Software Development Environments for Very Large Software Systems

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

Yoelle S. Maarek, Technion, Israel Institute of Technology, Computer Science Department, Haifa, 32000 ISRAEL

Dewayne E. Perry, AT&T Bell Laboratories, Computer Systems Research Lab, Murray Hill, NJ 07974

Robert W. Schwanke, Siemens Research and Technology Laboratories, Princeton Forrestal Center, Princeton, NJ 08540

December 1987

CUCS-279-87

Abstract

This technical report consists of the three related papers. *Living with Inconsistency in Large Systems* describes CONMAN, an environment that identifies and tracks version inconsistencies, permitting debugging and testing to proceed even though the executable image contains certain non-fatal inconsistencies. The next two papers are both from the INFUSE project. *Change Management for Very Large Software Systems* presents the new non-Euclidean hierarchical clustering algorithm used by the INFUSE change management system to cluster modules according to the strengths of their interdependencies. *Models of Software Development Environments* presents a general model of software development environments consisting of three components — policies, mechanisms and structures — and classifies existing and proposed environments into the individual, family, city and state classes according to the size of projects that could be adequately supported.

Prof. Kaiser is supported in part by grants from AT&T Foundation, Siemens Research and Technology Laboratories, and the New York State Center of Advanced Technology — Computer & Information Systems, and in part by a Digital Equipment Corporation Faculty Award. When this research was conducted, Ms. Maarek was a visiting PhD student at Columbia University.

Final version to appear in International Workshop on Software Version and Configuration Control, Grassau, West Germany, January 1988.

Living With Inconsistency in Large Systems

Robert W. Schwanke
Siemens Research and Technology Laboratories
Princeton, NJ 08540

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

31 August 1987

---

*Supported in part by grants from Siemens Research and Technology Laboratories, AT&T Foundation, and New York State Center of Advanced Technology - Computer and Information Systems, and in part by a Digital Equipment Corporation Faculty Award.

Abstract

Programmers generally want to be sure that the systems they are building are *consistent*, both with respect to source code versions used, and with respect to type safety. Most modern high-level language systems enforce this consistency upon the system instances they build. However, in a large system this can lead to very large recompilation costs after small changes. Therefore, programmers often circumvent enforcement mechanisms in order to get their jobs done. The CONMAN *con*figuration *man*agement project explores the premise that some degree of inconsistency is inevitable in software object bases, and that programming tools should be designed to analyze and accomodate it, rather than to abhor it. The CONMAN programming environment will help the programmer contend with inconsistency by automatically identifying and tracking six distinct kinds of inconsistencies, *without* requiring that they be removed; by reducing the cost of restoring type safety after a change, through a technique called *smarter recompilation*; and by supplying the debugger and testing tools with inconsistency information, so that they can protect the programmer from flaws in the code.

1. Introduction

Every programmer remembers wasting large amounts of time looking for a bug caused by changing and recompiling one source file and failing to recompile a related file. This kind of problem has made the Unix^{TM} *make* tool [3] very popular; when invoked after a change to a source file, *make* rebuilds every file derived (directly or indirectly) from the changed file.

Programmers generally want to ensure that the systems they are building are *consistent*. For example, they want to know that the object code they are running was built from the exact source code they are looking at, rather than from some previous version of the source code. They also want to ensure that the executable program is *type safe*; that is, that it satisfies the type rules of the programming language. Most modern high-level language systems enforce this consistency upon the system instances they build. In a large system, however, this can lead to very large recompilation costs even after small changes. Therefore, programmers often circumvent enforcement mechanisms in order to get their jobs done.

This practice is not only commonplace; it is commendable! The programmer can do it successfully by using design knowledge to decide which inconsistencies are harmless and which are dangerous. Allowing inconsistency can speed up the edit-compile-debug cycle, and can also reduce the coordination needed between programmers. Both benefits improve productivity dramatically.

The CONMAN *con*figuration *man*agement project is exploring the premise that some degree of inconsistency is inevitable in software databases, and that programming tools should be designed

to analyze and accomodate it, rather than to abhor it. The CONMAN programming environment helps the programmer contend with inconsistency by:

- Automatically identifying and tracking inconsistencies: CONMAN classifies each inconsistency into one of six categories, and tracks it for the programmer, *without* requiring her to remove it right away.

- Reducing the cost of type safety: CONMAN's type safety is based on a constraint called *link consistency*, which is less stringent than in conventional systems. This permits use of a technique called *smarter recompilation* to reduce the cost of restoring type safety after a change [15].

- Supporting debugging and testing: The debugger automatically stops execution upon reaching inconsistent code, thus helping to prevent crashes. The test coverage analyzer tells the programmer which tests can be executed in <the presence of an inconsistency.

This paper begins by presenting several scenarios in which allowing inconsistency is more cost-effective than removing it. Then it describes the six kinds of consistency that CONMAN recognizes automatically. Next, it explains how smarter recompilation uses link consistency to decide which modules really must be recompiled after a source code change. Finally, it describes how the CONMAN programming environment uses consistency analysis to help the programmer build, debug and test inconsistent systems.

2. Beneficial Inconsistency

Inconsistency is commonplace in software project libraries. A project library typically contains many system configurations, where each configuration might contain requirements, specifications, code, test data and documentation. Informally, a project library is inconsistent if it contains direct contradictions. For example, if a global data type is somehow defined differently in different parts of a configuration, this constitutes a contradiction (because most languages permit only one definition of each global identifier). On the other hand, two distinct system configurations may define the type differently, and that would not be a contradiction.

Inconsistency is likely to occur when permitting it is more cost effective than forbidding it. For example:

- Debugging and testing under deadline pressure. On fixing a bug, the programmer should recompile the minimum amount of code necessary to continue testing. She can wait to recompile the rest of the system until she goes home for the night.

- Debugging an incomplete implementation. In a language such as Ada$^{(R)}$, with specifications separated from package bodies, an early version of a package body might not contain all of the procedures. The programmer should not be distracted from her creative task by the tedium of writing stubs. (Wolf studies this form of incompleteness [18].)

- Changing requirements after implementation is under way. When requirements change, it may be easier to start by combining the new requirements with the old implementation -- even though they contradict each other -- rather than keeping them in separate system configurations until they agree.
- Handling "software rot". Sometimes a bug fix introduces new bugs. Until the new bugs are resolved, debugging may be easier if some parts of the system use the old version of the code, while others use the new version.
- Large teams debugging related changes. During large system maintenance, a single change request often involves several modules and the interfaces between them. Each team member would debug her changes independently, before integrating them with the work of others. To do so she should build an executable system instance with whatever versions of others' modules she deems appropriate, even if some of them still use obsolete, incompatible interface specifications.

This last example, when elaborated, provides many clues as to how a programming environment should support programming with inconsistency. Consider a typical operating system maintenance project, having [5]

- 1,000,000 lines of source code,
- 300 programmers,
- a new release about once per year,
- 300,000 lines of new or changed code per release,

Suppose there were one bug for every 30 lines of changed code, the syntax is correct but before any debugging or testing. That would add up to about 10,000 bugs per release.

Many module changes include modified interfaces. Suppose that each programmer has been assigned to modify a different module. Because tasks progress at different rates, and because some tasks must be redone, several new versions of each module will be produced. Each programmer is responsible for debugging and testing her own code as well as she can before releasing it to others. To do so, she selects the versions of other modules that she thinks will work best with her module. However, the ones she wants to use may not be ready yet. She might choose not to simulate them with a test harness, because test harnesses are often too expensive for early debugging and unit testing. They must be updated whenever the interface changes, which requires both manpower and calendar time. Therefore, programmers often build inconsistent configurations of the real system to use for debugging. In fact, large projects often assign their best analysts to figure out workable, albeit inconsistent, configurations for debugging and testing.

To build, debug and test inconsistent systems, programmers need tools that

- Identify and evaluate the severity of inconsistencies.
- Display the inconsistency information in a useful way, such as by incorporating it in a browser or by using it to compare several alternative module versions, none of which is completely compatible with the rest of the system.
- Protect the programmer from system crashes due to known inconsistencies, by placing firewalls around dangerous code.

3. Kinds of Consistency

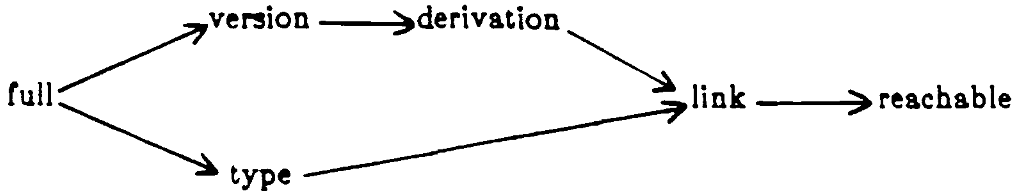
CONMAN formalizes the concept of inconsistency by defining six distinct kinds of consistency, to use for classifying inconsistencies it discovers in programs.

We use the term ***system instance*** to mean an executable representation of a program, typically created by compiling numerous separate program units and linking them together. We assume that the programming language specifies some form of static type checking, and that the programming environment provides a way of uniquely identifying versions of both source code files and derived files (such as object code files). The six kinds of consistency are:

- *Full consistency*: A system instance satisfies the rules that the programming language specifies for legal programs, insofar as they can be checked prior to execution. It also must be version consistent, as defined below.

- *Type consistency.* The system instance satisfies the static type checking rules of the programming language.
- *Version consistency.* The system instance is built using exactly one version of each logical source code file.
- *Derivation consistency.* The system instance is operationally equivalent to some version consistent system instance (which need not have actually been built).
- *Link consistency.* Each compilation unit is free of static type errors, and each symbolic reference between compilation units is type safe according to the rules of the programming language.
- *Reachable consistency.* All code and data that could be accessed or executed by invoking the system through one of its entry points are type safe.

The definitions above have the following partial ordering:



3.1. Full Consistency

The strongest form of consistency is full consistency. The definition tries to capture the ideal world. For example, a system written in Ada is consistent when it is built with exactly one version of each compilation unit, and the units have all been compiled without error in an order compatible with the inter-package dependencies, and then linked.

3.2. Type Consistency

Type consistency depends only on those language rules that deal with the types of identifiers. Operationally, a system instance is type consistent if the compiler reports no type errors for any separately compiled component, and if each identifier whose scope spans more than one compilation unit has the same type in every such unit. (For the C language, the rules checked by the Unix *lint* tool [6] define type consistency across boundaries of separately-compiled modules.)

3.3. Version Consistency

Version consistency is the system property enforced by Unix *make*. For example, if a system written in C contains a source file named "symtab.h", then *make* ensures that all files that `include` it (incorporate its text) are compiled with the latest version.

Version consistency is also important because it provides a practical means of ensuring (or circumventing!) type consistency. Many language systems implement type checking across separately compiled modules by using a file of definitions, called an "include file", to define the types of the identifiers exported from a compilation unit. If the same version of the include file is used to compile the exporting module and every importing module, then the exported identifiers will have the same type throughout their scopes. Conversely, one can trick a compiler into generating code for a module that is not type consistent with other modules, by using different versions of the include file when compiling different modules.

The definition of version consistency includes the word "logical" to cover a special class of systems in which two or more versions of a module are included by design. For example, a test configuration might be created to compare the behavior of two versions of a module. Its system construction model (cf. DSEE [9], Cedar [8]) would treat the two versions as separate logical entities during compilation and linking. A version consistent instance of this system could still use two different versions of the module, because the versions would implement two different logical modules.

3.4. Derivation Consistency

Derivation consistency includes the class of systems that one can build by foregoing unnecessary recompilations, and then use as if they were version consistent. For example, when a type is changed in an include file, only the modules that use the changed type need to be recompiled. Other modules that include the changed file, but do not use the type that was changed, need not be recompiled. Linking the object modules together produces a system that is equivalent to one where all modules were recompiled to use the new version of the changed include file.

3.5. Link Consistency

Link consistency is weaker than type consistency, because it enforces type safety pairwise between compilation units, rather than requiring types to be defined and used consistently system-wide. Nonetheless, this definition is sufficient to support debugging, because the actual executable code is all type safe according to the rules of the language. If each object module is internally type safe, and every data path between modules is type safe, then there is no place in the system where machine code that expects data of one type can operate on data of some other type.

Link consistency can be achieved without type consistency by using different versions of include files with different compilation units. Two units need to use equivalent versions of an included definition only if the link-time interface between them is affected (directly or indirectly) by that definition.

Link consistency describes some situations where a widely-used definition has been changed, but only some of the places where it is used have been rewritten to accomodate the change. Consider a system in which one module defines the type **linked list**, and two other subsystems each use **linked lists** internally, but do not pass **linked lists** between subsystems. This example is depicted in figure 3-1.

Suppose it is decided to change the implementation from singly-linked lists to doubly-linked lists, to enable sequencing in both directions. The programmer would like to try out the doubly-linked implementation in a limited context, before rewriting all of the places it is used. If she rewrites and recompiles the **linked list** module and just one of the subsystems that uses it, the system instance will be link consistent (because every module and every link is type safe), but not type consistent (because some modules were compiled with the singly-linked implementation, and some with the doubly-linked implementation). Assuming that the list representation is directly manipulated by the subsystems that use it (to increase efficiency), the programmer cannot compile the

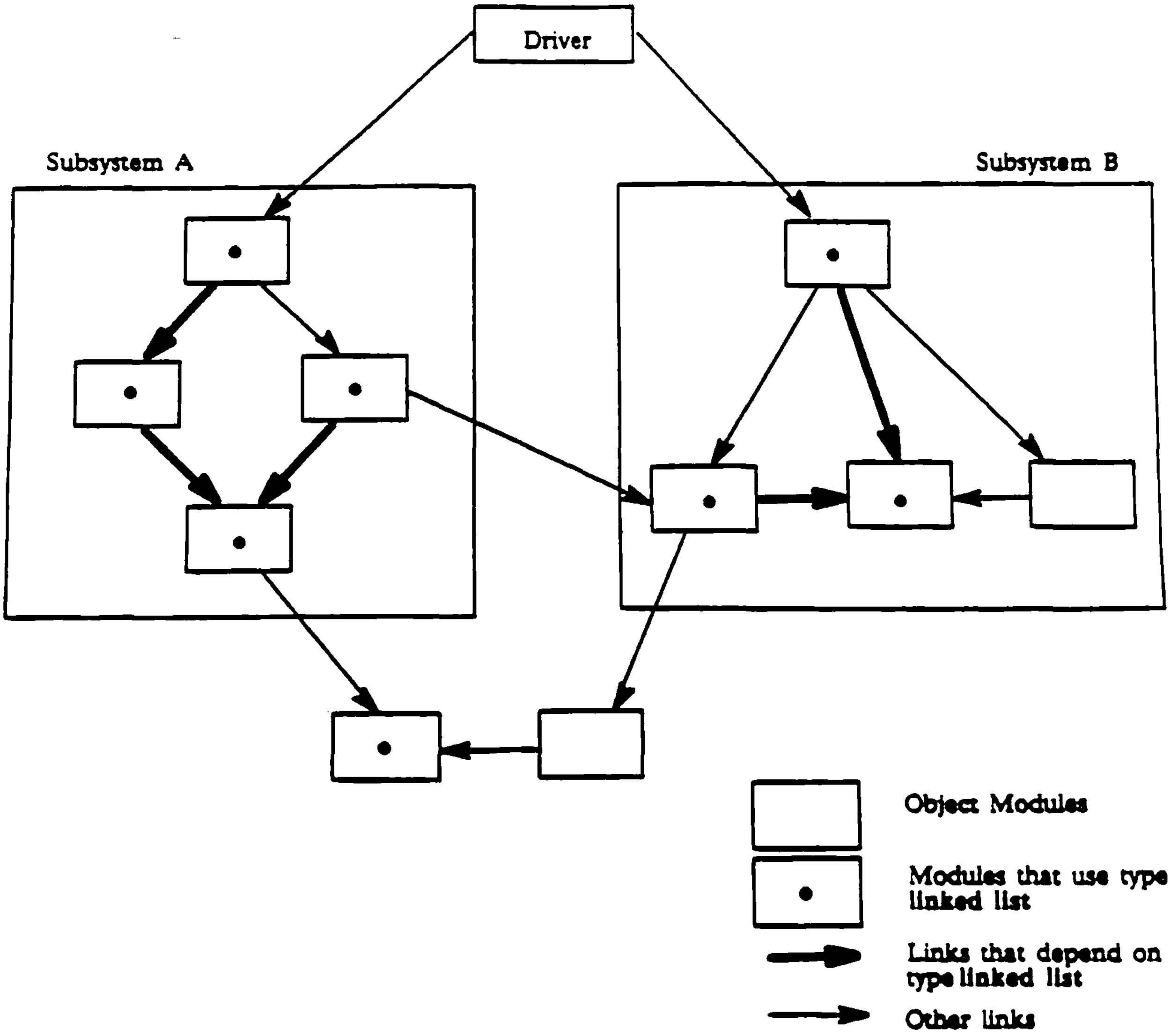

Driver
Subsystem A
Subsystem B
Object Modules
Modules that use type linked list
Links that depend on type linked list
Other links

Figure 3-1: Clusters That Use a Type Independently

second subsystem with the doubly-linked implementation until she rewrites it. Recompiling without rewriting would give lots of error messages, and probably no object code.

Such independent uses of a global type are consistent with sound design principles. A large system is frequently layered into levels, where each level uses services provided by the levels below it, and provides services to the levels above it. In a system that provides a broad range of end-user services, it is not unusual for the middle layers of the system to contain several subsystems that do not call each other at all. In that situation a service type defined by a lower level could be used independently by the subsystems at the next level.

Besides global types, several other language constructs permit multiple coexisting definitions without sacrificing link consistency. For example, Ada's inline procedures and generics both cause a definition to be instantiated separately at each place where it is used. Usually, separate instances of a generic package are treated as unrelated at run time, even though they were derived from a common definition. (Of course, Ada's rules currently forbid version inconsistency.)

3.6. Reachable Consistency

Reachable consistency is useful during development when service routines are written before the external interfaces that use them are ready. Any type errors in unused routines can not interfere with debugging the code that is reachable.

3.7. Automatic Checking

CONMAN checks all six kinds of consistency automatically. Version consistency is checked by straightforward configuration management methods. Type consistency and derivation consistency are checked by the methods used in *smart recompilation* [17]. (Full consistency simply means version consistency and no compilation errors.) Link consistency is checked by a simple method described in the next section. Reachability is checked by incremental, interprocedural data flow analysis, recently made efficient by Ryder and Carroll [14].

4. Reducing the Cost of Consistency

The Unix *make* tool restores version consistency by rederiving any output files that are older than the current versions of the input files from which they are supposed to be built. This can cause many recompilations after only a small change.

Toolpack [12] and smart recompilation reduce the cost of restoring consistency by restoring only *derivation* consistency. Both systems maintain a single, consistent version list of the "latest versions" of each file. They reduce recompilation costs by not rederiving a file when the existing derived file is operationally equivalent to what would be created by rederiving it with the new source file versions.

Toolpack defines "operationally equivalent" to mean "identical contents"; it permits certain attributes such as timestamps to be different. Toolpack uses the same "older than" rule as *make* to trigger recompilation, but avoids some processing steps by noticing when a certain step produces an output file with contents identical to the one it is replacing. This means that using the new output file in a subsequent translation step would be equivalent to using the old version, so the next step is avoided unless other inputs have changed.

Smart recompilation determines equivalence by extracting, from the inputs to a compilation, the set of declarations that actually affect the output files; two output files are equivalent if they are derived from equivalent extracted inputs. (The output files are also allowed to include unused code that differs.) Smart recompilation preprocesses each changed file to identify the declarations that have changed in it. The method then recompiles only the files that actually contain or use the changed declarations.

Smart recompilation succeeds because it performs only local semantic analysis, which it can do cheaply. Local semantic analysis examines each source file in isolation. Any identifiers occurring free in that file are assumed to be declared in some compatible way; they are typically bound by include statements to other files. The analysis produces a dependency file listing the identifiers exported by that file, and the free identifiers on which they depend. The details of smart recompilation are thoroughly explained in [17].

4.1. Checking Link Consistency

To simplify the following sections, we limit our discussion to a simple Pascal programming system, such as provided by the Berkeley Pascal compiler running on Berkeley Unix 4.2. This environment provides a version of Pascal that has been augmented with a separate compilation facility. Procedure headers can be separated from procedure bodies. Typically, the interface to a module is placed in a separate "include" file, which is included in the module that provides the interface and in every module that uses the interface. In the remainder of this paper, we use the term "module" to refer to a normal compilation unit, and "file" to refer to a module or an include file. Our discussion does not cover overloading nor identifiers that are moved between modules during a change. These extensions can be handled analogously to the way smart recompilation handles them.

Link consistency is defined on links between object modules. A link is a (definition, use) pair consisting of an identifier declared *global* in the object module that defines it, and *external* in the object module that uses it. A link is consistent if the definition and the use were compiled using equivalent declarations of the identifier's type. For example, if a procedure **P** with one parameter of type **T** is exported by one module and imported by another, then the two modules must agree that **P** has only one parameter, that its type is **T**, and that **T's** type is equivalent in both modules.

To check link consistency, we first identify the source code constructs that produce global and external references. Then, we use preprocessing methods derived from smart recompilation to analyze dependencies involving these constructs.

The only two kinds of object module links in Pascal are variables and procedures. Where Pascal programs define enumerations, records, constants, etc., the compiler translates them directly into object code, without leaving any links to external identifiers. We know, therefore, that a link exists only where a procedure or variable is exported from one module and imported by another.

To check link consistency, we augment the smart recompilation preprocessor in two ways:

- We divide dependencies into *interface* dependencies and *implementation* dependencies. For example,

  ```
  extern
  procedure P(a:T);
          var b:V;
              . . .
  ```

  This procedure has an interface dependency on type **T**, and an implementation dependency on type **V**.

- For each exported procedure and variable, we record its type signature, in which bound type names are replaced by their definitions, but free type names are treated as primitive. For example,

  ```
  (import type R)
  type Q is integer;
  type T is record
          a: Q;
          b: R
          end

  extern var v: T;
  ```

  In this case, v's type signature would be **record(integer,R)**. (This kind of type signature defines type safety by structural equivalence. It can be easily modified to use name equivalence instead.)

To test whether a link is consistent, we compare the versions of the identifiers that affect the definition site and the use site. We do so in the following steps:

1. Determine which source file versions to associate with the definition site, and which to associate with the use site. These can either be the files that were actually used, or files that are proposed to be used.

2. For both the definition and use sites, locate the source file version that defines the identifier's type.

3. Compare the two definitions for equivalence, as follows:

   a. If the version numbers are different, compare the type signatures. If they are different, the definitions are not equivalent.

   b. For each free identifier in the type signature, compare its two definitions (in the "definition site" versions and the "use site" versions) for equivalence, using this same algorithm recursively.

   c. If all the free identifiers in the type signature are equivalent. the definitions are equivalent.

4. (The results of every comparison should be saved for re-use should the type appear again elsewhere in the signature, or in the signature of another link between the same pair of modules.)

4.2. Smarter Recompilation

Smarter recompilation works by finding clusters of modules that must agree on certain identifier definitions in order to be link consistent. Specifically, clusters are defined with respect to a specific set of global identifiers. Two modules are in the same cluster if and only if they are connected by a link that depends on any of those identifiers. (Modules whose interfaces don't depend on the identifiers at all are not placed in any cluster.) Smarter recompilation saves processing time and programming time whenever a system contains two or more clusters with respect to a set of changed identifiers. The method reduces to smart recompilation when this definition causes all modules to be in the same cluster. It starts with the files that have changed, and at least one module that must be recompiled to test the changes. It then "grows" a cluster of modules that are transitively connected to the starting module via links affected by the changes. These are the other modules that must be recompiled. The algorithm proceeds as follows:

1. Begin with a previous system instance, all relevant source file versions, and the results of preprocessing each of the source files. These results are collected in a data structure that indexes all links, so that it is easy to find which links to check when deciding to recompile a module. The data structure is updated incrementally each time the system instance is modified.

2. Ask the programmer to select a set of file versions she wishes to debug or test. There can be at most one version of each logical module in the system, but the programmer need not choose versions of modules she does not care about.

3. Use smart recompilation to select a set of *build candidates*. Smart recompilation requires there to be a set of "new" file versions and a set of "old" file versions. For this purpose, the versions chosen by the programmer are the new ones, and any conflicting versions are the old ones.

4. Ask the programmer to select an initial *build set* from the candidates. These modules define the context in which she wants to debug or test her change.

5. For each new member of the build set,

    a. Determine which versions of the source files will be included when it is recompiled. Use heuristics to select versions that the user left unspecified, such as "latest", "whatever was used before", or "whatever has already been used in the build set".

    b. If the module's source code has changed, update the link index to reflect any changes.

    c. Using the proposed version bindings, check the consistency of each link between the new member and other modules.

    d. Augment the build set with any candidates that have become link-inconsistent with it.

The total time to check consistency is proportional to $B * I * T$, where B is the size of the build set, I is the average number of identifiers imported and exported from a module, and T is the average number of identifiers that must be tested for equivalence in the course of validating a link.

Smarter recompilation can be generalized to more complicated translation tools, and additional kinds of derived files. For example, consider a system written in Ada. The Ada compiler would generate interface files (`.int` files, containing compiled specifications) and object code files (`.obj` files, containing package bodies); the compiler would read in interface files when compiling modules that depended on them. Suppose main subprogram **X** depends on package specifications **Y** and **Z**, and package specification **Y** depends on **Z**. Compiling **X** requires a consistency check between **Y** and **Z**, to ensure that **Y** was compiled with a compatible version of **Z**. This processing model is diagrammed in figure 4-1.

In this situation, the concept of "link" generalizes to "name binding". Each compilation step resolves free names in some of its inputs by binding them to definitions exported by other inputs. Since any exported definition could be involved in a binding,

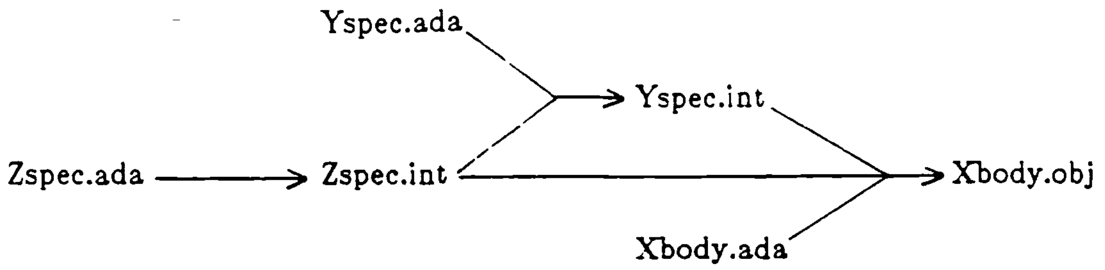


Figure 4-1: Compiling a Small Ada Program With Transitive Dependencies

the preprocessor would keep type signatures for all exported identifiers. Because the inputs to a compilation step are sometimes produced by other compilation steps, there can be version conflicts between inputs to compiles as well as to the link step. The consistency checking algorithm must be augmented to account for such complications in the version selection lists.

Smarter recompilation can be generalized further, to a broad class of translators and derived files, including program generators (such as Unix utilities *lex* and *yacc*), and distributed execution environments. "Compilation" generalizes to any translation step that produces an identifier definition or use based on input definitions and uses. For each "source code" language in the system, one would look for the kinds of identifier declarations that translate into unresolved references in derived files. For each such kind of identifier, a preprocessor would perform local semantic analysis to determine the equivalent of a type signature. Then, each tool that performs name binding can be preceded by an analysis step that uses version lists and type signatures to identify link inconsistencies.

In summary, smarter recompilation reduces the cost of restoring consistency by enforcing only link consistency, rather than derivation consistency. It interacts with the programmer to choose versions relevant to the current task, then performs the least number of compilations necessary to construct a system instance that is link-consistent with those choices.

5. An Environment for Programming with Inconsistency

CONMAN is a programming environment that helps the programmer interactively construct and debug inconsistent systems. The systems may contain different kinds of inconsistency in different places. The environment consists of an object base and a set of tools, consisting of a browser, a compiler, consistency analyzers, an incremental linker, a flow analyzer, a debugger, a test coverage analyzer, and an automated maintainer's assistant. Each is based on available technology, modified to handle inconsistent systems.

The object base is an integrated database of software artifacts [11, 1]. Each file is stored as an object, together with attributes and relations that represent its relationships to other parts of the system. The objects belong to a class hierarchy, with multiple inheritance. Tools in the system can be classified as either foreign tools or native tools. Foreign tools have no knowledge of the environment; they exchange data with the environment through an envelope that sets up an execution environment, calls the tool, and collects its results. Native tools can use the object base directly, such as to store dependencies between source files or to analyze inconsistencies in a desired system instance.

The compiler and linker are augmented with preprocessors to collect type signatures, which the analyzers then use to detect inconsistencies.

The browser helps the programmer construct a description to build. (We call this description a BCT for compatibility with the Domain Software Engineering Environment's (DSEE's) Bound Configuration Thread [9].) A structure editor is a promising type of browser for this application. Through it, the programmer can not only construct the BCT itself, but can also examine its connections with the rest of the object base.

The programmer starts by examining the BCT for some previous system build. The editor presents her with all the new module versions that have been created since the last system build, and asks her which ones she would like to use. The programmer assigns new version bindings to the derived objects she wants rebuilt. As the programmer makes the version choices, the editor highlights version inconsistencies and schedules background tasks to classify them further. Zooming shows details of an inconsistency, including its severity and the specific identifiers involved. The programmer can respond to an inconsistency by:

- Selecting modules to recompile.
- Choosing different source versions.
- Substituting previously compiled object files from the derived object pool (cf. DSEE).
- Approving the inconsistency.

As each part of the BCT is approved, its derivation begins. Any warning or error messages that result are presented to the programmer, who can further modify the BCT if she wishes.

The linker and debugger cooperate to protect the programmer from link inconsistencies. The linker inserts a debugger hook at each inconsistent link, so that execution will stop before the code that uses the link is executed. The debugger then permits the programmer to either move the program counter to a safer place, or continue execution at her own risk.

The BCT description language allows the programmer to permit two versions of an object module to coexist. The linker supports this by accepting multiple definitions of global identifiers, and linking each use to the definition with the correct type.

The test coverage analyzer produces a database for each test indicating the code it covers. On request, it compares this data to the link inconsistencies in a system instance, and tells the programmer which tests are safe and which are not.

The maintainer's assistant is facility for automating mundane programming tasks in a controlled way, called *opportunistic processing*. Whenever a programmer makes a manual change to a source file, it schedules appropriate analysis and compilation tools to run in background, as resources permit. It monitors the costs of compilation and linking, and uses them to estimate the costs of rebuilding after a change. This information is fed back to the user through the browser. The analyzer performs the consistency analysis in background, so that the information is ready when the programmer is ready to edit her BCT. It also maintains an agenda of modules needing rewriting due to changed interfaces.

This combination of tools helps the programmer keep track of inconsistencies, analyze their severity, estimate the cost of recompiling to remove them, and helps select test cases that avoid them. It also protects the programmer from inadvertently executing inconsistent code, while still allowing her to do so if she insists.

6. Implementation

Smarter recompilation has been implemented for the C language, as a Master's thesis at Columbia University [10]. It was constructed by making source code modifications to the portable C compiler and *make*. The prototype successfully handles such details as macros, structs, unions, and even bit field sizes and anonymous struct fields. Although it has not been tested on large systems, it demonstrates that the cost of adding the functionality to existing tools is reasonable.

The CONMAN programming environment is being assembled from a collection of other systems being developed and/or used at Siemens RTL. The object base and controlled automation system are being designed in conjunction with the Marvel project [7]. The browser is being implemented with the DOSE structure editor prototyping system [2]. The system modeling language draws ideas from both DSEE and Cedar, but adds facilities for conveniently naming and manipulating derived objects, and for mapping source-language dependencies into build step input-output dependencies. For example, a system model could declare that one source file called procedures in another source file; the system builder would automatically link the second file into system instances that used the first. The debugger will be the Sun Unix *dbxtool* [16], which will be primed with a set of breakpoint commands generated by the linker. Test coverage tools and methods will be drawn from the Asset project [13, 4]. Reachability analysis will be based on Ryder's methods, in a future version of the system.

7. Conclusions

Inconsistency is commonplace in real software projects. It is permitted to remain because it is often more cost-effective than consistency.

Automatically recognizing several gradations of consistency permits the programmer to choose the level appropriate to her task. Better tools can reduce the cost of restoring consistency, but not the cost of rewriting all the code affected by a change. Smarter recompilation permits derivation inconsistency without sacrificing run-time type safety, and thereby permits some rewriting to be deferred, reducing the length of the edit-compile-debug cycle and reducing the amount of synchronization needed between programmers.

The CONMAN configuration management project is developing a programming environment that helps a programmer to select different degrees of consistency in different parts of her system. The tools will recognize and keep track of inconsistencies for her, and place firewalls around them during debugging, but will not force her to remove them. By this approach, CONMAN will help the programmer live with inconsistency.

8. Acknowledgement

Our thanks to Walter Tichy for his many helpful comments on earlier versions of this paper, and to Harris Morgenstern for implementing smarter recompilation as his master's project at Columbia University.

References

[1] Philip A. Bernstein.
Database System Support for Software Engineering.
In *9th International Conference on Software Engineering*, pages 166-178. Monterey, CA, March, 1987.

[2] Peter H. Feiler, Fahimeh Jalili, and Johann H. Schlichter.
An Interactive Prototyping Environment for Language Design.
In *Proceedings of the Hawaii Conference on System Sciences*. January, 1986.

[3] Stuart I. Feldman.
Make -- A Program for Maintaining Computer Programs.
Software--Practice and Experience , April, 1979.

[4] P. G. Frankl and E. J. Weyuker.
A Data Flow Testing Tool.
In *Proceedings of the IEEE Softfair II*. San Francisco, December, 1985.

[5] Klaus Gewald.
Private Communication.
June, 1987.

[6] S. C. Johnson.
Lint, a C Program Checker.
In *Unix Programmer's Manual Supplementary Documents*. 4.2 Berkeley Software Distribution, 1984.

[7] Gail E. Kaiser and Peter H. Feiler.
An Architecture for Intelligent Assistance in Software Development.
In *Ninth International Conference on Software Engineering*, pages 80-88. IEEE, Monterey, CA, March, 1987.

[8] 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*, pages 1-13. June, 1983.

[9] David B. Leblang and Gordon D. McLean, Jr.
Configuration Management for Large-Scale Software Development Efforts.
In *Workshop on Software Engineering Environments for Programming-in-the-Large*, pages 122-127. June, 1985.

[10] Harris M. Morgenstern.
An Inconsistency Management System.
Master's thesis, Columbia University Computer Science Department, March, 1987.

[11] John R. Nestor.
Toward a Persistent Object Base.
In Reidar Conradi, Tor M. Didriksen and Dag H. Wanvik (editors), *Advanced Programming Environments*, pages 372-394. Springer-Verlag, Berlin, 1986.

[12] Leon J. Osterweil.
Toolpack -- An Experimental Software Development Environment Research Project.
IEEE Transactions on Software Engineering , November, 1983.

[13] S. Rapps and E. J. Weyuker.
Selecting Software Test Data Using Data Flow Information.
IEEE Transactions on Software Engineering 11(4):367-375, April, 1985.

[14] Barbara G. Ryder and Martin D. Carroll.
An Incremental Analysis Algorithm for Software Systems.
Technical Report CAIP-TR-035, Department of Computer Science, Rutgers University, March, 1987.

[15] Robert W. Schwanke and Gail E. Kaiser.
Smarter Recompilation.
ACM Transactions on Programming Languages and Systems , submitted for publication.

[16] *Debugging Tools for the Sun Workstation*
Sun Microsystems, Inc., 1986.

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

[18] Alexander L. Wolf, Lori A. Clarke, and Jack C. Wileden.
Ada-based Support for Programming-in-the-Large.
IEEE Software , March, 1985.

Change Management for Very Large Software Systems

Yoelle S. Maarek
Technion, Israel Institute of Technology
Computer Science Department
Haifa 32000, Israel

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

Abstract

Very large software systems tend to be long-lived and continuously evolving. Purely managerial means for handling change are often adequate for small systems, but must be augmented by technological mechanisms for very large systems simply because no one person can understand all the interactions among modules. Many software development environments solve part of the problem, but most consider change only as an external process that produces new versions. In contrast, INFUSE concentrates on the actual change process and provides facilities for propagating changes that affect other modules. INFUSE structures the set of modules involved in a change into a *hierarchy of experimental databases*, where each experimental database isolates a collection of modules from the changes made to other modules and the hierarchy controls the integration of changes made to separate subsystems. The focus of this paper is on the clustering algorithm that automatically generates and maintains this hierarchy according to the strengths of interdependencies among modules as they are added and modified during development and maintenance.

To appear in **Seventh Annual International Phoenix Conference on Computers and Communications**, Scottsdale, AZ, March 1988.

1. Introduction

A Very Large Software System (VLSS) is composed of a large number of interdependent modules that typically undergo numerous changes during their lifetime. By *module*, we mean a separately compilable syntactic unit, such as an Ada™ package, a Modula-2 module or a C source file. As such modules change, they often diverge from their specifications and the number of interface errors grows [12]. Change management tools are needed to coordinate programmers as they modify their modules, to propagate interface changes to dependent modules, and to enforce cooperation among programmers towards their goal of preventing interface errors. We describe a new algorithm that provides the basis for the INFUSE change management facility.

The change process in VLSS is considerably more complex than for small systems. For instance, determining the *extent* of a change (what is affected by the change) and its *implications* (what is necessary for restoring consistency after the change) is complicated by the sheer number of the interdependencies among pieces of the system. Moreover, an apparently simple change can easily *cascade* in unpredictable ways, requiring several rounds of changes for restoring consistency. Other problems such as the handling of temporary inconsistencies or the support of the iterative process of propagating changes become much more complex as the size of the system increases. INFUSE handles all these problems for *syntactic consistency*, that is, those inconsistencies that can be detected by a standard compiler; we are investigating extending INFUSE to semantic inconsistencies [14].

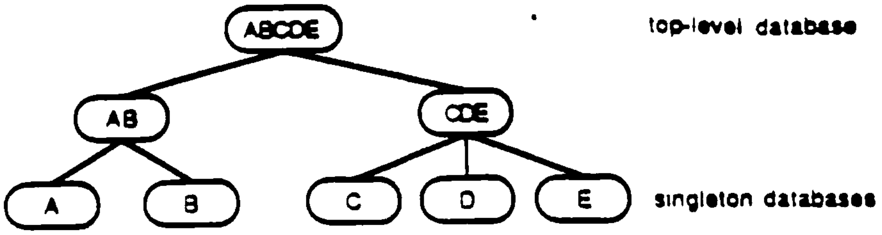
Several other tools have addressed simple cases of these problems. Make [3] automates recompilation of all dependent modules after source changes; it determines the extent of changes, and restores consistency by recompiling everything which might be affected, thus the first and fifth problems are solved in a rough way. Cedar's *System Modeller* [9] and Apollo's *Domain Software Engineering Environment* [10] (DSEE™) give programmers more control over dependencies among distinct versions of modules, but provide little more help than Make with respect to coordination and cooperation. None of these tools directly monitor the change process; DSEE permits each programmer to set up his own monitors to carry out specified actions whenever certain events occur, such as adding a new version to the baseline system. In contrast, INFUSE does not wait for deposit into the baseline system to perform its actions.

The NuMIL prototype [11] and Smile [6] are both much closer to INFUSE. The NuMIL prototype determines the impact of alterations based upon upward compatibility but provides analysis rather than control of the change process. Smile introduced the notion of an *experimental database*, which is a (virtual) copy of the baseline system that permits changes only to the subset of the system reserved by the user, isolating these changes from other programmers. INFUSE extends the notion of experimental database to a multiple-level hierarchy, and, unlike Smile, gathers *automatically* the modules into databases.

Previous papers on INFUSE have outlined its basic philosophy and discussed its automatic application of consistency-checking tools [15, 7]. In this paper, we briefly explain the INFUSE methodology and describe its use of a hierarchy of experimental databases for controlling and coordinating changes. We then present the algorithm INFUSE uses to automatically build and maintain this hierarchy.

2. The Hierarchy of Experimental Databases

INFUSE places all the modules involved in the change process in a distinguished experimental database: the *top level database*. This change set is normally chosen manually by a system analyst to attempt to satisfy the particular group of modification requests (MRs) appropriate for the next patch or release. Since the more numerous the modules in the change set, the more difficult the determination of the implications and the extent of changes, the top level experimental database is divided into several subsets that are themselves experimental databases. The implications and extent of changes in these smaller databases are easier to determine than in the top level one. By iteratively dividing the experimental databases into smaller and smaller databases, INFUSE limits the interactions that the programmers must cope with at one time. The hierarchy of experimental databases is the result of this division. The root of the hierarchy is the top level database, and each hierarchy level, from coarse to fine, is a partition of the original experimental database; a leaf contains a single module (see figure 1).



1. A hierarchy of experimental databases

The actual changes are made by editing the modules within their singleton databases. Once a singleton database is self-consistent it can be deposited into its parent database. An analysis tool is applied to determine this self-consistency: everything both defined and used within the module is used correctly with respect to its definition and everything used but not defined within the module is always used in a compatible manner. Once a singleton database is deposited, INFUSE coordinates and manages the iteration of changes by applying the following process recursively on every experimental database from the singletons to the top level (not included):

- When all child databases have been deposited into their parent, INFUSE invokes an analysis tool for performing change propagations within this parent database and checking the consistency among its subset of the changed modules. An analysis tool such as Lint [5] can be applied to the modules after all changes are made, or errors can be detected incrementally as by Mercury [8].
- If the database is self-consistent, then it can be deposited into its own parent database.
- If not, the local inconsistencies are detected and reported to the responsible programmers, who then negotiate and agree on new modifications for resolving the conflicts. The database, or only the part of it requiring further changes, is repartitioned into a subtree, and the singleton databases of that subtree are modified. The process above is reapplied to these experimental databases until the problematic database becomes self-consistent and can be deposited into its parent database.

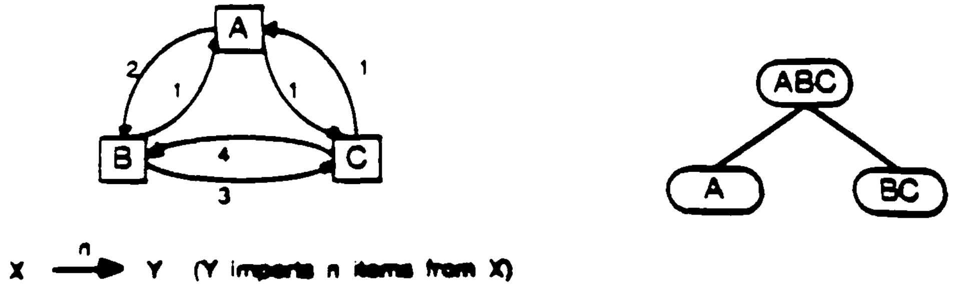
Finally, when all descendants have been deposited into the top level and it is both self-consistent and consistent with the modules of the baseline system that do not appear in the top level, the top level is itself merged back into the baseline.

The goal of this process is to support a widely accepted rule-of-thumb of software engineering: errors discovered early are much less costly to repair than those discovered late. The purpose of the hierarchy is to cluster together at the low levels those collections of modules where changes are most likely to lead to interface errors, ensuring early detection, and those collections of modules where the changes are unlikely to affect each other are not brought together until the high levels of the hierarchy.

Thus we need a measure for gathering collections of modules where changes are more or less likely to lead to interface errors. Our measure is the *interconnection strength* among pairs of modules, an approximation to the oracle that would tell us exactly how the future changes will effect other modules. Our approximation is based on the intuition that the probability of an interface error between modules M and

N is proportional to k, where module M uses i facilities imported from N, N uses j facilities from M, and k is the sum of i and j.

Consider three modules, A, B and C, importing and exporting items between each other, where an item is an importable syntactic unit of the programming language such as a procedure, a data type, *etc*. Since B and C are more strongly connected to each other than to A (see figure 2), they should be gathered in the same experimental database, A being added to them only at an upper level of the hierarchy.



2. Clustering according to the interconnection strength

3. Building a Hierarchy of Experimental Databases

There are two ways to build a hierarchy: top-down or bottom-up. The first way corresponds to *partitioning* methods and the second to *clustering*. In the partitioning approach we recursively divide the top level experimental database until reaching the singleton databases. When dividing a database, we need to know *a priori* the number of subsets we want to obtain; this approach is *model-driven*. Since the modules are available before beginning the construction of the hierarchy, we prefer the *data-driven* approach of clustering methods.

There is a strong analogy between the construction of a hierarchy of experimental databases and the hierarchical clustering of a set of objects. Clusters are groups of objects whose members are "more similar" to each other than to members of another group. The similarity between two clusters is measured by a *dissimilarity index*: the more similar any two clusters, the lower their dissimilarity index. There exist numerous hierarchical clustering algorithms [17] that differ only by the choice of the measure of similarity between clusters. Experimental databases correspond to clusters of modules, where the measure of similarity between clusters is the interconnection strengths between modules.

Hierarchical clustering is usually divided into two tasks: The first consists of applying the following general method [1] on the objects to be clustered.

- Identify the two clusters (initially a single object) that are the most similar according to the dissimilarity index.
- Merge them together into a single cluster.
- Repeat this process iteratively until there is only one cluster.

Every iteration in the clustering process forms a new *level clustering* by adding a new cluster and removing the merged clusters. The final output of the clustering process is often pictured as a hierarchy whose levels are these successive level clusterings; the hierarchy arises because each new cluster merges its two children in the immediately preceding level. The second task consists of selecting from this hierarchy the 'meaningful' level clusterings according to the needs of the application. This is usually done by an analyst since it requires knowledge of the application domain.

INFUSE expects a hierarchy where the arity of each experimental database is specific to the actual interconnection strengths of the modules in the change set. Our proposed algorithm combines the two tasks described above, without recourse to a human analyst; in particular, only the 'meaningful' level clusterings are actually generated, thus forming directly the hierarchy of experimental databases supported by INFUSE.

4. The Arity Controlled Clustering Algorithm

Unlike classical hierarchical clustering algorithms, our algorithm treats the level clusterings as temporary as long as they are not 'meaningful'. The temporary level clusterings are said to be *prospective*, whereas each level clustering that is selected is said to be *frozen*. The sequence of frozen level clusterings gives the hierarchy of experimental databases. To freeze level clusterings, the algorithm evaluates the similarity between the prospective level clusterings and an *exemplar*. We define the *arity* of an experimental database as its number of immediate descendants in the next level of the hierarchy. The similarity is computed by measuring the statistical dispersion of arities through a variance function defined as follows:

Let *LC* be a prospective level clustering and $\{x_1,x_2,\ldots,x_k\}$ the sequence of the arities of its k experimental databases; x_i represents the number of descendants that the i^{th} database of *LC* has in the previous frozen level clustering. The exemplar is defined by a single coefficient α. We define the measure v_α for evaluating the similarity between the *LC* and the exemplar by:

$$v_\alpha = \frac{1}{k}\sum_{i=1}^{k}(x_i-\alpha)^2$$

The initial frozen level clustering is composed of the singleton databases. Given this initial level clustering and an example arity for all the databases of the next level, the algorithm computes all the successive prospective level clusterings and freezes the one that minimizes our variance measure in order to determine the next level of the hierarchy. However, it is too costly to compute all the forthcoming level clusterings and to backtrack to the absolute optimum. In practice, the algorithm instead finds a local optimum, where the degree of locality is defined by a lookahead coefficient — that is, how many prospective level clusterings to generate.

The example arity is generated by the algorithm itself. It remembers past hierarchies involving the same software system, and uses previously successful values whenever possible. When not possible, such as in the early stages of the system's development when few changes have been made, the exemplar is chosen randomly or provided by an analyst.

Controlling the arity of experimental databases is reminiscent of the model-driven partitioning approach we rejected, where each partition splits an experimental database in a number of sets decided *a priori*. The similarity is misleading. When our algorithm controls the clustering arity of every level clustering, it treats this level arity as an exemplar that it is not necessary to meet. It chooses among several prospective level clusterings the one closest to the exemplar but does not force the construction of a level clustering identical to the exemplar.

We present a simplified version of our algorithm, with a lookahead equal to one, in figure 3. The overall time complexity of our algorithm is $O(n^2\log(n))$, the same as the classical clustering algorithms [16].

even though we introduce supplementary computation by controlling the variance of the arities.

Input:	The interconnection strength values between pairs of modules. The coefficients *a,b,c,d* for computing the interconnection strengths. The exemplar arity for every level clustering.
Output:	A hierarchy of experimental databases.

Start from the initial level clustering,
$L = \{\{m_1\},\{m_2\},\ldots,\{m_n\}\}$,
whose elements are the
singleton experimental databases reduced to a single module. Get
the value of α for the next level. The current prospective level
clustering is set to the previous frozen level clustering. The arity
of each of its experimental databases is set to 1.

While there are more than two experimental databases in the
current level clustering **do:**

1. **Construct** the next prospective level clustering, *NLC*, by merging together the two experimental databases of the current level clustering that maximize δ (if there is more than one pair of clusters which realize this maximum, one of them is chosen arbitrarily. This new experimental database is their ancestor in the hierarchy.
2. **Update** the interconnection strength values.
3. **If** the v_α of *NLC* is greater than that of the current level clustering, **freeze** the current level clustering. The arities of the experimental databases of the current level clustering are set to one. Get the value of α for the next frozen level.
4. **Else** the *NLC* becomes the current level clustering.

End While

Merge together the last two clusters of the current level clustering,
in order to form the last frozen level of the hierarchy.

---

3. The arity controlled clustering algorithm

The sequence of all the frozen level clusterings gives us the hierarchy of experimental databases.

5. Maintaining the consistency of the hierarchy

Changes made to modules may invalidate the hierarchy, in the sense that it no longer correctly reflects the interconnection strengths among modified modules. Two main classes of modifications can lead to invalidation:

1. Modifying the interface of a module, since the structure of the hierarchy is based on interconnection strength.
2. Adding a module to the hierarchy; a planned modification may involve creating a new module or conflict resolution may require modifying modules in the baseline but not in the original change set.

It is possible to treat a module whose interface has been modified in the same way as a new module. The older version is removed from the hierarchy, and the new one added. Therefore, we focus on adding a module to the hierarchy. The roughest way of updating the hierarchy is to recluster the entire change set, including the new module. This is too costly: Many experimental databases not affected by the modification would also be reprocessed, and deposits to these databases would have to be repeated. However, if we reject full reclustering and instead make only local changes, we cannot guarantee the resulting hierarchy is as 'good' as the one produced by our clustering algorithm. Fortunately, most practical cases (where relatively few interfaces are changed) affect only a small portion of the hierarchy and only this portion may not be the same as had full reclustering been applied.

In most cases, our *incremental* reclustering algorithm works as follows. The new module, M, is added to the top-level experimental database. Then it is merged into the next level experimental database with which it has the highest interconnection strength. This process is applied recursively until a singleton database is reached. The singleton is changed to contain two modules (the original and M) and has two new singleton children.

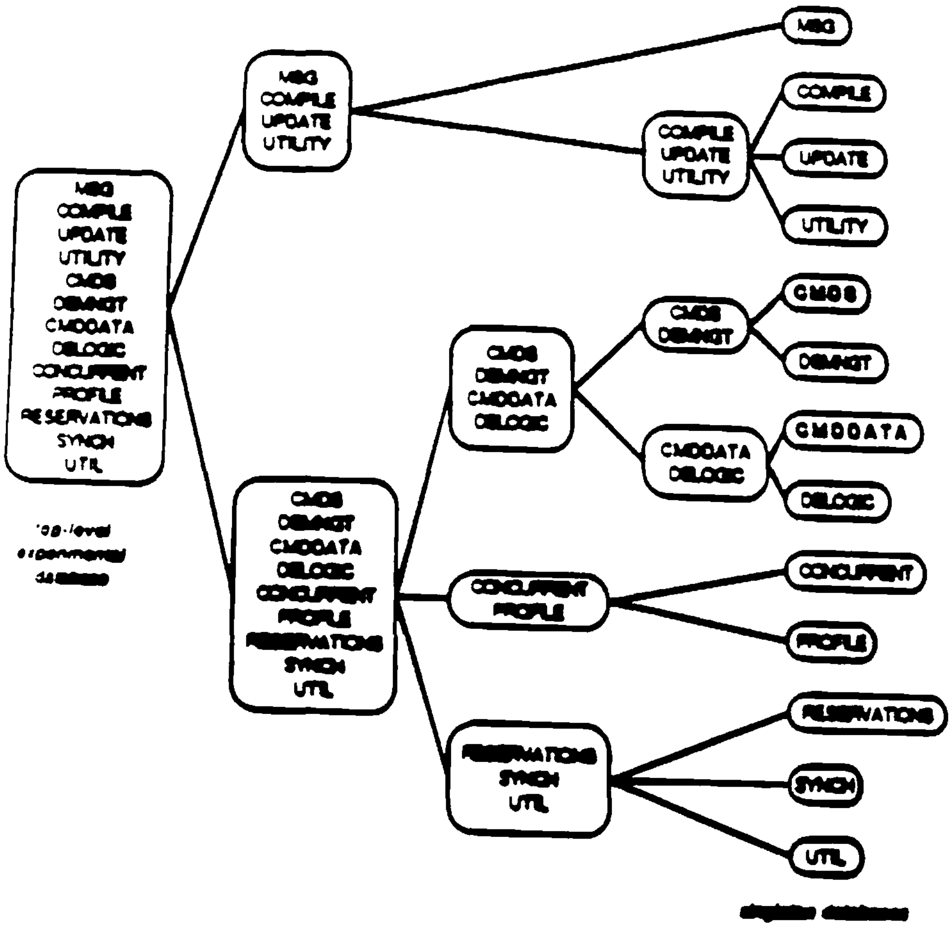
This naive algorithm works very nicely except for special cases where M is only weakly connected to each of the children of an experimental database, which occurs most frequently with a brand new module that is empty. Such a module is called an *outlier*. To determine that the module M is an outlier among several databases, $E_1, E_2, \ldots, E_k$, our incremental algorithm computes the interconnection strength values between every pair of databases in the set: $\{E_1, E_2, \ldots, E_k, \{M\}\}$. If the maximum is realized by a pair that does not include $\{M\}$, it means that M is less connected to any E_i than the E_i are interconnected among themselves. In this case, M is added as a new child of the parent experimental database.

6. Some empirical results

We selected Smile — a multiple-user programming environment for C developed as part of the Gandalf project [4] — as our test case for this paper since it is a medium sized system where the change processes involve few enough modules to be illustrated nicely in figures. We have also applied our clustering algorithm to the 60 modules of ALOE [2], also from the Gandalf project, as well as to several much smaller systems. Our example assumes that two Smile modules, CMDS and CMDDATA, are to be modified extensively. Therefore, the analyst also places the set of eleven modules related to them in the top-level experimental database, since these may also need to be modified. The interconnection strength values between these modules are automatically extracted from the program text and given in the following matrix (figure 4). Utility modules imported everywhere are not considered, since they are handled specially [15].

```
CMDDATA   | x
CMDS      | 27   x
COMPILE   | 17   8    x
CONCUR.   |  2  13    3    x
DBLOGIC   | 31  23   31    4    x
DBMGNT    | 42  51   10    1   25    x
RESERV.   | 18   9    0    0    1    6    x
UPDATE    |  5  21   52    3   30   46    2    x
UTILITY   | 27  21   28    2    0    6    0   31    x
MSG       |  1   0    0    0    0    0    0    0    0    x
PROFILE   |  0   2    1    3    0    2    0    0    3    0    x
SYNCH     |  2   5    0    0    0    4    1    0    0    0    0    x
UTIL      | 18  17    9    0    5   19    4    4   14    3    1    6    x
          +-----------------------------------------------------------------
             |    |    |    |    |    |    |    |    |    |    |    |    |
         CMDDATA  | COMPILE | DBLOGIC | RESERV. | UTILITY | PROFILE |   UTIL
                 CMDS     CONCUR.   DBMGT     UPDATE     MSG      SYNCH
```

4. Matrix of interconnection strengths



5. Hierarchy of experimental databases for Smile

Given this data, our algorithm produces a hierarchy (see figure 5) similar to the one manually identified by a Smile 'expert'. When applied to the larger ALOE, the hierarchies obtained are still very similar but not identical to the ones computed by hand.

7. Conclusion

We have described INFUSE, a software development environment that supports change management in addition to recompilation and version control after changes. Unlike other tools, INFUSE assists programmers during rather than after the change process. Conflicts are detected early when they are relatively inexpensive to repair, rather than later after the entire change process has completed and recompilation and testing has begun. The major contribution of this paper is the presentation of a new clustering algorithm which makes such conflict detection and resolution possible. From the change set, INFUSE automatically builds a hierarchy of experimental databases where the most strongly connected modules are collected together into the 'natural' clusters specific to the VLSS and negotiation of module interface errors are enforced. INFUSE thus provides practical support for managing and coordinating changes in very large software systems. We are currently extending INFUSE with mechanisms to combine stubs and test drivers hand-constructed for unit testing to operate as test harnesses for the integration among strongly connected clusters of modules.

Acknowledgments

Dewayne Perry and Gail Kaiser first developed INFUSE in the context of the Inscape project [13] at AT&T Bell Laboratories, where Peggy Quinn was responsible for extracting the Smile and ALOE dependency matrices. Travis Winfrey, Ben Fried and Pierre Nicoli completed the implementation under the direction of Bulent Yener. Galina Datskovsky, Michael Elhadad, Steve Popovich and the anonymous referees read earlier versions of this paper and made useful criticisms and suggestions.

This research was supported in part by grants from AT&T Foundation, IBM, Siemens Research and Technology Laboratories, and New York State Center of Advanced Technology — Computer and Information Systems, and in part by a Digital Equipment Corporation Faculty Award.

References

[1] E. Diday, J. Lemaire, J. Pouget and F. Testu.
Elements d'Analyse des Donnees.
Dunod, 1982.

[2] Peter H. Feiler and Raul Medina-Mora.
An Incremental Programming Environment.
IEEE Transactions on Software Engineering SE-7(5):472-482, September, 1981.

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

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

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

[6] Gail E. Kaiser and Peter H. Feiler.
Intelligent Assistance without Artificial Intelligence.
In *Thirty-Second IEEE Computer Society International Conference*, pages 236-241. San Francisco, CA, February, 1987.

[7] Gail E. Kaiser and Dewayne E. Perry.
Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution.
In *Conference on Software Maintenance*, pages 108-114. Austin, TX, September, 1987.

[8] Gail E. Kaiser, Simon M. Kaplan and Josephine Micallef.
Multiuser, Distributed Language-Based Environments.
IEEE Software :58-67, November, 1987.

[9] Bulter W. Lampson and Eric E. Schmidt.
Organizing Software in a Distributed Environment.
In *SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, pages 1-13. San Francisco, CA, June, 1983.
Proceedings published as *SIGPLAN Notices*, 18(6), June 1983.

[10] David B. Leblang and Gordon D. McLean, Jr.
Configuration Management for Large-Scale Software Development Efforts.
In *GTE Workshop on Software Engineering Environments for Programming in the Large*, pages 122-127. June, 1985.

[11] K. Narayanaswamy.
A Framework to Support Software System Evolution.
PhD thesis, University of Southern California, May, 1985.

[12] D.E. Perry and W.M. Evangelist.
An Empirical Study of Software Interface Errors.
In *International Symposium on New Directions in Computing*, pages 32-38. Trondheim, Norway, August, 1985.

[13] Dewayne E. Perry.
Programmer Productivity in the Inscape Environment.
In *IEEE Global Telecommunications Conference*, pages 428-434. December, 1986.

[14] Dewayne E. Perry.
Software Interconnection Models.
In *9th International Conference on Software Engineering*, pages 61-69. Monterey, CA, March, 1987.

[15] Dewayne E. Perry and Gail E. Kaiser.
Infuse: A Tool for Automatically Managing and Coordinating Source Changes in Large Systems.
In *ACM Fifteenth Annual Computer Science Conference*, pages 292-299. St. Louis, MO, February, 1987.

[16] J.C. Simon.
Etudes et Recherches en Informatique: La Reconnaissance des formes par algorithmes.
Masson, 1985.

[17] R. R Sokal and P. H. Sneath.
Principles of Numerical Taxonomy.
W. H. Freeman, 1973.

Final version to appear in 10th International Conference on Software Engineering, Singapore, April 1988.

Models of Software Development Environments

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

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

August 1987

Abstract

We present a general model of software development environments that consists of three components: policies, mechanisms and structures. The advantage of this formalization is that it distinguishes precisely those aspects of an environment that are useful in comparing and contrasting software development environments. We introduce four classes of models by means of a sociological metaphor that emphasizes scale: the individual, the family, the city and the state models. The utility of this taxonomy is that it delineates the important classes of interaction among software developers and exposes the ways in which current software development environments inadequately support the development of large systems.

Environments reflecting the individual and family models are the current state of the art. Unfortunately, these two models are ill-suited for the development of large systems that require more than, say, 20 programmers. We argue that there is a qualitative difference between the interactions among a small, "family" project and a large, "city" project and that this qualitative difference requires a fundamentally different model of software development environments. We illustrate the city model with Infuse and ISTAR, the only two environments we know of that instantiate this model, and show that there is a pressing need for further research on this kind of environment. Finally, we postulate a state model, which is in need of further clarification, understanding and, ultimately, implementation.

1. Introduction

A model is useful primarily for the insight it provides about particular instances and collections of instances. By abstracting away non-essential details that often differ in trivial ways from instance to instance and by generalizing the essential details into the components of the model, we derive a tool for evaluating and classifying these instances — in ways that we had not thought of before we constructed our model. It is with this purpose in mind — classification and evaluation — that we introduce a general model of software development environments (SDEs). Our model consists of three components: policies, mechanisms and structures.

Once we have defined this general model of software development environments, there are various points of view from which we might classify environments. We might, for example, classify the SDEs according to their coverage of the software life cycle; or classify them according to the kinds of tools that they provide, contrasting those that provide a kernel set with those that provide an extended set; etc. Each of these classifications yields useful comparisons and insights.

Another important point of view, which we have not seen in the literature, is a classification of SDEs relative to the problems of scale — what is required of software development environments for projects of different sizes taking into account the numbers of programmers and the length of the project as well as the size and complexity of the system. Note that the distinction between programming-in-the-small and programming-in-the-large [7] has some intimations of the problems of scale. However, this distinction is basically one of single-unit versus multiple-unit systems and captures only a small part of this problem. We build software systems that range from small to very large, and will be able to build even larger systems as hardware gets cheaper and more powerful. What has not been sufficiently considered is the effect of the scale of systems on the tools needed to build them*.

Thus, the main focus of this paper — and, indeed, of our research — is the problem of scale. We introduce a classification of SDEs in terms of a sociological metaphor that emphasizes this problem of scale and provides insight into the environmental requirements for projects of different sizes. This metaphor suggests four classes of models: individual, family, city and state. The individual and family classes are the

---

* For example, Howden [18] considers SDEs for medium and large systems only from the standpoint of capitalization and richness of the toolset.

current state of the art but are inadequate for building very large systems. We argue that the city model is adequate but that very little attention has been given to this class. Further, we argue that future research and development should address the city model and the proposed state model.

In section 2, we present our model of software development environments, discuss the individual components and their interrelationships, and illustrate various distinctions that we make with environments from the literature. In section 3, we classify SDEs into the four classes suggested by our metaphor, characterize these classes, present a basic model for each class, and categorize a wide variety of existing environments into the individual, family and city classes (we know of no examples of the state class). Finally, in section 4 we summarize the contributions of our model and classification scheme.

We confine our discussion in the sections below primarily to those environments concerned with the problems of implementing, testing, and maintaining software systems — that is, those environments that are concerned about the technical phases of the software development process. We believe that environments that concentrate on the full life-cycle and project management issues also could be described with this model and categorized according to our classification scheme presented in section 3.

2. A General Model of Software Development Environments (SDEs)

Our general model of software development environments consists of three interrelated components: policies, mechanisms and structures.

General SDE Model = ({ Policies }, { Mechanisms }, { Structures })

- Policies are the rules, guidelines and strategies imposed on the programmer by the environment;
- mechanisms are the visible and underlying tools and tool fragments;
- structures are the underlying objects and object aggregates on which mechanisms operate.

In general, these three components are strongly interrelated: choosing one component may have serious implications for the other two components and place severe limitations on them.

We discuss each of these components of the model, illustrate them with examples from the SDE literature, and discuss their interdependencies.

2.1 Policies

Policies are the requirements imposed on the user of the environment during the software development process. These rules and strategies are often hard-coded into the environment by the mechanisms and structures. For example, static linker/loaders generally require all externally referenced names to be defined in the set of object modules that are to be linked together. This requirement, together with the requirement that only linked/loaded objects may be executed, induces a policy of always compiling the modules before linking them. A different strategy is possible for execution preparation tools that provide dynamic linking and, hence, a different policy: for example, Multics' segmentation scheme [33] allows externally referenced names to be resolved at run-time. In most cases, the design of the tools and the supporting structures define or impose the policies.

But policies need not be hard-wired. A few architectures allow the explicit specification of policies. For example, Osterweil's process programming [34,49] provides the ability to program the desired policies with respect to the various mechanisms and structures available; Darwin's law-governed systems [30] consist of declaratively defined rules restricting the interactions of programmers and tools. The important distinction between hard-wired policies and process programs or rule systems is that the latter are architectures for building environments and provide a way of explicitly imposing policies on the developers independently of the mechanisms and structures.

Another distinction is between supporting and enforcing policies. If a policy is *supported*, then the mechanisms and structures provide a means of satisfying that policy. For example, suppose that top-down development is a supported policy. We would expect to find tools and structures that enable the developer to build the system in a top-down fashion; by implication, we would also expect to find tools and structures to build systems in other ways as well. If a policy is *enforced*, then not only is it supported, but it is not possible to do it any other way within the environment. We call this *direct enforcement* when the environment explicitly forces the developer to follow the policy. A slightly different kind of enforcement is that of *indirect enforcement*: policy decisions are made outside the environment either by management or by convention but once made they are supported but not enforced by the environment. For example, management decides that all systems are to be generated only from modules resident within the Source Code Control System (SCCS) [42]. The environment supports configuration management with SCCS; however, it is the management decision that forces the developers to control their modules within SCCS.

There is a further distinction to be made between those policies that apply to mechanisms and structures and those that apply to other policies. We refer to the second as *higher-order* policies. For example, 'all projects will be done in Ada' is a higher-order policy.

2.2 Mechanisms

Mechanisms are the languages, tools and tool fragments that operate on the structures provided by the environment and that implement, together with structures, the policies supported and enforced by the environment. Some of these mechanisms are visible to the developers; others may be hidden from the user and function as lower-level support mechanisms. For example, the UNIX™ System [25] tools for building systems are available to the user. However, in Smile [21] these tools are hidden beneath a facade that provides the developer with higher-level mechanisms that in turn invoke individual UNIX tools.

Policies are encoded in mechanisms in one of two ways: either *explicitly* by policy makers for a particular project, or *implicitly* by the toolsmiths in the tools that comprise the environment. In the first case, mechanisms such as shell scripts [19], Darwin's, CLF's [5] or Marvel's rules [20], or process programs enable the policy maker to define explicitly the policies to be supported by the system. Whether these can also be enforced depends on how well these mechanisms restrict the developer in what he or she uses in the environment. In the second case, the examples from the preceding section (illustrating hard-wired policies) exemplify implicit encoding. In most SDEs, policies are implicitly encoded in the mechanisms. There are good historical reasons for this situation: we must work out particular instances before we can generalize. Particular mechanisms and structures must first be built that implicitly encode policies in order to reach a sufficient understanding of the important issues. Once we have reached this level of maturity, we can then separate the specification of policies from mechanisms and structures.

2.3 Structures

Structures are those objects or object aggregates on which the mechanisms operate. In the simplest (and chronologically, earliest) incarnation, the basic structures — the objects with which we build systems — are files (as in UNIX, for example). The trend, however, is towards more complex and comprehensive objects as the basic structures. One reason for complex basic structures is found in integrated environments, particularly those centered around a syntax-directed editor [12, 50]. These SDEs share a complex internal representation such as an abstract syntax tree [9] or an IDL graph [26] to gain efficiency in each tool (because, for example, each tool does not need to reparse the textual form, but uses the intermediate, shared representation). The disadvantage of this approach is that it is difficult to integrate additional tools into the environment, particularly if the structure provided does not support well the mechanisms and their intended policies. Garlan's tool views [14] provide a partial* solution: a structure and a mechanism for generating the underlying common structure consistent with all the requirements of the different tools in the SDE.

Another reason for this trend is to maintain more information about software objects to support more comprehensive mechanisms and policies. For example, the use of project databases has been a topic of considerable interest in the recent past [1, 31]. The basic structure currently generating a large amount of interest is the *objectbase* [20, 49, 45] — it hoped that this approach will solve the deficiencies of files and databases.

These basic structures are the foundation for building more complex and more comprehensive higher-order structures. For example, Inscape [37, 38] maintains a complex semantic interconnection structure among the system objects to provide comprehensive semantic analysis and version control mechanisms and policies about semantic consistency, completeness and compatibility among the system objects. Smile's experimental database is a higher-order organization of basic structures that supports mechanisms and policies for managing changes to existing systems. The Project Master Data Base (PMDB) [36] provides an entity-relationship-attribute model [4] to represent, for example, problem reporting, evaluation and tracking processes. CMS's Modification Request Tracking System [43] builds a structure that is intertwined with SCCS's configuration management database (which in turn is built on top of the UNIX file system); it coordinates change requests with the actual changes in the system. Finally, Apollo's Domain Software Engineering Environment (DSEE) provides a comprehensive set of structures for coordinating the building and evolving of software systems; these structures support, for example, configuration control, planning and developer interactions.

---

* We say *partial* in the sense that Garlan's views do not help at all if the environment and its tools already exist independently of Garlan's mechanisms and new tools need to be added. It is a *full* solution in the sense that if one develops the entire environment with Garlan's views, then adding a new tool requires that one adds the view needed by that tool to the original set and generates the newly integrated structure.

In general, structures tend to impose limitations on the kinds of policies that can be supported and enforced by SDEs. Simple structures such as files provide a useful communication medium between tools but limit the kinds of policies that can be supported. The more complex structures required by integrated environments such as Gandalf [32] enable more sophisticated policies, but make it harder to integrate new mechanisms and policies into the environment. Higher-order structures such as Infuse's hierarchy of experimental databases [39] make it possible to enforce policies that govern the interactions of large groups of developers, but do not allow the policy maker the ability to define his or her own policies.

One fact should be clear: we have not yet reached a level of maturity in our SDEs with respect to structures. There is still a feeling of exploration about the kinds of structures that are needed. Indeed, there is the same feeling of exploration about the policies that can or should be supported by an SDE, particularly for those SDEs that are concerned with large-scale projects.

3. Four Classes of Models

We present a classification of SDEs from the viewpoint of scale: how the problems of size — primarily the numbers of developers, but by implication the size of the system as well — affect the requirements of an SDE that supports the development of those systems. Our classification is in terms of a sociological metaphor that is suggestive of the distinctions with respect to the problems of scale. Along what is a continuum of possible models, we distinguish the following four classes of SDE models:

Individual Family City State

There may be further distinctions to be made to the right of the family model; relatively little is known about the kinds of SDEs that support the city model and nothing is known about SDEs that support the state model. We concentrate our attention on those two classes — that is, the city and the state.

We present two orthogonal characterizations for each class. The first emphasizes what we consider to be the key aspect that distinguishes it in terms of scale from the others. These aspects are:

- *construction* for the individual class of models;
- *coordination* for the family class;
- *cooperation* for the city class; and

- *commonality* for the state class.

The second characterization emphasizes the relationships among the components. Historically,

- mechanisms dominate in the individual class;
- structures dominate in the family class;
- policies dominate in the city class; and
- higher-order policies dominate in the state class.

For each class of models we present a description of the class and support our characterizations with example SDEs. For convenience in the discussion below, we use the term *model* instead of *class of models*.

3.1 The Individual Model

The individual model of software development environments represents those environments that supply the minimum set of implementation tools needed to build software. These environments are often referred to as *programming environments*. The mechanisms provided are the tools of program construction: editors, compilers, linker/loaders and debuggers. These environments typically provide a single structure that is shared among mechanisms. For example, the structure may be simple, such as a file, or complex, such as a decorated tree. The policies are typically *laissez faire* about methodological issues and hard-wired for nano-management issues.

Individual Model =
(
{ *tool-induced policies*[*] } ,
{ *implementation tools* } ,
{ *single structure* }
)

These environments are dominated by issues of software *construction*. This orientation has led to an emphasis on the tools of construction — that is, the mechanisms — with policies and structures assuming secondary importance. The

---

[*] We use *italics* for general descriptions of the components and normal typeface for specific components.

policies are induced by the mechanisms — that is, hard-wired — while the structures are dictated by the requirement of making the tools work together.

We discuss four groups of environments that are instantiations of the individual model: toolkit environments, interpretive environments, language-oriented environments, and transformational environments. The toolkit environments are exemplified by UNIX; the interpretive environments by Interlisp [51]; language-oriented environments by the Cornell Synthesizer [50]; and the transformational environments by Refine [46].

The toolkit environments are, historically, the archetype of the individual model. The mechanisms communicate with each other by a simple structure, the file system. Policies take the form of conventions for organizing structures (as for example in UNIX, the bin, include, lib and src directories) and for ordering the sequence of development and construction (as exemplified by Make [13]). These policies are very weak and concerned with the minutiae of program construction. However, shell scripts provide the administrator with a convenient, but not very extensive, mechanism for integrating tools and providing support for policies beyond those encoded in the tools.

Toolkit Model =
(
{ *tools-induced policies, script-encoded policies, ...* } ,
{ editors, compilers, linker/loaders, debuggers, ... } ,
{ file system }
)

Interpretive environments are also an early incarnation of the individual model. They consist of an integrated set of tools that center around an interpreter for a single language such as Lisp or Smalltalk [15]. The language and the environment are not really separable: the language is the interface to the user and the interpreter the tool that the user interacts with. The structure shared by the various tools is an internal representation of the program, possibly with some accompanying information as exemplified by property lists. These environments are noted for their extreme flexibility and there are virtually no policies enforced (or, for that matter, supported). Thus, in contrast to the toolkit approach where the tools induce certain policies that force the programmer into certain modes of operation, programmers can essentially do as they please in the construction of their software.

Interpretive Model =
(
{ *virtually no restrictive policies* } ,
{ interpreter, *underlying support tools* } ,
{ intermediate representation }
)

Language-oriented environments are a blend of the toolkit and interpretive models. They provide program construction tools integrated by a complex structure — a decorated syntax tree. Whereas the tools in the toolkit environments are batch in nature and the tools in the interpretive are interactive, the tools in language-oriented environments are incremental in nature — that is, the language-oriented tools try to maintain consistency between the input and output at the grain of editing commands. A single policy permeates the tools in this model: early error detection and notification. These environments might be primarily hand-coded, as in Garden [40], or generated from a formal specification, such as by the Cornell Synthesizer Generator [41].

Language-Oriented Model =
(
{ error prevention, early error detection and notification, ... },
{ editor, compiler, debugger, ... } ,
{ decorated syntax tree }
)

Transformational environments typically support a wide-spectrum language (such as V [46]) that denotes a range of object and control structures from abstract to concrete. Programs are initially written in a abstract form and modified by a sequence of transformations into an efficient, concrete form. The mechanisms are the transformations themselves and the machinery for applying them. The structure is typically a cross between the intermediate representation of the interpretive model and the decorated syntax tree of the language-oriented model. As in the language-oriented environments, a single policy defines the style of the environment: the transformational approach to constructing programs (as, for example, in Ergo [444] and PDS [3]). Programmer apprentices, such as KBEmacs [53], are a variation of this policy in that the programmer can switch between the transformational approach and interpretive approach at any time.

Transformational Model =
(
{ transformational construction, ... },
{ interpreter, transformational engine, ... },
{ intermediate representation/decorated syntax tree, ... }
)

We have discussed four different groups of individual models and cited a few of the many environments that are examples of these different models. Most research environments and many commercial environments are instances of these individual models.

3.2 The Family Model

The family model of software development environments represents those environments that supply, in addition to the minimal set of program construction tools, facilities that support the interactions of a small group of programmers (under, say, 10). The analogy to the family for this model is that the members of the family work autonomously, for the most part, but trust the others to act in a reasonable way; there are only a few rules that need to be enforced to maintain smooth interactions among the members of the family. It is these rules, or policies, that distinguish the individual from the family model of environments: in the individual model, no rules are needed because there is no interaction; in the family model, some rules are needed to regulate certain critical interactions among the programmers.

Family Model =
(
{ ..., *coordination policies* } ,
{ ..., *coordination mechanisms* } ,
{ ..., *special-purpose databases* }
)

The characteristic that distinguishes the family model from the individual model is that of *enforced coordination*. The environment provides a means of orchestrating the interactions of the developers, with the goal that information and effort is neither lost nor duplicated as a result of the simultaneous activities of the programmers. The structures of the individual model do not provide the necessary (but weak form of) concurrency control. Because the individual model's structures are not rich enough to coordinate simultaneous activities, more complex structures are required. It is these structures that dominate the design of the environment, wherein the individual model the mechanisms dominated; the mechanisms and policies in the family model are adapted to the structures.

We discuss four groups* of environments that are instantiations of the family model: extended toolkit environments, integrated environments, distributed environments, and project management environments. The extended toolkit environments are exemplified by UNIX together with either SCCS or RCS [52]; the integrated environments by Smile; the distributed environments by Cedar [48]; and the project management environments by CMS.

The extended toolkit model directly extends the individual toolkit model by adding a version control structure and configuration control mechanisms (see, for example, UNIX PWB [8]). Programmer coordination is supported with these structures and mechanisms; enforced coordination is supplied by a management decision to generate systems only from, for example, SCCS or RCS databases. Thus, this kind of family environment provides individual programmers a great deal of freedom with coordination supported only at points of deposit into the version control database. The basic mechanisms for program construction from the individual toolkit model are retained. However, these tools must be adapted to the family model structure as, for example, Make must be modified to work with RCS or SCCS. Alternatively, the tools may be constructed in conjunction with a database — e.g., the Ada program support environments (APSEs) [2].

```
Extended Toolkit Model =
        (
          { ..., support version/configuration control } ,
          { ..., version/configuration management } ,
          { ..., compressed versions, version trees }
        )
```

The integrated model extends by analogy the individual language-oriented model, where the consistency policy permeates the tools. Here consistency is maintained among the component modules in addition to within a module. As in the individual model, the mechanisms determine consistency incrementally, although the grain size ranges from the syntax tree nodes of the Gandalf Prototype (GP) [16] to procedures in

---

* These groupings are not necessarily mutually exclusive. In particular, either distributed or project management aspects can be mixed and matched with either extended toolkit or integrated environments.

Smile, to entire modules in Toolpack [35] and R^n [6]. This model's structure is typically a special-purpose database, although in CLF it is generated from a specification. The structures vary in their support from simple backup versions to both parallel and sequential versions [17, 22].

Integrated Model =
(
{ ..., enforced version control, enforced consistency } ,
{ ..., version description languages, consistency checking tools } ,
{ ..., special-purpose database }
)

The distributed model expands the integrated model across a number of machines connected by a local area network. Additional structures are required to support reliability and high availability as machines and network links fail. For example, Mercury [23] is a multiple-user, language-oriented environment that depends on a special distributed algorithm that simulates a small shared memory to guarantee consistency among module interfaces; DSEE's database [27], on the other hand, is a simple extension Apollo's network file system.

Distributed Model =
(
{ ... } ,
{ ..., *network mechanisms* } ,
{ ..., distributed *objects* }
)

The project management model is orthogonal to the progression from the extended toolkit model to the distributed model. These environments provide additional support for coordinating changes by assigning tasks to individual programmers. In DSEE, structures and mechanisms are provided for assigning and completing tasks that may be composed of subtasks and activities [28]. CMS adds a modification request (MR) tracking system on top of SCCS in which individual programmers are assigned particular change requests and the changes are associated with particular sets of SCCS versions.

Project Management Model =
(
{ ..., support activity coordination } ,
{ ..., *activity coordination mechanisms* } .
{ ..., *activity coordination structures* }
)

The family model represents the current state of the art in software development environments. In general, it is an individual model extended with mechanisms and structures to provide a small degree of enforced coordination among the programmers. The policies are generally *laissez faire* with respect to most activities; enforcement of coordination is generally centered around version control and configuration management. The most elaborate instance of the family model with respect to mechanisms is DSEE; the most elaborate with respect to structures is CLF.

3.3 The City Model

As the size of a project grows to, say, more than 20 people, the interactions among these people increase both in number and in complexity. Although families allow a great degree of freedom, much larger populations, such as cities, require many more rules and regulations with their attendant restrictions on individual freedom. The freedom appropriate in small groups produces anarchy when allowed to the same degree in larger groups. It is precisely this problem of scale and the complexity of interactions that leads us to introduce the city model.

City Model =
(
{ ..., *cooperation policies* } ,
{ ..., *cooperation mechanisms* },
{ ..., *structures for cooperation* }
)

The notion of enforced coordination of the family model is insufficient when applied to the scale represented by the city model. Consider the following analogy. On a farm, very few rules are needed to govern the use of the farm vehicles while within the confines of the farm. A minimal set of rules govern who uses which vehicles and how they are to be used — basically, how the farm workers coordinate with each other on use of the vehicles. Further, these rules can be determined in *real time* — that is, they can be adjusted as various needs arise or change. However, that set of rules and mode of rule determination is inadequate to govern the interactions of cars and trucks in an average city: chaos would result without a more complex set of rules and mechanisms that enforce the cooperation of the people and vehicles; the alteration of rules, of necessity, has serious consequences because they affect a much larger population (consider the problem when Europe changed from driving on the left to driving on the right side of the road). Thus, *enforced cooperation* is the primary characteristic of the city model.

It is our contention that the family model is currently being used where we need a city model, and that the family model is not appropriate for the task. Because the family model does not support or enforce an appropriate set of policies to handle the problems incurred by an increase in scale, we generally have a set of methodologies and management techniques that attempt to stave off the anarchy that can easily occur. These methodologies and management techniques work with varying degrees of success, depending on how well they enforce the necessary cooperation among developers.

Little work has been done on environments that implement a city model — that is, that enforce cooperation among developers. We discuss two such environments: Infuse [39] and ISTAR [10]. Infuse focuses on the technical management of the change process in large systems whereas ISTAR focuses on project management issues. In both cases, the concern for policies of enforced cooperation dominate the design and implementation: in Infuse, the policy of enforced cooperation while making a concerted set of changes by many programmers has led to the exploration of various structures and mechanisms; in ISTAR, the contractual model and the policies embodied in that model dominate the search for project management structures and mechanisms.

The primary concern of Infuse* is the technical management of evolution in large systems — that is, what kinds of policies, mechanisms and structures are needed to automate support for making changes in large systems by large numbers of programmers. Infuse generalizes Smile's experimental databases into a hierarchy of experimental databases, which serves as the encompassing structure for enforcing Infuse's policies about programmer interaction. These policies enforce cooperation

---

* Infuse originated as, and still is, the change management component of the Inscape Environment (which explores the use of formal interface specifications and of a semantic interconnection model in the construction and evolution of software systems). However, the management issues of how to support a large number of developers are sufficiently orthogonal to the semantic concerns of Inscape to be applicable in a much wider context (for example, to environments and tools supporting a syntactic interconnection model) and to be treated independently.

among programmers in several ways [24].

- Infuse automatically partitions the set of modules involved in a concerted set of changes into a hierarchy of experimental databases on the basis of the strength of their interconnectivity (this measure is used as an approximation to the oracle that tells which modules will be affected by which changes). This partitioning forms the basis for enforcing cooperation: each experimental database proscribes the limits of interaction (however, see the discussion of workspaces below).

- At the leaves of the hierarchy are singleton experimental databases where the actual changes take place. When the changes to a module are self-consistent it may be deposited into its parent database. At each non-leaf database, the effects of changes are determined with respect to the components in that partition, that is, analysis determines the *local consistency* of the modules within the database. Only when the modules within a partition are locally consistent may the database be deposited into its parent. This iterative process continues until the entire system is consistent.

- When changes conflict, the experimental database provides the forum for negotiating and resolving those conflicts. Currently, there are no formal facilities for this negotiation, but only the framework for it. Once the conflicts have been resolved, the database is repartitioned and the change process repeats for that (sub-) partitioning.

- Because the partitioning algorithm is only an approximation of the optimal oracle, Infuse provides an escape mechanism, the *workspace*, in which programmers may voluntarily cooperate to forestall expensive inconsistencies at the top of the hierarchy.

Thus the rules for interaction are encoded in the mechanisms, with the hierarchy providing the supporting structure[*].

---

[*] We are also investigating the utility of this structure for cooperation in integration testing. With a notion of local integration testing analogous to our notion of local consistency checking, we expect to be able to assist the integration of changes further by providing facilities for test harness construction and integration and regression testing within this framework.

```
Infuse Model =
        (
          { ..., enforced and voluntary cooperation } ,
          { ...,  automatic partitioning,
                  local consistency analysis,
                  database deposit,
                  local integration testing,
                  ... } ,
          { ..., hierarchy of experimental databases }
        )
```

Whereas Infuse is concerned with the technical problems of managing system evolution, ISTAR is concerned with the managerial problems of managing system evolution. ISTAR is an integrated project support environment (IPSE) [29] and seeks to provide an environment for managing the cooperation of large groups of people producing a large system. To this end, it embodies and implements a *contract model* of system development. ISTAR does not directly provide tools for system construction but instead supports "plugging in" various kinds of workbenches. The contract model dictates the allowable interactions among component developers [47].

- The client specifies the required deliverables — that is, the products to be produced by the contractor. Further, the client specifies what the terms of satisfaction are for the deliverables — that is, the specific validation tests for the products.
- The contractor provides periodic reporting about the status of the project and the state of the product being developed. Clients are thus able to monitor the progress of their contracts.
- ISTAR provides support for amending the contracts as the project develops. Thus, the contract structure can change in the same ways that the products themselves can change.
- A contract database provides the underlying structure for this environment.

Thus the interactions between the clients and the contractors are proscribed by the underlying model and the mechanisms in the environment enforce those rules of interaction. The exact interaction of tools in the construction of the components of the system is left unspecified, but the means of contracting for components of a system are enforced by the environment.

ISTAR Model =
(
{ ..., contract model } ,
{ ..., *contract support tools* } ,
{ ..., contract data base }
)

3.4 The State Model

Pursuing our metaphor leads to the consideration of a state model. Certainly the notion of a state as a collection of cities is suggestive of a company with a collection of projects. There are, we think, intimations of this model in the following: the Department of Defense standardizing on one particular language, Ada, for all its projects; a company trying to establish a uniform development environment such as UNIX for all its projects; a company establishing a common methodology and set of standards to be used on all its projects. It is easy to understand the rationale behind these decisions: reduction in cost and improvement in productivity. If there is a uniform environment used by several projects, developers may move freely between projects without incurring the cost of learning a new environment. Further, reuse of various kinds is possible: tools may be distributed with little difficulty; code may be reused; design and requirements may be reused; etc.

State Model =
(
{ *commonality policies* } ,
{ *supporting mechanisms* } ,
{ *supporting structures* }
)

In this model, the concern for commonality, for standards, is dominant. This policy of commonality tends to induce policies in the specific projects (or, in their city model environments). Thus, the policies of the state model are higher-order policies because they have this quality of inducing policies, rather than particular structures and mechanisms.

While one can imagine the existence of instances of this model (and there are certainly many cases where it is needed), we do not know of one. Our intuition* suggests the following general description.

- Provide a generic model with its attendant tools and supporting structures for software development to be used throughout a particular company.
- Instantiate the model for each project, tailoring each instance dynamically to the particular needs of the individual project.
- Manage the differences between the various instances to support movement between projects.

Thus, while little is known about the state model, it appears to be a useful and fruitful area for investigation.

3.5 Scaling Up from One Class to the Next

Ideally, scaling up from one class to the next would be a matter of adding structures and mechanisms on top of an existing environment. In at least one case this has been done without too much difficulty: scaling up from the individual toolkit model to the family extended toolkit model. This example involves only a small increment in policy.

It is extremely attractive to think of the higher-level models as using the lower-level models as components upon which to establish new policies, mechanisms and structures. Unfortunately, there are several difficulties. First, there is the problem of the tightness of coupling between structures and mechanisms. Even in scaling up from the toolkit to the extended toolkit environments, retrofitting of old tools to new structures is necessary. This raises the fundamental question of whether it is more profitable to retrofit changes into the system or to reconstruct the entire environment from scratch. For example, even though Infuse is a direct generalization of Smile, Infuse's implementation is a reconstruction reusing some code from Smile rather than an extension on top of Smile. Because environment generators assume a common kernel that is optimized for a specific model, and often a particular group within the model, they are difficult to scale up. Similar to Infuse/Smile, Mercury scales up the Cornell Synthesizer Generator by extensive modifications to its common kernel rather than by adding something to coordinate generated editors.

• See various position papers and discussions in the 3rd International Software Process Workshop [11].

Second, problems arise from the lack of structures and mechanisms in the base-level environment suitable for the next level. For example, multiple-user interpretive environments are extremely rare. Further, this lack of suitable structures and mechanisms is particularly important in moving from the family model to the city model where enforcement is a much more serious issue. Building security measures on top of a permissive environment (such as UNIX) is particularly difficult; it is too easy to subvert the enforcement mechanisms.

Last, there is the problem of how well the granularity of the structures and the mechanisms of one level lend themselves to supporting the next level. For example, the file system in the toolkit approach is easily adapted to the extended toolkit. However, some of the higher-level structure of the extensions is embedded, by convention, within the lower-level structure, as in SCCS where version information is embedded by an SCCS directive within the source files.

Note that most of our examples illustrating scaling difficulties are from the individual to the family model. Since this increment is much smaller than from family to city, we can expect greater obstructions in scaling from the family to the city model.

4. Contributions

We summarize the contributions of this paper as follows:

- Our general model distinguishes precisely those aspects of an environment that are useful in evaluating software development environments: policies, mechanisms and structures.
- Our taxonomy delineates four important classes of interaction among software developers with respect to the problems of scale.
- The individual and family models represent the current state of the art in software development environments. We explain why these two models are ill-suited for the development of large systems.
- We show that the city model introduces the qualitative differences in the policies, structures and mechanisms required for very large software development projects.
- We propose a state model, which is in need of further clarification, understanding and implementation.

We conclude that there is a pressing need for research in both the technical and managerial aspects of city model environments.

Professor Kaiser is supported in part by grants from AT&T Foundation, Siemens Research and Technology Laboratories, and New York State Center of Advanced Technology - Computer and Information Systems, and in part by a Digital Equipment Corporation Faculty Award.

References

[1] Philip A. Bernstein. "Database System Support for Software Engineering", *Proceedings of the 9th International Conference on Software Engineering*, Monterey, CA, March 1987. pp 166-178.

[2] John N. Buxton and Larry E. Druffel. "Rationale for Stoneman", *Fourth International Computer Software and Applications Conference*, Chicago, IL, October 1980. pp 66-72.

[3] Thomas E. Cheatham, Jr., Glenn H. Holloway and Judy A. Townley. "Program Refinement by Transformation", *Proceedings of 5th International Conference on Software Engineering*, San Diego, CA, March 1981. pp 430-437.

[4] P. Chen. "The Entity-Relationship Model — Toward a Unified View of Data", *ACM Transactions on Database Systems*, 1:1 (March 1976). pp 9-36.

[5] Donald Cohen. "Automatic Compilation of Logical Specifications into Efficient Programs", *5th National Conference on Artificial Intelligence*, August 1986, Philadelphia, PA, Science Volume. pp 20-25.

[6] Keith D. Cooper, Ken Kennedy and Linda Torczon. "The Impact of Interprocedure Analysis and Optimization in the R^n Programming Environment", *ACM Transactions on Programming Languages and Systems*, 8:4 (October 1986). pp 491-523.

[7] Frank DeRemer and Hans H. Kron. "Programming-in-the-Large Versus Programming-in-the-Small", *IEEE Transactions on Software Engineering*, SE-2:2 (June 1976). pp 80-86.

[8] T.A. Dolotta, R.C. Haight and J.R. Mashey. "The Programmer's Workbench", *The Bell System Technical Journal*, 57:6-2 (July-August 1978). pp 2177-2200.

[9] Veronique Donzeau-Gouge, Gerard Huet, Gilles Kahn, and Bernard Lang. "Programming Environments Based on Structured Editors: The Mentor Experience", *Interactive Programming Environments*, David R. Barstow, Howard E. Shrobe and Erik Sandewall, editors. New York: McGraw-Hill Book Co., 1984. pp 128-140.

[10] Mark Dowson. "ISTAR — An Integrated Project Support Environment", *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Palo Alto, CA, December 1986. pp 27-33. Proceedings published as *SIGPLAN Notices*, 22:1 (January 1987).

[11] Mark Dowson, editor. *Proceedings of the 3rd International Software Process Workshop: Iteration in the Software Process*, Breckenridge CO, November 1986. IEEE Computer Society, 1987.

[12] Peter H. Feiler and Raul Medina-Mora. ''An Incremental Programming Environment'', *IEEE Transactions on Software Engineering*, SE-7:5 (September 1981). pp 472-482.

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

[14] David Garlan. ''Views for Tools in Integrated Environments'' , *Advanced Programming Environments*, Reidar Conradi, Tor M. Didriksen and Dag H. Wanvik, editors. Lecture Notes in Computer Science, 244. Berlin: Springer-Verlag, 1986. pp 314-343.

[15] Adele Goldberg and David Robson. *Smalltalk-80 The Language and its Implementation*, Reading, MA: Addison-Wesley Pub. Co., 1983.

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

[17] A. Nico Habermann and Dewayne E. Perry. ''System Composition and Version Control for Ada'', *Software Engineering Environments*, H. Huenke, editor. New York: North-Holland Pub. Co., 1981. pp 331-343.

[18] William E. Howden. ''Contemporary Software Devlopment Environments'', *Communications of the ACM*, 25:5 (May 1982). pp 318-329.

[19] William Joy. ''An Introduction to the C shell'', *UNIX User's Manual Supplementary Documents*, 1986. pp USD:4.

[20] Gail E. Kaiser and Peter H. Feiler. ''An Architecture for Intelligent Assistance in Software Development'', *Proceedings of the 9th International Conference on Software Engineering*, Monterey, CA, March 1987. pp 80-88.

[21] Gail E. Kaiser and Peter H. Feiler. ''Intelligent Assistance without Artificial Intelligence'', *Thirty-Second IEEE Computer Society International Conference*, February 1987, San Francisco, CA. pp 236-241.

[22] Gail E. Kaiser and A. Nico Habermann. ''An Environment for System Version Control'', *Twenty-Sixth IEEE Computer Society International Conference*, San Francisco, CA, February 1983. pp 415-420.

[23] Gail E. Kaiser, Simon M. Kaplan and Josephine Micallef. "Multiple-User Distributed Language-Based Environments", *IEEE Software*, (November 1987).

[24] Gail E. Kaiser and Dewayne E. Perry. "Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution", *Conference on Software Maintenance-1987*, Austin, TX, September 1987.

[25] Brian W. Kernighan and John R. Mashey. "The UNIX Programming Environment", *IEEE Computer*, 12:4 (April 1981). pp 25-34.

[26] David Alex Lamb. "IDL: Sharing Intermediate Representations", *ACM Transactions on Programming Languages and Systems*, 9:3 (July 1987). pp 297-318.

[27] David B. Leblang and Gordon D. McLean, Jr.. "Configuration Management for Large-Scale Software Development Efforts", *GTE Workshop on Software Engineering Environments for Programming in the Large*, Harwichport, MA, June 1985. pp 122-127.

[28] David B. Leblang and Robert P. Chase, Jr.. "Computer-Aided Software Engineering in a Distributed Workstation Environment", *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984. pp 104-112. Proceedings published as *SIGPLAN Notices*, 19:5 (May 1984).

[29] M. M. Lehman and W. M. Turski. "Essential Properties of IPSEs", *Software Engineering Notes* 12:1 (January 1987). pp 52-55.

[30] Naftaly H. Minsky. "Controlling the Evolution of Large Scale Software Systems", *Conference on Software Maintenance-1985*, November 1985. pp 50-58.

[31] John R. Nestor. "Toward a Persistent Object Base", *Advanced Programming Environments*, Reidar Conradi, Tor M. Didriksen and Dag H. Wanvik, editors. Lecture Notes in Computer Science, 244. Berlin: Springer-Verlag, 1986. pp 372-394.

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

[33] Elliott I. Organick. *The Multics System: An Examination of Its Structure*. Cambridge, MA: The MIT Press, 1972.

[34] Leon Osterweil. "Software Processes Are Software Too", *Proceedings of the 9th International Conference on Software Engineering*, Monterey, CA, March 1987. pp 2-13.

[35] L.J. Osterweil. "Toolpack — An Experimental Software Development Environment Research Project", *IEEE Transactions on Software Engineering*, SE-9:6 (November 1983). pp 673-685.

[36] Maria H. Penedo. "Prototyping a Project Master Database for Software Engineering Environments", *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Palo Alto, CA, December 1986. pp 1-11. Proceedings published as *SIGPLAN Notices*, 22:1 (January 1987).

[37] Dewayne E. Perry. "Software Interconnection Models", *Proceedings of the 9th International Conference on Software Engineering*, Monterey, CA, March 1987. pp 61-69.

[38] Dewayne E. Perry. "Version Control in the Inscape Environment", *Proceedings of the 9th International Conference on Software Engineering*, Monterey, CA, March 1987. pp 142-149.

[39] Dewayne E. Perry and Gail E. Kaiser. "Infuse: A Tool for Automatically Managing and Coordinating Source Changes in Large Systems", *ACM Fifteenth Annual Computer Science Conference*, St. Louis, MO, February 1987. pp 292-299.

[40] Steven P. Reiss. "A Conceptual Programming Environment", *Proceedings of the 9th International Conference on Software Engineering*, Monterey, CA, March 1987. pp 225-235.

[41] Thomas Reps and Tim Teitelbaum. "The Synthesizer Generator", *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984. pp 41-48. Proceedings published as *SIGPLAN Notices*, 19:5 (May, 1984).

[42] M. J. Rochkind. "The Source Code Control System", *IEEE Transactions on Software Engineering*, SE-1:4, (December 1975). pp 364-370.

[43] B. R. Rowland, R. E. Anderson, and P. S. McCabe. "The 3B20D Processor & DMERT Operating System: Software Devlopment System", *The Bell System Technical Journal*, 62:1 part 2 (January 1983). pp 275-290.

[44] W. L. Scherlis. "Software Development and Inferential Programming", in *Program Transformation and Programming Environments*, P. Pepper, editor. Berlin: Springer-Verlag, 1983. pp 341-346.

[45] Andrea Skarra and Stanley Zdonik. "The Management of Changing Types in an Object-Oriented Database", *ACM 1986 OOPSLA Conference*, Portland, OR, September 1986. pp 483-495.

[46] Douglas R. Smith, Gordon B. Kotik and Stephen J. Westfold. "Research on Knowledge-Based Software Environments at Kestrel Institute", *IEEE Transactions on Software Engineering*, SE-11:11 (November 1985). pp 1278-1295.

[47] Vic Stenning. "An Introduction to ISTAR", Chapter 1 in *Software Engineering Environments*, Ian Sommerville, editor. IEE Computing Series 7. London: Peter Peregrinus Ltd., 1986. p 1-22.

[48] Daniel Swinehart, Polle Zellweger, Richard Beach and Robert Hagmann. "A Structural View of the Cedar Programming Environment", *ACM Transactions on Programming Languages and Systems*, 8:4 (October 1986) pp 419-490.

[49] Richard N. Taylor, Lori Clarke, Leon J. Osterweil, Jack C. Wiledon and Michal Young. "Arcadia: A Software Development Environment Research Project", *2nd International Conference on Ada Applications and Environments*, IEEE Computer Society, Miami Beach, FL, April 1986.

[50] Tim Teitelbaum and Thomas Reps "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", *Communications of the ACM*, 24:9 (September 1981). pp 563-573.

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

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

[53] Richard C. Waters. "KBEmacs: Where's the AI?", *The AI Magazine*, VII:1 (Spring 1986). pp 47-56.