Infuse: Fusing Integration Test Management with Change Management

Gail E. Kaiser* Dewayne E. Perry William M. Schell
Columbia University AT&T Bell Laboratories AT&T Bell Laboratories
Dept of Computer Science Computer Systems Research Lab Computer Systems Research Lab
New York, NY 10027 Murray Hill, NJ 07974 Murray Hill, NJ 07974

Infuse is an experimental software development environment focusing on change coordination during the maintenance/evolution phase of large scale software projects. Its core philosophy is to integrate strongly connected modules first and more weakly connected sets of modules later, moving up a hierarchy from singletons to clusters of interdependent modules and, finally, merging the change set into the baseline. We have previously described how Infuse enforces static consistency at each level of the hierarchy. We now extend our work to dynamic consistency — *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. Infuse is implemented for C, and is used to support its own evolution, but the implementation does not yet provide the test management described here.

---

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

keywords: change coordination, integration testing, programming-in-the-many, regression testing, software development environments, software maintenance)

1. Introduction

The purpose of this paper is to present a novel approach to integration test management suitable for large-scale projects. Our approach has four primary contributions: hierarchical integration, semi-automatic construction of test harnesses, semi-automatic construction of regression test suites, and integration of these facilities into `Infuse`, an experimental software development environment.

- Hierarchical integration of modified modules and subsystems is hardly a new idea, but is rarely enforced (or even supported); `Infuse` enforces hierarchical integration, where the hierarchy may be selected by a distinguished user according to subsystem design or managerial concerns, or may be generated automatically by `Infuse` according to the strengths of dependencies among modules.
- Test harnesses consist of test drivers and stubs, where a stub for a module includes surrogates for all subroutines and objects expected to be provided by that module. At the unit testing level, where an individual user tests his own module(s) in isolation, `Infuse` can generate the headers for the stubs automatically but the contents must be constructed by hand or by some external mechanism. At higher levels in the hierarchy, however, `Infuse` reuses stubs, or components of stubs, from lower levels in the construction of the higher level test harnesses.
- Regression test suites consist of two classes of test cases: those specifically constructed for the collection of modules or subsystems at the current level of the integration hierarchy, and those previously executed on some subset of these modules at a lower level. The first class must be built by hand or by some external mechanism, but `Infuse` selects the second class by keeping track of those tests previously exercised on surrogates rather than components of the actual module.
- These facilities are not independent tools, but integral components of the `Infuse` environment. The test management facilities share the same object management system — including object repository and query interface — with the change management facilities. The hierarchical integration is directly enforced by the environment's concurrency control mechanism.

`Infuse` is a 'city model' software development environment (SDE) [18], that is, it addresses the special problems of developing and maintaining 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 [16, 20], we have presented our philosophy and the basic mechanisms for isolating groups of modules[1]
into a hierarchy of private databases. This hierarchy is based on the reserve/deposit (also known as reserve/replace) model prevalent in software development environments, where modules or other software artifacts are reserved (locked), copied to a private area, modified, and deposited (atomically updated to the public area and unlocked). We extend this model to a hierarchy in order to minimize the implications and extent of changes that groups of programmers as well as individual programmers must cope with at one time. This paper extends our previous work to support integration test management. In particular, we report the design of `Infuse` support for semi-automatic construction of test harnesses and selection of regression test suites at each (interior) level in the hierarchy.

A prototype implementation of `Infuse` has been completed, and is being used in its own further development. `Infuse` is implemented in C and runs on MicroVax, Sun and HP workstations. The prototype includes a hierarchical clustering algorithm [13], a simple object repository implemented using IDL [25] and RCS [27], a simple graphical browsing interface constructed using X windows, and a hierarchical reserve/deposit model that enforces syntactic consistency (using the Unix lint utility) before permitting deposit into the next higher level of the hierarchy.

---

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

In the next section, we give an overview of the change management facilities of `Infuse`. We then illustrate how `Infuse` supports both change and test management. The subsequent two sections describe the test harness and regression test suite construction facilities, respectively, in detail. We conclude by comparing our approach to related work.

2. `Infuse` Overview

The `Infuse` change management framework constructs and maintains a hierarchy of *experimental databases* (EDBs), where each EDB contains a subset of the *change set*, that is, the set of modules to be modified. At each level of the hierarchy, the subset of the modules in the parent EDB are partitioned into child EDBs. EDBs near the bottom of the hierarchy may be private to an individual programmer or private to a group of closely cooperating programmers, while EDBs near the top may be 'private' to large groups of programmers or to special integration groups. The change process consists of constructing the full hierarchy, making the actual edits in the leaves of the hierarchy, and enforcing the integration of the modules and subsystems within each EDB before permitting the EDB to be deposited into its parent.

By enforcing integration, we mean that `Infuse` allows the deposit of the child EDB into its parent only if it's *locally consistent*. The module(s) in the child EDB may have to be changed several times before local consistency is achieved. Local consistency requires both a static component, which applies some analysis tool to the module(s) to detect errors, and a dynamic component, which executes the module(s) to detect errors.

The rationale for this hierarchical integration is the widely accepted software engineering rule-of-thumb that errors detected early in the lifecycle are much less costly to repair than errors detected late [4, 2]. The same concept applies during maintenance: Interface errors detected earlier in the preparation for a patch or a new release are less expensive that those detected later.

There are two well-known mechanisms for structuring the modules of a system into a hierarchy: managerial and design. A 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` permits a distinguished user to select any of these three bases with respect to the new change set, and in the managerial or design cases the complete hierarchy must be given by the user.

`Infuse` generates the dependency-order hierarchy by *clustering*] of the change set, using a non-Euclidean similarity metric based on the interdependencies between pairs of modules. Typically, the *similarity* between two modules is defined as the total number of symbols (*e.g.*, subroutine names, object names, and so on) exported by one module and imported in the other, or more precisely, defined by one module and used in the other [28]. Other metrics could be used, for example, to weight according to the number of times each symbol is used or according to the number of semantic dependencies associated with each symbol [17]. 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; the details are outside the scope of this paper (see [13, 14]). Our similarity metric based on dependencies is used as 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 have more effects on strongly connected modules than on weakly connected ones, according to a simplistic proportionality argument.

In addition to the strict hierarchy of experimental databases, `Infuse` also supports *workspaces*, which cut across the tree to permit an arbitrary graph structure. Each workspace contains two or more experimental databases, which need not be at the same level of the hierarchy. Each workspace is constructed by human selection of a set of existing experimental databases. A workspace operates just like an experimental database for the purpose of integrating the modules and subsystems contained in the workspace. The difference is there is no ancestor to deposit the workspace into once consistency has been demonstrated. The rationale is to support early integration among modules that do not appear together until relatively high in the hierarchy. This is useful in those cases where the programmers' knowledge of the system and the specific change in progress indicates that this early integration is crucial. In the case where dependency-order rather than a managerial or design hierarchy is used, this alleviates the problems where change implications do not follow the strengths of module interdependencies — for example, reorganizations

where the interdependencies themselves are significantly modified and/or changes where weakly connected modules are significantly affected.

3. Using `Infuse`: An Example

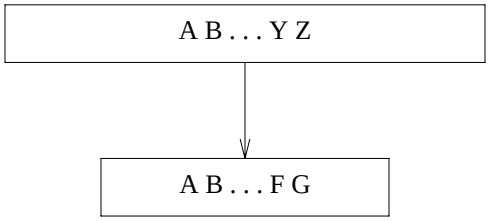

Figure 3-1: Baseline and Change Set

The `Infuse` change management system operates as follows for a scheduled set of changes, such as for a new release or patches to a previous release. The change set is selected manually by a system analyst or automatically by a modification request (MR) system, such as CMS [23]. `Infuse` checks out new revisions of the modules in this set from the version control system as shown in figure 3-1, 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 may be 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 3-2. 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 and the programmer is responsible only for this individual module; thus, we refer to leaf EDBs as *singletons*.

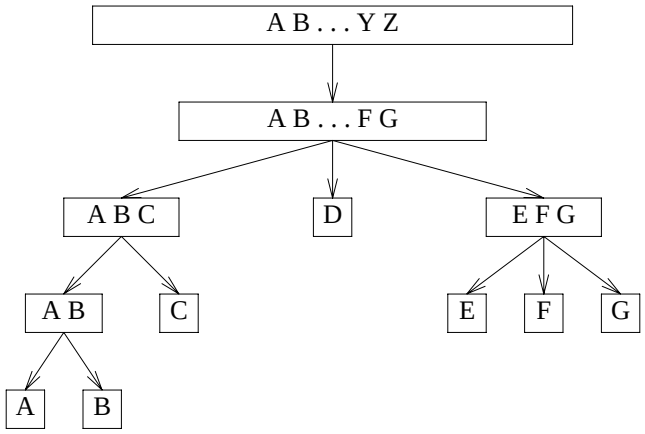

Figure 3-2: Hierarchy of Experimental Databases

When a programmer finishes editing his module, say *A*, he requests the consistency analysis tool, which determines whether or not *A* is locally consistent. Syntactic consistency requires that every identifier defined in *A* is used within *A* 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 *A*) must be consistent with all other uses of the same identifier within *A*. In the case of semantic consistency [19], 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 module *A* is statically consistent, the programmer builds a test harness. The harness consists of a driver D_A that invokes the module to perform the tests and a set of stubs S_A that perform, in an abstract sense, the functionality of those external modules referenced by *A*. In particular, stub $S_{A,M}$ represents all the subroutines and data defined by some module *M* and used in module *A*. `Infuse` compiles and links *A* together with D_A and S_A,

and then the programmer proceeds with testing and debugging.

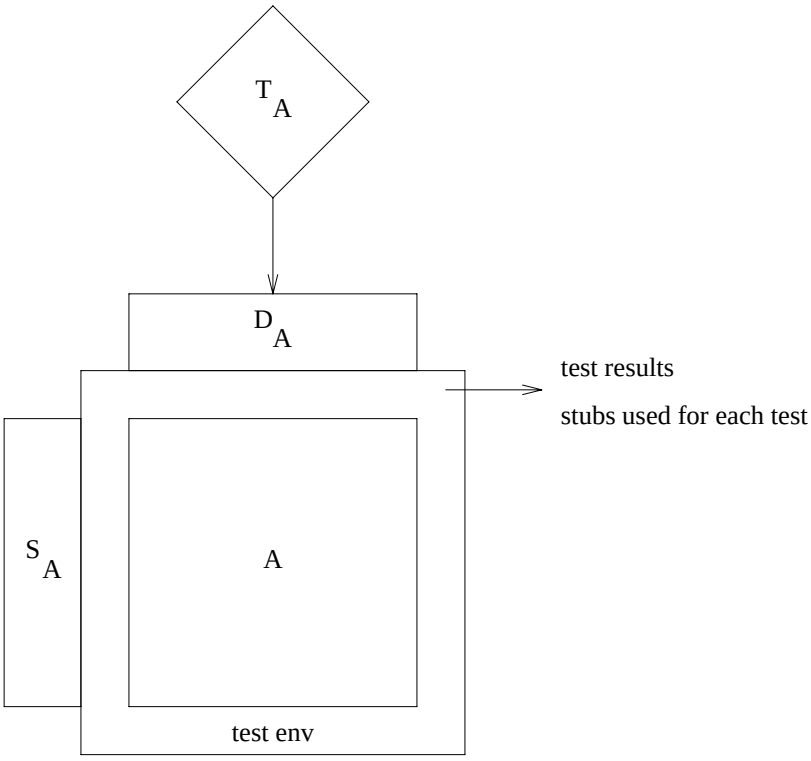

Figure 3-3: Unit Testing Stubs and Test Suite

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 [20]. The stubs, driver and unit test suite are associated with a singleton EDB as shown in figure 3-3. As the test suite T_A is applied, `Infuse` keeps track of which stubs are executed by which tests. After his module *A* has passed all these tests, the programmer enters a command to *deposit* it. Before allowing the deposit, `Infuse` requires that *A* is in fact locally consistent in the static sense of the analysis tool and in the dynamic sense of the unit tests. `Infuse` then checks the module into the version control system, makes this new version visible to the other modules in the parent EDB, and saves D_A, S_A, T_A and the association between tests and stubs during unit testing of *A*.

At some point, all the sibling (singleton) EDBs have been deposited into their parent EDB, which then contains several modules that are very strongly interdependent. In general, each EDB at every level of the hierarchy should have a relatively small number of children (we somewhat arbitrarily choose the range 2 to 5) to keep the set of new interactions relatively manageable. `Infuse` invokes the static analysis tool to check that these modules are locally consistent. If not, it informs the responsible programmers, who negotiate among themselves, agree on further 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.

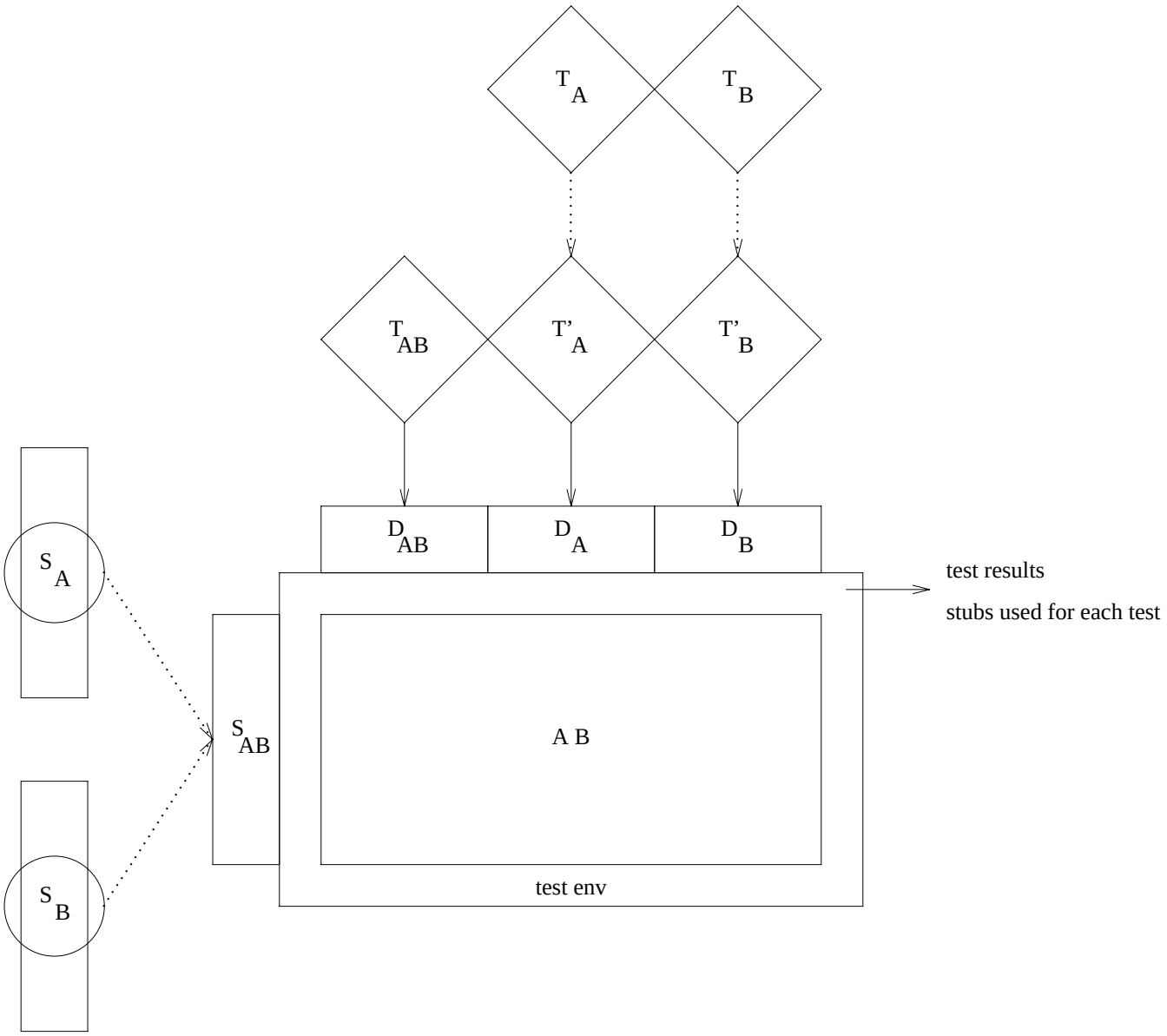

Figure 3-4: Automatically Selected Stubs and Tests

If the modules in the EDB *AB* are locally consistent to the extent that can be determined by a static analysis tool, `Infuse` constructs a set of stubs S_{AB} for integration testing from the sets S_A and S_B available from unit testing. `Infuse` automatically selects the regression test suite T_R (T'_A and T'_B) from the unit test suites T_A and T_B. Usually some programmer must build a new test driver D_{AB} to execute any new tests T_{AB}. The resulting drivers, stubs and tests are illustrated in figure 3-4. The tests are then 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 3-5.

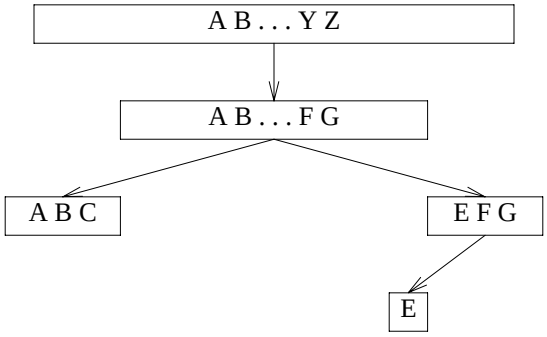

Figure 3-5: Hierarchy After Several Deposits

If errors are detected, however, 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 3-6. 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.

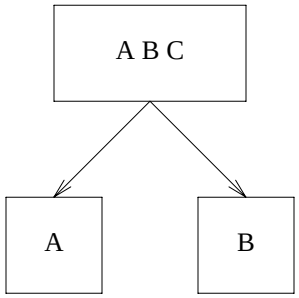

Figure 3-6: Repartioning for Further changes

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.

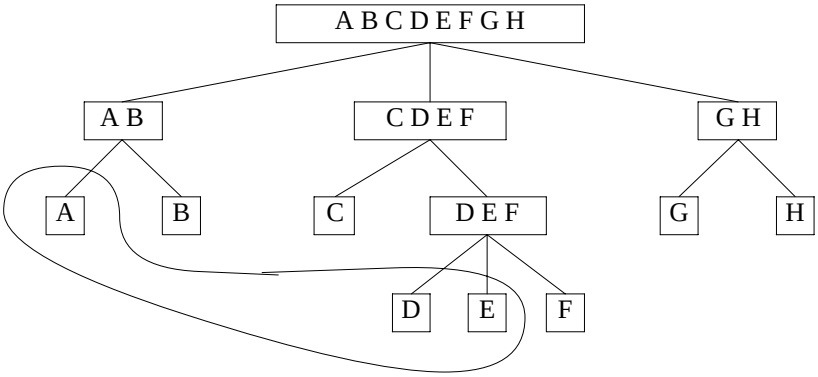

Figure 3-7: Change Simulation in an Individual Workspace

At some point during the change process, a programmer may realize there is a strong interaction among the changes he's pursuing in his EDB and a change being made by one or more other programmers in other EDBs, but their modules will not be considered for integration until relatively high in the EDB hierarchy. In these cases, the first programmer can construct a workspace consisting of all of these EDBs. An example is shown in figure 3-7, where the programmer responsible for module *E* wants to see how his changes interact with the new versions of modules *A*, *B* and *C*. The workspace is used for *change simulation*, as opposed to the change propagation necessary to deposit an EDB. Like change propagation, static and dynamic consistency checking is applied to the union of the modules from the EDBs in the workspace; however, there is no notion of depositing a workspace, and it can be simply dissolved at any time. Any errors become known **only** to the first programmer, who may then make corrective changes to his own modules or communicate with the other programmer.

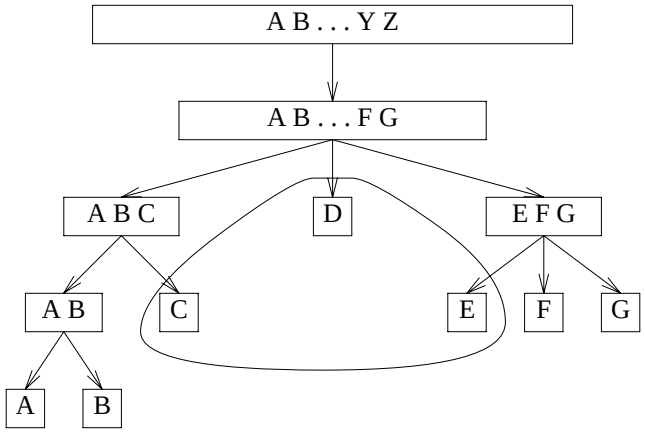

Figure 3-8: Change Simulation in a Group workspace

Alternatively, two or more programmers could construct a shared workspace consisting of their own EDBs. For example, in figure 3-8, the programmers responsible for *C*, *D* and *E* decide to integrate their modules early. In the case of a group workspace, change simulation notifies **all** these programmers of any errors, and they may negotiate early corrections. Note that this is a departure from our previous papers, where simulation implied notification of only the individual programmer who initiated the consistency checking. We now think of all consistency checking within a workspace, as opposed to an EDB, to be a form of change simulation rather than change propagation. Change simulation can still be performed with respect

to an EDB, where one programmer in a child EDB requests consistency checking with respect to the current contents of the parent EDB and is notified of any errors.

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 3-9. Finally, after acceptance testing, `Infuse` deposits the modules in the top-level EDB into the base-line database, checks off the MR, and performs any necessary updates regarding the configuration management system.

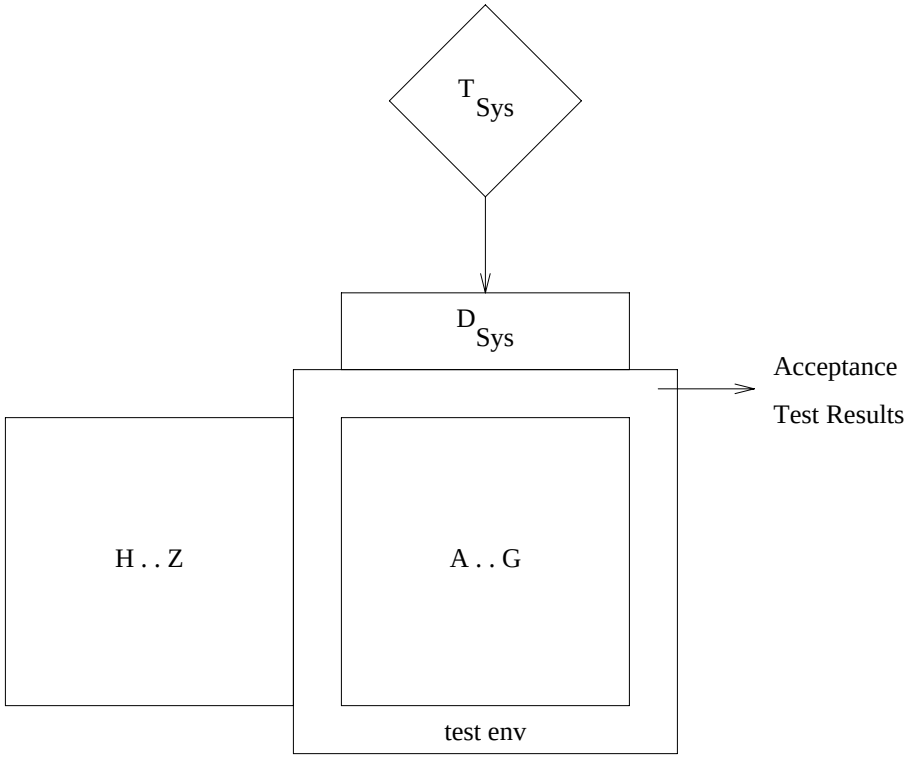

Figure 3-9: Acceptance Testing

4. 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 EDBs 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.

4.1 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.e.*, one of the M_i) in one of the child EDBs has a set of stubs S_M from its singleton EDB. `Infuse` operates according to the algorithm shown in figure 4-1 to construct the set of stubs for E.

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 $S^-_{M,N}$ 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
  $S_M \leftarrow \cup_N \{ S_{M,N\_} \}$
  $\forall\ O \in E$ 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 $\{ O \in E \mid O \neq M / \backslash S_{O,N} \in S_E \}$ then
      ask whether to keep, replace or merge
      and modify S_M accordingly
  $S_E \leftarrow S_E \cup S_M$

Figure 4-1: Algorithm for Stub Selection

`Infuse` examines each stub $S_{M,N}$, constructed to represent absent module N as used by module M. 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. Note that we 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 if it were just a 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 *Ssubx*,N by the lowercase variable x. Note that the duplication of a stub does **not** imply the duplicates are identical, and in fact the content of these stubs may be markedly 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 a 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.

4.2 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:

- tests that originate at this EDB and check the functionality, performance, *etc.* of the corresponding subsystem; and

- 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.

5. Regression Testing

Let E_i be the ith descendant of experimental database E in some standard ordering such as preorder,

T_R be the regression 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 associated with the ith descendent which actually excercised S'_i

T'_{R_i} be the subset of the regression test suite, from the test suite associated with the ith descendant, which actually exercised S'_i

In $T_i \leftarrow \bigcup T'_{R_i} \cup \bigcup T'_i$

Figure 5-1: Algorithm for Regression Test Selection

The preceding discussion of drivers suggests our approach to integration testing, which follows the algorithm shown in figure 5-2 (most of the algorithm appears in the let clause). In the worst case, the regression test suite T_R for an experimental database E is the union of all the test sets from the descendent EDBs. The complete test suite T_E also includes any additional subsystem tests added at this point by one or more of the relevant programmers. `Infuse` helps reduce the amount of testing, because a regression test is executed only when one or more stubs exercised by that test in a descendant EDB is replaced by actual code in the current EDB.

`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_R 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_R used for regression testing of E.

$$D_R = \{ D_i \mid T'_i \neq \emptyset \}$$

5.1 A Note on 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 [3] 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 [22].

The `Infuse` dependency-order hierarchy may not be the best for specification-based testing. There is typically a design hierarchy determined 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.

6. Related Work and Future Directions

Our notion of an experimental database was initially introduced in Smile [11], a multiple-user programming environment for C developed as part of the Gandalf project [7] at Carnegie Mellon University. `Infuse` extends this notion to (1) a hierarchy, (2) automatic partitioning of the change set into EDBs, and (3) integration testing. More recently, Sun Microsystem's Network Software Environment (NSE) [26] provides a more general hierarchy of environments with a copy/modify/merge model [1] oriented towards smaller teams of programmers. Copy/modify/merge is an optimistic variant of reserve/deposit where the module is not locked so multiple programmers can copy it to a private area and modify it; NSE prevents deposit until the changes have been merged with any other changes to the same module since the copy was made. Thus, NSE supports a form of static consistency checking but does not provide any special facilities for testing and requires manual partitioning.

There has been much previous research on testing strategies and tools as they relate to programming-in-the-small [31, 24], and some work on integration of subroutines [8]. SpecMan [15] supports cross-referencing of test cases with portions of the design document, to ensure that test cases are updated in response to design changes. To our knowledge, `Infuse` is the only large-scale SDE directly concerned with integration test management.

Our plans for the `Infuse` include:

- Adding the integration test management to the prototype implementation.
- Continuing our work on consistency and concurrency control, and eventually incorporate an extended transaction model suitable for software process activities [21].
- Extending `Infuse` to a distributed implementation by merging `Infuse` with Mercury [12], a generation system for multiple-user, distributed language-based environments (using incremental evaluation of attribute grammars)
- Integrating `Infuse` with Inscape [19] to support a semantic consistency model (*i.e.*, preconditions, postconditions and obligations with respect to each program unit) in addition to the current syntactic model.

Acknowledgements

Yoelle Maarek developed the hierarchical clustering algorithm used by `Infuse`. Ben Fried, Barry Goldberg, Bireley Jeng, Pierre Nicoli, Gretchen Taylor and Travis Winfrey worked on the implementation of `Infuse` under the direction of Bulent Yener. We would like to thank Yoelle Maarek, Maria Thompson, Michael van Biema, Alex Wolf and Bulent Yener for their useful comments on an earlier

version of this paper.

References

[1] Evan Adams, William Courington, Jonathan Feiber, Jill Foley, David Hendricks, Masahiro Honda, Tom Lyon, Terrence Miller, Russell Sandberg, and Daniel Scales. ‘‘Object Management in a CASE Environment’’, *11th International Conference on Software Engineering*, Pittsburgh PA, May, 1989.

[2] Barry W. Boehm, ‘‘A Spiral Model of Software Development and Enhancement’’, *Computer*, (May 1988), 21:5, pp61-72.

[3] 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), pp509-517.

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

[5] Phyllis G. Frankel and Elaine J. Weyuker, ‘‘Data Flow Testing in the Presence of Unexecutable Paths’’, IEEE Computer Society, *Workshop on Software Testing*, July, 1986, Banff, Canada, pp4-13.

[6] Phyllis G. Frankl and Elaine J. Weyuker, ‘‘A Data Flow Testing Tool’’, *SoftFair II A 2nd Conference on Software Development Tools, Techniques, and Alternatives*, December, 1985, San Francisco CA, pp46-53.

[7] A.N. Habermann and D. Notkin, ‘‘Gandalf: Software Development Environments’’, *IEEE Transactions on Software Engineering*, SE-12:12 (December 1986), pp1117-1127.

[8] Allen Haley and Stuart Zweben, ‘‘Module Integration Testing’’, *Computer Program Testing*, editors: B. Chandrasekaran and S. Radicchi, North-Holland Publishing Co., New York, 1981.

[9] William E. Howden, *Functional Program Testing & Analysis*, McGraw-Hill Book Co., New York, 1987.

[10] Gail E. Kaiser and Dewayne E. Perry, ‘‘Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution’’, ,I "Conference on Software Maintenance", Austin TX, September, 1987, pp108-114.

[11] Gail E. Kaiser and Peter H. Feiler, ‘‘Intelligent Assistance without Artificial Intelligence’’, *32nd IEEE Computer Society International Conference*, February, 1987, San Francisco CA, pp236-241.

[12] Gail E. Kaiser, Simon M. Kaplan and Josephine Micallef, ‘‘Multiuser, Distributed Language-Based Environments’’, *IEEE Software*, November, 1987, pp58-67.

[13] Yoelle S. Maarek and Gail E. Kaiser, ‘‘Change Management for Very Large Software Systems’’, ,.I "7th Annual International Phoenix Conference on Computers and Communications," March, 1988, Scottsdale AZ, pp280-285.

[14] Yoelle S. Maarek, ‘‘Using Structural Information for Managing Very Large Software Systems,’’ Technion — Israel Institute of Technology, August, 1988. This version is prior to final revisions.

[15] Thomas J. Ostrand, Ron Sigal and Elaine J. Weyuker, ‘‘Design for a Tool to Manage Specification-Based Testing’’, IEEE Computer Society, *Workshop on Software Testing*, July, 1986, Banff, Canada, pp41-50.

[16] Dewayne E. Perry and Gail E. Kaiser, ‘‘Infuse: A Tool for Automatically Managing and Coordinating Source Changes in Large Systems,’’ *ACM 15th Annual Computer Science Conference’*, February, 1987, St. Louis MO, pp292-299.

[17] Dewayne E. Perry, ‘‘Software Interconnection Models’’, *9th International Conference on Software Engineering*, Monterey CA, March, 1987, pp61-69.

[18] Dewayne E. Perry and Gail E. Kaiser, ‘‘Models of Software Development Environments’’, *10th International Conference on Software Engineering*, April, 1988, Raffles City, Singapore, pp60-68.

[19] Dewayne E. Perry, ‘‘The Inscape Environment’’, *11th International Conference on Software Engineering*, Pittsburgh PA, May, 1989, pp 2-12.

[20] Ronald E. Prather and J. Paul Myers, Jr., ‘‘The Path Prefix Software Testing Strategy’’, *IEEE Transactions on Software Engineering*, SE-13:7 (July 1987), .pp761-766

[21] Calton Pu, Gail E. Kaiser and Norman Hutchinson, ‘‘Split-Transactions for Open-Ended Activities’’, *14th International Conference on Very Large Data Bases*, August, 1988, Los Angeles CA, pp26-37.

[22] 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 (June 1987), pp677-682.

[23] B.R. Rowland and R.J. Welsch, ‘‘The 3B20D Processor & DMERT Operating System: Software Development System’’, *The Bell System Technical Journal*, 62:1, part 2 (January, 1983), pp275-289.

[24] ACM/SIGSoft and IEEE/CS Software Engineering Technical Committee. *2nd Workshop on Software Testing, Verification and Analysis*, IEEE Computer Society, Banff Canada, 1988.

[25] Richard Snodgrass and Karen Shannon, ‘‘Supporting Flexible and Efficient Tool Integration’’, *’Advanced* Programming *Environments"*, editors: Reidar Conradi, Tor M. Didriksen and Dag H. Wanvik. Springer-Verlag, Berlin, 1986, pp290-313.

[26] *Introduction to the NSE*, Sun Microsystems, Inc., Mountain View CA, March, 1988

[27] Walter F. Tichy, ‘‘Smart Recompilation’’, *ACM Transactions on Programming Languages and Systems*, 8, 3, July, 1986, pp273-291.

[28] Walter F. Tichy, ‘‘RCS — A System for Version Control’’, *Software — Practice and Experience*, 15, 7, July, 1985, pp637-654.

[29] Elaine J. Weyuker, ‘‘Axiomatizing Software Test Data Adequacy’’, *IEEE Transactions on Software Engineering*, SE-12:12 (December 1986), pp1128-1138.

[30] Alexander L. Wolf, Lori A. Clarke and Jack C. Wileden, ‘‘Ada-Based Support for Programming-in-the-Large’’, *IEEE Software*, 2:2 (March 1985), pp58-71.

[31] ACM/SIGSoft and IEEE/CS Software Engineering Technical Committee. *2nd Workshop on Software Testing*, IEEE Computer Society, Banff Canada, 1986.

CONTENTS