Infuse: A Tool for Automatically Managing and Coordinating Source Changes in Large Systems

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

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

September 1986

Abstract

In current change management tools, the actual changes occur outside the tool. In contrast, *Infuse* concentrates on the actual change process and provides facilities for both managing and coordinating source changes. Infuse provides facilities for automatically structuring the cooperation among programmers, propagating changes, and determining the consistency of changes, and provides a basis for negotiating the resolution of conflicting changes and for iterating over a set of changes.

1. Introduction

A number of tools address the problems of managing changes to large software systems. Most such tools provide a framework in which programmers can reserve modules[1] for change and in which the changes themselves occur outside of the tool. Examples include SCCS [Rochkind 75], Cedar's System Modeller [Lampson 83], Darwin [Minsky 85] and DSEE [Leblang 84]. When a change is to be made to a module, a programmer reserves the module, obtains an official copy of the module, and then proceeds by making changes with an editor. The tool has no knowledge of the changes as they are being made, though in most cases it notices the differences between the two versions once the new version have been deposited back into the database maintained by the tool. The changes, as they are made in the editor, are outside the scope of the change management tool.

In contrast, *Infuse* concentrates on the actual change process and provides facilities for both managing and coordinating source changes. The change process itself is supported within Infuse, not outside it. This enables Infuse to actively participate in the change process, assisting programmers by automating various aspects of the process, most notably *change propagation*. Change propagation consists of detecting and notifying those modules affected by a change in another module. Inconsistencies detected in propagating changes can be removed in two different ways: by making further explicit source changes (which of course have their own extent and implications) and by automated reprocessing such as recompilation and relinking (as required, for example, in the reordering of the fields in a record).

Infuse is concerned with assisting the programmer making source changes, not with automated reprocessing; tools already exist for restoring consistency by recompilation. For example, Make [Feldman 79] and similar tools deal with the problems of recompilation and relinking without having any knowledge of the actual source-level changes, except for the fact that changes have been made. However, Tichy's 'smart recompilation' [Tichy 86] deals nicely with syntactic dependencies, and provides a suitable basis for change propagation. We explain later how Infuse takes advantage of the dependency analysis component of smart recompilation.

We first describe the problems that must be solved in order to automate the process of changing multiple parts of a large system, where the changes are being made concurrently (i.e., simultaneously) by a number of programmers. Next, we outline our assumptions about the context of Infuse, including the model of syntactic interconnections that we use as the context for explaining Infuse's facilities.[2] Then we present Infuse's facilities for automating the management and coordination of source

---

1. We use the term "module" synonymously with "source file".
2. The primary motivation for Infuse is to provide the change management component for the Inscape Environment [Perry 85b, 86, and 87b]. Inscape implements the semantic interconnection model in addition to the syntactic model and is thus able to provide more comprehensive consistency checking and change propagation than we describe here. We restrict ourselves here to the syntactic model for reasons of simplifying the discussion; the framework is identical in both cases.

changes and discuss its three aspects: a hierarchy of experimental databases, a partitioning of the modules to be changed (and a merging back into the parent database), and notions of consistency within a database.

2. Problems of Coordinating Changes

The problems of making source changes in large, multi-programmer systems are different from those of single-programmer systems. In the latter, it is relatively easy to detect the modules affected by either single or multiple changes and restore consistency. Changes to large systems, on the other hand, have led to a wide variety of horror stories due to incomplete and/or inconsistent changes [Perry 85a, 87a].

The following are the basic problems in managing and coordinating source changes in software systems:

- *Determining the implications and extent of changes*. In spite of the effective use of techniques such as information hiding to localize the effects of change, changes often have effects beyond these limiting boundaries. For example, a seemingly simple change can easily *cascade* in complex and unpredictable ways. There are two factors in considering these effects: the *extent* of the change and the *implications* of the change. The extent of a change is determined by the number of modules that are affected while the implications of a change are determined by what is necessary to restore consistency to the system. Both the implications and the extent of changes become more complex in the presence of multiple, concurrent changes. The problem of cascading changes is thus further compounded.
- *Handling temporary inconsistencies during changes.* The problem of temporary inconsistencies is readily apparent in multiple, concurrent changes, but can arise due to single changes, since these can cascade. This problem is of particular importance when determining the effects of change either incrementally on the entire system or entirely on only part of the system.
- *Negotiating the resolution of conflicting changes.* The issue of change *negotiation* arises when independent changes lead to a conflict (that is, they are inconsistent with each other). When a conflict occurs, a forum is required in which to negotiate the solution to the inconsistencies. The usual form of this forum is managerial, i.e., determined entirely by humans without automated assistance.
- *Merging multiple changes into the baseline system.* Once negotiations have resolved conflicts and the changes have been made, then the changed modules must be merged back into the system. This may be done incrementally (that is, in planned stages) or it may be done haphazardly, for example, merging all the changed modules at once. Control of this merging process is usually managerial.
- *Supporting the iterative process of making further changes in response to changes in other modules.* As a result of inconsistencies and the subsequent change negotiation, iterations of changes occur. Some changes remain, others are undone, others are redone, and new changes are introduced. The result is that there are iterations in the modification cycle. Each iteration has its own set of required changes, complete with

their respective cascade of changes, that leads to the next iteration of changes, thereby causing a *yo-yo* effect.

2.1 Our Assumptions

We make four assumptions, which allow us to concentrate solely on the problems of automating the management and coordination of source changes.

- The initial set of modules to be changed has already been determined — for example, by an analyst.
- We have available some form of version and configuration control system (such as RCS [Tichy 85] or Gandalf's SVCE [Habermann 81, Kaiser 83]) that is used to determine the exact versions that comprise system.
- There exists a base system (which, for example, might be the previously released system) from which we evolve the new system.
- The syntactic objects of the programming language (procedures, types, variables, etc.) are used as the unit of interconnection and dependency [Perry 86c]. This model is exemplified by cross-reference listings [Teitelman 81] that can be generated as a side-effect of compilation.

3. Overview of Infuse

Infuse provides three main facilities for managing and coordinating source changes in a large system: a hierarchy of experimental databases; a means of automatically partitioning the modules to be changed into experimental databases and of merging the changed modules back into the parent database; and some notions of consistency within an experimental database.

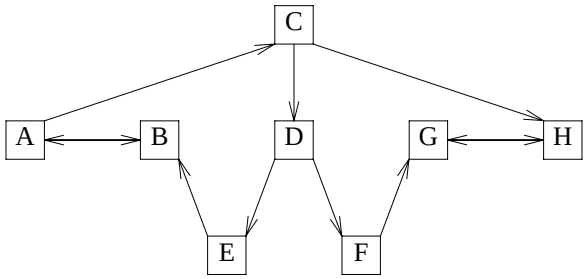
An *experimental database* is a virtual copy of the software system that permits a programmer or a group of programmers to make changes to reserved modules in isolation from the rest of the software team. Experimental databases provide the foundation for iteration and the forum for negotiation. By partitioning the modules into separate experimental databases recursively, we bound the determination of the implications and extent of changes. It is within an experimental database that we resolve the problems of inconsistencies among those modules before merging[3] the components back into the parent database where the problems of implication, extent and inconsistency are then addressed for the parent experimental database with respect to its own parent.

We use the following small example (Figure 1) to illustrate the various parts of the discussion below. The arrows indicate the direction of the dependencies (i.e., who imports from whom). A and B, G and H are strongly connected - that is, they depend

3. We use *merge* with respect to modules, that is, merging modules into an experimental database. We do not mean *merge* in the sense of merging several versions of the same module into a single version (as is possible in, for example, Cedar and RCS).

upon each other. E and F depend on D, D and H depend on C, C depends on A, B depends on E, and G depends on F.

Figure 1



Given this set of modules to be changed, Infuse behaves as follows. It partitions the set recursively into the set of hierarchical experimental databases. The leaves of the tree are singleton experimental databases, which contain a single module; all editing is done in singleton databases and then the databases are *deposited* into their parent database. When all the children databases have been deposited into their parent database, Infuse propagates the changes to the appropriate modules in that database and reports local inconsistencies to the responsible programmers. Conflicting changes, if they exist, are resolved by negotiation among those responsible for the changes and then Infuse repartitions the database recursively to the level of singleton databases to assist the programmers in making the agreed upon changes. When all the conflicts and local inconsistencies have been resolved, Infuse deposits this database into its parent experimental database. This process continues until all inconsistencies have been resolved and the top-level experimental database has been deposited back into the baseline system.

In addition to the basic facilities discussed below, Infuse also provides workspaces and simulation of change propagation [Kaiser 86]. Workspaces are a complementary facility to experimental databases and support immediate consistency checks with user-selected modules, independent of the hierarchical partitions. The simulation mechanism permits a programmer to determine the full extent and implications of a proposed change without actually committing the change.

4. Hierarchy of Experimental Databases

The notion of an experimental database was introduced in Smile [Krueger 85, Notkin 85]. We extend this notion to that of a *hierarchy* of experimental databases, where each database is subdivided automatically during partitioning into subcomponents. These subgroups then form the next level in the hierarchy. At the leaves of the tree are singleton experimental databases where the actual changes to the modules take place.

An experimental database is distinguished from the *main database*, or baseline system, in the following ways. A main database is guaranteed to contain source files that are consistent with each other whereas this need not be true for an experimental database. In an experimental database, the units within the database are self-consistent even if they are not consistent with each other.[4] At some point, however, the subcomponents become consistent with each other, that is, the database as a whole becomes self-consistent. Only when an experimental database is self-consistent will Infuse deposit it back into the parent database. Basically, then, a main database represents a stable, consistent system while an experimental database represents an unstable, changing part of a system.

---

4. We make an exception for singleton experimental databases: at any given point in the change process, the singleton database may be internally inconsistent; however, it must become self-consistent before Infuse will deposit it into its parent database.

The principal reasons for the hierarchy are to provide

- a structure for enforced cooperation among the various programmers responsible for making changes to a system, and
- a means for managing iterations inherent in the change process.

As a result of decomposing the set of modules into a hierarchy of experimental databases, we minimize the cost of consistency checking (by bounding the amount of checking), limit the extent of change propagation, and restrict the number of potential conflicts to be negotiated.

5. Partitioning/Merging

When sets of changes are made to large systems, the changes to individual modules are typically done in isolation and then the modules are merged together in some fashion (as, for example, in SCCS). In contrast, Infuse partitions the set of modules into subsets in order to provide a basis for the merging operation and to limit the amount of interaction that must be coped with at one time. The hierarchy of partitions limits the problems of determining the implications and extent of changes and provides a useful scoping mechanism for propagating the changes being made. DSEE has the notion of tasks and subtasks that might be used as the basis for partitioning the database. However, this is primarily a managerial approach in the form of tasklists in which tasks are completed and checked off; it does not address the technical problems that we solve with Infuse.

The partitioning algorithm is particularly important because of the costs involved in providing consistency checking and change propagation. These are the primary costs; secondary costs include the cost of creating and merging experimental databases.

Determining how to partition the modules into the appropriate subsets is a problem because we do not have available the optimal oracle: *which pieces will change and how*. We may be able to get a reasonably close approximation to the optimal oracle in the initial building of the top-level experimental database by simulating a change propagation to modules transitively affected by the proposed changes. By having a programmer indicate the specific procedures, types, etc. that will change and then automatically determining the (initial) implications of these changes, we can get a better idea of the interconnections that will be most likely to change. However, this fine grain of change indication may not be possible, either because it is not yet known or because it is too fine a level of detail to ask of the programmers prior to actually making the changes.

Instead, Infuse uses information that it can derive automatically from the structure of the system. We have chosen to use the strength of the dependency interconnections as the basis of partitioning. However, these interconnections give only an approximation of the possible effects of the actual changes; it could well be the case that the changes are between weakly interconnected pieces of the system rather than the heavily interconnected ones.

Given the costs of creating experimental databases, checking consistency, and propagating changes, there is a higher cost of finding inconsistencies at the upper levels of the hierarchy and a lower cost of finding them near the leaves. Therefore, the heaviest interconnections should be near the leaves, so that the changes with the most pervasive implications are considered earlier rather than later in the change cycle. Conversely, the number of potential inconsistencies — that is, the number of external interconnections between the partitions of a database — should be fewer in each succeeding upper layer.

There are certain kinds of interconnections that need to be filtered out of the interconnection structure used for the partitioning of the system. For example, the canonical module that is included by everything in the system serves only to provide an extra layer in the hierarchy and provides no useful information for partitioning the system; it is also the exception that proves the rule — there is no way to partition the system and still have this heavy a connection near the bottom of the hierarchy.

One further note on partitioning considerations: the interconnection structure changes with each iteration in the process of change. What was initially the dependency structure is modified by the changes and a new structure is created which must be used in the next cycle of partitioning (when changes conflict and must be resolved). We would also like to weight most heavily those dependencies actually involved in the

inconsistencies detected in the previous iteration.

5.1 Possible Partitioning Algorithms

Our partitioning problem is similar to *graph partitioning*, where the edges in the graph are directed and weighted. Each module to be changed is represented by a vertex in the graph. A syntactic object defined in one module and used in another is represented by a directed edge leaving the vertex representing the first module and entering the vertex for the second. If the second module references several syntactic objects defined by the first, the edge is weighted by the number of such dependencies.

Unfortunately, graph partitioning is intractable [Garey 79]. Therefore, we need an algorithm that *approximates* the correct partitioning. Because partitioning lasts only until a set of changes are resolved and is repeated for every set of changes, it is very important that the algorithm be fast. Moreover, as perfect graph partitioning is only an approximation to our oracle anyway, it is relatively less important that the partitioning algorithm be very close to correct in all cases.

Our partitioning problem has a number of characteristics in common with VLSI layout problems. We have investigated various graph partitioning algorithms [Kirkpatrick 83, Heller 82] applied in VLSI to see how they can be adapted to our problem of partitioning and distributing the cost of managing the change process. The Kernighan-Lin method [Kernighan 70, Dunlop 85] seems closest, but it unfortunately produces an N-ary partition at each level of the hierarchy, where N must be chosen in advance and cannot be automatically tailored to the graph under consideration.

We are currently working on improvements to the Kernighan-Lin method that adapt it to our situation. In the meantime, we have found several optimizations that can be applied to any graph partitioning algorithm to reduce the amount of work and to improve the results of the algorithm.

- Remove all edges with weight less than K1. The purpose of this strategy is to set a noise threshold — that is, to remove all the interconnections that are insignificant in order to reduce the amount of work to be done in the rest of the algorithm. This improves the chances of finding natural partitions. An intuitive refinement is to reduce K1 by half at each level of recursion.
- Remove from the graph all nodes whose number of edges is greater than K2. This step takes care of the situation where a module is too pervasive.

Experimentation is needed to determine strategies for finding appropriate values for K1 and K2. We would like to tune these values to the particular system under consideration, automatically adapting as the intermodule connections change. One attractive possibility is to plot the nodes of the graph according to the weight of the edges and choose K1 as the dividing point where the number of edges are less than K3 standard deviations below the norm and K2 as the point K4 standard deviations above the norm. The plot can be incrementally updated after each change to the interconnection structure. An intuitive value for both K3 and K4 is two standard deviations.

5.2 Example

Consider the small system composed of modules A through H as shown in Figure 1. Figure 2 illustrates how the dependencies between D, E and F might appear in an Ada-like source fragment.

Figure 2

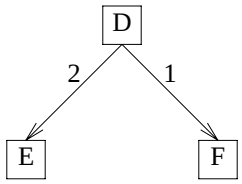
```
with C; use C;
package D is
    . . .
    procedure X ( . . . );
    procedure Y ( . . . );
    . . .
end D;

with D; use D;
procedure E is
    . . .
    X(. . .);
    . . .
    Y(. . .);
    . . .
end E;

with D; use D;
procedure F is
    . . .
    Y(. . .);
    . . .
end F;
```

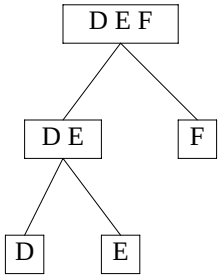
Note that procedure E uses two procedures from package D while procedure F uses only one procedure.

Figure 3



Because there is a stronger dependency relationship between units D and E than there is between D and F, D and E are partitioned together. Figure 4 depicts the portion of the hierarchical experimental database structure that results from the dependency graph depicted in Figure 3.

Figure 4

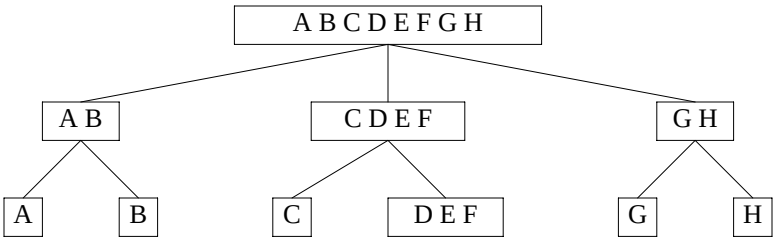


6. Consistency

Ensuring the consistency of changes where multiple programmers are concurrently changing many modules is a very difficult problem because of the complexity of the interactions among the modules. We use the hierarchical experimental databases to bound the complexity and the number of interactions: Infuse determines that a subset of components is self-consistent before it allows the merging of these components

back into the parent database.

Figure 5



In Figure 5, components D, E and F have each been changed and deposited back into the parent database. Infuse then checks for consistency: the changes in E and F are checked against the changes in D to determine whether they are self-consistent. When they are consistent, D, E and F are deposited into the parent database and Infuse checks the consistency of D and C. Similarly, Infuse propagates changes between A and B and between G and H. Once the databases at the second level are consistent, they are deposited into the database at the top level. Infuse then performs consistency checking for the changes that it did not propagate at the lower levels (between A and C; B and E; C and H; and G and F). At each level, if the consistency analysis fails, then that database is repartitioned on the basis of the inconsistent components and the appropriate changes are negotiated and made. Similarly, when the top-level database is deposited into the base system, Infuse checks the consistency of the changed modules against the base system and a new iteration of changes may be generated to restore consistency.

Smile and SVCE support a similar notion of consistency but require that a module be consistent before allowing its deposit back into the database. Infuse differs from this approach in that it does not require consistency at this point. Instead, Infuse treats each experimental database as a forum for determining consistency once all the components in that database have been changed and deposited.

Because Infuse localizes the determination of consistency within an experimental database, it limits the scope of handling the problem of temporary inconsistencies to the local experimental database. All Infuse requires is that the modules within a database be consistent with each other, and not with the rest of the system. This approach requires a concept of *local consistency*. PIC [Wolf 85] defines a notion of *conditional consistency* that allows partial consistency that is based on the incompleteness (specified by an incompleteness construct) of the parts that cannot be shown to be inconsistent. This is not quite the notion that we need here — there is no incompleteness in the sense of PIC, rather there are inconsistencies that Infuse ignores because they come from modules outside the local database.

The fundamental problem in determining the local consistency of changes is that of determining the implications of changes, deletions, additions and rearrangements of the units of interconnections. Our strategy is to ignore undefined objects (except when restoring consistency with the base-line system) and check the consistency only of those objects that are both defined and used within the modules being checked. One obvious refinement to this scheme is to check the consistency of only those objects that have not been previously checked — e.g., at the next lower level in the hierarchy.

Tichy's smart recompilation and Lint [Johnson 78] provide consistency checking at the level of module interfaces and could be modified to provide local consistency checking in a form appropriate for Infuse. For example, an appropriately structured symbol table could provide much of the information needed for local consistency checking. In this symbol table, the static semantic analyzer would flag definitions that have changed, keep track of deletions, note additions and flag rearrangements.

7. Conclusions

The contributions of our research are

- hierarchical experimental databases,

- grouping modules together in an experimental database according to the strengths of interdependencies,
- the notion of local consistency within the context of an experimental database,
- experimental databases as the forum for negotiating changes in the process of resolving inconsistencies, and
- experimental databases as the basis for iterating over a set of changes.

We are implementing Infuse as an extension of Smile, which already supports a single level of experimental databases. The advantage of building from Smile is we retain all its facilities, including its interfaces to programming tools and its mechanisms for locking databases and recovering after system failures. We are using a variant of Tichy's smart recompilation algorithm to support change propagation.[5] The main modification is ignoring references to symbols not defined within the current experimental database.

5. This variant of Tichy's algorithm provides us with the basis for syntactic change propagation. In addition, Infuse, in the context of Inscape, exploits the semantic interconnections to provide change propagation at the semantic level as well.

Acknowledgements

Yoelle Maarek is working with us on the implementation of Infuse as an extension of Smile. Maria Thompson and Mark Freeland provided careful readings and comments to earlier versions of this paper.

References

[Dunlop 85] Alfred E. Dunlop and Brian W. Kernighan. *A Procedure for Placement of Standard-Cell VLSI Circuits*. **IEEE Transactions on Computer-Aided Design**, CAD-4:1 (January 1985), pp. 92-98.

[Feldman 79] S. I. Feldman. *Make — a program for maintaining computer programs*. **Software — Practice & Experience**, 9 (1979). pp. 255-265,

[Garey 79] M. R. Garey and D. S. Johnson. **Computers and Intractability: A Guide to the Theory of NP-Completeness**. San Francisco: Freeman, 1979.

[Habermann 81] A. Nico Habermann and Dewayne E. Perry. *System Composition and Version Control for Ada*. in **Software Engineering Environments**. H. Huenke, editor. North-Holland, 1981, pp. 331-343.

[Heller 82] William R. Heller, G. Sorkin, Klim Maling. *The Planar Package for System Designers*. **19th Design Automation Conference Proceedings**, IEEE, 1982, pp 253-260.

[Kaiser 83] Gail E. Kaiser and A. Nico Habermann. *An Environment for System Version Control*. **Digest of Papers Spring CompCon '83**, IEEE Computer Society Press, February 1983. pp. 415-420

[Kaiser 86] Gail E. Kaiser and Dewayne E. Perry. *Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution*, Technical Report. Computer Technology Research Lab, AT&T Bell Laboratories, November 1986.

[Kernighan 70] B. W. Kernighan and S. Lin. *An efficient heuristic procedure for partitioning graphs*. **Bell System Technical Journal**, 49:2 (1970), pp. 291-308.

[Kirkpatrick 83] S. Kirkpatrick, C. D. Blatt, Jr., and M. P. Vecchi. *Optimization by simulated annealing*. **Science**, 220 (May 13, 1983), pp. 671-680.

[Krueger 85] Charles W. Krueger. **The SMILE User's Guide**. Carnegie-Mellon University, Department of Computer Science, The Gandalf Project, October 1985.

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

[Leblang 84] David B. Leblang and Gordon D. McLean, Jr. *Computer-Aided Software Engineering in a Distributed Workstation Environment*, **Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments**, **SIGPLAN Notices**, 19:5 (May 1984). pp. 104-112.

[Johnson 78] S. C. Johnson. *Lint, a C Program Checker*. **Unix Programmer's Manual**, AT&T Bell Laboratories, 1978.

[Minsky 85] Naftaly Minsky. *Controlling the Evolution of Large-Scale Software Systems*. **Workshop on Software Engineering Environments for Programming-in-the-Large**, June 1985, pp. 1-16

[Notkin 85] David S. Notkin. *The Gandalf Project*. **The Journal of Systems and Software**, 5:2 (May 1985), pp. 91-105.

[Perry 85a] Dewayne E. Perry and W. Michael Evangelist. *An Empirical Study of Software Interface Faults*. **International Symposium on New Directions in Computing** (IEEE), Trondheim, Norway, August 12-14, 1985.

[Perry 85b] Dewayne E. Perry. *Position Paper: The Constructive Use of Module Interface Specifications.* **Third International Workshop on Software Specification and Design**. IEEE Computer Society, August 26-27, 1985, London, England.

[Perry 86] Dewayne E. Perry. *The Inscape Program Construction and Evolution Environment*, Technical Report. Computer Technology Research Lab, AT&T Bell Laboratories, August 1986.

[Perry 87a] Dewayne E. Perry and W. Michael Evangelist. *An Empirical Study of Software Interface Faults — An Update*. 20th Hawaii International Conference on System Sciences. January 1987.

[Perry 87b] Dewayne E. Perry. *Software Interconnection Models.* To appear in the 9th International Conference on Software Engineering, March 30 - April 2, 1987. Now available as a Technical Report, Computer Technology Research Lab, AT&T Bell Laboratories.

[Rochkind 75] M. J. Rochkind. *The source code control system*. **IEEE Transactions on Software Engineering**, SE-1 (1975), pp. 364-370.

[Teitelman 81] Warren Teitleman and Larry Masinter. *The Interlisp Programming Environment*. **Computer** 14:4 (April 1981), pp. 25-34.

[Tichy 85] Walter F. Tichy. *RCS - A System for Version Control*. **Software — Practice & Experience**, 15:7 (July 1985). pp. 637-654.

[Tichy 86] Walter F. Tichy. *Smart Recompilation*. **ACM Transactions on Programming Languages and Systems**, 8:3 (July 1986), pp. 273-291

[Wolf 85] Allexander L. Wolf, Lori A. Clarke, and Jack C. Wileden. *Ada-Based Support for Programming-in-the-Large*. **IEEE Software**, 2:2 (March 1985), pp. 58-71.

Appendix — An Example of Semantic-Based Partitioning

In Figure A we present the interface specifications as they might occur in an Inscape-like environment for an Ada-like language. In addition to the syntactic objects that are defined in the interface specifications, we have the behavioral descriptions of the various components in the form of preconditions (predicates that must be true before execution — that is, assumptions that must be satisfied), postconditions (predicates that are guaranteed to be true afterwards — that is, descriptions of results and side-effects), and obligations (predicates that must eventually be satisfied — see [Perry 87b]). The keyword *sats* is an abbreviation for *satisfies* and *sat by* is an abbreviation for *is satisfied by*. (Note that the lists <...> denote statement labels, most of which are elided for the sake of brevity.) Preconditions and obligations are satisfied by postconditions; postconditions may satisfy both obligations and preconditions. The predicates, then, are the points of interconnection between components in the semantic model.

Figure A

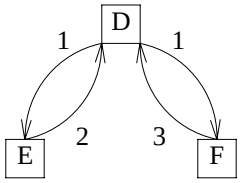
```
with C; use C;
package D is
    . . .
    procedure X ( . . . );
        Post: b(...)
        Obl:  c(...)
    procedure Y ( . . .)
        Pre:  d(...)
        Post: c(...), e(...), f(...)
    . . .
end D;

with D; use D;
procedure E is
    . . .
    <5> X(. . .);
        Post: b(...) sats <11>
        Obl:  c(...) sat by <8>
    . . .
    <8> Y(. . .);
        Pre:  d(...) sat by <2>
        Post: c(...) sats <5>
              e(...), f(...) sats <>
    . . .
end E;

with D; use D;
procedure F is
    . . .
    <4> Y(. . .);
        Pre:  d(...) sat by <3>
        Post: c(...), e(...) sats <5>
              f(...) sats <10>
    . . .
end F;
```

Figure B illustrates the dependency graph for the above system fragment. Note that there are two-way dependencies (actually there is also a dependency within D that is not shown here: the postcondition *c* of Y satisfies the obligation *c* of X): the operations in D each depend on postconditions provided by the implementations of both E and F; E uses only two of the combined postconditions supplied by X and Y while F uses all three of the postconditions supplied by Y. Hence the stronger connection is between D and F rather than between D and E.

Figure B



Because of the stronger connectivity between D and F, they are partitioned together as the illustration of the hierarchical experimental databases for this model shows in Figure C. Note that the partitioning is different from the syntactic example because of the differences in the interconnection structure.

Figure C