Coordinating Concurrent Development

William H. Harrison, Harold Ossher and Peter F. Sweeney

IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

ABSTRACT

Development of any large system or artifact requires the coordination of many developers. Development activities can occur concurrently. The goal of coordination is to enhance, not restrict, developer productivity, while ensuring that concurrent development activities do not clash with one another.

This paper presents a formal model of concurrent development, in which development consists of a collection of *modification activities* that change files, and *merges* that combine the changes. We define a notion of consistency called *coordination consistency* that ensures that changes are not inadvertently destroyed and that the changes of each modification activity are correctly propagated to subsequent modification activities. We briefly present a set of protocols for concurrent development using a hierarchy of stores that ensure coordination consistency.

1 INTRODUCTION

Large-scale development, whether of software systems, documents, engineering designs or other such material, requires the coordination of many developers. A key aspect of such coordination is ensuring that the developing artifact remains consistent in the face of concurrent modifications. This increases productivity by permitting developers to work in parallel without fear that their modifications will clash.

Recent work [Pu88] has employed traditional database notions such as serializability to guarantee consistency. However, in the course of large-scale development, developers often examine a great deal of material which provides general background to their work. If this material is treated as "read" from the point of view of serializability, too many conflicts arise to be acceptable. Traditional approaches to librarians, like that in Sun Microsystems' Network Software Environment (NSE)^{TM} [Sun88], employ weaker constraints than full serializability. The character of these constraints has not been precisely described.

This paper presents a formalization of the similar constraints employed in the RPDE^3 librarian [Harr90]. The resulting notion of consistency, called *coordination consistency*, is weaker than serializability but somewhat stronger than that provided by NSE. The paper also briefly presents a set of protocols that ensure preservation of coordination consistency during development.

This paper results from the design and use of support for distributed software development in a heterogeneous computing environment within the RPDE^3 project [Harr89]. An important aspect of RPDE^3 is that program constructs are represented

as objects that are linked together to form a software system. The software system is partitioned into files; however, manipulating an individual file in isolation is wrong since the cross-file links may become corrupted [West87]. For this type of linked representation, coordination consistency is essential to maintaining integrity. The approach we describe is also applicable to domains other than software and to different consistency requirements.

The next two subsections present our model and protocols informally. Section 2 then presents the formal model and Section 3 discusses the protocols. A summary of related work is given in Section 4 and of future directions in Section 5.

1.1 Informal Description of the Model

In our model, an artifact consists of a set of files kept in a *store* called the master store. A *modification activity* is a set of changes, made in isolation in a separate store. Multiple modification activities can occur concurrently, each in its own store. For the changes made during a modification activity to become visible outside its distinguished store, that store must be *merged* with other stores. Ultimately, all changes that are to become part of the artifact must be merged into the master store.

Since modification activities can proceed concurrently, they can modify the same file in different ways in their different stores. This gives rise to *collisions* when an attempt is made to merge the stores. A simple approach to merging, such as choosing the file from either store that has the latest time-stamp, could result in some of the changes being inadvertently destroyed. Version control is not a solution to this problem, though it can help by allowing merges to be delayed. Eventually, however, concurrent modifications to a single version do have to be merged. Configuration management is not a solution either. It deals with specifying the composition of a system and with building and releasing it, but not with handling concurrent modifications to it. Both version control and configuration management are thus orthogonal to the issues discussed in this paper.

Development in our model thus consists of modification activities and merges. Our objective is to ensure the following *coordination-consistency properties* throughout:

1. *Change-serializability.* Any change that results from a modification activity is not overwritten by a merge. This property ensures that when changes occur in parallel, one does not inadvertently supersede the other. A change can be deliberately undone or superseded by another modification activity.

2. *Atomicity.* Either all or none of the changes of a modification activity are involved in a merge.

3. *Completeness.* If the changes of modification activity A are used as the base of modification activity B, then all of the changes from earlier modification activities that were used as the base of A are also used as the base of B. This property ensures that the causal relation between modification activities is preserved.

These properties are defined more formally in Section 2.

Change-serializability is weaker than full serializability in that it makes no statement about files that are examined in the course of making a change, but are not changed themselves. For example, suppose modification activity A changes file f based on

the current details of file g, and modification activity B changes file g concurrently. Even though the changes that B makes to g might invalidate A, the two modification activities are change-serializable. Merging their results will preserve all changes; no work will be lost. The resulting artifact will be wrong, however, and will need to be fixed by another modification activity. Modification activities A and B are not serializable in the database sense, however; full serializability does trap the problem described.

The reason we use change-serializability despite the weakness illustrated above is that during the course of development much material is examined that can nonetheless be changed without adversely affecting the work in progress. In this situation, change-serializability permits much greater concurrency than full serializability, while still ensuring that actual work done is never lost. In practice, most concurrent modifications whose actual changes do not clash are in fact independent. In the example above, the fact that g was examined might have had little or no effect on the details of the change made to f; only occasionally is it of critical importance. Even then, the change made to g by B will often be independent and of no consequence to A. Even when the kind of clash in the example does occur, the necessary fixes can often be made quickly and easily after the fact, perhaps involving no more work or delay than would have been needed to avoid the clash. Only when deeply intertwined modifications are in progress concurrently does change-serializability permit serious clashes. Such modifications require either actual serialization or tight interaction among developers on an ongoing basis. Recent work by Kaiser [Kais90] and by Ellis and Gibbs [Elli89] explores the provision of support for such tight interaction.

We preserve coordination consistency during development by means of protocols that ensure that all merges are "safe" and that disallow extraction of isolated files from stores. Safe merges involve no collisions. Any collisions present when a merge is attempted must be *reconciled* before the merge is permitted. Reconciliation is an activity performed by a developer that it involves detailed and careful integration of all changes in colliding files.

We also provide *locking protocols* that detect potential collisions between concurrent changes when the changes occur, rather than much later at merge time. Locking anticipates an integration between two stores and a lock is a reservation that ensures that a specific file can be merged from the one into the other at a future time. The *strict locking protocol* requires that, before a file is modified, it be locked successfully in all stores into which it might later be merged. This protocol, if universally observed, prevents concurrent modification of the same file by different modification activities, and so guarantees that all merges will be safe. A *lenient locking protocol* warns a developer who is about to change a file that is locked to someone else, but allows development to proceed. This does not compromise coordination consistency, but does introduce the risk that collisions will occur on later merges and will need to be reconciled. It is important to permit this when explicitly desired by a developer to avoid holding up the work.

Use of a locking protocol is not necessary for the system to guarantee coordination consistency. It can be suspended, for example, in a situation in which workstations must run while disconnected from a central librarian. Locking is not discussed further in this paper. Details of locking within our model have been worked out and implemented, and are described elsewhere [Harr90].

2 FORMAL MODEL

2.1 Modification Activities and Consistency

A *modification history* is a sequence of *modification identifiers* that are unique and serve to trace all modifications to a file since its creation. A *file* is a triple (fn, M, c), consisting of a file name, a modification history, and the file contents. The notation $\mathcal{N}(f)$ denotes the file name of the file f.

A newly-created file with null contents has a null modification history. By our definition of a file, all files are immutable. A *modification* is a function from an input file (fn, M, c) to an output file $(fn, M \cdot m, c')$. Note the two restrictions: the file name fn remains unchanged, and the new modification history $M \cdot m$ is the prior one M with a new, unique modification identifier m appended to it.[1]

We say that file (fn, M', c') *is derived from* file (fn, M, c) if and only if M is a prefix of M'. Since M uniquely identifies a file, for convenience we also say that (fn, M', c) is derived from M if and only if M is a prefix of M'. The prefix need not be proper: a file is always derived from itself. As a notational convenience, we extend this definition to sets of files, as follows: A file f' is derived from a set of files F if and only if

$$(\forall f \in F)((\mathcal{N}(f) = \mathcal{N}(f')) \Rightarrow f' \textit{ is derived from } f)$$

A set of files F' is derived from a set of files F if and only if

$$(\forall f \in F)((\exists f' \in F')(f' \textit{ is derived from } f))$$

Thus F' must contain files derived from all files in F; it may also contain additional files.

We define a *file set* to be a set of files such that no two files in the set have the same file name. A *modification activity*, A, is a function from file sets to file sets. We call the argument the initial set, denoted $\mathit{initial}(A)$, and the result the final set, denoted $\mathit{final}(A)$, and require that $\mathit{final}(A)$ *is derived from* $\mathit{initial}(A)$. This models the intuitive notion that a modification activity involves a series of modifications to individual files. The set of files that has actually changed is called the *change set* of the modification activity, and is defined as follows:

$$\mathit{change}(A) = \mathit{final}(A) - \mathit{initial}(A)$$

With this notation, we can now formalize the three coordination-consistency properties as follows:

1. *Change-serializability.* Let A and B be two modification activities. Change-serializability is preserved if and only if

$$\begin{aligned}&(\forall f \in \mathit{change}(A))\\ &\quad(((\exists g \in \mathit{change}(B))(\mathcal{N}(g) = \mathcal{N}(f))) \Rightarrow g \textit{ is derived from } f) \vee\\ &(\forall g \in \mathit{change}(B))\\ &\quad(((\exists f \in \mathit{change}(A))(\mathcal{N}(f) = \mathcal{N}(g))) \Rightarrow f \textit{ is derived from } g)\end{aligned}$$

[1] Renaming a file is modeled with create-delete semantics. The new file contains the old file's contents, and its modification history consists of a single, unique modification identifier. Deleting a file is modeled by setting the file's contents to null and appending a unique modification identifier to its modification history.

In other words, if the change sets of two modification activities contain files of the same name, then one of the sets of commonly-named files must be derived from the other. If the files in the change sets have disjoint names, change-serializability imposes no restrictions. This differs from full serializability in that only change sets are involved rather than sets of files that are merely examined.

2. *Atomicity.* Let A and B be two modification activities. Atomicity is preserved if and only if

$$(\exists f \in change(A))((\exists f' \in initial(B))(f' \text{ is derived from } f)) \Rightarrow$$
$$initial(B) \text{ is derived from } change(A)$$

That is:

$$(\exists f \in change(A))((\exists f' \in initial(B))(f' \text{ is derived from } f)) \Rightarrow$$
$$(\forall g \in change(A))((\exists g' \in initial(B))(g' \text{ is derived from } g))$$

In other words, if the initial set of B contains a file derived from a change made by A, then it must also contain files derived from all changes made by A.

3. *Completeness.* Let A be a modification activity with $f \in change(A)$ and let B be a modification activity with $f' \in change(B)$, such that f' *is derived from* f. Completeness is preserved if and only if for any other modification activity C:

$$initial(C) \text{ is derived from } change(B) \Rightarrow$$
$$initial(C) \text{ is derived from } change(A)$$

That is:

$$(\forall h \in change(B))((\exists h' \in initial(C))(h' \text{ is derived from } h)) \Rightarrow$$
$$(\forall g \in change(A))((\exists g' \in initial(C))(g' \text{ is derived from } g))$$

In other words, if the initial set of C contains files derived from the changes made by B, then it must also contain files derived from all changes made by the prior modification activity A on which B depended.

2.2 Merges

We define the *merge* of the two file sets G and H, denoted "$G \oplus H$", as follows:

$$G \oplus H = \{f \mid (f \in G \wedge f \text{ is derived from } H) \vee (f \in H \wedge f \text{ is derived from } G)\}$$

Intuitively, if a file is in the merged file set then the file is in at least one of the input file sets and is derived from the other input file set. By definition, $G \oplus H$ is guaranteed to be a file set, with no two files having the same file name. It is a subset of $G \cup H$. The files that are in $G \cup H$ and not in $G \oplus H$ fall into two categories:

- Files in G from which files in H have been derived, and vice versa. These files are intentionally omitted so that only the "latest" files are retained in the merge.

- Files $(fn, M_g, c_g) \in G$ and $(fn, M_h, c_h) \in H$ such that neither file is derived from the other. These files are said to *collide*. They cannot both be included in $G \oplus H$ because they have the same file name and so would cause $G \oplus H$ to cease being a file set. Since neither is preferred over the other, they are both omitted.

$G \oplus H$ is said to be *safe* if and only if there are no collisions, i.e. if and only if $G \oplus H$ *is derived from* $G \cup H$:

$$(\forall f \in G \cup H)((\exists f' \in G \oplus H)(f' \textit{ is derived from } f))$$

The significance of safe merges is that they are guaranteed to preserve coordination consistency:

> **If concurrent development proceeds according to some protocol that produces file sets only by modification activities and safe merges, then coordination consistency will be preserved through those activities and merges that contribute to the production of each file set.**

Proofs that the properties of change-serializability, completeness, and atomicity are preserved in any serialization of these activities run along the following lines. All symbols are as defined in the formalization of the coordination-consistency properties given in Section 2.1:

1. *Change-serializability.* Change-serializability is symmetric in A and B, so we will assume that the merge of A (with the accumulated result of prior modification activities) occurs before the merge of B in the serialization. Then if g is not derived from f, the merge of B would not be safe. Hence g is derived from f.

2. *Atomicity.* If B comes before A in any serialization of the activities then f' will not be derived from f because of the unique coinage of modification identifiers. If A comes before B in the serialization, then all subsequent file sets in the (safe) merges leading to B will contain g or files derived from it.

3. *Completeness.* Any serialization of the activities in which A and B contribute to the resulting file set must merge B after A, otherwise the fact that f' *is derived from* f but $f' \neq f$ would prevent the merge of B. In this case, the merge of A would have introduced g into the resulting file set as well. Hence any activity after the merge of A must see some g' that is derived from g.

3 CONCURRENT DEVELOPMENT IN HIERARCHICAL STORES

The discussion in the previous section led to the conclusion that development in which file sets can be modified only by modification activities and by safe merges with other file sets results in the preservation of coordination consistency. A great variety of approaches to development can satisfy this requirement. However, ensuring that merges are safe requires checking of file derivations, and, in general, this involves maintenance of arbitrarily long modification histories. If file sets are stored in a hierarchy of *stores*, however, and all merges are between parent and child, the modification histories can be encoded simply as two bounded integers.

This section briefly describes our protocols for development using a hierarchy of stores, and the two-integer encoding. Additional detail is given elsewhere [Harr90]. These protocols are supported by the RPDE³ librarian, and are in constant use by members of the research group performing concurrent modifications to the approximately 1400 source files making up the RPDE³ system.

3.1 Protocols

Each modification activity is preceded by an *initiation activity* and followed by a *termination activity*, which might involve a *reconciliation activity*. The initiation, modification and termination activities simulate the copy-merge semantics of development typical of version control systems such as RCS [Tich85] or SCCS [Roth75]. The reconciliation activity resolves any collisions between changes that occurred concurrently in the parent and the child stores.

Initiation. Before a modification activity begins, a new, empty store S is created in which it is to be performed. S is placed in the hierarchy as a child of an existing, store, P. P is typically the master store, which is the root of the hierarchy. The file set $S \oplus P$ is stored in S.[2] The merge is guaranteed to be safe, since S's file set is originally empty. It, and all merges described below, are performed atomically.

Modification. The developer modifies files, using whatever tools he desires. While the modification activity is in progress, the files in S are not available to other modification activities.

Termination. When the developer is satisfied, by whatever criteria are appropriate, that the modification activity is complete and its result can be made available, termination takes place. Provided that the merge is safe, the termination activity replaces the contents of P with $P \oplus S$.

If $P \oplus S$ would not be safe, a *reconciliation activity* is needed. Its purpose is to resolve the collisions that rendered the merge unsafe. Reconciliation of stores S and P introduces a new intermediate store I between them in the hierarchy. I is initialized to contain copies of the files in P. The developer must process the relevant files in I to incorporate the conflicting changes in S, ensuring that all changes are integrated and none are lost. Files in both S and I are accessible during this activity, but only files in I can be modified. For the purposes of this paper, the manner in which conflicting changes to individual files are reconciled and integrated is irrelevant.[3]

Once reconciliation is complete, the store S is discarded. The termination activity can then resume, this time merging I into P. If new collisions occur, due to additional changes made to P while the reconciliation activity was in progress, another reconciliation activity is needed. The termination activity completes only after the merge $P \oplus S$ is safe. Once that has occurred, all changes made by the modification activity, and by any associated reconciliation activities, have been integrated into store P.

[2] In practice one will usually use an existing store whose modification activity has terminated, so as to limit the amount of copying required. All that is said of the newly created store is true of an existing store whose modification activity has terminated.

[3] Various tools provide assistance with collision integration. For example, by computing differences of text files [Tich85] or by using data flow analysis [Reps88].

3.2 Encoding of Modification Histories

In a context where all file transfer is in the form of safe merges between parent and child stores in a hierarchy, the modification histories needed to determine derivation relationships between files can be encoded as a pair of bounded integers, (pn, cn). The numbers are called the "parent number" and the "child number" respectively. If f is a file, we use $pn(f)$ and $cn(f)$ to denote the parent and child numbers of its modification history. Let S be a store, let P be S's parent in the store hierarchy, and suppose $f_s \in S$, $f_p \in P$, and $\mathcal{N}(f_s) = \mathcal{N}(f_p)$. The following invariants are maintained by our implementations of merges and modifications:

$$pn(f) \leq cn(f)$$

$$f_s \textit{ is derived from } f_p \Leftrightarrow pn(f_s) = cn(f_p)$$

$$f_p \textit{ is derived from } f_s \Leftrightarrow pn(f_s) = cn(f_s)$$

Details are given elsewhere [Harr90]. An additional property of this encoding is that a file f has been changed by the current modification activity if and only if $cn(f) \neq pn(f)$.

4 RELATED WORK

Several models of software development have been proposed that are sufficiently different from ours that the issue of coordination consistency do not apply. The Xerox PARC System Modeller [Schm82], for example, was designed to deal with multiple developers working on different components of a system, but specifically did not deal with multiple developers working on the same component.

The work by Reps, Horwitz, and Prins [Reps88,Horw88] uses data and control flow analysis to determine interference of parallel versions of a file from a base version. Their focus is on providing assistance with the reconciliation of conflicting files; they do not address the problem of enforcing coordination consistency. Their model of software development assumes a global check-in-check-out librarian.

Another model, initially presented by Kaiser, Kaplan and Micallef [Kais88] and further developed to incorporate version control and configuration management by Micallef and Kaiser [Mica88], statically partitions the software system, placing each partition into a separate store. Static semantic analysis is used to check consistency of the system. The static partitioning ensures that all merges are automatically safe, but makes it difficult for developers to make changes that span partitions. Our protocols facilitate "spanning changes" and check coordination consistency at merge time. The relationship between coordination consistency and static semantic consistency is a topic for future research, to be discussed in Section 5.

The Gordion system [Ege87] provides transactions and hard and soft locks as concurrency control primitives. Soft locks can be broken, with immediate or delayed notification of the lock holder. Hornick and Zdonik have proposed a wider variety of locks and notification mechanisms [Horn87]. Primitives such as these are mechanisms upon which concurrency control protocols can be built; the focus of this paper has been formalization of the properties that such protocols should have. Protocols that realize these properties can be built upon the primitives described above, though reliance on locking implies that development cannot continue if the central lock handler is inaccessible.

Recent work by Kaiser [Kais90] explores a transaction model that allows more flexibility than does full serializability. Ellis and Gibbs [Elli89] describe an algorithm for ensuring *precedence* and *convergence* properties in groupware systems. No transactions or locking are involved. Instead, operations are transformed when necessary; the algorithm must know some semantics of the operations. Both of these models are based on a much finer-grained sharing of low-level elements and a much tighter interaction between developers than is envisioned here.

Some systems provide check-in check-out facilities and version control, but make no attempt to ensure consistency across files. Tichy's RCS [Tich85] and Rochkind's SCCS [Roth75], for example, provide support for maintaining text files that evolve from each other and for selecting versions of text files. The unit of granularity is the individual text file. These systems do not provide support for encapsulating a set of files into an entity that can be manipulated as a unit. Interleaving of checking in and checking out of files between modification activities is possible, as is checking in or out of arbitrary subsets of files. Of the three coordination consistency properties, such systems guarantee change-serializability, but neither completeness nor atomicity.

Some more advanced systems do provide facilities for treating groups of files as units. Sun Microsystems' Network Software Environment™ (NSE) [Sun88] and Apollo's DOMAIN Software Engineering Environment [Lebl85] are examples. They both provide tools for configuration management. These tools are built on top of a version control system and provide a means to specify the components of a software system, the interelationships between the components, and the version of each component that is to be used. The semantics of how a set of changes are incorporated into a store does not ensure coordination consistency, however.

Of the two, NSE is the most closely related to our work. NSE provides support for concurrent program development that is based on working sets that are analogous to our notion of a store. Although in their software development model any working set can reference any other working set, their working sets are typically organized into hierarchies of release, integration, and development working sets.[4] NSE provides a detection protocol that is analogous to ours (we do not, however, require that the sequence numbers encoding modification histories be time stamps, nor do we require clock synchronization in a distributed computing environment). However, components partition the software system and are the units of transferral between working sets. When discussing RCS and SCCS we observed that partial or interleaved check-in and check-out operations on files can violate coordination consistency. Similarly, partial or interleaved operations on separate components in NSE can violate coordination consistency when modification activities occur that span multiple components.

5 FUTURE RESEARCH

The notion of coordination consistency defined in this paper attempts to capture consistency requirements for effective concurrent development without reference to the details of the artifact being developed. Coordination consistency is not a property of a set of files; it is a property of the development process giving rise to those files. It is also important to consider internal consistency of the files in the set. We call this *self consistency*. Different definitions of self consistency are appropriate for different systems and languages. Some examples are: consistent by definition

[4] Software releases are related by a connection between their corresponding release working sets in the hierarchy.

(*trivially-consistent*), consistent by developer's assertion (*asserted-consistent*), consistent with respect to cross-file links as in RPDE³ (*link-consistent*), or consistent with respect to uses and definitions of resources (*use-def-consistent*).

Self consistency is a property of a set of files, and is independent of the development process giving rise to those files. In general, neither self consistency nor coordination consistency implies the other. However, corresponding to any chosen notion of self consistency, x, there is a derived notion of consistency called *coordinated-x-consistency*. Coordinated-x-consistency is a property of a file set resulting from an *activity lattice* of modification activities and merges. The relationship of coordinated-x-consistency to incremental checking of x-consistency, and the additional information required to track and verify the preservation of x-consistency through safe merges, are topics of great interest to us.

6 SUMMARY

In this paper we have:

1. Developed a formal model for defining the activities of importance in coordinating concurrent development.
2. Defined the notion of *coordination consistency* to consist of three properties: *change-serializability, completeness* and *atomicity.*
3. Demonstrated that concurrent development consisting only of isolated modification activities and safe merges preserves coordination consistency.
4. Briefly described the RPDE³ protocols for performing concurrent development using a hierarchy of stores. Each modification activity occurs in its own store, and all file transfers are in the form of safe merges between parent and child stores.
5. Described an encoding of modification histories as a pair of bounded integers that is appropriate in the restricted world of the RPDE³ protocols.

REFERENCES

[Ege87] Aral Ege and Clarence A. Ellis. "Design and Implementation of GORDION, an Object Base Management System." In *Proceedings of the Third International Conference on Data Engineering,* IEEE Computer Society, pp. 226–234, February 1987.

[Elli89] C. A. Ellis and S. J. Gibbs. "Concurrency Control in Groupware Systems." In *Proceedings of the International Conference on the Management of Data,* ACM SIGMOD, pp. 399–407, June 1989.

[Harr90] William H. Harrison, Harold Ossher, and Peter F. Sweeney. "Coordinating Concurrent Development of Software." IBM Research Report RC15514, February 1990.

[Harr89] William H. Harrison, John J. Shilling, and Peter F. Sweeney. "Good News, Bad News: Experience Building a Software Development Environment Using the Object-Oriented Paradigm" In *OOPSLA'89 Conference Proceedings*, ACM, pp. 353–361, October 1989.

[Horn87] Mark F. Hornick and Stanley B. Zdonik. "A Shared, Segmented Memory System for an Object-Oriented Database." *ACM Transactions on Office Information Systems* 5(1), pp. 70-95, January 1987.

[Horw88] S. Horwitz, J. Prins, and T. Reps. "Integrating Non-Interfering Versions of Programs," *ACM Transactions on Programming Languages and Systems* 11(3), pp. 345–387, July 1989.

[Kais88] Gail E. Kaiser, Simon M. Kaplan, and Josephine Micallef. "Multiple-User Distributed Language-Based Environments." *IEEE Software*, pp. 58–67, November 1987.

[Kais90] Gail E. Kaiser. "A Flexible Transaction Model for Software Engineering." In *Proceedings of the Sixth International Conference on Data Engineering*, pp. 560–567, February 1990.

[Lebl85] David B. Leblang and Gordon D. McLean, Jr. "Configuration Management for Large-Scale Software Development Efforts." In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, pp. 112–127, June 1985.

[Mica88] Josephine Micallef and Gail E. Kaiser. "Version and Configuration Control in Distributed Language-Based Environments." In *Proceedings of the International Workshop on Software Version and Configuration Control*, Teubner Verlag, Stuttgart, FRG, pp. 119–143, January 1988.

[Pu88] Calton Pu, Gail E. Kaiser and Norman Hutchinson. "Split-Transactions for Open-Ended Activities." In *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, pp. 26–37, August 1988.

[Reps88] Thomas Reps, Susan Horwitz, and Jan Prins, "Support for Integrating Program Variants in an Environment for Programming in the Large." In *Proceedings of the International Workshop on Software Version and Configuration Control*, Teubner Verlag, Stuttgart, FRG, pp. 197–216, January 1988.

[Roth75] Marc J. Rochkind. "The source code control system." *IEEE Transactions on Software Engineering* SE-1(4), pp. 364–370, December 1975.

[Schm82] Eric E. Schmidt. "Controlling Large Software Development in a Distributed Environment." Xerox PARC Technical Report CSL-82-7, December 1982.

[Schw88] Robert W. Schwanke and Gail E. Kaiser. "Living with Inconsistency in Large Systems." In *Proceedings of the International Workshop on Software Version and Configuration Control*, Teubner Verlag, Stuttgart, FRG, pp. 98–118, January 1988.

[Sun88] Sun Microsystems. "Introduction to the Network Software Environment™." Part No: 800-2362-10, Revision A of 12 August 1988.

[Tich85] Walter F. Tichy. "RCS - A System for Version Control." *Software–Practice and Experience* 15(7), pp. 637–654, July 1985.

[West87] Brian A. Weston. "Segmenting an Object-Oriented Database." IBM Research Report RC12662, April 1987.