Maintaining Configurations of Evolving Software Systems

K. NARAYANASWAMY AND WALT SCACCHI, MEMBER, IEEE

Abstract—**Software configuration management (SCM) is an emerging discipline. An important aspect of realizing SCM is the task of maintaining the configurations of evolving software systems. In this paper, we provide an approach to resolving some of the conceptual and technical problems in maintaining configurations of evolving software systems. The approach provides a formal basis for existing notions of system architecture. The formal properties of this view of configurations provide the underpinnings for a rigorous notion of system integrity, and mechanisms to control the evolution of configurations. This approach is embodied in a language, NuMIL, to describe software system configurations, and a prototype environment to maintain software system configurations. We believe that the approach and the prototype environment offer a firm base to maintain software system configurations and, therefore, to implement SCM.**

Index Terms—**Configuration, module and subsystem families, module and subsystem interfaces, software configuration maintenance system, upward compatibility.**

I. Introduction

A COMPLEX product such as a large-scale software system is built through the interactions of numerous components. An arrangement of such components which constitutes the product is called a *configuration*. Configuration management (CM) is the discipline of developing uniform descriptions of a complex product at discrete points in its life cycle with a view to systematically controlling the manner in which the product evolves. For physical systems it is a well-understood discipline [7]. Software Configuration Management (SCM), the application of the tenets of configuration management to software systems, is an emerging discipline.

Software configuration management has four different elements [1], [5]:

1) Software Configuration Identification: The definitions of the different components, their *baselines* or milestones, and the changes to the components.

2) Software Configuration Control: Controlling the way components or configurations are altered with the necessary technical and administrative support.

Manuscript received September 2, 1985; revised June 13, 1986. This work was supported by AT&T Information Systems, TRW Defense Systems Group, IBM through Project Socrates at USC, and the Air Force Office of Scientific Research under Grant 810199.

K. Narayanaswamy was with the Department of Computer Science, University of Southern California, Los Angeles, CA 90089. He is now with the USC/Information Sciences Institute, Marina del Rey, CA 90292.

W. Scacchi is with the Department of Computer Science, University of Southern California, Los Angeles, CA 90089.

IEEE Log Number 8612410.

3) Software Configuration Auditing: Making the current status of the software system in its life-cycle visible to management, to determine whether the baselines meet their requirements.

4) Software Configuration Status Accounting: Providing an administrative history of how the software system has been altered by recording the activities necessitated by the other three SCM functions.

While the above definition accurately characterizes the goals of software configuration management, several technical problems remain with actually achieving those ends. Specifically, the phenomenon of *continuing change* in software systems [3] causes some conceptual and technical difficulties in the following areas:

- How should the components of a software system and its different configurations be specified so that one can maintain software configurations?
- What precisely is the *state* of an evolving software system configuration? What are the necessary conceptual mechanisms to understand how the state of a system may be altered?
- Typically, a system evolves not monolithically, but as a family of closely related entities [6], [27], [36]. How can the configurations of such software system families be represented?
- What mechanisms are needed to estimate the impact of contemplated or partially incorporated alterations on a software system configuration in order to evaluate their impact on the system?
- How can the answers to the above questions be used in providing a well-integrated and automated environment for the development and maintenance of software system configurations?

In this paper, we provide some answers to the above questions, and present an approach to maintaining configurations of large software systems. It is our belief that our approach and our prototype Software Configuration Maintenance System (SCMS) provides a firm conceptual and technical base for the maintenance of evolving system configurations and, therefore, for software configuration management.

In Section II, we present the background for our work, along with our particular view of what an evolving system configuration represents. Section III contains the details of a language, NuMIL, to describe evolving software components and their configurations, viewed as we advocate in Section II. The important formal properties of

our view of system configurations are examined in Section IV. Section V outlines the prototype SCMS we have implemented. Section VI summarizes our conclusions and outlines areas for further work.

II. Towards a Model of System Configurations

Until recently, software configuration maintenance has not been approached in a principled manner. Several of the issues which we raised in Section I were handled by the use of manual procedures and protocols. The task of assuring continued system integrity as the system evolved was accomplished by supervisory oversight. With the advent of more amicable software development environments such as UNIX™, which enabled the building of specialized computer tools, it became possible to automate some aspects of software configuration maintenance. Several currently existing UNIX tools provide some configuration maintenance facilities. For example, MAKE [9] accepts descriptions of system configurations, and can automatically construct the system from its descriptions. Tools such as the Source Code Control System (SCCS) [31] or the Revision Control System (RCS) [37] permit one to keep track of all the *textual* alterations made to a file.

The key problem, not addressed by any of the mentioned tools, is that of maintaining the *integrity* of the system configuration. Indeed, what precisely is "system integrity," especially when the system is constantly changing? What properties of a system configuration should be preserved and what may be altered? Which alterations are easy to carry out—which are not? These questions must be addressed in a fairly formal manner if we are to go beyond the current level of support for software configuration maintenance.

A. Module Interconnection Languages

Recent work in Module Interconnection Languages (MIL's) has been directed toward the *conceptual*, rather than the management aspect of maintaining software configurations. The concept of a MIL was introduced by deRemer and Kron [8] to permit the articulation of how system configurations could be constructed from their constituents (modules). The primary new idea of a MIL was to formally describe the *interdependencies* between the components of a system. This description was then to be used to control how the system evolved.[1] A MIL works at the level of "programming-in-the-large," i.e., at the level of interaction between modules, rather than "programming-in-the-small," which involves other programming activity like algorithm or data structure design, coding, etc. MIL's have also been incorporated into different system design methodologies as a basis for developing (or generating) the architectural structure of emerging software systems [28], [22].

The first such language, MIL-75 [8] recorded inter-module dependencies in a purely structural sense by listing the resources *provided* by each module, and a list of resources *required* by each module. A resource was regarded as something that had a representation in the implementation language. The primary focus of this work was to ensure that modules had access to the resources which they required (i.e., resources provided by other modules).

Tichy [36] and Cooprider [6] observed that if MIL's are to be used to record the *evolving* structure of a software system, one should recognize that most software systems exist as *families of related systems*. For instance, a Pascal compiler may evolve into a family of systems, each intended for a particular kind of target machine. The schemes proposed by these works allow for *module families*, i.e., closely related source files, and *subsystem families* which result from the configurations involving the different module families. Each member of a module (subsystem) family is called an *implementation* or a *version* of the module (subsystem) family. The problem of supporting evolution is then seen as controlling the interaction between module interfaces (*provide* and *require* resources, as in the conventional MIL's) and controlling the proliferation of versions resulting from evolution.

B. Our View of System Configurations

Our work is best understood against the background of the work discussed in the last section. We feel that the most effective way of maintaining software configurations is to focus on programming-in-the-large, describing the individual component interfaces (*provide* and *require* resources), describing how the components depend on each other, and, finally, preserving those dependencies between the modules of a system as the system evolves. We find it useful, as in [36], to view an evolving system as a family of related systems. However, beyond the similarities above, our framework differs in very substantial ways from the work on MIL's.

C. Vocabulary and Definitions

In the traditional manner, we view a system as essentially a directed acyclic graph where the leaf nodes are *modules*, and the internal nodes are *subsystems*. Each subsystem is realized by a *configuration* of its successor nodes which may, in turn, be other subsystems or modules. A module is directly realized as a source file which implements it. A subsystem is realized by putting together its successors using some unspecified construction rules (e.g., compile, link, and load). The successors of the subsystem are said to participate in a *configuration*.[2]

The currency of exchange between modules is called a *resource*. A resource is an entity that can be given some representation in the implementation language; for example, functions, procedures, type definitions, variables,

™UNIX is a trademark of AT&T Bell Laboratories.

[1] A comprehensive survey of existing MIL's appears in [29].

[2] The distinction between subsystems and modules is not important to our overall scheme, and is made for purely historical reasons. In object-oriented implementation languages like SMALLTALK, for instance, the distinction does not arise.

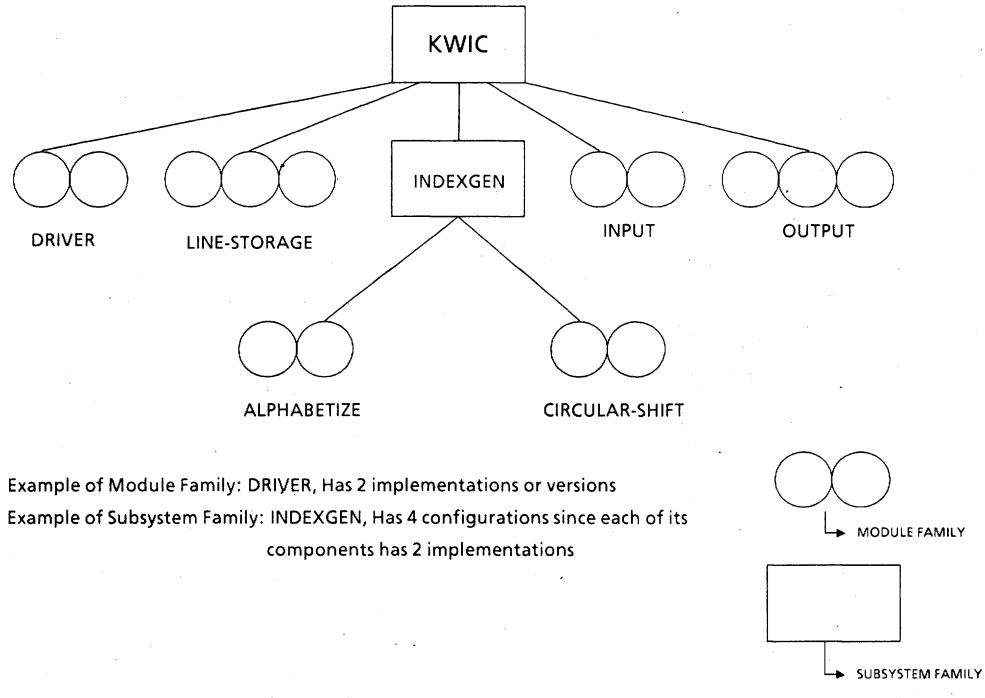


Fig. 1. Hierarchical structure of KWIC index system.

etc., can be resources [36]. Every module *provides* some resources to the other modules in the system, and in turn may *require* some resources from the other modules in the system in order to implement the resources it is supposed to provide. It is useful to think of the modules and subsystems as having *interfaces* through which they interact with other modules or subsystems. Thus, the interface of a module or subsystem is characterized by 1) the set of resources it provides, and 2) the set of resources it requires.

Because of the existence of module and system families, each node in the graph denotes not a single object, but a set of objects which actually realize the module's family. For example, each leaf denotes a module family, and each internal node a subsystem family. Each member of a module (subsystem) family is called either an *implementation* or *version* of that module (subsystem) family.[3] An example of a configuration in this style is given in Fig. 1, which shows the hierarchical structure corresponding to Parnas' KWIC-index system [26]. Note how the explicit existence of multiple versions is depicted.

The introduction of multiple versions requires that we be precise about the relationship between the versions and the module or subsystem of which they are members. Clearly, the criterion for family membership should be shared interface characteristics. What should these characteristics be?

D. Relationship Between Families and Their Members

With the conventional MIL's, a module interface is determined precisely by the *syntax* of the resources it provides and the *syntax* of the resources it requires [36]. All the implementations share exactly this interface. They cannot, for instance, require different resources, or have some resources with different syntax in different implementations. This is a serious inadequacy. For instance, the different implementations could embody completely different methods of doing the same thing. Therefore, it is conceivable that different implementations require different resources. Similarly, if a module *M* provided a resource "foo" as an integer, and module *N* provided "foo" as a real number, the two would be considered as belonging to *different families* regardless of whether the difference in the type of "foo" in the two cases is significant from the point of view of what *M* or *N* actually do.

More importantly, syntactic assurances about module interfaces are inadequate in addressing the more profound concern about the functional *properties* of the resources which are being used across module boundaries. For example, the "Matrix_Invert" and "Matrix_Transpose" operations in a particular library of mathematical routines have identical syntactic attributes (i.e., parameters and their types), and would therefore look identical in a conventional MIL, but clearly compute different functions. That two interfaces contain the same resources with the identical syntactic attributes does not really tell us much.

Rather than use the syntax of the resources in the interface as the unifying facet of all the members of a module family, we use the more powerful notion that each member of a module family satisfies the same *abstract interface specification*. We call this abstract interface specification the *module family template* for a family. This abstract interface specification is intended to be similar to the specification of a typical abstract data type [13]. In particular, we use the abstract model style of specification

[3]For conciseness, we will use module to mean a module family and a subsystem to mean a subsystem family in the rest of this narrative.

[16], [19], [25] to specify the module family template. Examples follow in the next section.

No two members of the family need to share the same *syntax* for the resources they provide and require, since this is not the chief criterion for family membership. In fact, the module family template is written in a manner independent of implementation language. Using an abstract module family template is appealing because, during evolution, we can permit an individual module to evolve in several concrete dimensions, so long as it still embodies the abstract interface properties specified in the module family template.

An additional important point is that there is no need to have required resources in the family template because *required* resources are those needed to *implement* the module or subsystem, and are thus a characteristic of each implementation rather than the family as a whole. Thus, each implementation may be free to require any resources in order to provide (at least) the resources in the family template.

In turn, each implementation is specified in terms of the implementation language as in the conventional MIL's, i.e., in concrete terms. Each implementation *provides* some resources, typically at least the resources in the family template, and *requires* some resources to implement the provided resources. Each resource is given syntactic attributes in terms of the implementation language. This description is called the *concrete* interface since it is specified using the syntax and semantics of the implementation language. The functional properties of the resources in the concrete interface are specified using pre- and postcondition assertations in a Hoare-like style. Examples follow in the next section.

III. A Language to Describe Software System Configurations

In this section, we present the features of a system configuration description language called *NuMIL* which embodies our view of large systems as an evolving ensemble of software (sub)system configurations. We use NuMIL to describe and identify software configurations in the manner suggested in Section I. Hence, the structure of the language closely resembles the concepts set forth in the previous section.

A. Module Family Specifications in NuMIL

Module families are specified first by a description of the family template which unites all the different versions or members of the family, and second by the information particular to each member (implementation) of the family. The NuMIL description of a module family separates the family template from the specification of the implementations. The template characterizes the properties of the resources provided by the module family in an implementation-independent manner, whereas inspection of the implementation specifications should reveal the structural and functional details peculiar to each implementation.

Specifying the Templates of Module Families: The structural portion of a module family template is specified by listing the (minimal) set of resources to be *provided* by the family. This characterizes the structure of all the members of the family which are constrained to provide at least the resources prescribed in the module family template.

```
module family LineStorage is
  provides Line {MaxLine,MaxWord,MaxChar,LineN,WordN,
                 CharN,Lines,PutChar,GetChar,DelLine,DelWord}
  properties
     word = string;
     eline = sequence of word;
     all_lines = sequence of eline;
     states :: all_lines
     type
        PutChar: integer integer integer character --->
        GetChar: integer integer integer ---> character
        DelLine: integer --->
        DelWord: integer --->

     pre.PutChar(n,w,c,ch) ==> 1<=n<=MaxLine and
                               1<=w<=MaxWord and c<=MaxChar
     post.PutChar(n,w,c,ch) ==> all_lines'[n,w,c]=ch

     pre.GetChar(n,w,c) ==> 1<=n<=MaxLine and
                            1<=w<=MaxWord and 1<=c<=MaxChar
     post.GetChar(n,w,c,r) ==> r=all_lines[n,w,c]

     pre.DelLine(n) ==> 1<=n<=MaxLines
     post.DelLine(n) ==> all_lines'[n] = nullseq

     pre.DelWord(n,w) ==> 1<=n<=MaxLines and
                                    1<=w<=MaxWords
     post.DelWord(n,w) ==> all_lines'[n,w] = nullseq
  implementations
    . . . < implementation specified in Figure 3. >
  end LineStorage
```

Fig. 2. Template specification for LineStorage module family.

The second part of the template specification lists the *functional properties* of each resource provided by the module family in a nonprocedural, implementation-independent manner. The specifications of these properties of the resources are essentially the same as those discussed in the specification of abstract data types or system prototypes described with operational specifications [2]. The specific language which is used to specify these properties depends on the method of specification selected. In particular, one could use algebraic techniques [13], [38], state machine techniques [30], abstract model techniques [16], [19], [25], or operational specification techniques [2].

The specific syntax for the structural and functional portions of the template specification are illustrated in the example specification of the LineStorage module family for the KWIC Index System in Fig. 2. For the purposes of illustrating the ideas in this work, we choose to employ the abstract model techniques of [16], [19], [25], and we borrow their notation as well. However, our use of the abstract model techniques is not to be construed as an advocacy of that technique alone.

The module family provides an abstract data type called Line, whose behavior is modeled by viewing the lines as a sequence of lines, with each line being a sequence of words. The properties of sequences are assumed to be specified already, perhaps in a standard library. The operations defined are specified by giving their pre- and postconditions, which are assertions about the lines before and after the operation is invoked.

Specifying Implementations in NuMIL: There may be several implementations of the same module family, each of which we consider to be a distinct member or version of the family. Therefore the implementation specification is essentially a list of version descriptions, each specifying one particular member of the family. The implementation specifications can be omitted completely in the initial phases of the architectural design description because the concrete details of an implementation may not yet be determined.

Each version specification has three parts:

1) The *realization* specification of the implementation, which merely provides the name of the source file which realizes that implementation.

2) The *concrete interface*, i.e., a list of provided and required resources, with the syntactic attributes of each resource described in the declarative syntax of the implementation language.[4]

3) The *functional properties* of each operation, based on the semantics of the underlying programming language, given in the form of pre- and postcondition assertions, in a Hoare-like notation.

An example of the specification of a particular version of the LineStorage module family is shown in Fig. 3. The example is based on Ada® [18] as the implementation language. The resources specified in abstract fashion in the template portion are given concrete syntactic attributes, described in the declarative syntax of Ada. The concrete pre- and postconditions of operations are also provided.

```
module family LineStorage
  < . . . Template specification from Figure 2. >
implementations
  version corearray {  realization corearry.ada;
    provides
      package Line is
       constant MaxLine, MaxWord, MaxChar : INTEGER;
       type LineN is range 1..MaxLine;
       type WordN is range 1..MaxWord;
       type CharN is range 1..MaxChar;
       readonly Lines:LineN;
       procedure PutChar(L:LineN,W:WordN,C:CharN,CH:CHARACTER);
       function GetChar(L:LineN,W:WordN,C:CharN)return CHARACTER;
       procedure DelLine(L:LineN);
       procedure DelWord(L:LineN,W:WordN);
      end Line;
    requires Storage_Error_Handler;
      properties
        private type lines is
        array[1..MaxLines,1..MaxWord,1..MaxChar] of CHARACTER;
        all_lines:lines;
        states:: all_lines

        pre.PutChar(n,w,c,ch,<all_lines>,<all_lines'>) ==>
          1<=n<=MaxLine and 1<=w<=MaxWord and 1<=c<=MaxChar
          post.PutChar(n,w,c,ch,<all_lines>,<all_lines'>) ==>
              all_lines'[n,w,c]=ch

        pre.GetChar(n,w,c,<all_lines>,<all_lines'>) ==>
          1<=n<=MaxLine and 1<=w<=MaxWord and 1<=c<=MaxChar
        post.GetChar(n,w,c,<all_lines>,<all_lines'>) ==>
          GetChar=all_lines[n,w,c]

        pre.DelLine(n,<all_lines>,<all_lines'>) ==>
          1<=n<=MaxLines
        post.DelLine(n,<all_lines>,<all_lines'>) ==>
            all_lines[n]=0

        pre.DelWord(n,w,<all_lines>,<all_lines'>) ==>
           1<=n<=MaxLines and 1<=w<=MaxWords
        post.DelWord(n,w,<all_lines>,<all_lines'>) ==>
            all_lines[n,w]=0
    }
end LineStorage
```

Fig. 3. Implementation specification for LineStorage module family.

B. Subsystem Family Specifications in NuMIL

Like a module family, each subsystem has a template specification to be satisfied by all its members. However, unlike a module family, each member of the subsystem is essentially a configuration. Each configuration is a list of named components which participate in the configuration, and any of the named components could, in turn, be a configuration. This permits the language to be used to specify the configurations of any software system. The second portion of a subsystem specification lists the configurations which realize the template for the subsystem family. The configuration specifications may be omitted entirely as shown above if only the subsystem template has been finalized. This often happens in the early stages of a software development project.

Each component in a configuration is a primitive module family member (a source file), or another configuration. Each individual component in the configuration is specified by providing the name of the (module or subsystem) family, and a selector which names a particular member of the family to be used in the construction of the configuration. For example, in Fig. 4, the configuration "KWICsmall" of the subsystem KWIC is constructed by using the "term" version of the "input" module, the "core" version of the "line-storage" module, etc. The selector used in naming the component may be omitted if it is not known which particular member of the family will be used in the configuration. However, such unspecified

```
subsystem KWIC is
   provides Kwic;
   configurations
      KWICsmall = { Input:term, LineStorage:core,
                    Index_Gen:core, Output:file ;
                    provides procedure Kwic;}
      KWICbig = { Input:file, LineStorage:isam,
                  Index_Gen:isam, Output:file ;
                  provides Kwic;}
end KWIC

subsystem Index_Gen is
  provides Alph, Ith, Shifted_Lines;
  requires Line, ISAMPackage, SAMPackage;
      configurations
         core = { Alphabetizer:core, Circular_Shifts:comp ;
            provides
               procedure Alph;
               procedure Ith(1:ShiftN) return ShiftN;
            requires Line;}
          isam = { Alphabetizer:isam, Circular_Shifts:index;
            provides Alph,Ith;
            requires Line,ISAMPackage;}
end Index_Gen

module Alphabetizer is . . . end
module Circular_Shifts is . . . end
module LineStorage is . . . end
module Input is . . . end
module Output is . . . end
```

Fig. 4. Subsystem specifications for KWIC system.

[4]The implementation language is not specified by us. Any programming language may be used, such as Fortran, C, or Ada [18]. However, we do impose the restriction that there is only one programming language. This eliminates the need to have a special language to specify the syntax of resources, as in [36].

®Ada is a registered trademark of the U.S. Department of Defense (Ada Joint Program Office).

selectors have to be resolved before the configuration can actually be constructed or checked for certain properties. The default value for the selector is designated, in case situations arise where the particular member of the family is unspecified. The syntax for specifying subsystems is illustrated in Fig. 4, and formally defined elsewhere [23], [24].

C. Guidelines to Developing NuMIL Specifications

NuMIL is a language to describe the architecture of software system configurations, no matter how they were designed. Several disciplines may be used in arriving at a particular architecture. For example, one could use information hiding [26], abstract data types [13], Hierarchical Development Methodology (HDM) [34], or module coupling and cohesion [21] in arriving at a particular modular structure for the system.

When one uses bottom-up design, one starts by specifying the primitive modules, realizing, either from experience or through other means, that the module is needed to implement the system. On the other hand, with top-down design, one would start with the subsystem hierarchy, until the primitive modules in the hierarchy are determined. Whatever the situation, interfaces of primitive modules must be specified. What sorts of resources should they provide? What should the properties of the resources be?

When answers to these questions are resolved, NuMIL can be employed to specify the templates of the different module families. These descriptions will include the list of provided resources and an abstract characterization of the properties of those resources. The abstract properties can be specified in several ways as we have suggested. Thus, one develops the different module family descriptions in the system characterized by the resources which they provide, and the functional properties of those resources.

The details of the NuMIL specification, such as what the particular concrete representations of resources ought to be in a particular version of the module family, or the concrete functional specifications are filled in gradually as more is understood about how the system is supposed to fit together. The syntactic representations chosen for each resource symbolize a deeper commitment to a particular configuration.

If alternatives for implementations are needed, the syntax of NuMIL can be used to gracefully accommodate the fact that there exists more than one implementation of the module family template. Recording this information in NuMIL will also aid potential users of the resources to determine whether they want to use the resources in the first place (from the module family template), and then a particular version, depending on the specific syntax of resources in the version.

IV. Formal Properties of System Configurations

By viewing a module family and a system configuration as we advocate, we can provide rigorous definitions for several important configuration maintenance concepts. These properties will be used in maintaining the integrity of the system as it evolves.

A. Criteria for Module Family Membership

While presenting the basic framework for characterizing module families, we have intuitively introduced several key properties of a module family template specification, implementation specifications, and the relationships between them. We can now introduce rigorous criteria for an arbitrary source module to be a member of a module family.

We require that a module conform to its module family template in two broad ways:

1) The structural portion of the module interface should conform to the module family template.

2) The functional properties of the resources from the module interface conform to those specified in the module family template.

We now explore these relationships in some detail.[5]

Structural Conformity: In the following definitions, let M^f be a module family, and M be any particular version or member of the family. In general, $p(M)$ and $p(M^f)$ denote the set of resources provided by the module M and the module family template for M^f, respectively. Similarly, $r(M)$ denotes the set of resources required by the module M.

A module M is said to be in *structural conformity* with respect to the template for M^f if and only if

$$p(M) \supseteq p(M^f)$$
$$p(M) \cap r(M) = \phi.$$

The first of the above assertions ensures that the module M provides the resources specified in the template for M^f, and the second assertion ensures that no module provides and requires the same resource.

Conformity of Resource Properties: While the notion of structural conformity relates merely the names of the resources from the module family template to the names of the resources in the concrete interface, we now define when we consider the properties of the resources in the module to be in conformity with the abstract properties in the module family template.

Every module is realized by a source file which is its concrete embodiment. Consider the module M and the template for the module family M^f. In order for such a module to satisfy the module template, there are two criteria:

- The source file for the module M satisfies the concrete interface for M.
- The concrete interface for M satisfies the abstract interface specification for the template for the module family M^f.

We will now show how each of these criteria can be shown to hold for a particular source file. These notions

[5]The criteria for family membership as set forth in this section represent natural extensions of Tichy's proposal in [36], molded to fit our specific view of module families.

are well-understood and have been explored in the formal specification literature [15], [16], [11], [4].

To show that a source file satisfies its concrete interface, one has to show that the source file is correct with respect to the concrete interface specification. This corresponds to the conventional notions of partial or total correctness depending on whether or not one includes a termination clause in an operation's postconditions. Since we stipulated that the pre- and postcondition clauses in the concrete interface must be consistent with the syntax and semantics of the implementation language, one can readily use the inductive assertion method of Floyd [10], the deductive systems of Hoare [15], or different testing techniques (e.g., see [12], [17]) to establish that the source file satisfies the concrete interface.

The concrete interface for any particular module must model the properties described in abstract terms in the module family template. This relationship is also well-understood. The abstract specification of the resource properties is written in implementation-independent terms, (i.e., it is based on some arbitrary mathematical model). We must now show that the concrete interface embodies those properties in the implementation language. The relationship is described in mathematical terms through the definition of a homomorphism from the concrete specification onto the abstract specification [19], [4], [15]. This is discussed in more detail elsewhere [23].

For our purposes here, it suffices to note that the notion of module family membership is readily related to the conventional notions of the correctness of implementations with respect to their specifications.

B. Well-Formed Configurations

For a configuration to be actually constructed, it must not violate any of the constraints implied by the interfaces of its components. The integrity of the assumptions that modules in a configuration may make about each other must be assured for a configuration to be meaningful. The characteristics of meaningful configurations are embodied in the concept of *well-formed configurations*. Such a notion first appeared in [14], where the properties of well-formed configurations are examined from a purely structural and syntactic viewpoint. However, we also need to consider the complications caused by the functional properties of the resources.

In purely structural terms, one can state that no configuration can provide a resource which is not (eventually) provided by a module in that configuration. Further, the configuration requires (from other modules or configurations) those resources which its constituents require, but which are not provided by other components in the configuration itself.

As with the implementations of module families, a configuration cannot provide and require the same resource. It is also important to ensure that no resource is provided in more than one component [14]. This constraint is necessary to simplify the rules for configuration of modules, and makes much intuitive sense by avoiding the confusion of having two different resources with the same name being exchanged between modules.

A configuration, $C = \{C_1, C_2, \cdots, C_n\}$, where each C_i may be a module or another configuration, is said to be *well-formed* if and only if

1) Every resource provided by C is provided by some C_i, i.e.,

$$p(C) \subseteq \bigcup_{i=1}^{n} p(C_i).$$

2) C requires those resources required by all the C_is except for the resources already provided by some other component in the configuration, i.e.,

$$r(C) = \left(\bigcup_{i=1}^{n} r(C_i)\right) - \left(\bigcup_{i=1}^{n} p(C_i)\right).$$

3) C does not provide and require the same resources

$$p(C) \cap r(C) = \phi.$$

4) The resources provided and required by each component of C are disjoint.

$$p(C_i) \cap r(C_i) = \phi.$$

5) No resource is provided by more than one component.

$$p(C_i) \cap p(C_j) = \phi, \quad \text{for all} \quad C_i, C_j \in C, i \neq j.$$

6) Uses of resources across module boundaries are syntactically consistent with their definitions (i.e., intermodule type checking).[6]

7) All the modules in the configuration C satisfy their respective module family templates.

The above definition characterizes those configurations in which the assumptions made by the participants about each other are preserved. This helps in enforcing the continued validity of these configurations as the system is altered. Last, this definition of configuration is similar to Tichy's in [36], but the conditions are described in line with our view of module and configuration families.

C. The Upward Compatibility Relationship

The properties discussed earlier are properties of software system configurations in a static sense. They provide us with useful characterizations of the integrity of the software system configurations without regard to change. However, both the descriptions and the software system which they describe are subject to alterations. This necessitates creating additional mechanisms to characterize how the software system may be altered. Which alterations are relatively easy to carry out in terms of not violating the assumptions that modules make about each other? The notion of *upward compatibility* between the members of a module family provides the means to answer these questions fairly efficiently.

[6]This is already provided in several modern programming languages such as Ada [18], CLU [20], etc.

Motivation for Upward Compatibility: Software systems are generally altered incrementally. As these alterations are carried out, it is of vital importance to ensure the continued validity of the system configurations. The alterations should not inadvertently violate the integrity of the intermodule dependencies. If the alterations were intended to alter the intermodule dependencies, then corresponding alterations must be made in the dependent modules so as to preserve consistency in the system configuration. Each system configuration embodies a set of design choices and some software system structuring principles. Any alteration to the existing architecture should be reviewed, and must therefore be detected and analyzed.

The discussion above raises the question of when changes to one module require corresponding changes in the modules which depend on it. We call this effect *propagation* of changes to ensure consistency in the configuration. The alterations which do not require any propagation to other modules are intuitively the easiest to effect because each module can be changed in isolation from other modules. Upward compatibility characterizes this notion rigorously. Further, by noting which parts of the upward compatibility criteria fail, we can better characterize the nature of the required propagation.

An initial notion of upward compatibility was presented by Tichy in [36] and [37]. He defined this compatibility in terms of the structure and syntax of the interfaces of the modules in question. In our case, we rigorously describe what the relationships between the resource properties in the two interfaces must be for the upward compatibility relationship to hold between members of a module or configuration family. Therefore, our formalization builds upon Tichy's initial foundation.

Upward Compatibility Relationship Between Versions: *A version* M_1 of a module family M^f is *upward compatible* to version M_2 (written M_1 **upc** M_2) if and only if

1) M_1 provides at least all the resources provided by M_2.

$$p(M_1) \supseteq p(M_2).$$

2) M_1 requires no resources which are not required by M_2.

$$r(M_1) \subseteq r(M_2).$$

3) The syntactic properties of the resources common to the two versions (provide and require clauses) are identical.

4) For each operation O in both M_1 and M_2, the following relationship holds in the *concrete* properties:

- $\text{pre}(O, M_2) \rightarrow \text{pre}(O, M_1)$
- $\text{post}(O, M_1) \rightarrow \text{post}(O, M_2)$.

Intuitively, the above definitions are devised to capture those situations when the usage of any particular resource R as provided in M_1 will satisfy usages of the same resource as provided in M_2. Therefore, M_1 may replace M_2 in any well-formed configuration, so long as the resources provided by M_1 are mutually disjoint with the resources provided by other components in that configuration.[7] In particular, consider the situation when M_1 is an altered form of M_2. So long as we ensure that the just discussed condition holds, we assert *upward compatible alterations preserve well-formed configurations.*

Upward Compatibility and Family Membership: Since the upward compatibility relationship preserves the integrity of intermodule dependencies, a relevant question is whether or not it also preserves membership in the family. In other words, we wish to discern whether or not if a module M_1 is upward compatible to M_2, and M_2 has already been shown to be a member of a particular family of modules, then M_1 also belongs to the same family.

Theorem: If M_1 upc M_2 and $M_2 \in M^f \rightarrow M_1 \in M^f$.

The proof of this claim is detailed elsewhere [23]. For this paper, we merely explore the implications of the result. From the result above, we can now assert that if M_1 is upward compatible to M_2, and if M_2 has already been shown to satisfy the template for a particular module family M^f, then M_1 also belongs to the same family. Consider the case when module M_2 is modified to get module M_1. In this case, M_2 is altered upward compatibly to get M_1. By the result, we know that not only can M_1 be used in all the configurations that M_2 is currently being used in, but if M_2 was a version of the family M^f, then so is M_1. This is essentially the version control problem, and therefore so long as a particular module is altered upward compatibly, the resulting module belongs in the same family as the original module. Thus, *upward compatible alterations preserve family membership.*

D. Upward Compatibility in Software System Evolution

In the previous section, we have shown how the upward compatibility relationship preserves intermodule dependencies and family membership. Hence, alterations which cause the concerned modules to change upward compatibly are the easiest to effect. If one considers the existing configurations of a software system as being the concrete embodiment of a particular set of assumptions between the different modules, upward compatible alterations leave the integrity of those assumptions, and hence the integrity of the configurations, unaltered.

Given the very diverse universe of alterations that software maintainers routinely carry out, it is fair to say that many alterations will *not* be upward compatible. However, the idea is that in carrying out the tests for upward compatibility, one will uncover those aspects of the tests which fail, providing important information to the maintainer about what else needs to be altered to preserve consistency. This would also ensure that a maintainer never inadvertently violates the consistency of configurations.

Another important aspect of upward compatibility is that it is geared towards incremental analysis of module

[7]Recall that a well-formed configuration is one where the sets of resources provided by each component were mutually disjoint. Since M_1 could provide more resources than M_2 by the above definition, one has to ensure that this condition holds.

interface properties. Since software systems are generally altered incrementally, upward compatibility is easier to show than the full family membership test and well-formed configuration tests discussed earlier in this section. For example, whether one uses either rigorous proof techniques or comprehensive testing strategies to establish family membership and well-formedness of configurations, it is extremely expensive to repeat the procedures each time a module is altered. If the alterations are indeed small, upward compatibility can be shown (manually or automatically) with much more facility than either family membership or well-formedness of configurations.

It can be shown that proving upward compatibility between unrelated source modules is unsolvable [23]. However, speaking intuitively, if one of the modules can be transformed into an incrementally modified form of the other, upward compatibility should be easier to show. Notwithstanding these difficulties, upward compatibility still remains a useful paradigm to adopt in analyzing incremental alterations to software systems. The prototype system which we discuss in the next section of this paper, uses upward compatibility as the primary analytical mechanism to analyze all structural and syntactic alterations.

V. Practical Support for Configuration Maintenance

The concepts discussed in this paper and the system description language NuMIL can be used to provide support for the development and maintenance of software configurations. The most important contribution of this work is the use of a fairly formal incremental analysis (i.e., upward compatibility) of module interface properties as the basis for controlling how a software system is altered and, indeed, understanding what the system looks like at any particular stage of its ongoing evolution.

Several major theoretical problems prevent the use of the complete checking of all the properties of the last section. The family membership criterion is akin to the notion of program correctness, a computationally intractable problem. Similarly, it cannot be automatically determined whether an arbitrary module is upward compatible to another [23]. Notwithstanding these computational barriers, the paradigm of incremental analysis to support software evolution remains a powerful notion. Thus, we decided to forego automating the functional portion of the integrity tests as a compromise to make the system implementable.

A prototype environment based on NuMIL was built to see if it was possible to support the range of configuration maintenance activities using upward compatibility wherever possible to reduce the cost of the analysis [22]. A secondary goal of the prototype environment is to experiment with and understand what sort of architectures are needed to support software configuration maintenance. For instance, we have used a general purpose relational database system, INGRES [35], to store, retrieve, and update all the NuMIL descriptions, enabling the users and tools alike to access the information in one canonical representation. This also permits browsing by users of the information in the database. Users can even tailor the user interface to suit their tastes. This aids in promoting better understanding of the system by its developers and maintainers. The architecture of the environment is shown in Fig. 5.

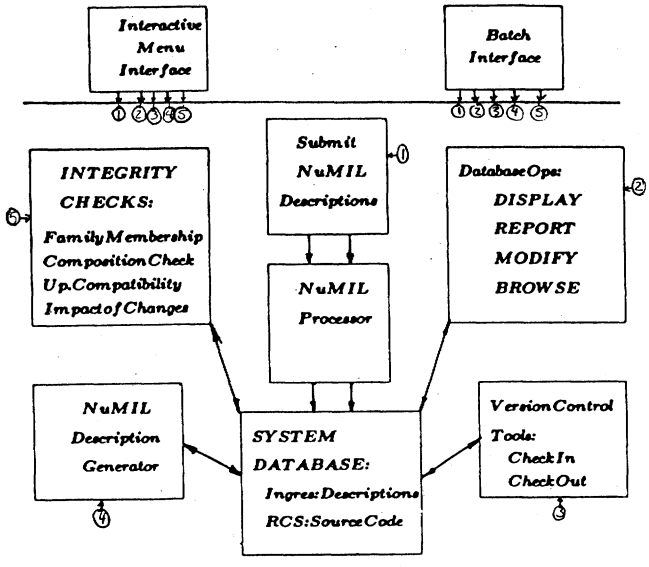


Fig. 5. Architecture of the NuMIL system.

A. Prototype System Architecture

The system consists of two central repositories of information. The first holds a processed form of the NuMIL descriptions of the developing system. The second consists of all the source files and their revisions. All the different tools that are needed in the environment operate on one or the other of these databases.

The use of database technology to aid at least in the information management portion of a software engineering environment has long been a popular idea in the literature. However, very little work has been done exploring the structure and design of a central repository of information for maintaining software system configurations. The NuMIL environment features one possible design of the database capable of handling such information, along with source code and revisions. The complete details of the design of this most important component of the environment are recorded elsewhere [22], [23], [24].

The source files and their revisions are stored using the Revision Control System (RCS) [37], available with the UNIX 4.2bsd system. RCS has mechanisms to store source files and their revisions in a space-efficient manner. Our basic idea in employing RCS was to use it to manage the storage of source files and revisions, while employing INGRES to store an intermediate form of the NuMIL system configuration descriptions [23].

B. Tools and Facilities

The different tools shown in Fig. 5 are built around the central database. The tools offered the user fall into several major categories:

1) Tools which handle the interactive or batch submission of new NuMIL system descriptions. These descriptions are translated into the appropriate relations in the INGRES database.

2) Facilities to help the user to interactively retrieve or update database information. A menu-based browse interface greatly aids in this area. Reports can be generated in any desired format regarding alterations to configurations, or descriptions of particular objects, etc.

3) Tools for routine maintenance, such as the tools which check-in and check-out source files. Concurrent updates to the same source file by two users are prevented by RCS. When the file is checked-in, its interface properties are checked by seeing if it is upward compatible to the file that was checked-out.

4) Tools to check and maintain the integrity of system configurations form the bulk of the system. All the notions of integrity discussed in the conceptual portion of this paper are included: family membership, well-formed configurations, and upward compatibility. Wherever possible, upward compatibility is used for analysis instead of the other tests, thereby greatly reducing the amount of computation required to assure system integrity.

The tools mentioned above are all integrated into a single system with a uniform menu-driven interface, enabling us to offer some of the basic software configuration management services outlined in [1], [5]. The first release of the prototype system is now being used by graduate students at USC in the development of large software systems as part of the System Factory Project [32], [33]. The feedback from the students is being used as a means to evaluate the validity of the ideas proposed in this work, and to provide suggestions for the compromises necessary to implement the complete framework.

VI. Conclusions

In this paper, we presented an approach to maintaining the integrity of evolving software system configurations. The contributions of this paper are:

- A view of software system configurations which uses the abstract interface properties as the basis of structuring a system into module families.
- Use of the formal properties of system configurations viewed as above in devising rigorous notions of system integrity.
- Proposal of the upward compatibility notion to analyze incremental alterations to system configurations.
- Discussion of the capabilities and operational of a language, NuMIL, and an accompanying prototype environment to maintain software system configurations.

The above aspects of our paper provide a sound technical basis to address the daunting problem of software configuration management.

We envisage much more research in the future in several areas which we have not fully explored in this paper. At a conceptual level, one has to investigate the compromises necessary to make the full NuMIL framework practicable. For instance, it is known that proofs of correctness of a program can be immensely hard to construct. The motivation to construct a proof virtually disappears if one has to construct a new proof every time a program changes. The notion of upward compatibility, based on incremental change, might be extended to include notions which guarantee the validity of the proofs for certain classes of changes. This is a fruitful area of investigation.

At a more practical level, one could consider coupling the checking of interface properties with a language-directed editor so that these checks can be carried out as the alterations are taking place. This will provide continuous feedback to maintainers as they are changing the system. While our work has focused primarily on source code alterations and their descriptions, a second line of investigations could include a comprehensive characterization of all the different structural and functional alterations that maintainers carry out, and their effects on the integrity of system configurations. This will enable us to reason more effectively about system alterations.

References

[1] W. Babich, *Software Configuration Management*. Reading, MA: Addison-Wesley, 1986.

[2] R. Balzer, N. Goldman, and D. Wile, "Operational specifications as a basis for rapid prototyping," *Software Eng. Notes*, vol. 7, no. 5, pp. 3–16, 1982.

[3] L. A. Belady and M. M. Lehman, "The characteristics of large systems," in *Research Directions in Software Technology*. Cambridge, MA: M.I.T. Press, 1979, pp. 108–138.

[4] H. K. Berg, *et al.*, *Formal Methods of Program Verification and Specification*. Englewood Cliffs, NJ: Prentice-Hall, 1982.

[5] E. Bershoff, V. Henderson, and S. Siegel, *Software Configuration Management*. Englewood Cliffs, NJ: Prentice-Hall, 1980.

[6] L. W. Cooprider, "The representation of families of software systems," Ph.D. dissertation, Carnegie-Mellon Univ., Apr. 1979.

[7] F. L. Czerwinski and T. T. Samaras, *Fundamentals of Configuration Management*. New York: Wiley-Interscience, 1971.

[8] F. DeRemer and H. H. Kron, "Programming-in-the-large versus programming-in-the-small," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 80–86, June 1976.

[9] S. I. Feldman, "MAKE—A program for maintaining computer programs," *Software—Practice and Experience*, vol. 9, pp. 255–265, 1979.

[10] R. W. Floyd, "Assigning meanings to programs," in *Proc. Symp. Appl. Math.*, Amer. Math. Soc., 1967.

[11] J. A. Goguen, J. W. Thatcher, and E. G. Wagner, "Initial algebra approach to specification, correctness, and implementation of abstract data types," in *Current Trends in Programming Methodology*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

[12] J. Goodenough and S. L. Gerhart, "Towards a theory of test data selection," *IEEE Trans. Software Eng.*, vol. SE-1, no. 3, pp. 179–191, 1977.

[13] J. V. Guttag, "The specification and application to programming of abstract data types," Ph.D. dissertation, Univ. Toronto, 1975.

[14] A. N. Habermann and D. E. Perry, "System composition and version control for ADA," in *Software Engineering Environments*. Amsterdam, The Netherlands: North-Holland, 1981.

[15] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, pp. 576–580, Oct. 1969.

[16] ——, "Proof of correctness of data representations," *Acta Inform.*, vol. 1, no. 3, pp. 271–281, 1972.

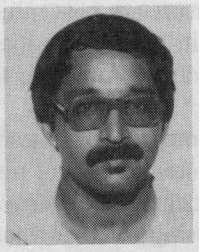
[17] W. E. Howden, "Functional program testing," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 162–169, 1980.

[18] J. D. Ichbiah, *et al.*, "Preliminary ADA reference manual," *SIGPLAN Notices*, vol. 14, no. 6, 1979.

[19] C. B. Jones, *Software Development—A Rigorous Approach*. Englewood Cliffs, NJ: Prentice-Hall, 1980.

[20] B. H. Liskov, *et al.*, *CLU Reference Manual* (*Lecture Notes in Computer Science*). New York: Springer-Verlag, 1981.

[21] G. Myers, *Composite/Structured Design*. New York: Van Nostrand Reinhold, 1978.
[22] K. Narayanaswamy and W. Scacchi, "An environment for the development and maintenance of large software systems," in *Proc. 2nd SOFTFAIR*, IEEE Comput. Soc., 1985, pp. 11–23.
[23] K. Narayanaswamy, "A framework to support software system evolution," Ph.D. dissertation, Univ. Southern California, May 1985.
[24] K. Narayanaswamy and W. Scacchi, "A database foundation to support software system evolution," *J. Syst. Software*, 1987, to be published.
[25] J. R. Nestor, W. A. Wulf, and D. A. Lamb, "IDL—Interface description language: Formal description," Dep. Comput. Sci., Carnegie Mellon Univ., Tech. Rep., 1981.
[26] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM.*, vol. 15, pp. 1053–1058, Dec. 1972.
[27] ——, "Designing software for ease of extension and contraction," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 128–137, Mar. 1979.
[28] M. H. Penedo and D. Berry, "The use of a module interconnection language in the SARA system design methodology," in *Proc. 4th Int. Conf. Software Eng.*, 1979, pp. 294–307.
[29] R. Prieto-Diaz and J. Neighbors, "Module interconnection languages: A survey," Univ. California, Irvine, ICS Tech. Rep. 189, 1982.
[30] L. Robinson and O. Roubine, "SPECIAL—A specification and assertion language," Stanford Res. Inst., Rep. TR CSL-46, Jan. 1977.
[31] M. J. Rochkind, "The source code control system," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 364–370, Dec. 1975.
[32] W. Scacchi, "The system factory approach to VLSI and software engineering," in *Proc. 2nd AFCET Software Eng. Conf.*, 1984, pp. 149–157.
[33] ——, "A software engineering environment for the system factory," in *Proc. 19th Hawaii Int. Conf. Syst. Sci., Software*, vol. 2, 1986, pp. 822–829.
[34] B. Silverberg, "An overview of the SRI hierarchical development methodology," SRI International, Tech. Rep. CSL-116.
[35] M. Stonebraker, P. Kreps, and G. D. Held, "The design and implementation of INGRES," *ACM Trans. Database Syst.*, vol. 1, no. 3, pp. 189–222, 1976.
[36] W. F. Tichy, "A data model for programming support environments and its application," in *Automated Tools for Information System Design and Development*, H.-J. Schneider and A. I. Wasserman, Eds. Amsterdam, The Netherlands: North-Holland, 1982, pp. 31–48.
[37] W. F. Tichy, "RCS—A system for version control," *Software—Practice and Experience*, vol. 15, no. 7, pp. 637–654, 1985.
[38] S. N. Zilles, "Data algebra: A specification technique for data structures," Ph.D. dissertation, Massachusetts Inst. Technol., 1975.



K. Narayanaswamy received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Madras, the M.S. degree in computer science from the University of Nebraska—Lincoln, and the Ph.D. degree in computer science from the University of Southern California, Los Angeles.

He is now a Research Computer Scientist with the USC/Information Sciences Institute, Marina del Rey, CA. His main research interests are in formal software specification, software development environments, and the application of AI techniques to software engineering.

Walt Scacchi (S'77–M'80), for a photograph and biography, see this issue, p. 323.