Environment Support for The Software Testing Process

Steven Wartik

Computer Science Report No. TR–88–09
April 19, 1988

This research was supported in part by NSF grant DCR–8602674.

Abstract

This paper discusses how a software development environment should support the testing process—that is, the coordination of interactions between developers and testers. For a large project, the problem is a serious one, involving a significant amount of communication overhead that can result in long delays.

The testing process is first formally defined. Next, a solution is presented, based on the typical communication capabilities present in software development environments. This solution has been implemented, and the user interface of the tools that do so are described. An experience using the tools is related. The experience shows that an environment can indeed influence the ease with which testing occurs.

Key words and phrases: software environment, software process, software testing, software configuration management.

1 INTRODUCTION

The coordination of the interactions between project personnel and software is a neglected aspect of software testing. Unit testing tools, path analysis techniques, requirements traceability tools, static and dynamic analysis aids and symbolic execution tools [10,5,8,6] help developers and testers build tests, execute them, and interpret the results—but do not provide guidelines on the order in which tests should be applied, who must see the results, or how the results affect the software integration process.

The problem is often relegated to project management (see [14]). This is because it involves notification of project personnel: a person must know the current state of all modules in the system on which his or her work depends. However, the problem encompasses many other areas. It is partially a configuration management problem, since it involves the interaction of an evolving set of software modules [4]. While environments with configuration management support (*e.g.*, [13,19,11]) provide for such interactions, they do not model the interactions of people and modules that occur during testing.

The relationship of the software development environment to the testing process must also be considered, as an environment greatly influences the testing process. In particular, it defines the mechanisms by which software can be integrated, and by which personnel can know the status of testing. Environments such as TOOLPACK [16] treat software integration aspects, but not specifically in relation to testing. Ideally, an environment will make testing status information readily available. In practice, environments often provide no such information, and people must learn of it by word of mouth.

An environment's inability to provide this information can result in much misspent time and effort. Every time a module is accepted from a round of testing, all modules that depend on that module are affected. Until the developers of those modules know of the acceptance, they are operating with out-of-date knowledge. If they cannot continue their work in the meantime, they must wait idly; if they are working in the belief that the accepted version does not exist, they face inconsistencies. Similarly, testers typically spend much of their time waiting to be informed that a module is ready to test, and trying to coordinate their schedules with other testers. These situations cause delays that add up over the course of a project. Testing a large project requires frequent communication between many people, over many communication paths. If this is done manually, it is likely to be slow and error-prone.

A software development environment could therefore improve its users' productivity by considering these interactions. To do so requires a model of the testing process, defining the interactions between developers, testers and the modules of the system they are building. Beizer presents an informal model [3], but only lists the entities that comprise the process; his model is not rigorous enough to automate. A formal model would both define the interactions that may occur, and identify what an environment needs to support effective testing. However, the model cannot be too restrictive. The focus of this paper is on the process of software integration that leads to a complete, tested system, as distinguished from the generation and execution of

test cases on individual or integrated modules. The model must define the former without constraining ways to accomplish the latter.

This paper has two goals. The first is to provide a formal model that defines how people interact during testing to yield a working product. The second is to show how this model can be used to define an "ideal" project testing environment, containing what we see as a useful set of abilities for project personnel. The model we define is sufficiently flexible as to not interfere with a software environment's integration mechanisms; thus the environment can incorporate one's favorite testing tools and techniques. Section 2 defines the problem more precisely. Section 3 states a solution to the problem. Section 4 describes an automated version of the solution present in the OVERSEE configuration management tool set [17]. Section 5 gives our experiences using the tools. Section 6 presents conclusions and directions.

2 PROBLEM SPECIFICATION

The problem may be summarized as follows. For a given project, how can the testing process be carried out so that as little time as possible is wasted waiting for people to finish their portion of the work?

Let us define the problem in detail. A project has a set of d developers $D = \{D_1, \ldots, D_d\}$, a set of t testers $T = \{T_1, \ldots, T_t\}$, and m modules $M = \{M_1, \ldots, M_m\}$. Assume that the system is "non-trivial", meaning that d, t and m are greater than one. (The model in this paper may be used on trivial systems, but it has little effect on a project's productivity.)

Each module must be developed, unit-tested, integrated and integration tested, in that order. These terms' definitions vary throughout the literature, so some definitions are necessary. *Development* involves creating or modifying a module. It is done by some developer D_i; each D_i is responsible for a set of modules $DR_i \subset M$. The set $DR = \{DR_i\}$ is the set of all responsibilities for all developers in a project. Elements of DR are not necessarily disjoint; two developers might be assigned to build an Ada package together, for instance. They probably would not share responsibility for the same procedure (at least not simultaneously), but might both want to make changes to the package's specification part.

Unit-testing requires executing a set of test cases that exercise a single module's capabilities. Unit-testing is done by the module's developer. It occurs in isolation from other testing activities; that is, compiling the module and executing tests on it affect no one but the invoker. Compiling module M_i requires some other set of modules MD (possibly empty) on which M_i depends. When a unit-test module is compiled, the modules in MD may be either used or *stubbed*—simulated by simple modules that permit compilation and little more.[1] Unit-testing results in either *acceptance* or *rejection* of a module. If accepted, it may be integrated; if rejected, it must be re-developed.

[1] For the purposes of this paper, stubs and test harnesses (see [9]) accomplish the same goal.

Integration, also done by developers, is the incorporation of one or more unit-tested modules into a set of unit-tested modules. Integration is accomplished according to an integration strategy IS, a relation defining a partial order on the power set of M. $MOD_i < MOD_j$ implies $MOD_i \subset MOD_j$, and means the modules in MOD_i are to be integrated before those in MOD_j. For example,

$$\{\{M_1\} < \{M_1, M_2\}, \{M_1\} < \{M_1, M_3\}, \{M_1, M_3\} < \{M_1, M_2, M_3\}\}$$

means the developers will first integrate the two pairs (M_1, M_2) and (M_1, M_3), then integrate all three modules.

A system's software structure partially dictates the nature of IS. The type of integration used—top-down, bottom-up, sandwich, *etc.*—defines the exact order, and which module is highest. Note that a partially integrated system can exist, but must contain stubs if it is to be compiled.

Integration testing is the testing of a set (possibly with one element) of integrated modules. Whereas unit-testing of a module tests the characteristics of that module (even if it uses other modules), integration testing tests whole subsystems. Testers, not developers, do integration testing, to assure that each part of a system has been validated by someone other than its developer. Testers have responsibilities defined by $TR = \{TR_i\}$; each TR_i is a set that defines the modules T_i is testing. Each element of TR_i is a set of modules. For instance, if $TR_i = \{\{M_1\}, \{M_2, M_4, M_5\}\}$, then T_i currently has two testing responsibilities: module M_1 and the integration of M_2, M_4 and M_5. (Nested sets distinguish the above case from a tester who is independently testing four modules.)

If $TR_i = M$, then T_i is testing the entire system. This is *acceptance testing*, the final step of the testing process.

Integration-testing, like unit-testing, results in either acceptance or rejection. Rejection requires re-development of the offending module(s). Acceptance is more complicated, and is explained below.

We make no assumptions about the amount of integration testing. In the worst case, none will occur. In the best case, testers will test each individual module and each step of the integration process. A common intermediate case is to have the testers evaluate only the entire system, after all modules have been integrated. That is, testers perform only acceptance testing.

Each module has a current *stage* that defines what it must undergo before being part of an accepted system. The stage is one of "development," "unit-testing," "accepted from unit-testing," "integration-testing," or "accepted from integration testing." Stages are cyclic: a rejected module reverts back to development, and must be unit-tested again.

The sets DR and TR evolve quite differently throughout the testing process. A developer usually has responsibility for a module throughout the testing process, so DR is fairly static. Also, overlapping elements of DR are the exception rather than the rule. Neither of these two properties hold for TR. The following are reasons why:

1. Testers often share responsibilities. Several testers are usually assigned to a complex module, for instance. Hence elements of TR may overlap.

2. Assuming the "best-case" integration-testing mentioned above, each element of TR will initially be some subset of M, and all elements of TR will be disjoint. For a tester to test the entire, integrated system, the set of modules he is testing must equal M.

3. Usually, several people test the entire system. Thus, once the system is ready for acceptance testing, several elements of TR may be equal.

4. Unlike developers, testers do not necessarily see a module through from start to finish. The following scenario is more common. Suppose tester T_a is responsible for M_i, and tester T_b for M_j; once they finish, the integration strategy requires that M_i and M_j be integrated, and the result will be tested by T_c. Thus, initially $TR_a = \{\{M_i\}\}$, $TR_b = \{\{M_j\}\}$, and $TR_c = \emptyset$; afterward, $TR_a = TR_b = \emptyset$ and $TR_c = \{\{M_i, M_j\}\}$.

5. When a tester rejects a module, he is not responsible for it until it has passed through unit-testing again. Reason 4 shows that he may not be immediately responsible for it even then.

Therefore, DR specifies responsibilities that exist throughout a project; TR reflects the stages of each module, and changes throughout the testing process.

Reason 4 implies another important distinction between unit and integration testing: a tester may accept a module from integration testing without changing its stage. Modules change from "integration-testing" to "accepted from integration-testing" when acceptance testing ends. The stage change then occurs simultaneously for all modules in the system. Prior to this, a module can be accepted by a tester but still be undergoing integration testing.

Testers therefore need an integration strategy that follows the developers'. It is given as a set $TIS = \{TIS_1, \ldots, TIS_s\}$. Each $TIS_i = (MOD_i, TST_i)$ where $MOD_i \subseteq M$ and $TST_i \subseteq T$. Each element specifies that, when the system is sufficiently integrated to allow all modules in MOD_i to be tested, the testers in TST_i will have the responsibility. Each MOD must be valid according to IS: if $M_j < M_k$ then TIS may not contain an element (MOD, TST) where $M_k \in MOD$ and $M_j \notin MOD$. In fact, IS can be used to define a partial ordering on TIS as well. This ordering might not be identical to IS, however. The developers might perform a systematic, comprehensive integration, but the testers might opt to test only the entire system. If so then $TIS = \{(\{M\}, T)\}$, and the partial ordering on TIS is vacuous.

Whereas TR reflects the current state of testing, TIS specifies all responsibilities for all testers over the entire testing process. Note that there is not necessarily a subset of TIS that corresponds to a given state of TR. In particular, there might exist an element $(MOD, \{T_a, T_b\})$, naming the two people as responsible

for the modules in MOD; however, T_a might be ready before T_b. If so then TR_a will contain MOD before TR_b does.

The testing process can be defined as transformations on module stages and TR, as constrained by the integration strategies IS and TIS. For example, given:

$$\begin{aligned} M &= \{M_1, M_2, M_3\} \\ D &= \{D_1, D_2\} \\ DR &= \{\{M_1, M_2\}, \{M_3\}\} \\ IS &= \{\{M_1\} < \{M_1, M_2, M_3\}, \{M_2\} < \{M_1, M_2, M_3\}\} \\ T &= \{T_1, T_2, T_3\} \\ TIS &= \{(\{M_1\}, \{T_1\}), (\{M_2\}, \{T_2\}), (\{M_1, M_2, M_3\}, \{T_2, T_3\})\} \end{aligned}$$

then the following is a reasonable way to conduct testing. Before testing begins, $TR = \emptyset$, and all modules are in "development". During unit-testing, D_1 takes M_1 and M_2, and D_2 takes M_3, through the "unit-testing" and "accepted from unit-testing" stages. Next, integration-testing begins. This moves the modules to the "integration testing" stage, and makes $TR = \{(\{M_1\}, \{T_1\}), (\{M_2\}, \{T_2\})\}$. After T_1 and T_2 complete their tests, $TR = (\{M_1, M_2, M_3\}, \{T_2, T_3\})$. After T_2 and T_3 completed these tests, the modules move to the "accepted from integration testing" stage.

A project has the responsibility to see that the transformations occur in an orderly way, such that each module is available for integration in a timely fashion. In practice, precise specification of the sets prior to testing is virtually impossible. The most obvious reason is that the sets D and T are changeable, due to such mundane realities as vacations and personnel changes. Moreover, unexpected difficulties in development may cause delays that necessitate, if not major rewrites of, then at least minor alterations to the integration strategy.

Effecting these transformations is a challenging task. It is typically done *ad hoc*, with information on stage recorded informally and transmitted to other people orally. Another common solution is to delay stage changes until all modules are ready. Neither solution is satisfactory. The first is error-prone, and the second is slow.

The discussion so far has concerned changes in stage. The real problem, however, is making the changes known and available to other people. Testing "in the large" should not only verify a subsystem, it should make a module part of a configuration where it may be utilized (or tested) by others. The testing process aims to systematically combine all modules into a working product; this cannot occur unless those people involved in each part of the process are made aware that the modules they require are ready and accessible.

These problems are termed *information flow* problems. They arise from the difficulties in transmitting information about project status to all concerned parties (see [17]). Designating a module's stage as "accepted from unit-testing" is not enough. Certain people need to be made aware of the change in stage. In other

words, information must "flow" from party to party.

The aforementioned delays in software development often result from failure of information flow. In an environment lacking support for it, the flow is slow and uncertain. It may evolve into routes requiring intermediate parties (*i.e.*, people hear things only through their managers). Automated information exchange using electronic mail is quicker and more reliable. Even mail, however, is error-prone: someone must remember to send it, it must be addressed to the correct people, and must be sent in a way that assures the most timely delivery.

3 A SOLUTION TO THE PROBLEM

This section presents a solution to the problem defined in section 2. The presentation describes what we consider an ideal set of abilities that an environment should provide during the testing process—the full powers people would like to have at their command.

The following initial conditions hold. First, some suitable assignment of values exists for the sets D, T, M, DR, IS and TIS; in this paper, we assume that these sets do not change. Second, $TR = \emptyset$ (since testers do not have responsibilities until they are actually assigned a module).

Every module must pass through the stages listed in section 2. Each of the following subsections describes actions needed to move a module into the next stage (or reject it and send it back to development). These actions are constrained by two factors: the module's stage, and the stage of other modules that the module requires. Each subsection also describes the consequences of each action, in terms of the information flow that should result from the action, and the transformations on TR. Some of these consequences are necessary for the testing process to proceed. Many, however, are what we feel users of our idealized environment would want to proceed most effectively. The consequences are distinguished by the use of "must" versus "should", respectively.

3.1 Beginning Unit-Testing

Since unit-testing is directed towards an individual module, the action of beginning unit-testing is largely independent of other actions. A developer who is not using stubs must wait until the required modules are available, but that describes a constraint on, rather than one imposed by, the action. However, unit-testing cannot commence unless one's unit compiles. Therefore the action depends on knowing that other modules are sufficiently developed. Section 3.2 describes how this happens.

The action has only one required consequence: the module's stage must change from "development" to "unit-testing". If exactly one developer is responsible for that module, then no one else need know. However, if several developers are responsible for it, they should be made aware of the action and prevented

from making changes until the testing has ended. Also, other people may wish to know that the module is close to being accepted from unit-testing. For scheduling purposes, then, the action should notify all testers assigned to the module, and all developers whose modules require it.

3.2 Unit-Test Acceptance

Suppose a module is in the "unit-testing" stage, and has passed all the tests its developer devised. It is then ready to be accepted from unit-testing. It is not ready to be part of an accepted system, because it has not yet undergone testing by an independent party (*i.e.*, integration testing). However, other developers might require it, or at least a portion thereof, to build their own software. This is a typical problem of concurrent development [1]: since software modules built in parallel must ultimately merge, they should be available to everyone as quickly as possible. The issue is how soon that can be. Many projects allow developers to informally share code at any time, which is risky. Untested code is often too bug-ridden to be useful, and is subject to unannounced changes. Alternately, the developers could wait for a module until it is accepted from integration testing, but that is often overly restrictive and generally necessitates long delays.

Our view is that a module should be available as soon as it is accepted from unit-testing. Such a module has been subjected to at least some testing (by definition) and, while it has not been independently reviewed, can at least be considered useful, "caveat emptor". In practice, we have found this approach quite desirable. It lets developers access a module at the earliest possible time when that module's developer has some confidence (as established by his unit-tests) that the module is correct. Moreover, it eliminates most careless errors. Even if the module still contains bugs, it is compilable (an uncompilable module cannot pass unit-testing), which is usually what other developers need to continue their own work. For instance, an Ada programmer might need to use a package via a `WITH` clause. That person cannot compile his module without the package's specification. Thus he needs the specification, in a reasonably stable configuration, as soon as possible. Furthermore, the person can accomplish a reasonable amount of testing on his own module even with a slightly flawed implementation of the package, either by testing areas not related to the package or by creating stubs. A flawed implementation is undesirable, but can still be useful. Also, it should be expected. Few modules are 100% correct even after several rounds of testing.

A module accepted from unit-testing should therefore be in a central area, accessible to all developers, but removed from their own areas. This area is called the *mini-environment*. Developers and testers reference others' work exclusively through this area, not by direct access to other developers' areas. This separation is important during integration testing, as section 3.6 shows.

A module that is ready for unit-testing can be developed before the modules it requires are fully ready. If so it can be tested with stubs. It could even be accepted from unit-testing with stubs; this is once again the principle of making software available early. However, other developers making use of it should be aware

that it contains stubs. Also, as the modules on which it depends are accepted from unit-testing, they can be substituted for the stubs. This process follows the system integration strategy.

The action of accepting a module from unit-testing requires notifying the following people:

1. Developers who will be using the module.

2. The testers assigned to perform integration testing on the module.

3.3 Unit-Test Rejection

Under the above scenario, a module's unit test can fail due either to a problem in the module or in a module that it uses. If the former, the module must be rejected and sent back for modifications to correct the problem. After doing so, its developer may begin unit-testing it again, repeating the cycle until it is ready to pass. No change to the mini-environment takes place until the module is accepted, which does not happen until until the developer has some confidence that it is correct.

If the problem is in another module, then this module must be in the mini-environment, meaning its developer has accepted it from unit-testing. The action to be taken depends on who is developing the faulty module. If the same person is responsible for both, he must withdraw the faulty module from acceptance and fix the problem. If it is someone else, that person must be notified.

All developers using the faulty module should be made aware of the flaw. Sometimes—if the fault renders a system unusable, and a previous version is available, for example—replacing the version in the mini-environment with a previous version might be desirable.

Because developers' responsibilities do not change, the first three operations have no effect on the sets of section 2. Only the module's stage changes.

3.4 Beginning Integration Testing

Testers need capabilities similar to developers; however, the differences in how they work alters the manner in which they communicate. These differences are as follows. First, developers unit-test individual modules, but testers may test individual modules or groups of related modules; they may also have responsibility for several unrelated modules. Second, testers need to be able to inherit responsibilities from other testers. Third, testers may inherit responsibilities piecemeal. If a tester has to test the integration of ten modules, and if these tests involve code reviews, he will want to begin reviewing each module as soon as it is available.

Tester T_i may begin integration-testing module set MOD_j only if the following two conditions hold:

1. There exists $(MOD_j, TST_j) \in TIS$, where $T_i \in TST_j$, and $MOD_j \notin TR_i$.

2. If the partial ordering on TIS defines $(MOD_k, TST_k) < (MOD_j, TST_j)$, then all testers in TST_k have accepted MOD_k.

Beginning integration testing sets $TR_i \leftarrow TR_i \cup MOD_j$, and changes the stage of modules in MOD_j to "integration testing" if they were "accepted from unit-testing". Furthermore, it requires communication. A module's developer should know that a tester has started examining his code. Other testers assigned to the module should be notified that their partners have begun.

3.5 Integration-Test Acceptance

When a tester accepts a module (or modules) from integration testing, he adds it to the set of modules that constitute the partially integrated system. The module must then be integrated with other modules, and new test cases must be prepared and run. Only when a system passes acceptance testing can a module be accepted; at that point, all modules are accepted simultaneously. Note that multiple testers may be assigned to a set of modules. If so, acceptance does not occur until all have completed their testing. That is, given some $(MOD_k, TST_k) \in TIS$, all testers in TST_k should accept all modules in MOD_k before TR changes.

Integration test acceptance requires notification of the people who will be responsible for the next round of tests. Also, the developers of the modules should be informed that their code is, so far, adequate.

The transformations are as follows. If $MOD_k = \{M_1, \ldots, M_m\}$, the entire system is accepted; all modules become "accepted from integration testing", and $TR \leftarrow \emptyset$. Otherwise, TR is transformed according to TIS; testers in TST_k are relieved of responsibility for MOD_k, and modules in MOD_k become available for further integration. More precisely, for each i such that $T_i \in TST_k$, $TR_i \leftarrow TR_i - MOD_k$. New responsibilities are not assigned until testers begin their next round of integration testing, as discussed in section 3.4.

This definition ignores one detail: the transformation to TR does not occur until all testers in TST_k have accepted all modules in MOD_k. Synchronizing that is an implementation problem that does not affect the sets. It does affect notification, however. Each acceptance prior to the final one should notify the developers of the modules, and the testers in TST_k and TST_n, so that they are aware of the progress being made. The final notification should inform these people that a new phase of work may begin.

3.6 Integration-Test Rejection

Rejection sends a module back to development. An important feature of rejection from integration testing is that it can apply to individual modules as well as groups of modules. If a tester is testing a set of three modules, and finds a problem, then there is no reason to reject all of them if the error can be localized. However, if the error cannot be localized, all modules may have to be rejected.

Rejection differs from acceptance in that a single rejection stops the integration testing. This does not

mean that all integration testing must stop, only that the current version of the rejected module(s) cannot be accepted. The re-developed version must undergo unit-testing again (remember, this is a statement that its developer has verified the correction). Ideally, it should also be subjected to the same integration-testing steps; this may only be practical if regression-testing capabilities are present, however.

Rejection requires, for each rejected module, notification of the module's developer. All testers currently responsible for the module should also be alerted, as should those testers who will next inherit the module.

Rejection transforms TR by deleting the module(s) from the appropriate tester responsibility sets. If a tester rejects all modules for which he is responsible, the entire integration-testing process up to that stage must be repeated. If he rejects a proper subset of his modules, however, only those modules need such treatment.

4 OVERSEE TOOLS FOR THE TESTING PROCESS

The capabilities presented in section 3 provide a solid foundation for an automated set of tools to control the testing process. Most of the capabilities currently exist in the OVERSEE configuration management system. This section describes the tools that provide them, and their implementation.

The tools' user interface is straightforward; the capabilities can be provided with only five commands, some permitting small variations in their arguments. Figure 1 shows the command syntax. They are UNIX shell-level commands, although flags are words, rather than single characters, for readability. The meanings are as follows. First, there is no explicit command to initially place a module in development; that is its stage when created (an OVERSEE command sequence not covered here). A developer may then execute the `unit-test` command, and perform his tests. After doing so, he can `accept` the module. An accepted module is automatically installed in the mini-environment, and those developers and testers who require the module are notified (notification in OVERSEE is via electronic mail). He can also `reject` the module, correct problems, and run `unit-test` again. This command also notifies those concerned, so they may have an idea of the project's status.

If a developer has inserted stubs into the module, he can use the `-stubs` flag with the `accept` command. This installs a stubbed version, and people are explicitly informed that the version in the mini-environment is not fully functional. As modules become available to replace the stubs, the developer re-executes `accept`, naming those modules; his module is re-installed, without the stubs, in the mini-environment. Note that this technique shows how the final system is built. Whoever is responsible for the main program of the system is ready to build the full system whenever all subordinate modules (and their subordinates) contain no stubs. The acceptance process defines this precisely.

Once a module is accepted, a tester can run `intr-test` to begin integration testing. He may simultane-

Developer Commands	Tester Commands
`unit-test`	`intr-test module [ module ... ]`
	`intr-test -add module [ module ... ]`
`accept`	`accept`
`accept -stubs`	
`accept module-name [ module-name ... ]`	
`reject`	`reject [ module ... ]`
`withdraw`	

Figure 1: OVERSEE Command Summary

ously integration-test several modules, by giving several module names on the command line. in addition, if he is assigned to test a group of modules, he may begin testing one module and then use the `-add` flag to add them as they become available, or as he feels he is ready. He may then **accept** the modules; however, the current implementation does not support inheritance, so acceptance can be done only to the full set of modules. (So far, people have handled this either by hand-simulating inheritance or sharing accounts.) He may also reject all the modules, or any subset thereof.

Sometimes a developer will discover a problem in a module he has already accepted. If so then he may execute the `withdraw` command. The module reverts to the development stage, where the developer may correct the error. The command notifies other developers who are using the module, and to testers who are testing it.

Each tool operates on a module or set of modules. In some contexts, the module is known; in others, it must be named explicitly. The rules to achieve this are as follows. First, a developer using OVERSEE always has a "current module" determined by the context in which he is working. It is on this module that a developer's commands always operate. Second, a tester who is integration-testing a set of modules may opt to simultaneously reject all of them. If so then the modules need not be named.

Each tool has three purposes. First, it verifies that the operation requested is legal. Second (assuming the operation *is* legal), it changes the stage of the module(s) on which it operates. Third, it announces the change to all concerned parties. A fourth purpose (performed only by **accept**) is to install software in the mini-environment. A fifth purpose would be to support the integration strategy, but the current implementation does not do so directly. (Although one may be implemented using a technique known as policies [17].) Partly due to this, the current implementation also takes the safe but sure method of broadcasting announcements to all testers and developers, rather than select groups. sizable

5 EXPERIENCE

The OVERSEE testing tools were recently used in the development of a small database accounting system, consisting of nine modules, which when completed contained approximately four thousand executable lines of code written in the C programming language. The system was written as the project for a graduate course in software engineering. The students started with a design written in Ada PDL [12], and had two weeks to convert it into C and test the result. Eighteen students participated in the project. Ten of these were developers, and eight were testers. The two groups were egoless teams [18]; the usual warnings of a six-person maximum [2] were deliberately ignored to induce large amounts of communication.

The model of the testing process proved both feasible and useful. Despite the usual constraints of an academic project—limited time and students not able to devote their complete attention to the matter—the system was operational (although not fully robust) within the required time. The testing process occupied about one week. During that time, each module went through several rounds of formal unit and integration testing. Also, the testers were able to verify (or more often, invalidate) the developers' integration at each step. This represents excellent productivity on everyone's part. It was possible in part because little time was wasted, due to the techniques and tools discussed in this paper.

A sizable part of that productivity stems from the incorporation of communication mechanisms into tools. Having the tools change module stages without notifying anyone would have been far less successful. Some people worked off-site (one was 100 miles away) and so were inaccessible except through electronic mail. People saw the automatic notification facilities of the OVERSEE tools as an invaluable aid. The current OVERSEE implementation delivers mail to everyone, however, and people quickly complained of information overload. They had to read about 50 messages each day, many of which were irrelevant to them. Thus we feel that the notification scheme described in section 3 is the correct approach. Manually routing mail to a subset of project personnel would be too error-prone. Having the tools perform this function is far surer.

The project used object-oriented development [7]. Each module therefore had a separate specification and implementation. To make the specification available early, we devised a system where the two parts where separate submodules. This worked well but was not conceptually desirable. It suggests that unit-test acceptance should be applicable to entities other than complete modules. This is an area we are currently exploring.

When the project began, we had not appreciated the need for tester groups. It was quickly revealed as testers waited for modules that were supposed to be ready. Since OVERSEE does not currently support tester groups, we shared user accounts. Based on the resulting interactions between testers, we modified the model to that described in section 3. We have not yet automated this approach, but believe it accurately reflects how the testing occurred on the accounts that were used.

6 CONCLUSIONS AND DIRECTIONS

This paper has presented a model of the interactions that occur between developers, testers, and software modules. The model defines a set of abilities granted to project personnel that promote speedy integration and testing of a system, with minimal interference due to communication problems. Previous systems have not treated this problem specifically, adopting information solutions and making simple use of automated tools. However, most of the interaction between people that occurs during software development takes place during testing. Indeed, testing may be said to involve the most complicated interactions of the software process. Special support, such as that defined in this paper, helps solve the problem.

An automated tool set is an essential part of any solution. The tools must blend into the environment, however, so as to not constrain how the rest of the software development process occurs. The OVERSEE tools discussed in this paper are one such set. They separate testing into two parts: managing the testing, and running the tests. By concentrating on the former, they allow the integration mechanisms of the host environment to accommodate any tools that are already present. Note that they automate the configuration management process, by assuming the role of configuration manager. That is, a developer controls, via the `accept` operation, when software is installed in the mini-environment, and does not need anyone's approval to do this; however, to execute `accept`, a developer must have conducted a certain amount of testing. Furthermore, OVERSEE tools assume the responsibility of notification, rather than requiring it as a separate responsibility of project personnel. Determining who to notify is not simple, as section 3 showed. The notification scheme we actually used would not have worked if the project were much larger. Having tools solve the problem relieves people of a large chore.

We have deliberately not discussed what goes on between the `unit-test` and `accept` operations—namely, the construction, execution and verification of test cases. That is not a simple problem, but it is outside the scope of this paper. Note that it may require little amount time in comparison to development and subsequent re-development, especially if one has regression testing tools available. OVERSEE currently supports them in a rudimentary form, with functional capabilities similar to those found in SPMS [15]. A developer may specify a set of test cases to be run each time he executes the `unit-test` command. This practically automates unit testing of all rounds following the first, because if all tests succeed, acceptance can follow immediately.

Section 3 assumed that all sets except TR are static—meaning that people never join or leave a project. Such changes cannot be formally predicted, but are not problematic. They do, however, have information flow implications. New people need to be made aware of their modules' stages, and other people need to know of the change. This should probably be automated as well.

An interesting possibility that was not covered in this paper is using the model of section 2 to analyze the desirability of a given allocation of testers. We are currently investigating an algorithm that, given M,

IS, and the complexity of each module in *M* as inputs, produces as its output an assignment to *TIS*.

We are also enhancing OVERSEE to fully support the model of section 3. This is part of a larger project to understand the information flow that occurs in software development environments. Information flow complexities are particularly apparent in testing, and have provided one on the most interesting areas of study. A formal definition of information flow helps in evaluating an environment, and the productivity that is possible within it.

References

[1] Mikio Aoyama, "Concurrent Development of Software Systems: A New Development Paradigm", *ACM Software Engineering Notes*, 12(3):20–24, July 1987.

[2] F. Baker, "Chief Programmer Team Management of Production Programming", *IBM Systems Journal*, 11(1):56–73, 1972.

[3] Boris Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, New York, NY, 1983.

[4] Edward Bersoff, "Elements of Software Configuration Management", *IEEE Transactions on Software Engineering*, SE–10(1):79–87, January 1984.

[5] Barry Boehm, Maria Penedo, Arthur Pyster, Don Stuckle, and Robert Williams, "A Software Development Environment for Improving Productivity", *Computer*, 17(6):30–44, June 1984.

[6] Susan Brilliant, *Testing Software Using Multiple Versions*, PhD thesis, University of Virginia, Charlottesville, VA, September 1987.

[7] Brad Cox, *Object-Oriented Programming*, Addison Wesley, Reading, MA, 1986.

[8] Richard Fairley, "Ada Debugging and Testing Support Environments", *SIGPLAN Notices*, 15(11), November 1980.

[9] Richard Fairley, *Software Engineering Concepts*, McGraw-Hill, New York, NY, 1985.

[10] Richard Fairley, "Static Analysis and Dynamic Testing of Computer Software", *Computer*, 11(4), April 1978.

[11] Ferdinando Gallo, Regis Minot, and Ian Thomas, "The Object Management System of PCTE as a Software Engineering Database Management System", in *Proceedings of the Second Symposium on Software Development Environments*, pages 12–15, December 1986.

[12] Hal Hart, "Ada for Design: An Approach for Transitioning Industry Software Developers", in *Proceedings NSIA Software Group Conference*, Alexandria, VA, October 1981.

[13] Patrick Hawley, "DACOM: A Design and Configuration Management System", in *COMPSAC'83*, pages 580–587, Chicago, IL, November 1983.

[14] Harlan Mills, Richard Linger, and Alan Hevner, *Principles of Information System Analysis and Design*, Academic Press, Orlando, FL, 1986.

[15] Peter Nicklin, "The SPMS Software Project Management System", in *Unix Programmer's Manual (4.2 Berkeley Software Distribution)*, University of California, Berkeley, CA, 1983.

[16] Leon Osterweil, "Toolpack—An Experimental Software Development Environment Research Project", *IEEE Transactions on Software Engineering*, SE–9(6):673–685, November 1983.

[17] Steven Wartik, "Rapidly Evolving Software and the OVERSEE Environment", *SIGPLAN Notices*, 22(1):77–83, January 1987.

[18] Gerald Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinhold, New York, NY, 1971.

[19] Sandra Zucker, "Automating the Configuration Management Process", in *SOFTFAIR*, pages 164–172, Arlington, VA, June 1983.