# A Graph Model for Software Evolution

LUQI

*Abstract*—**This paper presents a graph model of software evolution. We seek to formalize the objects and activities involved in software evolution in sufficient detail to enable automatic assistance for maintaining the consistency and integrity of an evolving software system. This includes automated support for propagating the consequences of a change to a software system.**

*Index Terms*—**Configuration control, consistency, management, maintenance, software evolution.**

## I. Introduction

EVEN though the evolution of software systems accounts for the bulk of their cost, there is currently little automated support for evolution, especially when compared to other aspects of software development. This state of affairs is partially due to lack of tractable formal models for the process of software evolution. We propose a graph model of software evolution to help address this problem, and show how our model can help in maintaining the consistency of a changing system. We are particularly concerned with large and complex systems, which often have long lifetimes and undergo gradual but substantial modifications because they are too expensive to discard and replace. Computer assistance is essential for effective and reliable evolution of such systems because their representations and evolution histories are too complex for unaided human understanding. Computer-aided evolution is particularly important in rapid prototyping, where exploratory design and prototype demonstrations guide the development of the requirements via an iterative process that can involve drastic conceptual reformulations and extensive changes to system behavior [9].

Software evolution involves change requests, software systems, and evolution steps as well as customers, managers, and software engineers. Customers include the people and organizations who use software systems and have funded their development and evolution. Change requests come from customers, and the corresponding changes are controlled by the managers of the software system. Change requests that are approved by the management trigger evolution steps which produce versions of the system incorporating the requested changes. The evolution steps are scheduled by the management, and are carried out by the software engineers.

Manuscript received December 15, 1989; revised April 9, 1990. Recommended by M. Zelkowitz. This work was supported in part by the National Science Foundation under Grant CCR-8710737.

The author is with the Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943.

IEEE Log Number 9036535.

Both software systems and evolution steps typically have hierarchical structures. Software systems are viewed and manipulated as structured collections of software components of many different types, such as requirements, specifications, design descriptions, source code modules, test cases, manuals, etc. Similarly, evolution steps are viewed and scheduled as structured collections of related substeps, such as job assignments for organizations and individuals, and changes to subsystems and individual software objects. A software component or a step is *composite* if it can be viewed as a collection of related parts, and is *atomic* otherwise. The customers are usually directly concerned only with the top levels of these structures, which correspond to delivered systems and responses to change requests, respectively. Top-level components and steps can be either atomic or composite. For large systems, top-level components and steps are usually composite, with several levels of decomposition between the top level and the atomic parts.

As systems change, they go through many different versions. An *object* is a software component that is subject to change. Objects can be either composite or atomic, and can represent both systems and individual modules. A *version* is an immutable snapshot of an object. Versions have unique identifiers. New versions can be created, but versions cannot be modified after they are created. Objects can be changed only by creating new versions. Because previous versions are not destroyed when a new version is created, the state of an object consists of a partially ordered set of versions, rather than a single version.

This paper explores the general class of objects subject to version control. We view each type of software object—such as a specification or a program—as a subclass of the general class "versioned-object." Each subclass provides additional operations and properties relevant to each kind of software object. Our discussion is independent of the additional operations and properties provided by the more specific subclasses.

A distinguishing characteristic of versioned-objects is that they are persistent. Thus versioned-objects are more closely related to the objects in object-oriented databases than they are to the objects in object-oriented programming languages. In the rest of this paper we refer to versioned-objects simply as objects.

Large systems change gradually, in relatively small steps. The direct effect of each step in the evolution of a system is a change in one or more of the component objects comprising a system. These changes affect the functionality and the performance of the system as well as its

U.S. Government work not protected by U.S. copyright

representation, and must respect many dependencies between the components to avoid damaging the system. Considering the complexity of current software systems and the scope and frequency of the changes they typically undergo, complete and effective control over the set of configurations is imperative for successful system evolution. Two of the main objectives of version management are ensuring that consistency constraints are met and coordinating concurrent updates to subcomponents of a system.

Evolution steps can be represented as dependency relations between versions. There are often very many evolution steps in the lifetime of a system, and some of these steps may fork off new branches to create families of alternative versions of the system, which may differ in functionality of performance, and may interface to different operating environments, peripherals, or external systems. The complexity of this structure and the dependence of future changes on past design decisions makes it important to record the evolution history of a system.

We assume that each object has one or more alternative *variations*. A variation of an object is a totally ordered sequence of versions of the object which represents the evolution history of an independent line of development. Each version of an object belongs to exactly one variation. Each variation has a unique identifier. All variations of an object share some common properties which characterize the identity of the object. A new variation for an object is created when one of its lines of development branches. Different variations of an object have different properties of interest to the designers, such as those listed above. Variations can be organized using *generalization per category* [8], which provides a structure useful for supporting browsing tools and mapping values for sets of categorical properties meaningful to the users into internal unique identifiers, thus supporting retrieval and specification of variations based on information familiar to the users.

We seek to formalize the evolution history to provide computer aid for maintaining the consistency of the configurations in the product repository. After summarizing some relevant previous work in Section II and presenting the details of our formalized graph model in Section III, we describe the possibilities for computer aid in Section IV and present some conclusions in Section V.

## II. Summary of Previous Work

We briefly review some recent work on configuration management, emphasizing the aspects most closely related to this paper. Our work adapts and extends some of the concepts and structures introduced in earlier models [3], [4]. This section concludes with a summary of the relevant aspects of these models. The rest of the paper refines and extends these concepts to reflect specifics of software evolution.

### A. Related Work on Configuration Management

The goals of configuration management include recording the development history of evolving systems, maintaining the integrity of such systems, and aiding the management of the systems in guiding and controlling their evolution. Until relatively recently, configuration management was carried out via a combination of manual and administrative procedures. Early attempts at providing automated support for these functions were aimed at identifying and efficiently storing many versions of the same document, and at keeping versions of mechanically derived software objects up to date.

The problem of maintaining the integrity of an evolving configuration has been addressed more recently via module interconnection languages [12]. The purpose of a module interconnection language is to record the interdependencies between the components of a system. The approach reported in [12] includes specifications of functional properties of modules, in addition to the structural and syntactic properties captured by earlier approaches. This language provides a textual form for recording which versions are compatible with which other versions in a related family of software systems. A major contribution of this work is the recognition that specifications can be ordered by an upward compatibility relationship, and that one component can be substituted for another even in cases where the specifications of the components differ, provided that the specification of the new component is an upwards compatible extension of the specification for the original component. This idea is orthogonal to our contribution, and can be beneficially combined with the formulation presented in this paper. The Inscape environment [13] provides several refined versions of upward compatibility, as well as strict compatibility and implementation compatibility. Implementation compatibility is a weaker restriction than upward compatibility which allows one specification to be substituted for another in particular contexts. The concept of obligations in the Inscape environment also supports automatically determining whether an induced step needs to actually make any changes, and locating the aspects of a component that must be changed by an induced step.

Our work is concerned with clarifying the concepts associated with configurations to enable automated tool support for exploring design alternatives in the context of prototyping, for providing concurrency control in situations where many designers are simultaneously working on different aspects of the same system, and for aiding management in controlling and directing the evolution of complex systems at a conceptually manageable level of detail. Rather than introducing a special language for recording dependencies, we rely on existing specification languages to represent semantic constraints, and include specification objects in the configuration. This simplifies the dependencies between components, resulting in a graph structure that can be represented and maintained via established database technology, and can be treated uniformly for all types of software objects. Our work explicitly provides frozen versions, which are necessary to provide stability in a project setting, and applies to all kinds of software objects. The work reported in [12] is limited to just specifications and programs.

Other work has addressed the problem of maintaining mechanically-derivable software objects [6]. The main contribution of this work is to do opportunistic evaluation of derived components based on forward chaining and a set of rules and strategies that represent a model of the user's intentions and the systems capabilities. These rules can also be applied via backward chaining, which provides a mechanism for the system to deduce what tools must be applied to which components to achieve a state requested by the user. The problem of maintaining rederivable components is not addressed in this paper, and the solutions to these problems can be profitably integrated with our approach. The work reported in [6] does not directly address frozen versions and the details of managing configurations. Our view of dependencies between source objects can readily be encoded in rules of the style reported in [6], and our model can be implemented using the system described there. Transactions are considered in [7], which proposes some programming language constructs for realizing nested atomic transactions that can take long periods of time without blocking other concurrent activities. This work does not characterize the integrity properties of the software configuration that should be maintained, and does not link the commit protocols of the transactions to management controls. We provide a graph model that captures these integrity properties, and extend the transaction commit protocol to provide management controls on a high level that can be mechanically extended to the detailed evolution steps that realize a high level change.

The work described above [6], [7], [12], [13] appears to be based on the implicit assumption that only the current version of the object is useful. Our work focuses on maintaining the entire history of each object, not just the most current version. This is most important for groups of evolving systems that share evolving reusable components. Histories are considered in the Cosmos system [14], which provides a distributed database for supporting software development environments. This work is compatible with ours, and provides a means for realizing our graph model in a practical setting. Their consistent domains and domain relative addressing provide solutions for the problems of providing concurrency control that allows a high degree of concurrency in a distributed environment without risk of deadlock. Our work sheds some additional light on the properties of consistent domains, and indicates how the boundaries of a consistent domain might be established by automatic means: a consistent domain consists of the results of an evolution step and all of its induced steps. Nested steps give rise to nested consistent domains.

### B. Concepts from the Model of Software Manufacture

The model of software manufacture [3] was developed to aid in managing versions of mechanically derived objects, with the goals of minimizing the number of objects that must be rederived in response to a change, and of automatically estimating the computing costs associated with installing a proposed change. Our main concern is with the source objects which are produced under the direct control of the software engineers. Several of the concepts of [3] can be readily adapted for formalizing evolution histories in addition to providing support for creating and managing mechanically derived objects.

The model of software manufacture formalizes the concept of a *configuration*. Configurations are intended to capture all of the information that can distinguish between two different versions of a system. The concept of a configuration is important in software evolution because each top-level evolution step produces a new configuration of the evolving system. A configuration is represented as a triple $[G, E, L]$, where $G = [C, S, I, O]$ is a bipartite directed acyclic graph, $E \subseteq C$ is a set of exported components, and $L$ is a labeling function giving unique identifiers for both components and manufacturing steps. The nodes in the graph represent software components ($C$ nodes) and manufacturing steps ($S$ nodes), and the two kinds of nodes alternate on every path in the graph. The arcs in the graph represent input relations between components and manufacturing steps ($I \subseteq C \times S$), and output relations between manufacturing steps and components ($O \subseteq S \times C$).

Exported components correspond to the deliverable parts of a system. Each configuration contains all of the components that can affect the production of the deliverable parts of a specific version of a system, and no other components. The model of software manufacture has a broad view of components, which can include software tools such as compilers and flow analyzers, and tool inputs such as command line options as well as traditional software objects such as test data files and source code modules.

The unique identifiers assigned by the labeling function ensure that each component is the result of a unique manufacturing step. This can be expressed formally as follows.

$$\text{ALL}(m1\ m2{:}S,\ c{:}C{::}\ [m1, c] \in O \,\&\, [m2, c] \in O \Rightarrow m1 = m2) \quad (1)$$

Different invocations of the computations representing a manufacturing step are considered to be distinct, have different unique id's, and produce two different sets of output components, which also have distinct id's. Thus two components are considered to be distinct if they have different derivation histories, if even if the values of the components happen to be the same. The model of software manufacture is constructed in this way to avoid the assumption that all manufacturing steps must be repeatable, so that derivations can involve computations which have persistent states or may be affected by transient hardware faults. The unique identifiers for components and manufacturing steps also allow the graphs corresponding to several different configurations to be combined via graph unions without loss of information. Since the labeling functions are required to give globally unique identifiers with respect to the set of all possible components

and steps rather than with respect to just the components and steps in a single configuration, there is no possibility of losing the distinction between parts of different configurations.

The software components and manufacturing steps in the model of software manufacture correspond to our component versions and evolution steps. However, the model of software manufacture focuses on steps that can be completely automated, such as compilation, and on components that can be automatically generated, such as object code. A typical manufacturing step is illustrated in Fig. 1. Since we are concerned mainly with coordinating the activities of a team of people responsible for the evolution of a software system, rather than on the coordination of a set of programs, our model includes only source objects as component versions and activities involving human interaction as evolution steps. A typical evolution step is illustrated in Fig. 2.

The model of software manufacture provides formal definitions of some concepts useful for describing software evolution. The set of *primitive components* $P$ consists of the components that are not produced by any step, and can be defined formally as follows.

$$P = \{c:C \mid ALL(s:S::[s, c] \notin O)\} \qquad (2)$$

In the context of the model of software manufacture, primitive components are the source objects: those which cannot be mechanically generated. In the context of software evolution, the primitive components form the initial configuration of the system, as delivered by the developers. Modified versions of these original source objects are considered to be derived rather than primitive in our model of software evolution.

The dependency relation $D+$ is defined to be the transitive closure of the relation

$$D = (I \cup O) \qquad (3)$$

induced by the arcs in the graph. Since the relation $D$ is acyclic, $D+$ is a strict partial ordering. The dependency relation represents the dependencies among the components and the steps in terms of the derivation structure. For example, component $c1$ depends on component $c2$ if $[c2, c1] \in D+$ and step $m1$ depends on step $m2$ if $[m2, m1] \in D+$. This dependency relation plays a central role in both the model of software manufacture and in our model of software evolution. For example, it can be used to define the set of steps affected by a change in a component $c$ as follows.

$$\text{affected-steps}(c:C) = \{s:S \mid [c, s] \in D+\} \qquad (4)$$

This set is used to determine which derived components must be recomputed when the component $c$ is changed [3]. In Section IV we develop a refinement of this concept suitable for identifying induced evolution steps.

The model of software manufacture must be extended to represent the issues relevant to sofware evolution because it does not include any representations for future plans, does not admit parts of derivations that do not lead to delivered products, and does not include any representation of the hierarchical structures involved in software evolution. The relations between components in the model of software manufacture are limited to just the dependencies induced by the derivation history. In particular the model of software manufacture has no representation of whether or not two components are different versions of the same object, or whether one component is a part of another component.


Fig. 1. A typical manufacturing step.


Fig. 2. A typical evolution step.

### C. Concepts from the Graph Transform Model

The motivation for the graph transform model [4] is similar to that for the model of software manufacture, and several of the concepts of that model are useful in our context. We classify software objects into two categories: *rederivable* and *nonrederivable*. Rederivable objects can be automatically reconstructed by applying a software tool to a set of software objects, without the need for human intervention. All other objects are nonrederivable. An example of a nonrederivable object is a representation of the user input guiding a computer-aided software design tool. The software objects in the graph transform model can have attributes, which can specify computational procedures that can be applied to the components to perform specific transformations.

There are two important relations between nonrederivable and rederivable objects: *uses* and *derives*. These relations have a direction and have natural representations as directed graphs, as illustrated in Fig. 3.

The relation "derives" is defined between general objects and rederivable objects. The relation represents possible transformations of one or more software objects into other objects (e.g., compilation of source code into object code). The "derives" transformations are associated with the use of software tools in the process of programming and are usually invisible to the management and the customers. In the applications of the graph transform model the set of transformations is usually fixed. Individual transformations from the set can be applied automatically using information about the type of the software object and the attributes associated with the objects. The "derives" relation is used for automatically managing derivable objects, which can be either stored or computed on demand depending on the relative importance of response time and use of storage space. This is an important function in a computer aided evolution environment, which

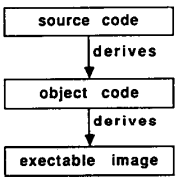


Fig. 3. The derives relation.

we propose to integrate with our approach via the "derives" relationship. The primary focus of our work, however, is the problem of managing the non-re-derivable components of an evolving system.

The "uses" relation is defined between non-re-derivable objects, and represents situations where the semantics or implementation of one software object depends on another software object. An example of this kind of relation is the dependency between Ada packages represented by Ada "with" statements. The "uses" relations between code modules are part of the module decomposition of a software system. These relations may be either defined directly in the components themselves via compiler directives or programming language constructs (e.g., "#include" in C, "with" in Ada, "COPY" in some Cobol dialects) or may be contained in externally specified attributes representing additional information used by the software tools (e.g., library specifications in linking commands). In both cases the relation is defined explicitly, and is not changed often in the evolution process compared to the properties of the individual components. The "uses" relation for implementation modules can be derived automatically from the source code and the external attributes for most programming languages.

In addition to recording dependencies between source code modules, the "uses" relation can include dependencies involving other types of software objects. For example, a clear box test case for a module "uses" the source code of the module, the source code of a module "uses" the behavioral specification for the module as well as the behavioral specifications and the concrete interfaces of the other modules it invokes, the behavioral specification of a module "uses" definitions of properties of the environment from the requirements model, a specification for a user function "uses" the requirements satisfied by the user function, and lower-level requirements "use" the higher-level goals they achieve. Similar relations expressing dependencies not directly related to the source code are that a user manual entry for a user function "uses" the specifications for the user function, and that a black box test case for a code module "uses" the behavioral specifications for the module. These relations are illustrated in Fig. 4. The relationship "*a* uses *b*" is denoted by an arrow directed from *a* to *b*. It is worth noting that in the event a composite module is decomposed into submodules, the implementation of the composite module uses the its own specification and the specifications of the submodules, *but not the implementations of the submodules*. This limits the impact of evolution steps

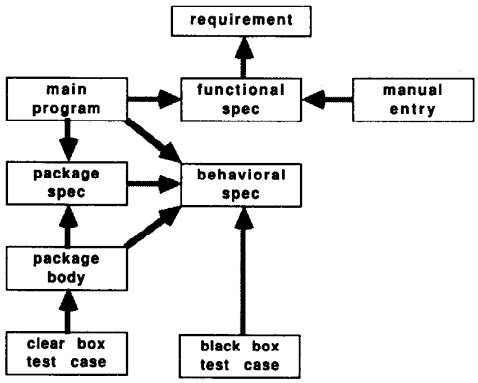


Fig. 4. The uses relation.

and provides in incentive for using formal specifications in software evolution. The uses relation can serve as the basis for automatically identifying inputs of proposed evolution steps and the identifying induced steps triggered by a proposed step, as indicated below.

## III. Model of Software Evolution

The main objective of our model of software evolution is to provide a framework that integrates software evolution activities with configuration control. The model is not concerned with the mechanics and the details of the tasks carried out by the software engineers and evolution programmers. The model is a refinement of some recent work [11] based on a set of organizational paradigms consistent with the ANSI/IEEE standard on Software Configuration Management [1], which are summarized as follows:

1) The management of the software evolution organization exercises a formal type of change control, so that the system configuration changes only as a result of an evolution action authorized by the management.

2) A software configuration management system is used as a tool to coordinate evolution activities for a system.

3) All of the verified software objects are contained in a controlled software library (the configuration repository) and all changes to components of the configuration repository must be authorized by the management.

4) The actual programming work is done using the programmer's workspace, which is outside the configuration repository. When a programmer is assigned to perform an evolution activity, appropriate software objects are copied from configuration repository to the programmer's workspace, where the programmer has free access to them. Final results of the activity are transferred from the programmer's workspace to the configuration repository when the work has been tested, verified and accepted.

5) The deliverable products of the configuration (e.g., user manuals and executable software objects) are derived from the system's configuration repository and installed at the "production" site, which is outside the configuration repository. These software products are the "exports" of the configuration.

6) Since product derivation may be required at any point of time, the system's configuration must be consistent at all times, i.e., the derivation of deliverable objects may not be compromised at any time because of consistency problems in existing software objects.

Such organizational paradigms are common to most software development and evolution organizations that deal with medium and large sized software systems.

### A. Definition of the Model

The model of software evolution is composed of two basic elements: system components and evolution steps. We refer to these as components and steps.

Components are versions of nonrederivable software objects: they are immutable, and correspond to the components in the model of software manufacture with the exception that the components must have concrete existence, since they cannot be automatically reconstructed on demand.

The evolution steps correspond to manufacturing steps of the model of software manufacture with the following differences.

1) A top-level evolution step is a representation of an organizational activity concerned with initiation, analysis and implementation of one request for a change in the system.

2) An evolution step may be either atomic or composite.

3) An atomic evolution step produces at most one new version of a system component.

4) The inputs and outputs of a composite step correspond to the inputs and outputs of its substeps.

5) The model of software evolution allows empty steps that do not produce any output components.

6) The model allows steps that do not lead to production of exported components. Such steps represent design alternatives that were explored but not incorporated into any configuration in the repository.

7) Automatic transformations are not considered to be evolution steps and are not represented in the model.

8) The model covers multiple systems which can share components, alternative variations for a single system, and a series of configurations representing the evolution history of each alternative variation of a system.

9) A scope is associated with each evolution step which identifies the set of systems and variations to be affected by the step.

The evolution history is an acyclic bipartite graph $G$ with a global labeling function $L$, as defined in Section II-B. We interpret $C$ nodes as system components and $S$ nodes as evolution steps. The output edges $O$ relate an evolution step to the components it produces. The input edges $I$ relate a step to the set of system components which must be examined to produce output components that are consistent with the rest of the system. Cycles are not allowed in the graph $G$, so that sets of software objects with circular dependencies must be packaged as single atomic components in the repository. For example, a set of mutually recursive subprograms must be packaged as an atomic component, as must the data declaration and the operations comprising an abstract data type.

Every configuration in the repository consists of an initial subgraph of $G$, a set of exported components $E$, and the global labeling function $L$. The evolution history graph represents a snapshot of the evolution history at some point in time. New versions of this graph may be created only by consistent extension, i.e., all evolution history graphs representing past states must be initial subgraphs of the current evolution history graph, and must be subject to the same global labeling function.

We formalize some of the above principles and definitions. The set of input components and the set of output components of an evolution step are defined in terms of the arcs in the evolution history graph.

$$\text{input}(s{:}S) = \{c{:}C \mid [c, s] \in I\},$$
$$\text{output}(s{:}S) = \{c{:}C \mid [s, c] \in O\} \quad (5)$$

The "part-of" relation for steps represents the relationship between a substep of a composite step and the composite step. This relation defines a tree-structured decomposition for each top-level evolution step. Atomic steps are defined as follows.

$$\text{atomic}(s{:}S) = \neg\,\text{EXISTS}(s'{:}S :: s' \text{ part-of } s) \quad (6)$$

We also need to introduce a relationship that captures indirect dependencies between components. One component *affects* another if both components are identical or if the first component is involved in the derivation of the second component, expressed formally as follows.

$$\text{ALL}(c1\ c2{:}C :: c1 \text{ affects } c2 \Leftrightarrow c2 \text{ uses* } c1) \quad (7)$$

The relation "uses*" is the reflexive transitive closure of the "uses" relation introduced in Section II-C. The "affects" relation can be derived from the structure of the non-primary inputs of the evolution steps for all components except for the primitive components in the initial configuration. The "uses" relations among the primitive components must be specified when the initial configuration is defined.

The restriction on inputs and outputs of steps can now be stated as follows.

$$\text{ALL}(s{:}S :: \text{atomic}(s) \Rightarrow |\text{output}(s)| \leq 1) \quad (8)$$

Atomic steps produce at most one output.

$$\text{ALL}(s1\ s2{:}S,\ c{:}C ::$$
$$s1 \text{ part-of } s2\ \&\ c \in \text{output}(s1) \Rightarrow c \in \text{output}(s2)) \quad (9)$$

The output of a composite step includes all of the outputs of its substeps.

The inputs to both composite and atomic steps are restricted by their parent steps. The inputs to a composite step could be defined to consist of all the inputs of its substeps, but it is more useful to aggregate inputs using a kind of generalization based on the dependencies between components. This simplifies descriptions of composite

steps, and supports planning via estimates of the expected sets of inputs to the top-level steps before implementation begins. We achieve this via the following restriction.

$$\text{ALL}\big(s1\ s2 : S,\ c1 : C ::$$
$$s1 \text{ part-of } s2 \ \&\ c1 \in \text{inputs}(s1) \Rightarrow$$
$$\text{EXISTS}\big(c2 : C :: \big(c2 \in \text{input}(s2) \ \&\ c2 \text{ affects } c1\big) \text{ or}$$
$$\big(c2 \in \text{output}(s1) \ \&\ c2 \text{ uses } c1\big)\big)\big)$$

Every input to a substep must either be affected by some input to the parent step, or must affect some output of the substep. For example, if a high-level step changes the specification of a user function, then the substeps can take as input the specifications and code of the modules implementing the user function, since all of those components directly or indirectly "use" the specification of the user function. These indirect dependencies let the inputs to top-level steps be small sets of high-level components which are meaningful to managers, such as the set of requirements affected by the top-level step. If the substep corresponds to a major change that introduces subcomponents that were not used by the previous version, thus introducing some new dependencies, then the specifications of those additional subcomponents will also be inputs to the substep. This is an example of a situation where all of the inputs to the substeps cannot be anticipated in advance. This type of step need not introduce potentials for deadlocks, however, because there is no need to acquire new locks after a transition has started: either the additional subcomponents can be reused without any modifications, or they can be modified by branching off a new variation of the object in question, without affecting any of the other contexts where the object is used.

### B. States of Evolution Steps

To model the dynamics of the evolution process, we associate states with evolution steps. We define the following five states of a evolution step.

*1) Proposed:* In this state a proposed evolution step is analyzed to determine costs, benefits, and potential impact on the system. This includes identifying the software objects in the step's input set. In this state implementation of the change has not yet been approved.

*2) Approved:* In this state the implementation of the change has been approved, but has not yet been scheduled, and references to generic input objects are not yet bound to particular versions.

*3) Scheduled:* In this state the implementation has been scheduled, the people responsible for doing the work have been assigned, all inputs of the step have been bound to particular versions, and the work may be in progress. When the step is in this state unique identifiers have been assigned for its output components, but the corresponding components are not yet part of the configuration repository.

*4) Completed:* In this state the outputs of the step have been verified, integrated, and approved for release. When a top-level step reaches this state, all output versions associated with the step and all of its direct and indirect substeps are incorporated in the configuration repository. This the final state for all successfully completed steps.

*5) Abandoned:* In this state the step has been canceled before it is completed. The outputs of the step do not appear as components in the evolution history graph or in the configuration repository. All partial results of the step and the reasons why the step was abandoned are stored as attributes of the step for future reference. The "abandoned" state is the final state for all evolution steps that were not approved by the Software Configuration Control Board or were canceled by the management in the "approved" or "scheduled" states.

Each state corresponds to several phases of the evolution process as they are defined in [10], and corresponding substates can be defined for each of the above states in a detailed implementation of the model.

Transitions of an evolution step from one state to another correspond to explicit decisions made by the management of the evolution organization. By controlling the states of the evolution steps, the management exercises direct control over both the software evolution process and the system configuration. The possible transitions are illustrated in Fig. 5. Evolution steps in the "scheduled" state can be "rolled back" into the "approved" state. Such an action corresponds to a long term delay in a step, and releases the bindings of generic input objects to specific versions. Since this may result in the loss of some or all of the work invested in the step, due to changes in the input objects that may occur before the step returns to the "scheduled" state, decisions to take such transitions should be made with insight and great care. This disadvantage can be reduced by tools for automatically applying a given change to another version of the object. However, automatically combining the results of several steps is the subject of current research [2], [5], and completely automated tools for performing this task reliably have not yet become available for practical use.

### C. Constraints on State Transitions

Evolution steps have a tree structure described by the "part-of" relation. In order to ensure consistency in evolution histories containing both composite and atomic steps, we impose the following constraints on some state transitions of composite steps and their substeps:

1) When a step changes from the "proposed" to the "approved" state all of its substeps make the same transition automatically.

2) A step changes automatically from the "approved" state to the "scheduled" state if one of its substeps makes this transition.

3) When a step changes from the "scheduled" to the "approved" state all of its substeps make the same transition automatically.

4) A composite step changes automatically from the "scheduled" state to the "completed" state when all of its substeps have done so.

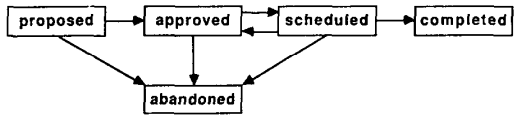


Fig. 5. State transitions for evolution steps.

5) A composite step changes automatically to the "abandoned" state when all of its substeps have done so.

6) When a step changes to the "abandoned" state all of its substeps make the same transition automatically.

7) When a new substep is created, it enters the same state as its parent superstep and inherits all version bindings associated with the parent step.

These rules help to ensure that inconsistent configurations are not entered into the repository and that the version bindings for a step are consistent with the version bindings of its substeps. They also reduce clerical effort by allowing management decisions to be explicitly recorded only for the largest applicable composite steps, with mechanical propagation down to the detailed substeps as appropriate.

### D. Input Classification for Atomic Evolution Steps

The purpose of an atomic evolution step is to incorporate a single change in a single component of the system. The result of the change is the single output of the atomic step. In order to capture dependencies between different software objects, we distinguish between primary inputs and nonprimary inputs of an atomic step. An input to an atomic step is *primary* if and only if it is a version of the same variation of the same object as the output of the step. Recall that alternative variations of an object represent parallel lines of development for the object which correspond to alternative design choices. The most common case, in which there is exactly one primary input, is illustrated in Fig. 6. We use the notation $c(a, b)$ to denote version $b$ of variation $a$ of object $c$. The primary input of the step is $c(v, n)$, which is the most recent version of the affected object variation $c(v)$ before the step, and the next to most recent version after the step. The version $c(v, n+1)$ represents the output from the step.

An atomic step without any primary inputs can arise in several situations. For example, an atomic step may create a new software object as part of a major change which affects the decomposition of the system. An atomic step can also create a new variation of an existing object in cases where the evolution of the object must split into two independent branches. Such a situation can arise in cases where a software object is shared between different systems, and an evolution step acting on one of these systems $S1$ has created a new version of an object which is not suitable for another system $S2$. This new version is therefore not incorporated in any configuration of $S2$. When a later step acting on $S2$ affects the same object, this change must be based on a version of the object that is not the most current one, thus creating a parallel branch in the development of the object, corresponding to a new alternative variation for the object. This is illustrated in Fig. 7. Primary inputs are shown as heavy arrows and nonprimary inputs are shown as thin arrows. Variations are represented as paths with heavy arrows. The versions $c(1, 1..n)$ are shared by systems $S1$ and $S2$, and all belong to variation 1 of the object $c$. Step $s1$ implements an enhancement to system $S1$ and produces a version $c(1, n+1)$ which is compatible with system $S1$ but not with system $S2$. Steps $s2$ and $s3$ introduce later chnages in the object $c$ for implementing enhancements to system $S2$. Step $s2$ creates the new variation with index 2, and cannot have a primary input because there are no versions belonging to variation 2 until after step $s2$ is completed. The later step $s3$ has a primary input $c(2, 1)$ which belongs to the same variation as the output version $c(2, 2)$.

(a) History of the affected object variation c(v) before the step: [c(v, 1), ... , c(v, n)]

(b) History of the affected object variation c(v) after the step: [c(v, 1), ... , c(v, n), c(v, n+1)]

Fig. 6. The effect of an atomic step.

We assume that an atomic step has at most one primary input, since it makes sense to include two different versions of an object as inputs to the same step only if those versions have different purposes, and in such a case the two versions should belong to two different alternative variations of the object. An atomic step which acts on several different variations of the affected object represents a change that combines the features of all the variations of a software object. Such a change can either be treated as an enhancement to one of the input variations, in which case there is one primary input corresponding to the existing variation associated with the output, or it can be treated as the creation of a completely new variation of the object, in which case there are no primary inputs.

We can formalize the concept of a primary input by introducing the attributes *object-id* and *variation-id*, both of which apply to versions, and yield unique identifiers for the object and variation associated with the version. Two versions belong to the same variation if they are versions of the same variation of the same object.

$$\begin{aligned}\text{ALL}(&c1\ c2 : C :: c1 \text{ same-variation } c2 \Leftrightarrow \\ &\text{object-id}(c1) = \text{object-id}(c2)\ \& \\ &\text{variation-id}(c1) = \text{variation-id}(c2))\end{aligned} \tag{11}$$

The property *primary-input* can then be defined as follows.

$$\begin{aligned}\text{ALL}(&s : S,\ c1 : C :: c1 \text{ primary-input } s \Leftrightarrow \\ &c1 \in \text{input}(s)\ \& \\ &\text{EXISTS}\ (c2 : C :: c2 \in \text{output}(s)\ \& \\ &\qquad c1 \text{ same-variation } c2))\end{aligned} \tag{12}$$

Some of the nonprimary inputs of an evolution step can be derived from the "uses" relation, since a step can depend on all of the components used by its primary input.

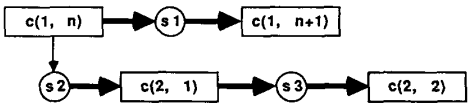

Fig. 7. Creation of a new variation.

This can be expressed formally as follows.

$$\text{ALL}(c1\ c2{:}C,\ s{:}S ::\ c1 \text{ uses } c2 \ \&\ c1 \text{ primary-input } s \Rightarrow c2 \in \text{input}(s)) \quad (13)$$

The set of nonprimary inputs to a step should ideally contain all of the component versions used by the *output* of the step. The above rule approximates this set by the set of component versions used by the primary input of the step, and is intended to define a mechanically derived initial approximation to the set of nonprimary inputs. This initial approximation may need some manual adjustment, since design changes associated with the evolution step can introduce dependencies that did not exist in the previous version, and can remove some dependencies that did exist.

### E. Specifying Inputs to Evolution Steps

Inputs to an evolution step can be specified by a reference to either a generic object or a specific version. Generic object references are usually the most common. Informally a generic object reference denotes the "current" version of the object.

Formally a generic object reference consists of an identifier for an object and an identifier for a variation of that object. Each variation of an object consists of a sequence of versions ordered by the dependency relation $D+$, or equivalently by the completion times of the versions. Generic object references for any step are bound to specific versions based on the scheduling of its top-level superstep, at the time the top-level step makes the transition from the "approved" state to the "scheduled" state. The top-level super-step top(s) is defined by the following properties:

$$s \text{ part-of* } \text{top}(s) \ \&\ \neg\ \text{EXISTS}(s'{:}S :: \text{top}(s) \text{ part-of } s') \quad (14)$$

where "part-of*" is the reflexive transitive closure of the irreflexive "part-of" relation. The top-level superstep is unique because a step cannot be "part-of" two different supersteps.

The inputs to a step can be specified by generic object references only while the step is in a "proposed" or "approved" state, and must be resolved to specific versions before the step can enter the "scheduled" or "completed" states. The version bindings of a composite step are inherited by its substeps to ensure consistency. Configurations in the repository are completely bound, in the sense that they do not contain any generic object references.

Specific object references are usually used to define inputs to steps in cases where the current version of an object has features that are not desirable for the proposed new configuration, and some earlier version of the object is acceptable. Specific object references often coincide with the creation of new variations, as discussed in Section III-D.

## IV. Evolution Consistency

An important practical problem in the evolution of a large system is ensuring the consistency of each new configuration. While the certification of semantic consistency involves several computationally undecidable problems in the general case, some related consistency criteria based on structural considerations can be maintained automatically with practical amounts of computation. Such support should extend the abilities of an organization responsible for the evolution of a software system to maintain control over the system. We propose to base such support on the concept of an *induced evolution step*.

### A. Induced Evolution Steps

A change in a component of a software system can require changes in other components to maintain the consistency of the system. We refer to those other changes as *induced evolution steps*. In this section we define some relationships that enable induced evolution steps to be identified mechanically. These relationships are based on structural considerations, and provide a conservative estimate of the impact of a change. A human designer must either examine the induced steps to determine if they need to produce new versions, or must define uniform policies similar to the "difference predicates" of [3] for filtering out some of the common cases where an induced step can be safely implemented by the identity transformation. Tools for automatically recognizing instances of upwards compatibility relationships [12] would also be useful for this purpose. The purpose of induced evolution steps is to alert the software engineers and the management to the impact of proposed changes and to prevent problems due to incomplete propagation of the consequences of a change. A change in one module can trigger a change in another, which can trigger further changes in a chain of indirect effects. The extent of such chains can be difficult to predict without computer assistance, especially for complex systems.

We call a step that originates such a chain an *inducing step*. The set of induced steps triggered by an inducing step updates the current versions of all components which are affected by the inducing step and are within the scope of the current top-level evolution step. There is need for concern about the scope because there may be multiple systems in the evolution history, which are distinct but may share components. We do not wish to create induced steps which implement unauthorized changes to systems that are not involved in the current top-level evolution step. The purpose of the induced steps is to produce versions of their primary inputs which are consistent with the output version of the inducing step.

A component is current if there is no later version of the same variation of the same object.

$$\text{ALL}(c1 : C :: \text{current}(c1) \Leftrightarrow \neg\ \text{EXISTS}(c2 : C :: c1\ D + c2\ \&\ c1 \text{ same-variation } c2)) \quad (15)$$

The scope of a top-level step consists of the components affected by its inputs.

$$\text{scope}(s : S) = \{c1 : C \mid \text{EXISTS}(c2 : C :: c2 \in \text{input}(\text{top}(s))\ \&\ c2 \text{ affects } c1)\} \quad (16)$$

This formulation assumes that the inputs to the top-level steps are the highest-level objects that are affected by the step, such as the system requirements affected by the change.

The set of induced steps can now be characterized as follows.

$$\text{induced-steps}(s1) = \{s2 : S \mid \text{EXISTS}(c1\ c2 : C :: c1 \text{ primary-input } s1\ \&\ c2 \text{ primary-input } s2\ \&\ c1 \text{ affects } c2\ \&\ \text{current}(c2)\ \&\ c2 \in \text{scope}(s1))\} \quad (17)$$

Since the inputs of a top-level step are bound to specific versions at the time the step is scheduled, the set of induced steps cannot be influenced by any changes due to parts of any other top-level steps that may be executed concurrently. The predicate "current" is evaluated in the state defined by the version bindings of the top-level step.

An example of induced steps in a small system implemented in Ada is shown in Fig. 8. The initial configuration of the system shown in the figure consists of the three components in the top row. The step $s1$ changes the main program without affecting the package specification, and does not trigger any induced steps. The step $s2$ changes the package body without affecting the package specification, and does not cause any induced steps because there are no other components that use the package body. The step $s3$ does change the package specification, and triggers induces steps $s3.1$ and $s3.2$, which must update the main program and the package body to conform to the new package specification. These induced steps can be derived from the "uses" relationships according to definitions (15)–(17). For all but the initial versions of the components, the "uses" relationships can be derived from the evolution history graph by reversing the directions of the nonprimary input relationships.

In realistic situations there can be longer chains of induced steps, corresponding to paths in the "uses" relation similar to those illustrated in Fig. 4. An example of indirectly induced steps is shown in Fig. 9. Step $s1$ triggers the induced step $s1.1$, which in turn triggers the indirectly induced step $s1.1.1$.

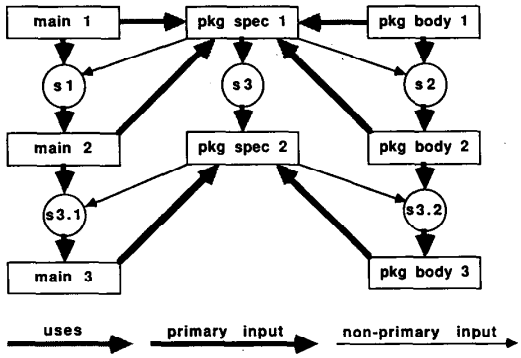


Fig. 8. Induced evolution steps.

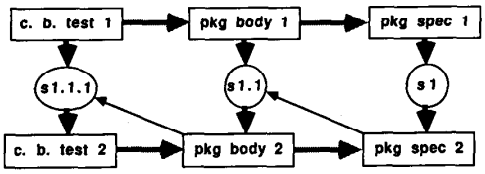


Fig. 9. Indirectly induced steps.

### B. Induced State Transitions

To maintain the consistency of the configuration, an inducing step and all of the induced steps must be carried out as an atomic operation. This means that all of the steps in such a set must make their transitions from the "approved" to the "scheduled" states and from the "scheduled" to the "completed" states without other intervening transitions. This can be accomplished by the following rules.

1) An inducing step can enter the "completed" state only if all its induced steps are completed.

2) An induced step enters the "scheduled" state automatically when its inducing step does so.

3) Any "roll back" transition of an inducing step from the "scheduled" state to the "approved" state causes the same transition to be performed on all its induced steps.

4) An induced step can be "rolled back" only by "rolling back" all of its inducing steps.

5) Abandoning an inducing step causes all of its induced steps to be abandoned.

6) An induced step can be abandoned only by abandoning its inducing step.

The first rule ensures that the effects of an inducing step are entered into the repository together with the effects of all the directly and indirectly induced steps. The second rule ensures that the version bindings of the induced steps are consistent with those of the inducing steps. The remaining rules deal with propagating the effects of rollbacks and canceled steps.

## V. Conclusion

A formal model of the process of software evolution is needed to serve as a basis for smarter software tools. This paper describes an initial version of such a model and in-

dicates how the model can be used to help maintain the consistency of an evolving system and to help organize and coordinate the activities involved in the evolution of large systems. The model can support aspects of software evolution that are not described in detail in this paper. Some areas for future applications of the model include tools for estimating the cost of proposed changes, and scheduling approved evolution steps. We have found that ideas similar to those underlying techniques for automatically managing versions of automatically rederivable software objects can be applied to nonrederivable source objects, if dependencies between objects are recorded and maintained. We have refined previous approaches to opportunistic construction of derived objects by introducing a link to management approval via the scope of an evolutionary step, as defined in equation (16). This prevents unauthorized and unintended changes to systems caused by propagation of changes to components shared by several systems.

Our model can also serve as the basis for organizing the repository of configurations. Our work has suggested that the configuration repository should contain representations of the steps as well as of the resulting software products. The minimal set of attributes associated with a step are the sets of inputs and outputs. This information is useful for reconstructing the "uses" relation, which is needed in determining the set of induced steps triggered by a proposed evolution step. Other attributes that might be useful include records of time and effort spent on the step, and records of the justifications for the decisions made and the alternatives that were considered and rejected.

Work is needed to address the additional problem of providing computer-aided explanations of the evolution history to support the decisions of the software engineers. This problem is a natural extension of the questions addressed in this paper. Decision support of this type is needed because the groups of people responsible for building a system and those responsible for evolving it are often disjoint, and tend to serve in their positions for short periods of time relative to the lifetime of the system. Thus the evolution history should serve as a "corporate memory," and be capable of supporting current decisions by supplying relevant information about decisions made in the past about the design of the system and past evaluations of alternative designs that were not adopted. Effective representations and analysis procedures for providing adequate decision support for the engineers responsible for system evolution are important areas for future research. We believe such representations can be developed as compatible extensions of the model described in this paper.

## References


[1] *IEEE Guide to Software Configuration Management*, American National Standards Inst./IEEE, New York, Standard 1042-1987, 1988.

[2] V. Berzins, "On merging software extensions," *Acta Inform.*, vol. 23, no. 6, pp. 607-619, Nov. 1986.

[3] E. Borison, "A model of software manufacture," in *Advanced Programming Environments*, R. Conradi, T. Didriksen, and D. Wanvik, Eds. New York: Springer-Verlag, 1986, pp. 197-220.

[4] D. Heimbigner and S. Krane, "A graph transform model for configuration management environments," in *Proc. ACM Software Eng. Notes/SIGPLAN Notices Software Engineering Symp. Practical-Software Development Environments*, 1988, pp. 216-225.

[5] S. Horowitz, J. Prins, and T. Reps, "Integrating non-interfering versions of programs," *Trans. Program. Lang. Syst.*, vol. 11, no. 3, pp. 345-387, July 1989.

[6] G. Kaiser, P. Feiler, and S. Popovich, "Intelligent assistance for software development and maintenance," *IEEE Software*, pp. 40-49, May 1988.

[7] G. Kaiser, "Modeling configurations as transactions," in *Proc. 2nd Int. Workshop Software Configuration Management*, IEEE, Princeton, NJ, Oct. 1989, pp. 129-132.

[8] M. Ketabchi and V. Berzins, "Generalization per category: Theory and Application," in *Proc. Int. Conf. Information Systems*, 1986; also Tech. Rep. 85-29, Dep. Comput. Sci., Univ. Minnesota.

[9] Luqi, "Software evolution via rapid prototyping," *Computer*, vol. 22, no. 5, pp. 13-25, May 1989.

[10] R. Martin and W. Osborne, *Guidance on Software Maintenance*, Nat. Bureau Standards, U.S. Dep. Commerce, Dec. 1983.

[11] I. Mostov, Luqi, and K. Hefner, "A graph model of software maintenance," Dep. Comput. Sci., Naval Postgraduate School, Tech. Rep. NP552-90-014, Aug. 1989.

[12] K. Narayanaswamy and W. Scacchi, "Maintaining configurations of evolving software systems," *IEEE Trans. Software Eng.*, vol. SE-13, no. 3, pp. 324-334, Mar. 1987.

[13] D. Perry, "The Inscape Environment," in *Proc. 11th Int. Conf. Software Engineering*, IEEE, 1989, pp. 2-12.

[14] J. Walpole, G. Blair, J. Malik, and J. Nichol, "A unifying model for consistent distributed software development environments," *Software Eng. Notes (Proc. ACM Software Engineering Symp. Practical Software Development Environments)*, vol. 13, no. 5, pp. 183-190, Nov. 1988.



**Luqi** received the B.S. degree from Jilin University, China, and the M.S. and Ph.D. degrees in computer science from the University of Minnesota.

She worked on software research, development, and maintenance for the Science Academy of China, the Computer Center at the University of Minnesota, and International Software Systems. She is currently an Associate Professor at the Naval Postgraduate School, Monterey, CA. Her research interests include rapid prototyping, real-time systems, and software development tools.