplied as regression tests because the integration makes it possible for the results of the tests to be different now than when previously performed at a lower level of the hierarchy.

For tests in the first class, the new set of drivers must usually be constructed by the programmers, perhaps by merging several existing drivers associated with descendant EDBs. For carrying out tests in the second class, however, INFUSE can automatically retain the original drivers.

4. Regression Testing

The preceding discussion of drivers suggests our approach to integration testing, which follows the algorithm shown in figure 4-1. In the worst case, the test suite T_E for an experimental database E is the union of all the test sets from the descendent EDBs, plus the additional subsystem tests added at this point by one or more of the relevant programmers. In the typical case, INFUSE helps reduce the amount of testing, without degrading the opportunities to detect

Let E_i be the ith descendant of experimental database E in some standard ordering such as preorder,
T_E be the test suite for E,
S_i' be the set of stubs, among those associated with the ith descendent, which were replaced in E (using the algorithm given previously),
T_i' be the subset of the test suite, from the test suite associated with the ith descendant, which actually exercised S_i'
In

$T_E \leftarrow$ *new tests for subsystem E*
$\forall M \in E$ *do*
$T_E \leftarrow T_E \cup T_M'$

Figure 4-1: Algorithm for Test Selection

errors. It does this by automatically marking as correctly completed all tests in T_E that it knows could not produce different results in the integrated forum than in the descendant.

INFUSE determines the tests to mark as follows. Any tests applied directly to a module that continues to use (transitively) exactly the same set of stubs as in the relevant descendant EDB is assumed to return the same results for the same inputs. This is of course true only if the stubs guarantee repeatability; INFUSE cannot automatically reduce T_E if the stubs and/or modules involve nondeterminism (e.g., values based on the system clock, concurrency).

We assume S_i' is computed as an extension of the previous algorithm, making this algorithm relatively simple. Note the following implication for the driver (actually a set of drivers) D_E used for testing E.

$$D_E \supseteq \{D_i \mid T_i' \neq \emptyset\}$$

4.1. Program-based versus Specification-based Testing

So far, we have ignored the questions of how the tests are produced and how a test suite is determined to be "adequate" according to some standard. These questions can be answered in two different ways, following the two divergent forms of test case coverage that have been proposed [9], program-based and specification-based. *Program-based* testing implies inspection of the source program and selection of test cases that together cover all possibilities, where the possibilities might be statements, branches, control flow paths or data flow paths. In practice,

some intermediate measure such as essential branch coverage [1] or feasible data flow path coverage [5] is most likely to be used, since the number of possibilities might otherwise be infinite or at least infeasibly large.

In the case of program-based testing, the test suite for each EDB would consist of the new tests for the module(s) introduced by the EDB, plus additional tests to deal with the combinatorics between the paths through these modules and the paths through the modules at the next lower level of the program. The INFUSE notion of dependency-order hierarchy thus fits well with program-based testing, since the massively connected modules are tested early. However, particular program-based testing tools (such as Asset [6]) might require a different ordering.

Unlike program-based testing, *specification-based* ('black-box') testing does not consider the source program. It instead addresses the (functional and non-functional — for instance, performance) specification of the system, and hopefully the specifications of its subsystems and individual modules. The current state of the art permits automatic test case generation and/or test adequacy determination for only a few special cases — for example, mathematical subroutines [23]. In the general case, the best that can be done beyond auditing is to cross-reference tests with portions of the design document [16].

The INFUSE dependency-order hierarchy may not be the best for specification-based testing. There is typically a design hierarchy developed from the specification, where the specification-based tests are associated with the units of this design. Even though the design hierarchy often implies the initial interdependencies among modules, and thus the initial dependency-order hierarchy, the two may not be very similar after a sequence of changes. But INFUSE does not require a dependency-order hierarchy; the clustering component of the system can be replaced with some other mechanism for partitioning the change set. INFUSE still uses the same rules to determine whether or not to apply regression tests at each level of the hierarchy, independent of how the hierarchy is derived.

5. Conclusions

We have previously reported the partitioning of a change set into a hierarchy of experimental databases according to the strengths of interdependencies among modules. We have also described a suitable clustering algorithm and described how to do consistency checking in this context. This paper presents our more recent work on extending INFUSE from compile-time to

execution-time crowd control, that is, to integration testing for large scale software projects. The new contributions of this paper are:

- A formalization of integration testing.
- A framework for integration testing management.
- Semi-automated support for test harness construction.
- Semi-automated support for test suite selection.

There has been much previous research on testing strategies and tools as they relate to programming-in-the-small [31], and some work on integration of subroutines [8]. INFUSE is to our knowledge the only system that applies the results of such research to large scale software systems.

Acknowledgements

Yoelle Maarek developed the clustering algorithm used by INFUSE. Travis Winfrey, Ben Fried and Pierre Nicoli, under the direction of Bulent Yener, completed the implementation of an earlier version of INFUSE with syntactic consistency analysis and without testing support. Peggy Quinn participated in discussions that led to the development of our city model of software development environments. We would like to thank Yoelle Maarek, Michael van Biema and Bulent Yener for their useful comments on an earlier version of this paper.

References

[1] Takeshi Chusho.
Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing.
IEEE Transactions on Software Engineering SE-13(5):509-517, May, 1987.

[2] Andrea Pohoreckyj Danyluk.
The Use of Explanations for Similarity-Based Learning.
In *Tenth International Joint Conference on Artificial Intelligence*, pages 274-276. Milan, Italy, 1987.

[3] Richard Fairley.
Software Engineering Concepts.
McGraw-Hill Book Co., New York, 1985.

[4] S.I. Feldman.
Make — A Program for Maintaining Computer Programs.
Software — Practice & Experience 9(4):255-265, April, 1979.

[5] Phyllis G. Frankel and Elaine J. Weyucker.
Data Flow Testing in the Presence of Unexecutable Paths.
In *Workshop on Software Testing*, pages 4-13. IEEE Computer Society, Banff, Canada, July, 1986.

[6] Phyllis G. Frankl and Elaine J. Weyucker.
A Data Flow Testing Tool.
In *SoftFair II A Second Conference on Software Development Tools, Techniques, and Alternatives*, pages 46-53. San Francisco, CA, December, 1985.

[7] A.N. Habermann and D. Notkin.
Gandalf: Software Development Environments.
IEEE Transactions on Software Engineering SE-12(12):1117-1127, December, 1986.

[8] Allen Haley and Stuart Zweben.
Module Integration Testing.
Computer Program Testing.
North-Holland Publishing Co., New York, 1981.

[9] William E. Howden.
Software Engineering and Technology: Functional Program Testing & Analysis.
McGraw-Hill Book Co., New York, 1987.

[10] S.C. Johnson.
Lint, a C Program Checker.
Unix Programmer's Manual.
AT&T Bell Laboratories, 1978.

[11] Gail E. Kaiser and Dewayne E. Perry.
Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution.
In *Conference on Software Maintenance*, pages 108-114. Austin, TX, September, 1987.

[12] Gail E. Kaiser and Peter H. Feiler.
Intelligent Assistance without Artificial Intelligence.
In *Thirty-Second IEEE Computer Society International Conference*, pages 236-241. San Francisco, CA, February, 1987.

[13] David B. Leblang and Robert P. Chase, Jr.
Computer-Aided Software Engineering in a Distributed Workstation Environment.
In *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 104-112. Pittsburgh, PA, April, 1984.
Proceedings published as *SIGPLAN Notices*, 19(5), May 1984.

[14] Yoelle S. Maarek and Gail E. Kaiser.
Change Management for Very Large Software Systems.
In *Seventh Annual International Phoenix Conference on Computers and Communications*. Scottsdale, AZ, March, 1988.
To appear.

[15] Edward Miller and William E. Howden.
TUTORIAL: Software Testing & Validation Techniques.
IEEE Computer Society Press, Washington, DC, 1981.

[16] Thomas J. Ostrand, Ron Sigal and Elaine J. Weyucker.
Design for a Tool to Manage Specification-Based Testing.
In *Workshop on Software Testing*, pages 41-50. IEEE Computer Society, Banff, Canada, July, 1986.

[17] David J. Panzl.
Automatic Software Test Drivers.
Computer :44-50, April, 1978.
Reprinted in [15].

[18] Dewayne E. Perry.
Programmer Productivity in the Inscape Environment.
In *IEEE Global Telecommunications Conference*, pages 428-434. December, 1986.

[19] Dewayne E. Perry and Gail E. Kaiser.
Infuse: A Tool for Automatically Managing and Coordinating Source Changes in Large Systems.
In *ACM Fifteenth Annual Computer Science Conference*, pages 292-299. St. Louis, MO, February, 1987.

[20] Dewayne E. Perry.
Software Interconnection Models.
In *9th International Conference on Software Engineering*, pages 61-69. Monterey, CA, March, 1987.

[21] Dewayne E. Perry and Gail E. Kaiser.
Models of Software Development Environments.
In *Tenth International Conference on Software Engineering*. Raffles City, Singapore, April, 1988.
To appear.

[22] Ronald E. Prather and J. Paul Myers, Jr.
The Path Prefix Software Testing Strategy.
IEEE Transactions on Software Engineering SE-13(7):761-766, July, 1987.

[23] Robert P. Roe and John H. Rowland.
Some Theory Concerning Certification of Mathematical Subroutines by Black Box Testing.
IEEE Transactions on Software Engineering SE-13(6):677-682, June, 1987.

[24] B.R. Rowland and R.J. Welsch.
The 3B20D Processor & DMERT Operating System: Software Development System.
The Bell System Technical Journal 62(1):275-289, January, 1983.

[25] Rudolph E. Seviora.
Knowledge-Based Program Debugging Systems.
IEEE Software :20-32, May, 1987.

[26] Richard N. Taylor, Lori Clarke, Leon J. Osterweil, Jack C. Wiledon and Michal Young.
Arcadia: A Software Development Environment Research Project.
In *2nd International Conference on Ada Applications and Environments*. IEEE Computer Society, Miami Beach, FL, April, 1986.

[27] Walter F. Tichy.
RCS — A System for Version Control.
Software — Practice and Experience 15(7):637-654, July, 1985.

[28] Walter F. Tichy.
Smart Recompilation.
ACM Transactions on Programming Languages and Systems 8(3):273-291, July, 1986.

[29] Elaine J. Weyucker.
Axiomatizing Software Test Data Adequacy.
IEEE Transactions on Software Engineering SE-12(12):1128-1138, December, 1986.

[30] Alexander L. Wolf, Lori A. Clarke and Jack C. Wileden.
Ada-Based Support for Programming-in-the-Large.
IEEE Software 2(2):58-71, March, 1985.

[31] ACM/SIGSoft and IEEE/CS Software Engineering Technical Committee.
Workshop on Software Testing, IEEE Computer Society, Banff, Canada, 1986.

Adequate Testing and Object-Oriented Programming

Dewayne E. Perry
AT&T Bell Laboratories
Murray Hill, NJ 07974

Gail E. Kaiser*
Columbia University
Department of Computer Science
New York, NY 10027

October 1988

Abstract

One of the primary advantages frequently cited for object-oriented programming is the inheritance of reusable code from superclasses. It is commonly assumed that properly constructed reusable units such as abstract data types and classes can be tested once in isolation and reused without retesting in a wide variety of contexts. Although this is intuitively appealing, it turns out to be a false assumption for certain widely accepted testing criteria.

In this article, we consider the *adequate testing* of programs written in object-oriented languages. We explain the axioms of adequate testing developed in the testing community, discuss their application to specification-based and program-based testing, and examine their implications for object-oriented programming. Contrary to one's intuition, we discover that, in the general case, inherited code must be retested in most contexts of reuse. We illustrate this by applying the adequacy axioms to encapsulation and single inheritance, overriding of methods, and multiple inheritance, respectively.

* Supported in part by National Science Foundation grants CCR-8858029 and CCR-8802741, in part by grants from AT&T, DEC, IBM, Siemens, Sun, and Xerox, in part by the Center for Advanced Technology and by the Center for Telecommunications Research.

1. Introduction

Brooks, in his paper "No Silver Bullet: Essence and Accidents of Software Engineering" [3], states:

> *Many students of the art hold out more hope for object-oriented programming than for any of the other technical fads of the day. I am among them.*

We are among them as well. However, we have uncovered a flaw in the general wisdom about object-oriented languages — that "proven" (that is, well-understood, well-tested and well-used) classes can be reused as superclasses without retesting the inherited code. On the contrary, inherited methods must be retested in most contexts of reuse in order to meet the standards of adequate testing. In this paper, we prove this result by applying test adequacy axioms to certain major features of object-oriented languages — in particular, encapsulation in classes, overriding of inherited methods, and multiple inheritance pose various difficulties for adequately testing a program. Note that our results do not indicate that there is a flaw in the general wisdom that classes promote reuse (which they in fact do), but that some of the attendant assumptions about reuse are mistaken (that is, those concerning testing)

Our past work in object-oriented languages has been concerned with multiple inheritance and issues of granularity as they support reuse [10,11]. Independently, we have developed several technologies for change management in large systems [12,14,20] and recently have been investigating the problems of testing as a component of the change process [13], especially the issues of integration and regression testing. When we began to apply our testing approach to object-oriented programs, we expected that retesting object-oriented programs after changes would be easier than retesting equivalent programs written in conventional languages. Our results, however, have brought this thesis into doubt. Testing object-oriented programs may still turn out to be easier than testing conventional-language programs, but there are certain pitfalls that must be avoided.

First we explain the concepts of specification- and program-based testing, and describe criteria for *adequate testing*. Next, we list a set of axioms for test data adequacy developed in the testing community for program-based testing. We then apply the adequacy axioms to three features common to many object-oriented programming

languages, and show why the axioms may require inherited code to be retested.

2. Testing

By definition, a program is deemed to be *adequately tested* if it has been covered according to the selected criteria. The principle choice is between two divergent forms of test case coverage reported by Howden [9]: specification-based and program-based testing.

Specification-based (or "black-box") testing is what most programmers have in mind when they set out to test their programs. The goal is to determine whether the program meets its functional and non-functional (for example, performance) specifications. The current state of the practice is informal specification, and thus informal determination of coverage of the specification is the norm. For example, tests can be cross-referenced with portions of the design document [19], and a test management tool can make sure that all parts of the design document are covered. Test adequacy determination has been formalized for only a few special cases of specification-based testing — most notably, mathematical subroutines [23].

In contrast to specification-based testing, *program-based* (or "white-box") testing implies inspection of the source code of the program and selection of test cases that together cover the program, as opposed to its specification. Various criteria have been proposed for determining whether the program has been covered — for example, whether all statements, branches, control flow paths or data flow paths have been executed. In practice, some intermediate measure such as essential branch coverage [4] or feasible data flow path coverage [5] is most likely to be used, since the number of possibilities might otherwise be infinite or at least infeasibly large. The rationale here is that we should not be confident about the correctness of a program if (reachable) parts of it have never been executed.

The two approaches are orthogonal and complimentary. Specification-based testing is weak with respect to formal adequacy criteria, while program-based testing has been extensively studied [6]. On the one hand, specification-based testing tells us how well it meets the specification, but tells us nothing about what part of the program is executed to meet each part of the specification. On the other hand, program-based testing tells us nothing about whether the program meets its intended functionality.

Thus, if both approaches are used, program-based testing provides a level of confidence derived from the adequacy criteria that the program has been well tested whereas specification-based testing determines whether in fact the program does what it is supposed to do.

3. Axioms of Test Data Adequacy

Weyuker in "Axiomatizing Software Test Data Adequacy" [29] developed a general axiomatic theory of test data adequacy and considers various adequacy criteria in the light of these axioms. Recently, in "The Evaluation of Program-Based Software Test Data Adequacy Criteria" [30], Weyuker revises and expands the original set of eight axioms to eleven. The goal of the first paper was to demonstrate that the original axioms are useful in exposing weaknesses in several well-known program-based adequacy criteria. The point of the second paper is to demonstrate the *insufficiency* of the current set of axioms, that is, there are adequacy criteria that meet all eleven axioms but clearly are irrelevant to detecting errors in programs. The contribution of our paper is that, by applying these axioms to object-oriented programming, we expose weaknesses in the common intuition that programs using inherited code require less testing than those written using other paradigms.

The first four axioms state:

- **Applicability.** *For every program, there exists an adequate test set.*
- **Non-Exhaustive Applicability.** *There is a program P and test set T such that P is adequately tested by T, and T is not an exhaustive test set.*
- **Monotonicity.** *If T is adequate for P, and T is a subset of T' then T' is adequate for P.*
- **Inadequate Empty Set.** *The empty set is not an adequate test set for any program.*

These (intuitively obvious) axioms apply to all programs independent of which programming language or paradigm is used for implementation, and apply equally to program-based and specification-based testing.

Weyuker's three new axioms are also intuitively obvious.

- Renaming. *Let P be a renaming of Q; then T is adequate for P if and only if T is adequate for Q.*
- Complexity. *For every n, there is a program P, such that P is adequately tested by a size n test set, but not by any size n-1 test set.*
- Statement Coverage. *If T is adequate for P, then T causes every executable statement of P to be executed.*

A program P is a *renaming* of Q if P is identical to Q except that all instances of an identifier x of Q have been replaced in P by an identifier y, where y does not appear in Q, or if there is a set of such renamed identifiers. The first two axioms are applicable to both forms of testing; the third applies only to program-based testing. The concepts of renaming, size of test set, and statement depend on the language paradigm, but this is outside the scope of this article.

4. Antiextensionality, General Multiple Change, Antidecomposition, and Anticomposition Axioms

We are interested in the four remaining (not so obvious) axioms: the antiextensionality, general multiple change, antidecomposition and anticomposition axioms. These axioms are concerned with testing various parts of a program in relationship to the whole and *vice versa*, and certain of them apply only to program-based and not to specification-based adequacy criteria. They are, in some sense, negative axioms in that they expose inadequacy rather than guarantee adequacy.

Antiextensionality. If two programs compute the same function (that is, they are *semantically close*), a test set adequate for one is not necessarily adequate for the other.

There are programs P and Q such that P ≡ Q, [test set] *T is adequate for P, but T is not adequate for Q.*

This is probably the most surprising of the axioms, partly because our intuition of what it means to adequately test a program is rooted in specification-based testing. In

specification-based testing, adequate testing is a function of covering the specification. Since equivalent programs have, by definition, the same specification [22], any test set that is adequate for one must be adequate for the other. However, in program-based testing, adequate testing is a function of covering the source code. Since equivalent programs may have radically different implementations, there is no reason to expect a test set that, for example, executes all the statements of one implementation will execute all the statements of another implementation.

General Multiple Change. When two programs are syntactically similar (that is, they have the *same shape*), they usually require different test sets.

> *There are programs P and Q which are the same shape, and a test set T such that T is adequate for P, but T is not adequate for Q.*

Weyuker states: "Two programs are of the *same shape* if one can be transformed into the other by applying the following rules any number of times: (a) Replace relational operator r1 in a predicate with relational operator r2. (b) Replace constant c1 in a predicate or assignment statement with constant c2. (c) Replace arithmetic operator a1 in an assignment statement with arithmetic operator a2." Since an adequate test set for program-based testing may be selected, for example, to force execution of both branches of each conditional statement, new relational operators and/or constants in the predicates may require a different test set to maintain branch coverage. Although this axiom is clearly concerned with the implementation, not the specification, of a program, we could postulate a similar axiom about the syntactic similarity of specifications, as opposed to source code.

Antidecomposition. Testing a program component in the context of an enclosing program may be adequate with respect to that enclosing program but not necessarily adequate for other uses of the component.

> *There exists a program P and component Q such that T is adequate for P, T' is the set of vectors of values that variables can assume on entrance to Q for some t of T, and T' is not adequate for Q.*

This axiom characterizes a property of adequacy as well as an interesting property of testing — that is, a program can be adequately tested even though it contains

unreachable code. But the unreachable code remains untested, adequately or otherwise. The degenerate example is that in which Q is unreachable in P and T' is the null set. By the Inadequate Empty Set axiom of the previous section, T' cannot be adequate for Q. In the more typical case, some part of Q is not reachable in P but is reachable in other contexts; hence, T' will not adequately test Q. While this axiom is written in program-based terms, it is equally applicable to specification-based testing. In particular, the enclosing program P may not utilize all the functionality defined by the specification of Q and thus could not possibly test Q adequately.

Anticomposition. Adequately testing each individual program component in isolation does not necessarily suffice to adequately test the entire program. Composing two program components results in interactions that cannot arise in isolation.

> *There exist programs P and Q, and test set T, such that T is adequate for P, and the set of vectors of values that variables can assume on entrance to Q for inputs in T is adequate for Q, but T is not adequate for P;Q.* [P;Q is the composition of P and Q.]

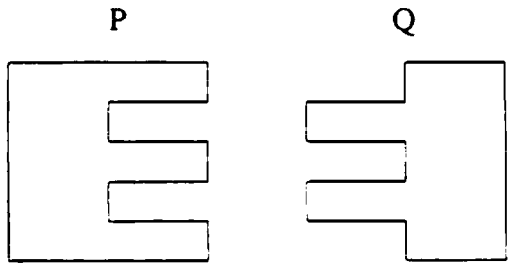

Figure 1

This axiom is counter-intuitive if we limit our thinking to *sequential* composition of P and Q. Consider instead the composition illustrated in figure 1, which can be interpreted as either P calls Q multiple times or P and Q are mutually recursive. In either case, one has the opportunity to modify the context seen by the other in a more complex manner than could be done using stubs during testing of individual components in isolation.

If the composition of P and Q is in fact sequential, then the axiom is still true — just less useful. The proof is by a simple combinatorics argument: If p is the set of paths through P and q is the set of paths through Q, then the set of paths through P;Q may be as large as $p \times q$, depending on the form of composition and on reachability as considered by the previous axiom. However, T applied to P;Q generates at most p paths. A larger test set may be needed to induce the full set of paths. This is an issue for specification-based as well as program-based testing when the specification captures only what the program is supposed to do, not including what it is not supposed to do.

5. Encapsulation in Classes

In this and the following two sections, we consider only abstractions of encapsulation, overriding of inherited methods and multiple inheritance, respectively, rather than concern ourselves with the details of specific object-oriented languages, such as Smalltalk-80 [7], Flavors [18], CommonLoops [1] and C++ [28].

Encapsulation is a technique for enforcing information hiding, where the interface and implementation of a program unit are syntactically separated. This enables the programmer to hide design decisions within the implementation, and to narrow the possible interdependencies with other components by means of the interface. Encapsulation encourages program modularity, isolates separately developed program units, and restricts the implications of changes. In particular, if a programmer changes the implementation of a unit, leaving the interface the same, other units should be unaffected by those changes. Our initial intuition, grounded in specification-based testing, is that we should be able to limit testing to just the modified unit. However, the anticomposition axiom reminds us of the necessity of retesting every dependent unit as well, because a program that has been adequately tested in isolation may not be adequately tested in combination. This means that integration testing is always necessary in addition to unit testing, regardless of the programming language paradigm.

Fortunately, one ramification of encapsulation for testing is that the dependencies tend to be explicit and obvious. If a programmer changes only the implementation of a unit, he need only retest that unit and any units that explicitly depend on it (call it, use

its global variables, etc), as opposed to the entire program. Similarly, if the programmer adds a new unit, he need only test that unit and those existing units that have been modified to use it (plus unmodified existing units that previously used a different unit that is now masked due to a naming conflict).

One would assume that the classes of object-oriented languages would exhibit this behavior, so that it would be both necessary and sufficient to retest those classes explicitly dependent on a changed class as well as the modified class itself. We would expect that, when a superclass is modified, it would be necessary to retest all its subclasses since they depend on it in the sense that they inherit its methods. What we don't expect is the result of the **antidecomposition** axiom — that, when we add a new subclass (or modify an existing subclass), we must retest the methods inherited from each of its ancestor superclasses. The use of subclasses adds this unexpected form of dependency because it provides a new context for the inherited components — that is, the dependency is in both directions where we thought it was only in the one direction.

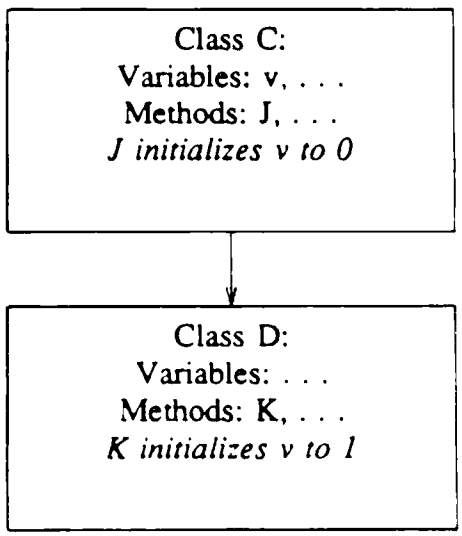

Figure 2

For example, consider a class C with method J; we have adequately tested J with respect to C. We now create a new class D as a subclass of C; D does not replace J but inherits it from C. According to the **antidecomposition** axiom, it is necessary to retest J in the context of class D. There may be new errors when in the context of D, with its enlarged set of methods and instance variables — and perhaps subtly different

local *meanings* for instance variables inherited from C. The bug illustrated in figure 2 (the conflicting assumptions about instance variable v) would not be detected without retesting J in the context of D.

In order to make this example more concrete, consider C to the class WindowManager, D to be the class SunWindowManager, J is the method InitializeScreen, and K is SetScreenBackground. J initializes to a blank screen, while K puts a digitized picture in the background. There are obvious problems if K is invoked first and then J, and vice versa.

There is one case where adding a new subclass does not require retesting the methods inherited from the superclass in order to meet the adequacy axioms. This is when the new subclass is a pure extension of the superclass, that is, it adds new instance variables and new methods and there are no interactions in either direction between the new instance variables and methods and any inherited instance variables and methods.

At least one object-oriented language has solved this problem in the general case, by prohibiting unexpected dependencies: CommonObjects [25,26] removes all implicit inheritance — that is, inherited methods must be explicitly invoked. This, in effect, inserts ''firewalls'' between each superclass and its subclasses, in the same sense that encapsulation inserts firewalls between a class and its clients.

6. Overriding of Methods

Almost all object-oriented languages permit a subclass to replace an inherited method with a locally defined method with the same name, although some support a subtyping hierarchy that restricts the method to have the same specification [24]. In either case, it is obvious that the overriding subclass has to be retested. What is not so obvious is that a different test set is often needed. This is expressed by the antiextensionality axiom: although the two methods compute semantically close functions, a test set adequate for one is not necessarily adequate for the other.

For example, consider figure 3 where class C has subclass D, and method M is defined in C but not in D. Say there exists an object O that is an instance of class D, which receives a message containing the method selector M; M applied to O has already been adequately tested. Now we change class D to add its own method M, which is

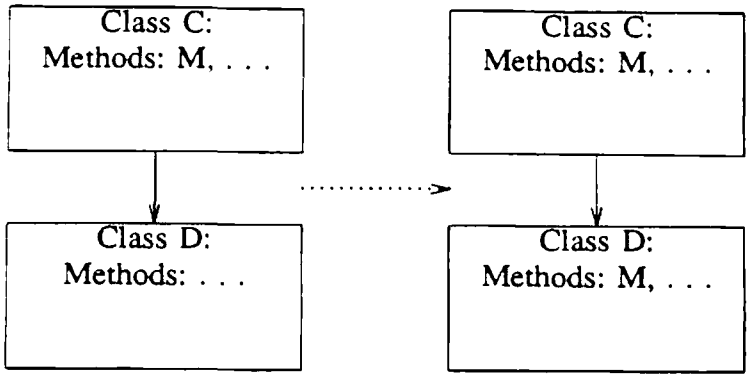

Figure 3

similar to C.M (by "C.M", we mean the method M from superclass C). Obviously, we need to retest class D. Intuitively we would expect that the old test data would be adequate, but the **antiextensionality** axiom reminds us that it may not be adequate. Thus, we may have to develop new test cases for two reasons. First, remember that program-based testing considers the details of the program formulation, attempting to cover, for example, each statement or branch. The test data would necessarily be at least slightly different for C.M and D.M if the formulation in terms of statements and branches were different; the test data would probably be very different if C.M and D.M used different algorithms. Second, it is very likely that the underlying motivation for overriding a method affects not only the internal structure of the overriding method but its external behavior as well — that is, it changes the functional specification. Hence, in addition to test cases to exercise the different structure of the method, we need test cases to test the different specification of the that method.

More concretely, consider C to be the class WindowManager, D to be the class SunWindowManager, C.M to be the method RefreshDisplay that rewrites an entire bitmapped screen, and D.M to be the method RefreshDisplay that repaints only the "damaged" part of a bitmapped screen. In this case, the specifications as well as the implementations of the two methods might be different, in which case different test sets would be required for specification-based as well as program-based testing.

In the previous section, we treated the two-way dependency between classes and superclasses and explained how the **antidecomposition** axiom requires testing of inherited methods in each inheriting context as well as the defining context. What we did not discuss there was the application of the **antiextensionality** axiom to this

additional testing: different test sets may be needed at every point in the ancestor chain between the class defining the overriding method and its ancestor class defining the overridden method.

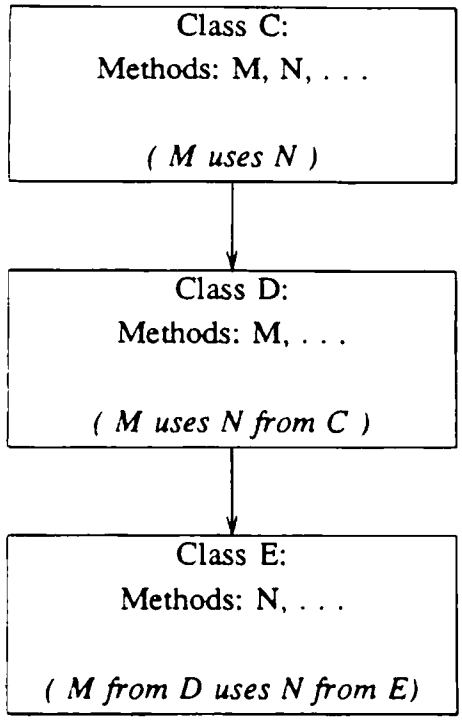

Figure 4

In figure 4, class C has subclass D, which in turn has subclass E; C has methods M and N; D has method M, which uses method N (from C); class E does not have method M but does have method N (overriding the N inherited from C). The antiextensionality axiom reminds us that we need different test data for M with respect to each of the classes C, D, and E. This is obvious with respect to instances of C and D, since they invoke distinct methods M in response to the message M; even if these methods are semantically close, test data adequate for one may not be adequate for the other. This is less obvious with respect to D and E, since they invoke the identical method M. But when we consider that M calls C.N for D whereas it calls E.N for E, it becomes clear that different test sets are required since the formulation and algorithms used by C.N and E.N are likely to be different in functionality as well as structure.

Again, more concretely, let class C be WindowManager where method M is RefreshDisplay and method N is DrawCharacter, using bitmapped fonts; let class D be SunWindowManager where method M is D's replacement for the method RefreshDisplay; and let class E be NeWS where method N is E's replacement for the method DrawCharacter, using Postscript fonts.

7. Multiple Inheritance

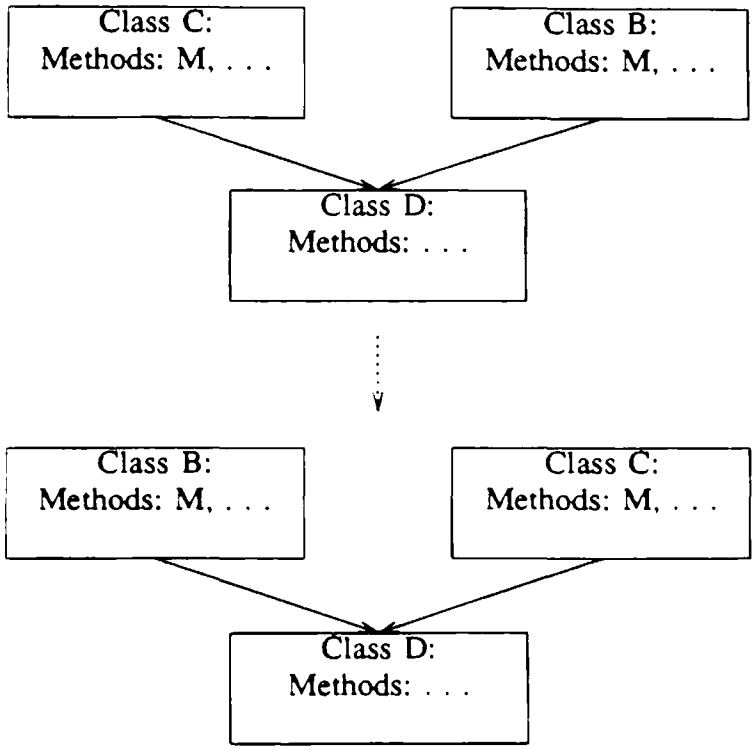

Figure 5

Some, but not all, object-oriented languages support multiple inheritance [2], where each class may have an arbitrary number of superclasses. The so-called "multiple inheritance problem" arises when the same component may be inherited along different ancestor paths. Solutions to this problem typically define a precedence ordering, which linearizes the set of ancestors so that there is a unique selection (or a unique ordering if the semantics of the language are such that all conflicting inherited methods must be invoked) [27]. These solutions, unfortunately, cause very small syntactic changes to have very large semantic consequences. Fortunately, the general multiple change axiom reminds us that programs that are syntactically similar usually

require different test sets.

In figure 5, class D lists superclasses C and B, in that order, and the language imposes the precedence ordering C, B. Method M is defined by both C and B but not by D. Class D is then changed so that the ordering of the superclasses is B and C (meaning that the precedence ordering is B, C). Not only must class D be retested, since it now uses B.M rather than C.M, but most likely a different set of tests must be used. Since C and B are independent, and perhaps developed separately, there is no reason that B.M would be either syntactically or semantically similar to C.M — and even if it were, the **antiextensionality** and **general multiple change** axioms remind us that even then different test sets may be necessary.

As a concrete realization of this example, let class C be TextWindowManager where method M is RefreshDisplay (that repaints the window from a text description), let class B be GraphicsWindowManager where method M is RefreshDisplay (that repaints the window from a bit-mapped representation), and let class D be SunWindowManager.

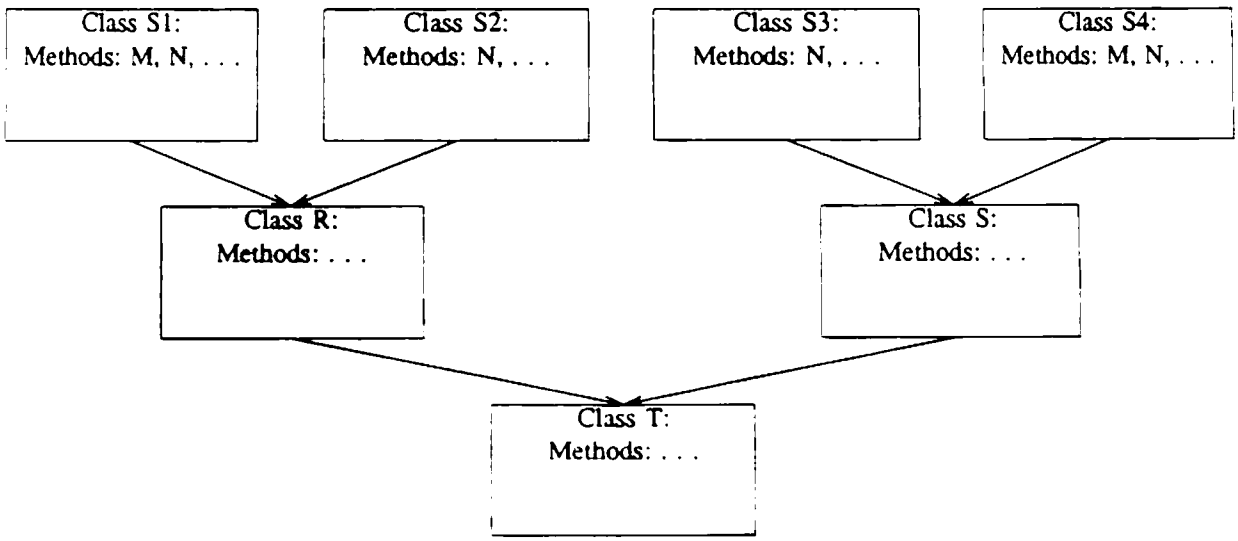

Figure 6

The example in figure 6 shows the inherent compounding effects of multiple inheritance.

This implication of the **general multiple change** axiom is probably the most significant result of applying the test data adequacy axioms to object-oriented

languages, but also the least surprising to the object-oriented languages community. Multiple inheritance is already widely recognized as both a blessing and a curse [15,16,17].

8. Conclusions

Inheritance is one of the primary strengths of object-oriented programming. However, it is precisely because of inheritance that we find problems arising with respect to testing.

- Encapsulation together with inheritance, which intuitively ought to bring a reduction in testing problems, compounds them instead.
- Where non-inheritance languages make the effects of changes explicit, inheritance languages tend to make these effects implicit and dependent on the various underlying, and complicated, inheritance models.

Brooks concludes his section on object-oriented programming:

> *Nevertheless, such advances can do no more than to remove all the accidental difficulties from the expression of the design. The complexity of the design itself is essential, and such attacks make no change whatever in it. An order-of-magnitude gain can be made by object-oriented programming only if the unnecessary type-specification underbrush still in our programming language is itself nine-tenths of the work involved in designing a program product. I doubt it.*

While object-oriented programming clears away much of the accidental underbrush of design, we have noted ways in which it adds to the accidental underbrush of change management and testing. We conclude that there is a pressing need for research on testing of object-oriented languages. We have begun work on this in the context of a data-oriented debugger for concurrent object-oriented languages [8] and in the context of semantic analysis (applying the approach of Inscape [21] to C++).

Acknowledgements

Jim Coplien, Narain Gehani, Jim Krist, and Alex Wolf provided useful criticism and suggestions on earlier versions of this paper.

References

[1] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. "CommonLoops: Merging Lisp and Object-Oriented Programming", *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, Portland OR, September 1986. pp 17-29.

[2] Alan Borning and Daniel Ingalls. "Multiple Inheritance in Smalltalk-80", *Proceedings of the National Conference on Artificial Intelligence*, Pittsburgh PA, 1982. pp 234-237.

[3] Frederick P. Brooks, Jr. "No Silver Bullet: Essence and Accidents of Software Engineering", *Computer* 20:4 (April 1987). pp 10-20.

[4] Takeshi Chusho. "Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing", *IEEE Transactions on Software Engineering* SE-13:5 (May 1987). pp 509-517.

[5] Phyllis G. Frankel and Elaine J. Weyuker. "Data Flow Testing in the Presence of Unexecutable Paths", in *Workshop on Software Testing*, Banff, Canada, July 1987. pp 4-13.

[6] David Gelperin and Bill Hetzel. "The Growth of Software Testing", *Communications of the ACM* 31:6 (June 1988). pp 687-695.

[7] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*, Reading MA: Addison-Wesley, 1983.

[8] Wenwey Hseush and Gail E. Kaiser. "Data Path Debugging: Data-Oriented Debugging for a Concurrent Programming Language", *Proceedings of the ACM SIGPlan/SIGOps Workshop on Parallel and Distributed Debugging*, Madison WI, May 1988. pp. 236-246.

[9] William Howden. *Software Engineering and Technology: Functional Program Testing and Analysis*, New York: McGraw-Hill Book Co., 1987.

[10] Gail E. Kaiser and David Garlan. "Melding Software Systems from Reusable Building Blocks", *IEEE Software*, July 1987. pp 17-24.

[11] Gail E. Kaiser and David Garlan. "MELDing Data Flow and Object-Oriented Programming", *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, Kissimmee FL, October 1987. *SIGPlan Notices* 22:12 (December 1987). pp 254-267.

[12] Gail E. Kaiser and Dewayne E. Perry. "Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution", *Proceedings of the Conference on Software Maintenance*, Austin TX, September 1987". pp 108-114.

[13] Gail E. Kaiser and Dewayne E. Perry. "INFUSE: Integration Testing with Crowd Control". Technical Report. Computing Systems Research Laboratory, AT&T Bell Laboratories, January 1988.

[14] Gail E. Kaiser, Simon M. Kaplan and Josephine Micallef. "Multiuser, Distributed Language-Based Environments", *IEEE Software*, November 1987. pp 58-67.

[15] Norman Meyrowitz. *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, Portland OR, September 1986. Special Issue of *SIGPlan Notices*, 21:11 (November 1986).

[16] Norman Meyrowitz. *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, Orlando FL, October 1987. Special Issue of *SIGPlan Notices*, 22:12 (December 1987).

[17] Norman Meyrowitz. *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, San Diego CA, September 1988. Special Issue of *SIGPlan Notices*, 23:11 (November 1988).

[18] David A. Moon. "Object-Oriented Programming with *Flavors*", in *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, Portland OR, September 1986. pp 1-8.

[19] Thomas J. Ostrand, Ron Sigal, and Elaine Weyuker. "Design for a Tool to Manage Specification-Based Testing", in *Workshop on Software Testing*, Banff, Canada, July 1987. pp 41-50.

[20] Dewayne E. Perry and Gail E. Kaiser. "Infuse: A Tool for Automatically Managing and Coordinating Source Changes in Large Systems", *Proceedings of the ACM Fifteenth Annual Computer Science Conference*, St. Louis MO, February 1987. pp 292-299.

[21] Dewayne E. Perry. "Software Interconnection Models", *Proceedings of the 9th International Conference on Software Engineering*, Monterey CA, April 1987. pp 61-69.

[22] Dewayne E. Perry. "Version Control in the Inscape Environment", *Proceedings of the 9th International Conference on Software Engineering*, Monterey CA, April 1987. pp 142-149.

[23] Robert P. Roe and John H. Rowland. "Some Theory Concerning Certification of Mathematical Subroutines by Black Box Testing", *IEEE Transactions on Software Engineering* SE-13:7 (July 1987). pp 761-766.

[24] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. "An Introduction to Trellis/Owl", in *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, Portland OR, September 1986. pp 1-8.

[25] Alan Snyder. "CommonObjects: An Overview", *Object-Oriented Programming Workshop Proceedings*, Yorktown Heights NY, June 1986. pp 19-29.

[26] Alan Snyder. "Inheritance and the Development of Encapsulated Software Components", *Twentieth Hawaii International Conference on System Sciences*, Kona HI, January 1987. volume II, pp. 227-238.

[27] Mark Stefik and Daniel G. Bobrow. "Object-Oriented Programming: Themes and Variations", *The AI Magazine* (Winter 1985). pp 40-62.

[28] Bjarne Stroustrup. *The C++ Programming Language*. Reading MA: Addison-Wesley, 1986.

[29] Elaine J. Weyuker. "Axiomatizing Software Test Data Adequacy", *IEEE Transactions on Software Engineering* SE-12:12 (December 1986). pp 1128-

1138.

[30] Elaine J. Weyuker. "The Evaluation of Program-Based Software Test Data Adequacy Criteria", *Communications of the ACM* 31:6 (June 1988). pp 668-675.