Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution

Gail E. Kaiser*
Columbia University
Department of Computer Science
New York, NY 10027
(212) 280-3856

Dewayne E. Perry
AT&T Bell Laboratories
Murray Hill, NJ 07974
(201) 582-2529

November 1986

Abstract

We introduce and compare two models of cooperation among programmers during software maintenance. Enforced cooperation is the normal mode of operation when the sheer size of the software maintenance effort makes *laissez-faire* management infeasible. Voluntary cooperation is more common when a small group works together to enhance a small system or modify a small portion of a large system. We describe a tool, Infuse, that provides change management in the context of both models of cooperation. We demonstrate how Infuse automates change propagation and enforces negotiation of conflicts for the enforced model, but provides less restrictive aids for maintaining consistency for the voluntary model.

* Supported in part by a grant from the AT&T Foundation, in part by a grant from Siemens Research and Technology Laboratories, and in part by a Faculty Award from Digital Equipment Corporation.

1. Introduction

The maintenance and evolution of large systems requires cooperation among multiple programmers, but there are very few models and corresponding tools to support this cooperation. We introduce and compare two models of cooperation among programmers doing software maintenance: enforced and voluntary cooperation. We describe a tool, Infuse, that provides change management in the context of both models.

Enforced cooperation among programmers is necessary for software projects involving many (for example, 20 or more) programmers. Maintenance typically consists of a sequence of coordinated, scheduled source code changes each involving a large collection of modules.** Programmers must cooperate to maintain consistency among these modules during the modification and testing process. We present a software engineering environment, Infuse, that automates change management by enforcing such cooperation. Infuse automatically partitions these modules into a hierarchy of *experimental databases*. The partitioning may be determined by the syntactic and/or semantic dependencies among the modules or according to the dictates of project management. Each experimental database provides consistency checking among its modules and provides a forum for the programmers assigned to its modules, or their managers, to negotiate the interactions among the modules. Infuse requires that the modules within an experimental database be consistent with each other before permitting these modules to be merged back into the parent experimental database. This process repeats until the entire collection of consistently changed modules is merged back into the new release of the software system.

Voluntary cooperation is possible for projects involving a small number of programmers (for example, less than 10). Maintenance typically consists of partially ordered sequences of changes made by individual programmers to a small collection of modules, where additional modules are dynamically added to each collection as the programmer deems necessary. Cooperation is required only when multiple changes are made by different programmers at the same time. Infuse automates this more relaxed form of change management by providing a *workspace* in which the programmer makes his changes. Individual workspaces are combined if the programmers choose to coordinate their changes; we are concerned in this paper only with these combined workspaces. Each workspace provides consistency checking among its modules and thus aids the programmers in making sure the changes are consistent with each other. Infuse also aids the programmers in merging their changes back into the baseline version of the software system.

---

* Supported in part by a grant from the AT&T Foundation, in part by a grant from Siemens Research and Technology Laboratories, and in part by a Faculty Award from Digital Equipment Corporation.

** The work described in this paper is independent of any particular programming language. For example, the term "module" applies to both assembler source files and Ada packages.

Infuse supports both experimental databases and workspaces, and both may be used for the same project where warranted. In particular, workspaces may cut across the divisions imposed by the experimental databases to permit flexible modes of cooperation among programmers.

We begin describing the problems of change management during software maintenance. We present Infuse's automated support for enforced and voluntary cooperation and explain how Infuse automates change management using existing programming tools. We compare Infuse to related systems, none of which support an enforced model of cooperation. We conclude by listing our contributions.

2. Changes to Software Systems

Cooperation is required when source code changes to a software system involve the efforts of multiple programmers. In a medium-to-large software project (involving at least 100,000 lines of source code), there are several reasons why maintenance activities are rarely carried out by a single programmer. The sheer volume of the effort may be prohibitive. There may not be any single programmer with knowledge about the modules and the system sufficient to make the changes correctly and rapidly. There may be administrative considerations that make it inappropriate for one programmer from one managerial unit to modify the modules assigned to another programmer reporting to a different manager.

Instead, a group of programmers normally cooperates on a change that involves several modules, particularly a change that cuts across subsystem boundaries. When changes are being made in several modules, it is necessary for the changes to be consistent with each other. The changes must be *syntactically* consistent [15], meaning that the interfaces between the modules must be correct and that the modules can compile and link successfully. Consider the case where module M exports procedure *p*, where procedure *p* has two formal parameters of types *t1* and *t2*. Module N imports procedure *p* from module M and defines procedure *q* that calls procedure *p*; the call is made with two arguments, of types *t1* and *t2*.

The change to this software system includes modifying procedure *p* to add a third parameter, of type *t3*. This change requires that module N also be changed, so procedure *q* calls *p* with a third argument, which must be of type *t3*. It is necessary for the programmer responsible for module M to communicate this change to the programmer responsible for module N and for the two programmers to coordinate their changes and make sure they are consistent.

When changes are made in several modules, it is also necessary for the changes to be *semantically* consistent [9]. Consider the case where the new parameter of type *t3* is a pointer to a buffer. Procedure *p* assumes that procedure *q* has allocated space for this buffer before calling *p*. If this has not been done, then procedure *p* will cause a run-time error when it attempts to write into the buffer implied by its third argument. Assumptions of this kind are not normally reflected in the interfaces among modules, and thus cannot be detected by the compiler linker. It is therefore particularly crucial for the programmers involved to communicate their assumptions and cooperate on making the desired changes in the software system.

Infuse provides the basis for automating two kinds of consistency checking, change simulation and change propagation; both may be applied with either syntactic or semantic consistency. A programmer may request *change simulation* to check for any interface errors [10] introduced by his most recent changes without immediately notifying other programmers of these changes. This permits a programmer to reconsider a change that has undesirable effects on other modules. Change simulation may be performed with respect to only the modules in the programmer's workspace, with respect to only the modules in the programmer's experimental database, or with respect to a baseline version of the software system.

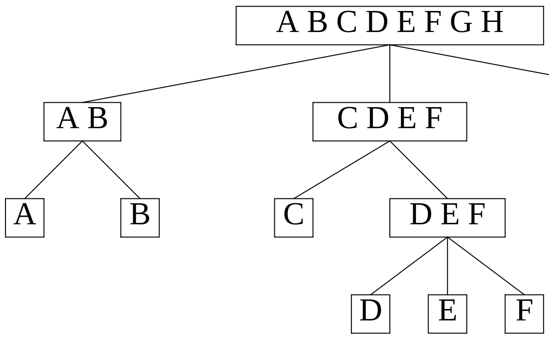
Infuse automatically performs *change propagation* under certain circumstances, discussed below. The tool checks for interface errors among the modules in a workspace or among the modules in an experimental database. For each interface error found, Infuse notifies the programmers assigned to the relevant modules and informs them of the change(s) that introduced the error.

There are three important distinctions between change simulation and change propagation. The first is that Infuse propagates changes automatically while it simulates changes only in response to a programmer's request. The second difference is that Infuse can simulate any change, but propagates only 'committed' changes. The third distinction is in the way that Infuse reports inconsistencies. It reports errors found during change propagation to every programmer whose module was involved in the inconsistency; this permits the programmers to negotiate further changes with full knowledge of any problem areas. In contrast, it reports errors found during change simulation only to the individual programmer who initiated the analysis; this enables the programmer to back out of uncommitted but troublesome changes without entering into negotiation with other programmers.

3. Experimental Databases

An experimental database contains a collection of reserved modules. Modules are placed in a particular experimental database according to some criteria that partition the collection of modules involved in a change. This normally occurs when a planned change (or set of planned changes) in the software system requires coordinated changes in a set of modules. The programmers responsible for these modules are automatically associated with the experimental database.

Experimental databases are hierarchical. The top-level experimental database consists of a virtual copy of the entire software system. The second level experimental database consists of only those modules involved in the change. This database is then partitioned into child experimental databases, where disjoint subsets of the modules are placed in each child. The partitioning is repeated recursively until the experimental database at each leaf contains one or more modules assigned to the same programmer. We refer to a leaf as a ‘singleton’ experimental database. An example hierarchy (without the base level) is illustrated in Figure 1.



This software system consists of modules A through Z, so the top-level experimental database (not shown) contains each of these 26 modules. Modules A through H are involved in the change, so the second-level experimental database contains copies of A, B, C, D, E, F, G and H. The third level of the hierarchy consists of three experimental databases; the first contains A and B, the second C, D, E and F, and the third G and H. The second of these experimental databases is further subdivided into two experimental databases at the next level. The leaves of the hierarchy each contain a single module.

Figure 1: Hierarchy of Experimental Databases

The hierarchy of experimental databases may be partitioned manually to reflect the management hierarchy of the software maintenance team. Another possibility is to partition, again manually, according to the subsystem organization of the target software system. However, both alternatives often result in higher communication and propagation costs than other methods of partitioning that reflect the interconnections among the modules.

We have developed an approach to partitioning based on the strengths of the syntactic and semantic dependencies among the modules [11]. This approach attempts to minimize the communication and propagation costs, and has the additional advantage that it can be automated. An optimal partitioning would place together in the same subtree of experimental databases those modules whose new changes will interact most strongly. Unfortunately, it is not possible to determine an optimal partitioning *a priori* because the full costs cannot be determined until after the changes have been completed. We propose strengths of dependencies as a good approximation.

The partitioning tool must be embedded within Infuse. However, it is not necessary for an experimental database to interact with the tools used by the programmers to modify their modules. Infuse places no requirements of any sort on these tools. When the programmer completes the changes to his module (or small set of modules), the programmer must give the **deposit** command to notify the experimental database that his part of the overall change is complete.

Infuse invokes an analysis tool to determine whether the module is self-consistent. Self-consistency within a module means that no errors are detected during lexical analysis or parsing and that any symbols that are both defined and used internally are used correctly with respect to their definitions. For example, the front-end of a compiler is easily modified to check syntactic consistency only with respect to a particular module, ignoring consistency with other modules. The deposit of a singleton experimental database is not permitted unless the module is self-consistent.

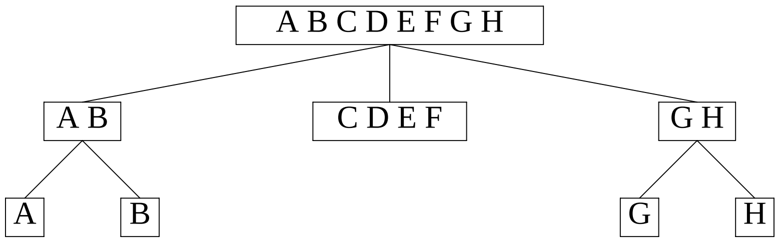
After all the modules in an experimental database have been deposited, Infuse invokes an analysis tool to propagate changes. The tool checks that the modules in the database are consistent among themselves. Intermodule consistency requires that any symbol defined by one module and used by another is used correctly with respect to its definition. Modules outside the experimental database are not considered. It is not difficult to include external modules in the analysis, but this leads to an information overload and communication difficulties as many error messages are generated due to temporary inconsistencies and other transient conditions. The goal of experimental databases is to limit change propagation to a manageable level.

Infuse checks for three possible results. The first is that the modules are consistent; Infuse automatically deposits the entire experimental database into its parent experimental database. Once all child experimental databases have been deposited, Infuse then analyzes the parent experimental database as it analyzed each of the children.

The second possibility is that the modules are consistent, but some of the modules in the experimental database must be recompiled due to changes in imported modules. Infuse marks these modules for recompilation, and then deposits the entire experimental database into the parent database as above. It is generally best that recompilation not proceed immediately, since inconsistencies involving these modules may be detected at a higher level of the hierarchy.

The third possibility is that the modules are inconsistent; Infuse informs all the programmers associated with the experimental database about the conflicts. It provides precise information about the localized conflicts for each affected module and appropriate instructions for resolving the conflicts. The programmers or their managers must now negotiate changes to their modules that will resolve the conflict. Infuse does not permit a deposit of the experimental database into its parent until the modules in the experimental database are consistent among themselves.

The relevant programmers choose some subset (perhaps all) of the conflicting modules for an additional set of changes to remove the inconsistencies. This group of modules is then recursively partitioned in the same manner as the original partition into a hierarchy of experimental databases. The root of this new hierarchy is the experimental database in which the inconsistencies were detected, so this new partitioning does not affect the rest of the original hierarchy. This process is illustrated in Figures 2 and 3.



Modules D, E and F were changed and deposited into the experimental database containing only D, E and F. Infuse invoked the analysis tool, which determined that there were no inconsistencies among these three modules. The experimental database was then deposited into its parent, which contains C, D, E and F. Module C had also been changed and deposited into this experimental database.

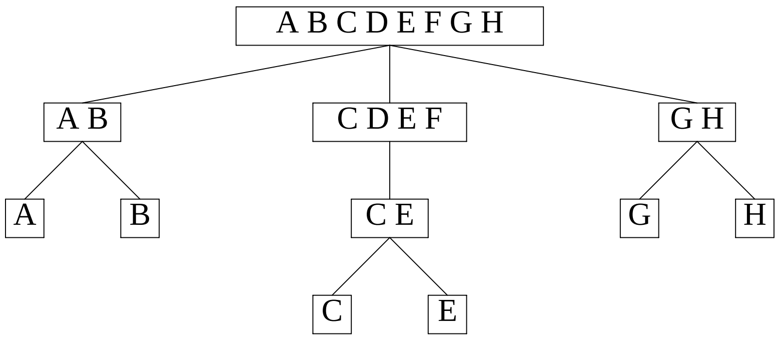
Figure 2: Deposit into Parent Experimental Databases

Then the resolution process repeats as if there had been no previous resolution. The modules in the leaf experimental databases are modified again, and deposited again into their parent. If there are new conflicts, or if previously detected inconsistencies have not been correctly resolved, then additional negotiation and change must follow.

When part of a hierarchy is deposited, inconsistencies may be detected, requiring repartitioning of the subhierarchy. We call this the ‘yo-yo’ effect. When the yo-yo effect occurs frequently, particularly within the same portion of the hierarchy, this usually reflects poor communication or a lack of cooperation among the participating programmers. Infuse can be set up to monitor instances of this problem and to notify appropriate managers of the problem.

3.1 Tool Integration and Change Simulation

We have discussed how Infuse operates when the programming tools are not integrated



Infuse again invoked the analysis tool, finding a conflict between modules C and E and between modules C and F. Subsequently, the programmers chose to modify C and E (but not D or F) to remove the inconsistency. Infuse creates a new child experimental database containing C and E, which is further partitioned so the leaves of the hierarchy again contain a single module.

Figure 3: Repartitioning into Subhierarchy of Experimental Databases

into Infuse, and each programmer must explicitly notify Infuse when his changes are complete by issuing the **deposit** command. This works well for change propagation, but difficulties arise when we apply it to change simulation. The goal of simulating changes is to inform the programmer of any conflicts caused by his changes before the changes have been deposited, while the programmer is still in the context where the change is made. To do this effectively, each modification tool must be integrated into Infuse so that the **simulate** command can be invoked from within the tool.

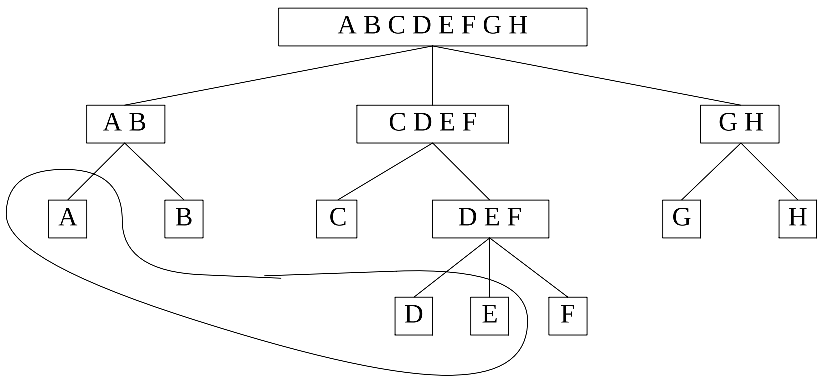
This is a rudimentary level of ‘integration’, but it has the significant advantage over non-integration that the programmer can try a small change and immediately find out how it affects the modules in the parent experimental database. Without such integration, the programmer has to leave the modification tool and perhaps sacrifice its internal state to obtain early notification of any problems.

If we add a further level of integration, then Infuse provides more extensive information regarding any problems. For example, Infuse highlights any errors detected in the part of the source code currently being displayed by the modification tool; if this tool supports multiple windows, Infuse pops up a window to display the conflicting source code from other modules. This makes it easier for a programmer to decide whether he wants to commit the proposed change and, if so, to inform other programmers of the effects on their modules.

4. Workspaces

Integration makes it possible for Infuse to inform programmers immediately of conflicts between their module and other modules in the parent experimental database, but it does not help programmers keep track of simultaneous changes being made to other modules. Sometimes it is important for a small subset of the programmers involved in a large change to get immediate feedback about the interactions among the changes to their modules. In these cases, it is appropriate to propagate changes at agreed intervals, such as on demand or after each checkpoint, rather than waiting until all of the modules have been deposited. However, immediately propagating changes does not solve the whole problem, because partitioning does not necessarily place these modules in the same experimental database. These issues motivated the development of workspaces, which support interactive consistency checking among selected modules. Workspaces can cut across the partitioning of the hierarchy of experimental databases, or can be used alone to support changes involving a small group of programmers.

Like an experimental database, a workspace contains a collection of modules. In general, the modules have been placed in the workspace because the programmers responsible for the modules have decided to coordinate their changes to the software system. Figure 4 shows a workspace that cuts across the divisions imposed by experimental databases.



The most crucial part of the change to the software system involves modules A, D and E, so the changes to these modules must be carefully coordinated. Thus a workspace is defined that contains these three modules and provides the three programmers with immediate feedback about interactions among the changes to their modules.

Figure 4: A Workspace

Unlike an experimental database, the workspace must interact with the tools used by the programmers to modify their modules. The tools may include text editors, syntaxdirected editors, program transformation systems, *etc.* Infuse does not assume any particular technology for making changes in modules. Each programmer can use his favorite collection of tools, or use the standard collection provided at his site, and there is no need for the modules in the workspace to be modified using the same tools, as long as all these tools are appropriately integrated with Infuse.

The minimal requirement is that each tool notify the workspace when a modification session terminates; this involves a minor extension to each tool used within Infuse. Many such ‘sessions’ may be required to complete a change in a module. For example, the end of a session may correspond to a checkpoint, and editing of the same module may continue immediately. A session may involve editing the entire module, or a subunit of the module such as a procedure.

When each modification session ends (or on demand), Infuse invokes an analysis tool to propagate changes. It determines the syntactic and/or semantic changes that have taken place. For example, a theorem prover is an possible tool for semantic analysis. In particular, the Inscape environment [9, 12] provides a notation for specifying the preconditions, postconditions and obligations of each procedure and automatically determines whether all preconditions and obligations have been satisfied by previous and subsequence postconditions, respectively.

Infuse checks for three possibilities. The first is that changes in the module do not affect any other modules in the workspace. In this case, the workspace does nothing except monitor further changes in the module.

The second is that the changes in the module require recompilation of other modules resident in the workspace, but do not require the source code of any of these other modules to be modified. When this occurs, the necessary recompilation is noted for later. There is no need to inform the programmers of such requirements, since Infuse performs these recompilations automatically.

The third possibility is that the changes in the module require source modifications in one or more of the other modules resident in the workspace. This happens when there is a syntactic and/or semantic inconsistency between the changed module and each module that now requires a change. In this case, both the programmer who made the change and the programmer responsible for the affected module are immediately notified of the conflict. Since Infuse propagates changes frequently, it is particularly important to keep the number of modules in a workspace small to restrict the error messages and resulting negotiations among programmers to a manageable level.

4.1 Integration of Analysis Tools

Infuse is most useful when there is a high level of integration between it and the modification tools used by the programmers. For example, notification of new interface errors can take several forms. At the minimal level of integration, Infuse relies on electronic mail messages or simply dumps error messages on the participating programmers' screens. At higher levels of integration, Infuse provides immediate visual feedback and displays the relevant portions of all modules involved in each conflict, enabling the programmers to negotiate changes to their modules as soon as possible.

In any case, the level of detail included in the notification is dependent on the capabilities of the analysis tool(s) used and the level of integration between Infuse and the analyzer(s). The more detail that is provided, the easier it will be for the participating programmers to negotiate changes without confusion. Integration requires that the minimal analysis tool match error messages with textual lines of source code. When the modification tool is a structure editor [4, 13] that shares its internal representation with the analysis tool, this can be improved to attach error messages to portions of the abstract syntax tree.

If the analysis tool is moderately intelligent, in the sense of the intelligent assistants described by Winograd [16], it might suggest how to to fix the problems.

Certain difficulties arise when a conflicting module is involved in several distinct coordinated changes, and thus the module is resident in several overlapping workspaces. Changes made to the other modules in one workspace must not leak to the other modules in other workspaces through the error messages regarding the common module. Infuse prevents this problem by propagating changes to only one workspace at a time. This problem does not arise for change simulation because only the requesting programmer receives the error messages.

5. Related Work

The following tools also propagate changes among modules. However, none support the enforced model of cooperation among programmers necessary for large maintenance projects, since they permit programmers to modify source code at will, in some cases requiring that no other programmer is modifying the same module at the same time. Only one provides an automated forum for change negotiation among programmers.

Make	When requested, the Unix Make tool [3] rebuilds the entire software system. It invokes the tools specified in the 'makefile' on changed files and all files that depend on changed files. Make is normally used to recompile and relink.
Build	The Unix Build tool [2] is an extension to Make that permits various users to have different views of the target software system. A 'viewpath' defines the sequence of directories searched by Make to find the files named in the makefile. Viewpaths are similar to the underlying support for experimental databases.

Cedar	The Cedar System Modeller [7] combines an advanced version of the Make tool with version control. It invokes the tools on selected versions of modules. A distinguished programmer called the ‘Release Master’ is informed of syntactic interface errors between modules; the Release Master is responsible for communicating with the programmers responsible for these modules.
DSEE	The Apollo Domain Software Engineering Environment [8] also combines Make-like facilities with version control. In addition, DSEE provides a monitoring mechanism that allows programmers and managers to request that they be notified when selected modules are changed.
Masterscope	Interlisp’s Masterscope tool [14] automatically maintains cross-referencing information among program units. It approximates change simulation by answering queries about syntactic dependencies among program units.
SVCE	The Gandalf System Version Control Environment [5] performs incremental consistency checking across the modules in its database and notifies the programmer of errors as soon as they occur. The consistency checking is limited to detecting syntactic interface errors. SVCE supports multiple programmers working in sequence, but does not handle simultaneous changes.
Smile	The Smile programming environment [4, 6] introduced the notion of an experimental database, but does not support a hierarchy of experimental databases and does not automate partitioning. It both simulates and propagates changes at the syntactic level, and does not permit a deposit of an experimental database until its modules are syntactically consistent among themselves and with the other modules in the main database. However, Smile restricts each experimental database to a single programmer, preventing multiple programmers from cooperating on changes. We have used Smile extensively, and have drawn on that experience in developing Infuse.

6. Conclusions

The primary contributions of this research in software engineering environments are:

- Change management that supports the specific problems of large numbers of programmers maintaining large systems.
- Integration with facilities specific to small group cooperation within large projects.

Acknowledgements

Yoelle Maarek has developed a clustering algorithm for partitioning the change set into hierarchical experimental databases. Bulent Yener is working with us on the implementation of hierarchical experimental databases as an extension of Smile. David Bellanger and David Hanssen provided careful readings and comments on an earlier version of this paper.

References

[1] David R. Barstow, Howard E. Shrobe and Erik Sandewall, eds. **Interactive Programming Environments**. McGraw-Hill, 1984.

[2] V. B. Ericson and J. F. Pellegrin. *Build — A Software Construction Tool*. **AT&T Bell Laboratories Technical Journal**, 63:6 (July-August 1984), 1049-1059.

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

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

[5] Gail E. Kaiser and A. Nico Habermann. *An Environment for System Version Control*. In **Digest of Papers of the Twenty-Sixth IEEE Computer Society International Conference**, February 1983, 415-420.

[6] Gail E. Kaiser and Peter H. Feiler. **Intelligent Assistance without Artificial Intelligence**. **Proceedings of the Thirty-Second IEEE Computer Society International Conference**, February 1987, 236-241.

[7] Butler W. Lampson and Eric E. Schmidt. *Organizing Software in a Distributed Environment*. In **Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems**, June 1983, 1-13. **SIGPLAN Notices** 18:6 (June 1983).

[8] David B. LeBlang and Robert P. Chase, Jr. *Computer-Aided Software Engineering in a Distributed Workstation Environment.* In **Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments**, April 1984, 104-112. **SIGPLAN Notices** 19:5 (May 1984).

[9] Dewayne E. Perry. *Software Interconnection Models*. **Proceedings of the 9th International Conference on Software Engineering**, March 30 - April 2 1987, 61-71.

[10] D. E. Perry and W. M. Evangelist. *An Empirical Study of Software Interface Faults*. In **Proceedings of the International Symposium on New Directions in Computing**, August 1985, 32-38.

[11] 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**, February 1987, 292-299.

[12] Dewayne E. Perry **The Inscape Program Construction and Evolution Environment**. Technical Report, Computer Technology Research Lab, AT&T Bell Laboratories, August 1986.

[13] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator*. In **Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments**, April 1984. **SIGPLAN Notices** 19:5 (May 1984).

[14] Warren Teitelman and Larry Masinter. *The Interlisp Programming Environment*. **IEEE Computer**, 14:4 (April 1981), 25-34. Reprinted in [1].

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

[16] Terry Winograd. *Breaking the Complexity Barrier (Again)*. In **Proceedings of the ACM SIGPLAN/SIGIR Interface Meeting on Programming Languages — Information Retrieval**, November 1973. Reprinted in [1].

CONTENTS

1. Introduction 2

2. Changes to Software Systems 3

3. Experimental Databases 4
   3.1 Tool Integration and Change Simulation 7

4. Workspaces 9
   4.1 Integration of Analysis Tools 10

5. Related Work 11

6. Conclusions 12
   Acknowledgements 13
   References 14