A Generic Model for Semantics-Based Versioning in Projects

S. Srinath, R. Ramakrishnan, and D. Janaki Ram, *Member, IEEE*

***Abstract*—Large projects generally involve a number of phases and evolve over a period of time. Several revisions of the individual artifacts consisting of the project take place during the various phases. These revisions and refinements are normally captured as different versions using configuration/version management tools. But the semantics of these refinements with respect to the project are not captured by existing mechanisms. In addition to the above, a change in the semantics of a project artifact may require suitable changes in other related artifacts. Existing mechanisms for configuration management do not provide mechanisms for change propagation based on the change semantics. In this paper, we propose a generic model for semantics-based version management in projects, which can be built over existing tools. The model also provides support for capturing how changes propagate in a project. We then elucidate the generality of the model by applying it to a project involving a computer aided design (CAD) framework and a software development project (SDP).**

***Index Terms*—CAD, change propagation, configuration management, equivalent, software development, version, version management.**

I. Introduction

LARGE projects are essentially based on an iterative paradigm where each iteration provides successive refinements over previous iterations. Refinements in such projects are managed by maintaining different configurations of the various artifacts of the project.

Support for version management is available from configuration management (CM) tools. CM techniques have been widely used in computer aided design (CAD) projects. CM mechanisms in CAD projects implemented using object databases are normally realized in terms of class and object versions. In software development projects (SDP's), CM was initially addressed in the form of maintaining different versions of an artifact [17], [22], but has expanded in scope to address other issues like change tracking, process control, workspace management, concurrent development, etc. [2], [7]. CM activities are also recorded in the form of software configuration management (SCM) processes consisting of procedures like identification of components and structures, control of changes and releases, status accounting, audit and review, etc. [8].

Manuscript received July 10, 1998; revised October 18, 1999. This work was supported by the Department of Science and Technology, Government of India, Grant CSE979803DST&DJAN. This paper was recommended by Associate Editor J. M. Tien.

The authors are with the Distributed and Object Systems Group, Department of Computer Science and Engineering, Indian Institute of Technology, Madras, Chennai 600 036, India (e-mail: janaki@lotus.iitm.ernet.in).

Publisher Item Identifier S 1083-4427(00)01734-3.

There are a number of similarities in the issues addressed by projects involving a CAD framework and an SDP. Such similarities have also been addressed in order to develop common frameworks for CM [5], [6].

However, most techniques for CM consider a project as only a set of artifacts of different kinds, whose configuration should be managed. Hence, they only address versioning of individual artifacts, such as files. The project itself is not considered as a semantic entity where versioning of each of the individual artifacts have different semantic meanings with respect to the project. In addition, changes in a project rarely occur in isolation. Various kinds of dependencies exist among the artifacts that make up a project. Changes in an artifact normally require corresponding changes in other dependent artifacts. For example, in a CAD project for designing airplanes, there might be a dependency between the design of the wing and that of the tailplane. Hence, a change in the design of one requires a corresponding change in the design of the other. Similarly, in a software project, a change in requirements may induce changes all along the project life cycle.

This paper addresses the above issues by providing mechanisms for semantic-based version management in projects. Semantics-based versioning have been addressed in CAD environments in the form of design versions [16] and user level versions [18]. In [16], different versions of a design artifact are maintained in the form of *design versions* and *design equivalents*. Design versions record semantic changes in the design of an artifact in terms of changes in the design attribute values. Design equivalents represent changes in the values of nondesign attributes which do not affect the semantics of the design. This paper extends the model to provide a more general framework for semantics-based versioning in projects by representing a project as a directed graph and introducing semantics-based versioning on the nodes of the graph. The model is generic and has been applied to both CAD and SDP.

The rest of the paper is organized as follows. Section II introduces a representation for projects which is used to build the generic model. Section III describes the proposed model. Section IV considers a CAD project and illustrates how the proposed model can be applied. Section V illustrates the application of the model to a SDP. Section V-D presents a set of tools which have been built to support the paradigm. Finally, Section VI concludes the paper.

II. Representation of a Project

We use the concept of URA [21] to represent projects. This section briefly introduces the URA concept to represent projects in a unified manner.

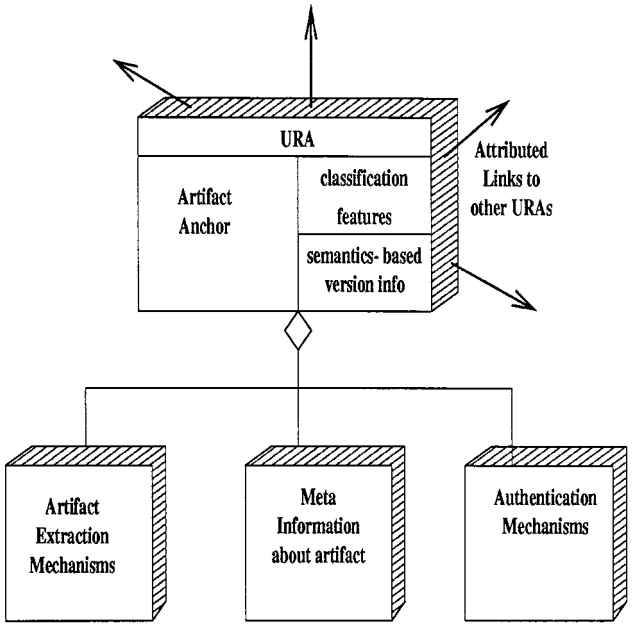


Fig. 1. Structure of a URA.

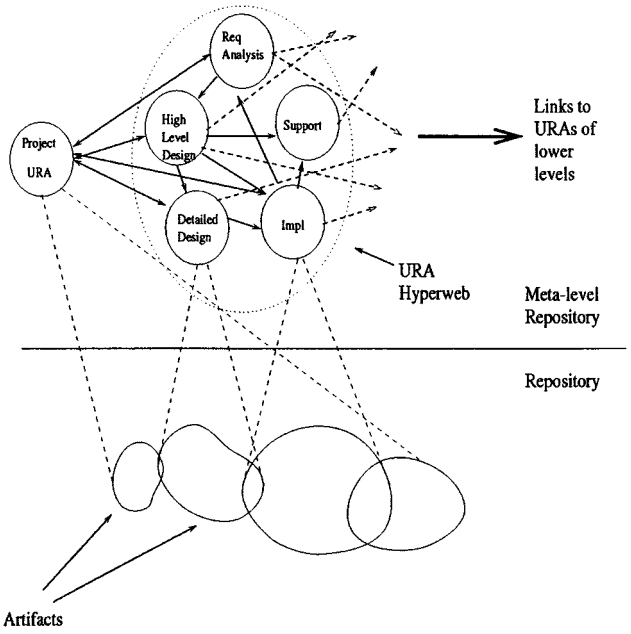


Fig. 2. Representation of a project using URA's.

Any large project may be viewed along different dimensions. People participating in the project have specific sets of roles in the project. Each role would normally have a set of *logical entities* associated with the project. For example, in a SDP, developers are concerned with entities like classes, objects, libraries, etc., while project managers are concerned with entities like schedule, cost, etc. We define an *artifact* as *any logical entity of interest*. Artifacts map to physical entities in different ways depending on the role played by the artifact. A project consists of artifacts of different kinds and granularities which are interrelated.

A meta-level entity called a *unified representation of an artifact (URA)*[1] is defined, which represents an artifact of any type or granularity. The semantics of a URA is shown in Fig. 1.

A URA consists of three main components. These are as follows.

- A component which can extract the artifact from the information system whenever required.
- A component containing context and other meta information about the artifact.
- A component enforcing authentication mechanisms to ensure the integrity of the extracted artifact.

In addition to these, there are links to other URA's which reflect the relationship between the artifacts which the two URA's represent. A set of "features" are associated with each URA which allows it to be classified and queried. A set of semantics-based version information enables a URA to keep track of the evolution of the artifact that it represents.

Since URA's contain links to other URA's, they can be connected in any fashion. Also, since URA's represent logical entities of any kind and granularity, a graph of URA's can be used to represent an entire project. Fig. 2 shows a URA graph for a SDP.

A formal definition for a URA and the project graph may be provided as follows. An artifact A in an information system, has a URA U_A associated with it. The semantics of U_A is defined as a 6-tuple $U_A = (S, C_a, C_m, C_t, L, F)$ where $S(U_A)$ is a tuple of the form (ver, eqv) containing version information about A. The next section explains the (ver, eqv) format of the version information.

$C_a(U_A), C_m(U_A), C_t(U_A)$ are called the *components* of U_A.

$C_a(U_A)$ is a set containing mechanisms for extracting A.

$C_m(U_A)$ is a set containing mechanisms for obtaining information about A.

$C_t(U_A)$ is a set containing mechanisms for validating the artifact obtained from any mechanism $m \in C_a(U_A)$.

$L(U_A)$ is a set of URA's $\{U_{a_1}, U_{a_2}, \cdots, U_{a_n}\}$ such that for any $U_{a_i} \in L(U_A)$ there is a relationship between A and a_i in the information system.

$F(U_A)$ set of the form $\{name_1 = value_1, name_2 = value_2, \cdots, name_k = value_k\}$ which indicates the features of A.

The *semantics* of an artifact A is defined as the set of all features of A which describe the functionality and interfaces of A. Based on this, the set of features $F(U_A)$ of an artifact may be partitioned into two subsets $F_v(U_A)$ and $F_e(U_A)$, where $F_v(U_A)$ would be the set representing the semantics of the artifact A. The set $F_e(U_A)$ denotes the set of auxiliary features which do not directly govern the functionality and interfaces of A.

A project is defined as the set of all activities and information generated from the time beginning from the conceivement of the product, to the time when the product is no longer in use. The

[1]In our earlier work [21], URA expanded to *unified reuse artifact*, since it was primarily meant for reuse. However, since then, URA's have been used for other purposes also and the present expansion has been adopted. The acronym URA still remains the same.

URA representation of a project is defined as a directed graph $G = (V, L, E)$, where

- V set of all URA's in the project;
- L set of all URA links of the form $(U_A, U_B, attr)$ $U_A, U_B \in V$, which represents artifact relationships;
- E set of all links of the form $(U_A, U_B, cohesive)$ $U_A, U_B \in V$, which represents links through which changes propagate.

Normally, the set of links for change propagation would be a subset of the links representing artifact relationships.

For a given link $l \in L$, the following functions are defined:

- $source(l) = U_A$ such that $l = (U_A, U_B, attr)$. source(l) depicts the source node of a link;
- $target(l) = U_B$ such that $l = (U_A, U_B, attr)$. target(l) depicts the target node of a link;
- $attr(l) = attr$ such that $l = (U_A, U_B, attr)$. attr(l) depicts the attribute of a link.

For a given link $e \in E$, the following are defined in a similar manner—$source(e)$, $target(e)$, and $cohesive(e)$.

A link $e \in E$ may either represent a "strong" or a "weak" relationship between artifacts. A strong relationship between two artifacts indicates that the semantics of the target artifact is dependent on the semantics of the source artifact. A strong relationship is indicated by setting the value of $cohesive(e)$ to **true** and a weak relationship is indicated by setting the value of $cohesive(e)$ to **false**.

During the life cycle of the project, artifacts of the project and the structure of the project graph undergo changes. If the project has to be considered as a semantic entity in itself, the versioning mechanism used by the project should reflect the impact of changes on the project graph.

The next section describes such a versioning mechanism based on the URA graph representation for projects.

III. A Generic Model for Semantics Based Version Management

The previous section introduced a paradigm that enables a project to be considered as a semantic entity (namely, a directed graph of URA's), when addressing CM needs. This section proposes a CM mechanism based on changes in the semantics of an artifact.

A directed graph represents the project, and the graph is said to evolve as changes occur in the project. The evolution of the graph is captured as a set of 12 *instances*. A description of the model is as follows:

$G = (V, L, E)$ is a directed graph, where the set V corresponds to the set of all nodes in the graph, and L is the set of all triplets of the form $(a, b, attr)$, $a, b \in V$, which denotes a link of type $attr$, from node a to node b. E is the set of all links of the form $(a, b, cohesive)$, $a, b \in V$, which denotes the links through which changes propagate.

G can be considered to have a set of *subgraphs*, $G_1, G_2, \cdots, G_k$. A nonempty subgraph represents a subsystem inside the project. The nodes in the subgraph represent the artifacts of the subsystem. For each subgraph G_i, a unique node pv_i is identified as the *pivot* node. The pivot node is the node which represents the artifact corresponding to the entire subsystem depicted by the subgraph. For example, if the subgraph corresponds to the graph consisting of all aspects of a fuselage design, the pivot node corresponds to the design of the entire fuselage. The pivot node is representative of the subgraph in the sense that, the subgraph is said to have changed into a new version when the pivot node changes into a new version. By the definition of a pivot node, it is imperative that the pivot node of a subgraph should be reachable by all nodes in the subgraph.

Each node in the graph has a set of features or attributes. The set of attributes for node *a* in the graph is denoted by $F(a)$. A link of the form $(a, b, attr) \in L$ is considered to be an attribute of node a. A set $F_v(a) \subseteq F(a)$ denotes the set of *versioning attributes* or the semantics of the node *a*. A set $F_e(a) \subset F(a)$ denotes the set of *nonversioning attributes* of node *a*. $F_v(a)$ and $F_e(a)$ partition the set of attributes of a, i.e., $F_v(a) \cup F_e(a) = F(a)$ and $F_v(a) \cap F_e(a) = \phi$.

The set of links E is partitioned by two sets C and N ($C, N \subset E$ and $C \cup N = E$ and $C \cap N = \phi$), such that $C = \{e | e \in E \text{ and } cohesive(e) = \textbf{true}\}$ and $N = \{e | e \in E \text{ and } cohesive(e) = \textbf{false}\}$.

A. Evolution

Evolutions in the graph are captured as a set of 12 instances. The instances are divided into different *classes*, categorized roughly based on the similarity of the changes occurring.

The instances of graph evolution are explained below. Fig. 3, shows a schematic representation of all the instances discussed.

1) Evolution of a Graph:

Instance 1: The evolution of a subgraph is characterized by the evolution of its pivot node.

Fig. 3(a) depicts two nodes A and B, interconnected by a cohesive link (link with a filled bubble). The attributes of A are **ar1** and **ar2**, and **ar1** is the versioning attribute. Similarly, **br1** is the versioning attribute of B. The term $A_{m,n}$ denotes the nth equivalent of the mth version of A, while A_m denotes the mth version of A. If the name of a node does not contain any version information, then it denotes the latest (or whatever is set to be the **head**) version of the node. Thus, if $A_{0,1}$ and $B_{3,2}$ are the latest versions of A and B, a link of the form $(A, B, attr)$ denotes a link between $A_{0,1}$ and $B_{3,2}$. The following holds for $A_{0,0}$:

$$F(A) = \{ar1, ar2\}$$
$$F_v(A) = \{ar1\}$$
$$(A_{0,0}, B_{0,0}, \textbf{true}) \in C.$$

If A is considered to be the pivot node for the subgraph containing the nodes A and B, the subgraph is said to change its semantics whenever the semantics of A change.

2) Versions and Equivalents:

Instance 2: A new *version* of node A is created when an attribute $a \in F_v(A)$ changes. A new version indicates a change in the semantics. This is represented formally as : $for\ any\ a \in F_v(A),\ a \mapsto a' \Rightarrow A_{m,n} \mapsto A_{m+1,0}$.

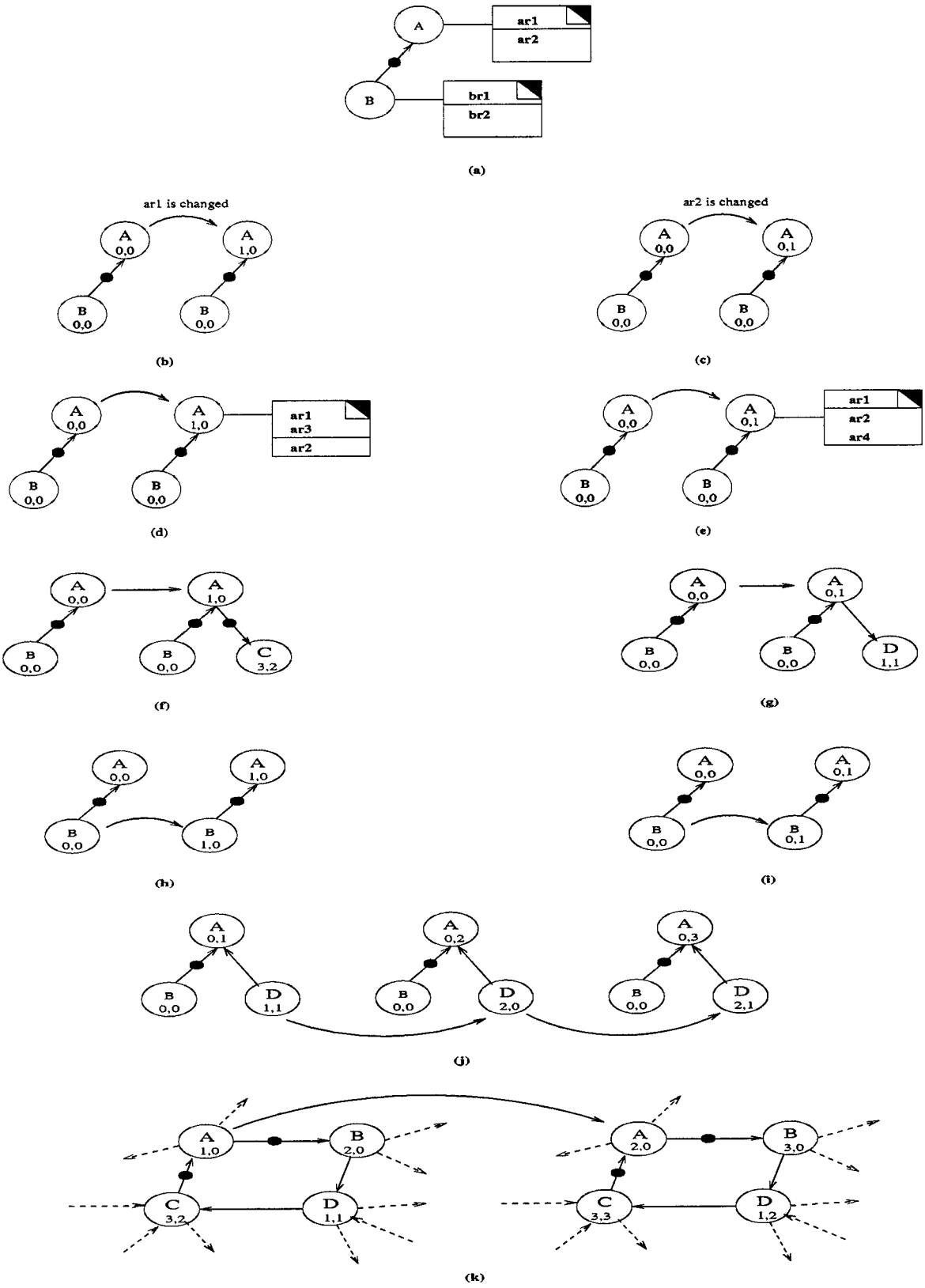


Fig. 3 Evolution of a URA graph. (a)–(i) Different instances of change propagation and corresponding graph evolution.

As shown in Fig. 3(b), when **ar1** $\in F_v(A)$ changes, a new version $A_{1,0}$ is created. The change may be in the value of **ar1** or in the type of **ar1**.

Instance 3: A new *equivalent* of node A is created when an attribute $a \in F_e(A)$ changes. This is represented as—$for\ any\ a \in F_e(A),\ a \mapsto a' \Rightarrow A_{m,n} \mapsto A_{m,n+1}$.

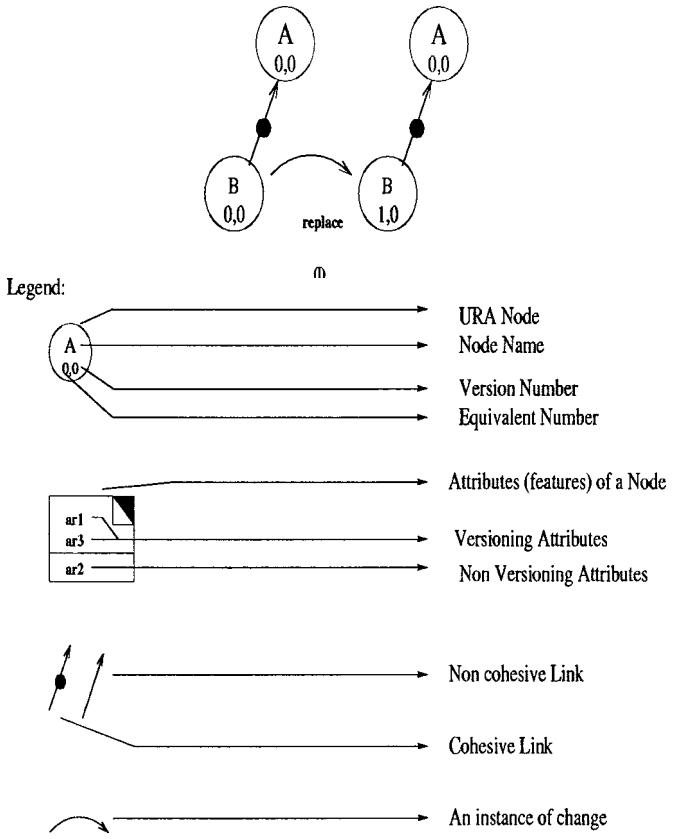


Fig. 3. *(Continued.)* Evolution of a URA graph. Different instances of change propagation and corresponding graph evolution.

As shown in Fig. 3(c), when **ar2** $\in F_e(A)$ changes, a new equivalent, namely $A_{0,1}$ is created.

Instance 4: A new *version* of node A is produced if $|F_v(A)|$ changes, where $|F_v(A)|$ denotes the cardinality of the set $F_v(A)$. This is represented as—$[(F_v(A) = F_v(A) \cup \{a\})\ or\ (for\ any\ a \in F_v(A), F_v(A) = F_v(A) - \{a\})] \Rightarrow A_{m,n} \mapsto A_{m+1,0}$. As shown in Fig. 3(d), when a new versioning attribute **ar3** is added to $A_{0,0}$ causing $|F_v(A)|$ to change, a new version $A_{1,0}$ is created.

Instance 5: A new equivalent of node A is created if $|F_e(A)|$ changes. This is represented as—$[(F_e(A) = F_e(A) \cup \{a\})\ or\ (for\ any\ a \in F_e(A), F_e(A) = F_e(A) - \{a\})] \Rightarrow A_{m,n} \mapsto A_{m,n+1}$. As shown in Fig. 3(e), when a new nonversioning attribute **ar4** is added to $A_{0,0}$, causing a change in $|F_e(A)|$, a new equivalent $A_{0,1}$ is created.

Instance 6: A new *version* of node A is created if a link of the form $(A, B, \textbf{true})$ is added or deleted from C. This is represented as—$[(C = C \cup \{(A, B, \textbf{true})\})\ or\ (for\ any\ (A, B, \textbf{true}) \in C, C = C - \{(A, B, \textbf{true})\})] \Rightarrow A_{m,n} \mapsto A_{m+1,0}$. As shown in Fig. 3(f), when a new cohesive link is added from $A_{0,0}$, a new version $A_{1,0}$ is created. Hence $A_{1,0}$ is now characterized by two cohesive links, one from $B_{0,0}$ and another to $C_{3,2}$.

Instance 7: A new *equivalent* of node A is created if a link of the form $(A, B, \textbf{false})$ is added or deleted from N. This is represented as—$[(N = N \cup \{(A, B, \textbf{false})\})\ or\ (for\ any\ (A, B, \textbf{false}) \in N, N = N - \{(A, B, \textbf{false})\})] \Rightarrow A_{m,n} \mapsto A_{m,n+1}$. As shown in Fig. 3(g), when a noncohesive link is added from $A_{0,0}$, a new equivalent namely $A_{0,1}$ is created.

3) Propagation of Versions and Equivalents:

Instance 8: A new *version* of node A is created if a new version of a node which connects A by a cohesive link is created. This is represented as—$[(B_{i,j} \mapsto B_{i+1,0})\ and\ ((B, A, \textbf{true}) \in C)] \Rightarrow A_{m,n} \mapsto A_{m+1,0}$. As shown in Fig. 3(h), when a new version of node $B_{0,0}$ is created, the change is propagated upward to result in a new version of A, namely $A_{1,0}$.

Instance 9: A new *equivalent* of node A is created if a new equivalent of a node which connects A by a cohesive link is created. This is represented as—$[(B_{i,j} \mapsto B_{i,j+1})\ and\ ((B, A, \textbf{true}) \in C)] \Rightarrow A_{m,n} \mapsto A_{m,n+1}$. As shown in Fig. 3(i), when a new equivalent of node $B_{0,1}$ is created, the change is propagated upward to result in a new equivalent of A, namely $A_{0,1}$.

Instance 10: A new *equivalent* of node A is created if a new version or equivalent of a node connecting node A by a noncohesive link, is created. This is represented as—$[(B_{i,j} \mapsto B_{i+1,0})\ and\ ((B, A, \textbf{false}) \in N)] \Rightarrow A_{m,n} \mapsto A_{m,n+1}$ $[(B_{i,j} \mapsto B_{i,j+1})\ and\ ((B, A, \textbf{false}) \in N)] \Rightarrow A_{m,n} \mapsto A_{m,n+1}$. As shown in Fig. 3(j), when a new version of $D_{1,1}$ is created, the change propagates to A resulting in a new equivalent $A_{0,1}$. Further, when a new equivalent $D_{2,1}$ is created, this will create yet another equivalent $A_{0,2}$. In both of the above cases only equivalents of A_0 are created since it is connected to $D_{1,1}$ by a noncohesive link.

4) Limiting Scope of Change Propagation:

Instance 11: When a node evolves, it moves to a *transient* state. The evolution is then propagated to other connected nodes. The propagation stops when it cannot proceed further. A node cannot evolve if it is already in a transient state. When the propagation stops, the evolved nodes move out of their *transient* states. Consider the configuration as shown in Fig. 3(k). When $A_{1,0}$ changes to a new version $A_{2,0}$, it enters a *transient state*. The change is propagated to B, creating a new version $B_{3,0}$ as the link is a cohesive link. Further propagation causes new equivalents of D and C. As there is further a cohesive link between $C_{3,3}$ and $A_{2,0}$, the change propagates through that link, but as $A_{2,0}$ is already in the transient state, further evolution of $A_{2,0}$ does not take place. This mechanism helps in preventing infinite propagation due to cycles in the graph.

Instance 12: If a node is *replaced* by another node, the change will not be propagated to other connected nodes. As shown in Fig. 3(l), when node $B_{0,0}$ evolves into $B_{1,0}$ in the *replace* mode, the original link between $A_{0,0}$ and $B_{0,0}$ is lost, and is now replaced by a link between $A_{0,0}$ and $B_{1,0}$. A replace mode of change is used to explicitly halt the propagation of change. This is applicable in cases

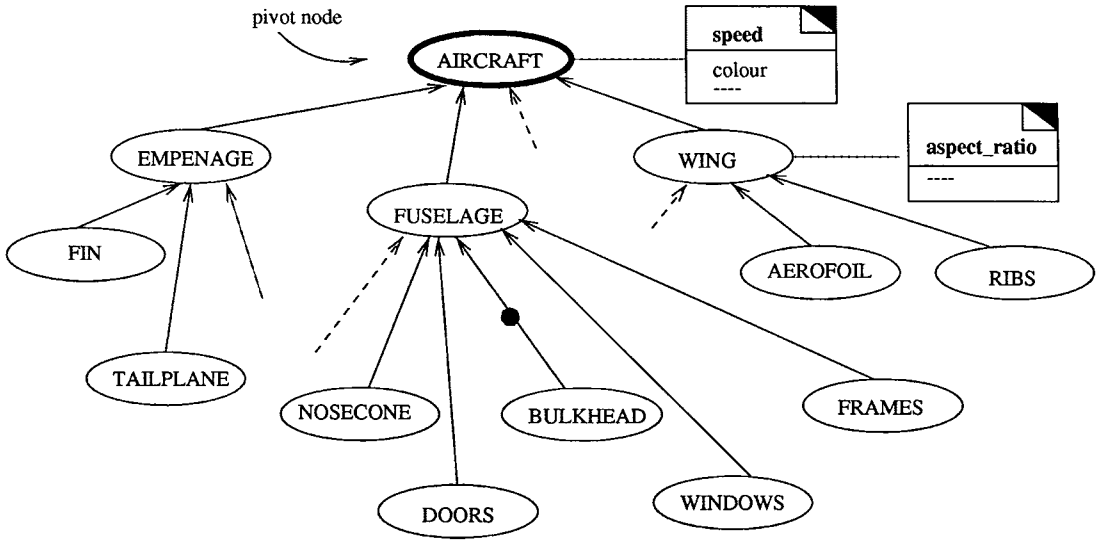


Fig. 4. Aircraft project.

where a defective version of a component in a product is replaced by a corrected version.

The next section applies the instances explained above to a project involving a CAD framework.

IV. Evolutions in a CAD Project

This section illustrates an application of the proposed model for projects involving CAD applications. CAD applications involve the management of evolutionary designs of numerous components of artifacts such as aircrafts, ships, and cars. Different versions of the numerous components have to be adequately managed. The versioning mechanism should be based on design semantics, i.e., it should create new versions based on whether a new design of the component is created or not. Necessary versions should not be missed and unnecessary versions should not be created. Creating a new version of an artifact due to some very minor changes is certainly not desirable. Presently, version control in CAD which involve object oriented databases are predominantly based on the concept of class versions and object versions [1], [3], [11]. However, these concepts do not take into account the design semantics. They do not distinguish between designs that are same and designs that are different. Further, the concepts of change propagation as discussed in [4], [10], [11], [19] are not based on design semantics. In this section, we illustrate how the proposed generic model achieves semantics-based versioning of the numerous design components. We first highlight the important characteristics of the design process in CAD, and then elucidate the applicability of the proposed model for design semantics-based version management.

A. CAD Design Process

This subsection briefly enumerates the characteristics of a typical design process in a CAD application.

- In the design department involved in designing a complex artifact, individual designers are assigned different components to design. For example, in the case of aircraft design, one designer designs the wing, another designer designs the tailplane, etc. The designers work collaboratively and design the artifact.
- The design process is inherently evolutionary in nature. Routine designing often involves retrieval of earlier designs and modifying them suitably so as to meet the new requirements.
- Typically, the design department is involved in creating an improved design of its product. The improvement may be in terms of better design or better performance or to suit different requirements. Hence, for any industry, there exist different product versions which use different versions of the low level components.
- The concept of *design* is critical for systems targeted at design applications. The *design* of an artifact characterizes the functionality and behavior of the artifact. A system that manages the different versions of an artifact should distinguish between designs that are same and designs that are different, i.e., it should be based on design semantics.

Fig. 4 shows a graph of URA's representing a project for aircraft design. Each component is represented by a URA, and it has links to its low level components, which are again represented by URA's. The aircraft is represented through a graph of URA's depicting the part-of hierarchy of the aircraft. In the next subsection, we illustrate how the concept of versions and equivalences achieve semantics-based version management.

B. Versions and Equivalences

The versioning attributes of an artifact are the key attributes that govern the design of the artifact. For example, consider the design of an elastically stressed element. The attributes of the artifact considered include Young's modulus, thermal conductivity, ductility, Poisson's ratio, color, etc. Of all these attributes, only a few of them govern the design. For example, Young's modulus governs the design of the artifact since it is elastically stressed. Any change in the Young's modulus can be consid-

ered to be a new design of the artifact. However, changes to the values of the nonkey attributes such as color, ductility, etc., will not create a new design of the artifact. Further, if it is a thermally stressed element under consideration, changes to Young's modulus do not affect the design, while changes to the thermal conductivity will affect the functionality of design.

Hence, it can be observed that

- depending on the design characteristics, only a few of the attributes of the component govern the functionality and hence the design of the component, and
- the set of attributes that affect the functionality of the design vary with the desired functionality and interfaces of the artifact.

Consider the fuselage of an aircraft (Fig. 4). It has lower level components such as bulkhead, frames, doors, windows, etc. Changes to the design of the door will not affect the design of the fuselage. However, changes to the design of the bulkhead changes the design of the fuselage (the bulkhead is designed for critical shear stress, and hence governs the design). This illustrates the fact that the design of a component is governed by the design of certain key low level components.

The proposed model addresses these issues through the concept of versioning attributes and cohesive links. A version is characterized by its versioning attributes and their values. Different versions of a component would differ in their design. Equivalents have the same set of versioning attributes and the same values for these attributes, and essentially represent the same design. For instance, in the case of elastically stressed element, Young's Modulus can be the versioning attribute. The link between the bulkhead and the fuselage can be a classified as a cohesive link, as a change in the design of the bulkhead changes the design of the fuselage. The design of the bulkhead, hence, governs the design of the fuselage. This is depicted through the filled circle link between the fuselage URA and the bulkhead URA (Fig. 4). It is thus seen that the concept of versions and equivalents is based on design semantics, and distinguishes between designs that are same and designs that are different.

C. Illustration

The previous subsection illustrated how the concept of versioning attributes and cohesive links achieve semantics-based version management. Versions differ in their design, while equivalents represent the same design. The critical factor is the versioning attributes, based on which version management is achieved. In this subsection, we illustrate how the instances that were described in Section III relate to the case of version management for CAD applications.

Instance 1: Consider the aircraft as depicted in Fig. 5. The root node aircraft is the pivot node. The versioning attributes of the aircraft include attributes such as speed, rate of climb, etc., while nonversioning attributes include attributes such as color, etc. Versions of aircrafts are distinguished based on their speeds, namely—subsonic, supersonic, etc. The evolution of the aircraft is hence characterized by the evolution of the pivot node.

Instances 2, 3: If the value of the attribute, speed, is changed from subsonic to supersonic, the design of the aircraft changes, and hence, a new version of the aircraft is created. However, if the value of the attribute color is changed, the design of the aircraft remains the same, and hence a new equivalent of the aircraft is created.

Instances 4, 5: Consider the case of the elastically stressed element. The attribute Young's modulus may be identified as the versioning attribute. However, if the element is also to withstand thermal stress, then, thermal conductivity also becomes a versioning attribute. This would create a new version of the stressed element as the design has changed from being designed to just withstand elastic stress to one which can withstand both elastic and thermal stress.

Instances 6, 7: If the lower level component, windows, is deleted from the fuselage, a new equivalent of the fuselage is created as the design of the windows does not affect the design of the fuselage.

Instances 8, 9, 10: A new version of the bulkhead causes a new version of the fuselage since the design of the fuselage changes when the design of the bulkhead changes. This is represented through the cohesive link between the bulkhead and the fuselage. Similarly, a new equivalent of the bulkhead would create a new equivalent of the fuselage. A new version or equivalent of the door will create only equivalences of the fuselage as the design of the door does not affect the design of the fuselage.

Fig. 5 depicts the evolution of the fuselage design. The node fuselage is the pivot node for the subgraph depicting the part-of hierarchy for the fuselage. Version 0 of the fuselage is characterized by a cohesive part-of link between bulkhead and fuselage, and a cohesive dependency link between frames and doors. This reflects the fact that the design of the doors used in that particular version of the fuselage was governed by the design of the frames. This is not the case in Version 1 of the fuselage, where there is no dependency between the doors and the frames. It can be seen that the version of the fuselage is governed by the version of the bulkhead and the cohesive links connected to the fuselage. Evolution of the graph occurs due to evolution of the structure of the graph, and evolution of nodes linked through cohesive links.

This section illustrated how the proposed generic model achieves design semantics-based version management for CAD applications. The next section illustrates how this model can be used for CM in a SDP.

V. Evolutions in a SDP

This section applies the generic model for semantics-based versioning, to a SDP. A software project, although similar in complexity to CAD projects, has some issues which are unique to it. Some of the major differences include the lack of standard design components and processes in software projects and the intangible nature of a software product. However similarities between the two environments have been pursued in order to benefit both CAD and SDP. For example, Dart [5] provides a comparison between software development and CAD environments with respect to CM.

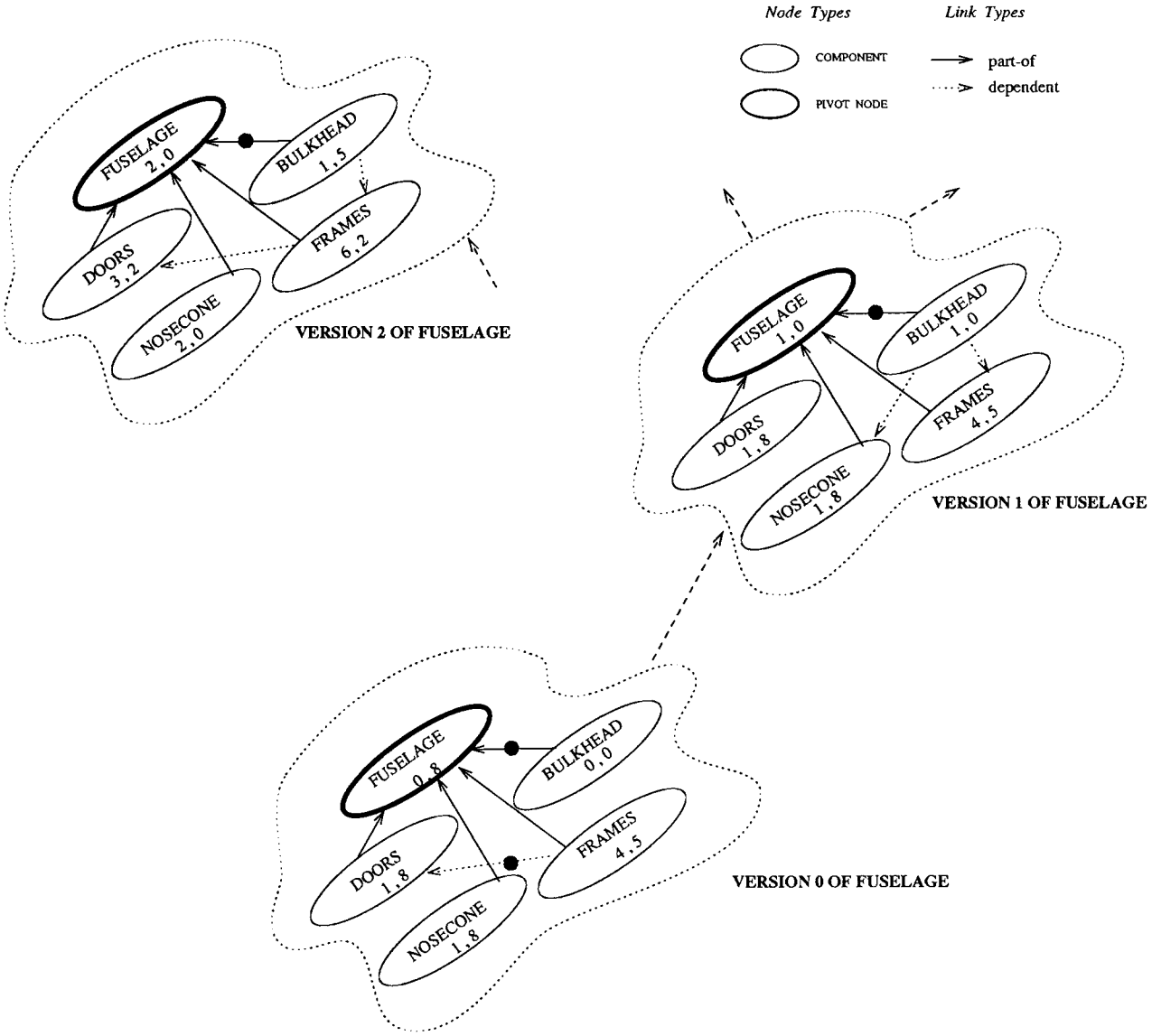


Fig. 5. Evolution of project graph.

A. Software Projects and Processes

In a general sense, software projects involve entities that are intangible. Due to this, the complexity of the project is hidden. This makes change management difficult as the impact of a change cannot be reliably determined without the knowledge of the complexity involved. In addition to the above, the intangible nature of software development makes it very difficult to accurately define development processes at the start of the project. Small granule process steps often tend to be *ad hoc*, and a sufficiently large number of such *ad hoc* changes can easily lead to a chaotic situation.

A main concern in software development is that of changing requirements. Software is expected to be flexible and easily adaptable to changing requirements. The emphasis on software development processes has hence shifted from a cascading model to an iterative model. The emphasis on iterative development has increased with component-based development and OO reusable frameworks being utilized for the development of large applications. Studies involving iterative models and OO reusable frameworks can be found in [14], [15], [23]. However, there is as yet no standard iterative model which unifies all kinds of project needs.

The following subsections apply the generic model for CM to a software project. We use the URA paradigm introduced earlier, to represent artifacts of a software project, and a directed graph of URA's to represent the project itself. Evolution of the project now essentially means changes in the semantics and structure of the directed graph.

B. An Example Project

This subsection considers an example project concerning the building of an operating system and constructs a partial graph representation for the project.

A typical process model is assumed, which is in the form of a cascading model containing the following set of phases:

- Market survey

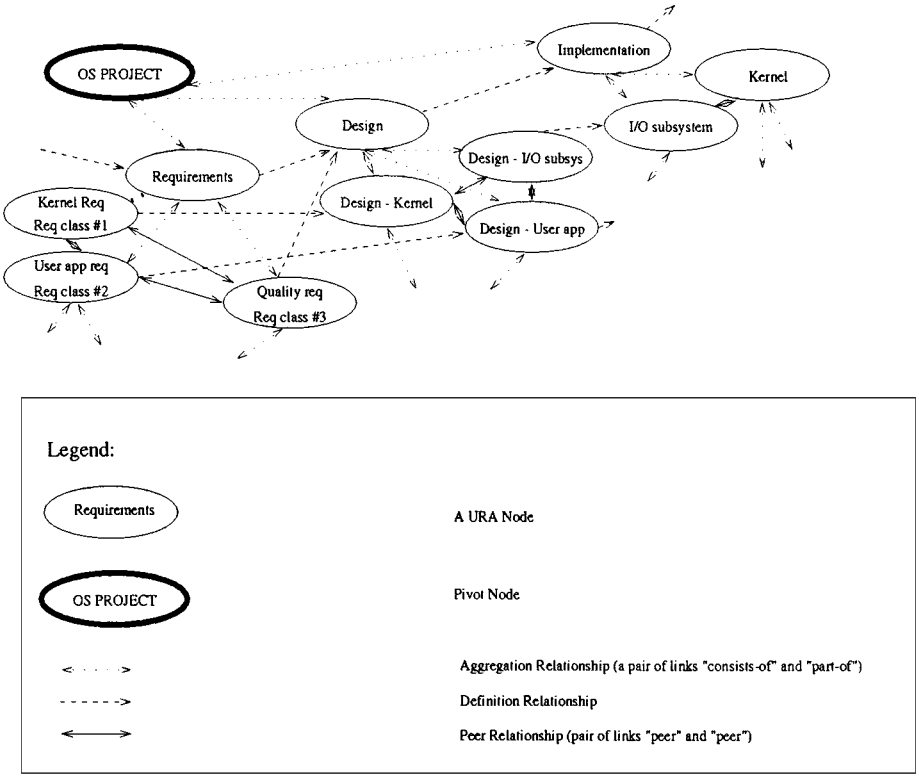


Fig. 6. OS project URA graph.

- Requirements gathering and analysis
- Design
- Implementation and integration
- Maintenance and enhancement
- Post mortem analysis

The process model considered above is provided only as an example. The URA representation may be utilized by other kinds of processes as well.

When the project starts, a URA is created to represent each of the the above phases. In addition, a solitary URA is also created to represent the entire project. This URA links each of the other URA's with a "consists-of" link; while the other URA's themselves link to the project URA with a "part-of" link. Initially the URA's created would be "stubs" to be filled later. As the project proceeds, appropriate URA's are filled.

We shall consider activities in each of the phases, and relate the kinds of URA's that go into building the project graph.

Market Survey: The market survey addresses the type of market for a proposed product, the competition scene, the depreciation rate, the rate of obsolescence, the prevailing standards, etc. The market survey is thus divided into different *classes* of survey, and URA's are created to represent each class. All the URA's thus created, are linked by the market survey URA, and each URA in turn links to the URA for market survey.

Requirements Gathering and Analysis: From the market survey, a rough requirements document for the kind of operating system desired, is drafted. This is represented by a URA which links to the URA for the requirements phase. From the rough draft, a finer draft is created, and the requirements aredivided into different requirement classes. Each class addresses one or more of the issues in market survey. For example, the URA for the prevailing standards in the market, indicates a large market for products conforming to a certain quality audit standard. Correspondingly, a requirements URA is created which enumerates the quality standards for different aspects of the project according to the conformance that was decided on. A link is then created from the URA for the prevailing market standards, to the URA for the quality conformance requirements.

Design: From the requirements gathering and analysis performed in the previous phase, a rough design is drafted, which shows the various subsystems, and their interaction. This is represented by a URA. URA's are then created for more detailed designs of each of the subsystems. Other "corollary" URA's are created under the design phase, which address the design process itself. For instance, a URA is created which describes the process of design review, and conformance checks. Appropriate links are created from previous phases to the design phase.

Implementation: URA's can be created at varying levels of abstraction for the implementation. Each relate to implementation and reuse relevant to that particular abstraction level. For example, reusable methods, classes and subsystems can all be represented and retrieved in a uniform way. URA's created in this phase should relate to URA's created in earlier phases. We see that integration becomes part of

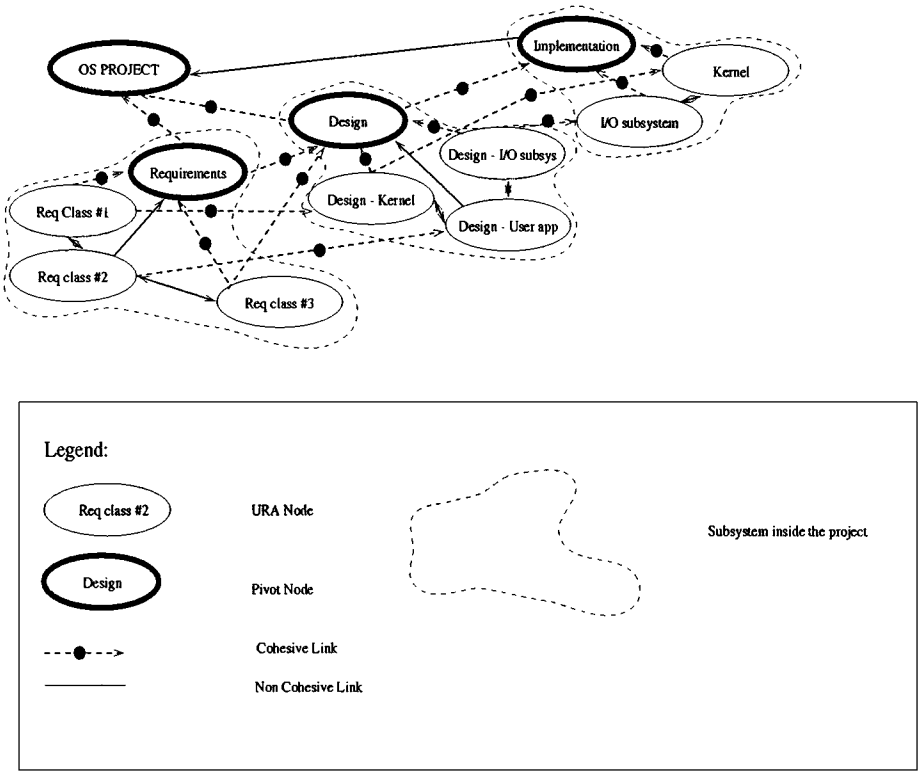


Fig. 7. Evolution propagation in the OS project.

the implementation phase, since an implementation phase concerns itself with varying levels of abstraction.

Maintenance: The maintenance phase consists of providing support to the released software, and implement enhancements on it. The maintenance phase predominantly involves changes to the project graph which has been built. Dependencies, changes, etc., are appropriately reflected back onto the graph. This phase would be the phase in which semantics-based versioning and change propagation would be most useful. Its importance is especially apparent when maintenance and enhancements have to be performed by a different set of developers who have not participated in the design and development of the software. In this phase, each bugfix request may be represented by a URA which is integrated into the project graph. A constraint may be placed that a semantic change may only be initiated by such a URA. This helps in recording the source of every semantic change.

Post Mortem Analysis: The post mortem analysis is performed after the product has completed its life cycle, and has been replenished from the market. The project graph, along with all the changes to it that have been stored in the CM, forms the starting point of this phase. The project graph records not only the product that was released to the market, but also the analysis and design decisions that were made to release such a product. The performance of these can now be analyzed to get better insights for forthcoming projects.

Fig. 6 shows a schematic diagram of a part of the graph that was built to represent software projects.

C. Semantics-Based Versioning

Here we illustrate the process of refinements that take place as the project that was explained in the previous section undergoes iterations and reworks due to changing requirements.

Fig. 6 shows a graph of URA's representing the project. The root node of the graph is the pivot node for the entire graph; its artifact part is the entire project. Similarly, URA's are created which depict each phase of the process, and other artifacts of varying levels of granularity.

As the project evolves, the components of the existing URA's (including their links with other URA's) undergo change, and URA's are added and deleted from the graph. Such nonmonotonic changes to the URA graph are typical of iterative projects.

Fig. 7 shows the graph constructed depicting links through which evolution information propagates. Cohesive links are depicted by a filled circle as part of the link. The links by which evolution propagates may or may not be related to the links depicting URA relationships.

As shown in the figure, the project graph is divided into many subgraphs, with each subgraph having its own pivot node. The pivot node for a subgraph should be the URA which represents the artifact depicted by the entire subgraph. For example, the "design" URA forms the pivot node for the entire subgraph of URA's concerned with the design phase of the project. The root node of the project graph forms the pivot node for the entire project.

We consider all the instances of graph evolution that were described in conjunction with the abstract model and apply them to the above software project.

Instance 1: The version and equivalent number of the entire project is available in the root node of the project graph. Similarly, the requirements URA provides information about the version and equivalence of the requirements gathering phase.

Instances 2, 3: Consider the URA for kernel design. The memory management module and the process management modules are considered as versioning attributes. The module for user authentication is considered as a nonversioning attribute. A change in the design of either the memory management module or the process management module will create a new *version* of the kernel design, while a change in the design of user authentication module creates a new *equivalent* of the kernel design.

Instance 4: The user authentication module is now changed to be a versioning attribute. This creates a new version of the kernel design.

Instance 5: The design for a module for providing networking support is added as a nonversioning attribute. This creates a new equivalent kernel design.

Instance 6: The cohesive link from requirements class 2 to the URA containing the design of user applications is now removed. This creates a new version of the URA containing requirements class 2.

Instance 7: A noncohesive link is added from the URA containing requirements class 1, to the URA containing designs of I/O subsystems. This forms a new equivalent of the "requirements class 1," URA.

Instance 8: Since the URA containing kernel design is linked to the URA for the overall design, by a cohesive link, a new version of the kernel creates a new version of the overall design.

Instance 9: Similarly, a new equivalent of the kernel design creates a new equivalent of the overall design.

Instance 10: A new equivalent of the overall design is created whenever a new version or equivalent of the user application design is created. This is because the user application URA connects the overall design URA with a noncohesive link.

Instance 11: Consider the URA pair consisting of a URA for the I/O subsystem implementation and a URA for kernel implementation. They have noncohesive links in both directions among themselves. When the I/O subsystem implementation moves into a new equivalent state, the evolution is also propagated to the kernel implementation which moves into a new equivalent state itself. But this change is not propagated back to the I/O subsystem URA, since the URA's are now in a transient state. Only when propagation stops, will the new version/equivalence be committed.

Instance 12: Consider the case where a new design of the operating system is created due to the creation of a new version of the kernel. Now, if the design URA decides to *replace* the older version, then further propagation will not take place from this point. Suppose the project were in Version 2 equivalence 5, and the design in Version 5 and equivalence 0, then even though the design changes to version 6 equivalence 0, the project still has Version 2 and equivalence 5.

A process model based on URA's is provided in [21]. However, a graph representation of a project can be used to represent any project regardless of the process employed. In an iterative process, the project is started with rough design or a prototype, over which iterations are performed in order to refine the output. Correspondingly, when the project graph is used in an iterative project, the project is started with only the main URA's (the project URA, URA's for each of the process stages, etc.). More URA's are added as the project grows along, and the evolution is propagated along the graph as per the semantics discussed.

D. URA Tools as a Case Study

In this section we illustrate our experiences with changes, constraints and inconsistencies while building tools for the URA paradigm. A prototype toolset for building, retrieving and querying URA's had already been built. We illustrate the changes that occurred in the URA graph for these tools when we tried to incorporate versioning, change propagation and constraints. This section is divided into three parts. The first part introduces the tools that were built earlier to support the URA paradigm. The second part introduces the tools which support change, constraint and inconsistency propagation on a URA graph. The third part introduces the URA graph for these tools and tracks some of the changes resulting from the incorporation of the new features. It also illustrates some of the implicit constraints that became explicit only when the tools were in operation. In this paper we have proposed a generic model where the links have binary values (viz 0 or 1). But in cases where it cannot be determined exactly whether the link is cohesive or not, the links may have values ranging from 0 to 1 as has been in the case study.

1) URA Tools: One of the main issues that has to be addressed by URA tools is that of representing different kinds of logical entities or *artifacts*, in a uniform fashion. This involves being able to extract the artifact from the information system whenever requested, and integrating it into the user's environment.

Since it is not possible for a single extraction and integration mechanism to cater to all kinds of artifacts, the architecture of the URA environment is based on pluggable "support" components. This is shown schematically in Fig. 8.

The URA toolset is broadly divided into two categories—"server-side" tools and "client-side" tools. The "server-side" tools are available on all machines which contain URA's. They maintain a local URA database and cater to requests from client-side tools. The "client-side" tools operate from the user's side. They either interact with the user directly (ex. for building and browsing URA's), or interact with server side tools (ex. to integrate an artifact into the user's environment). Each URA that is stored in a server database is in the form of an object which contains "holes". These "holes" can be used to plug in a set of "support components". There are two types of support components—"client-side support components" and "server-side support components". Each server-side support component provides extraction mechanisms for a given class of artifacts. A client-side support component provides mechanisms to integrate different kinds of artifacts into the user's environment.

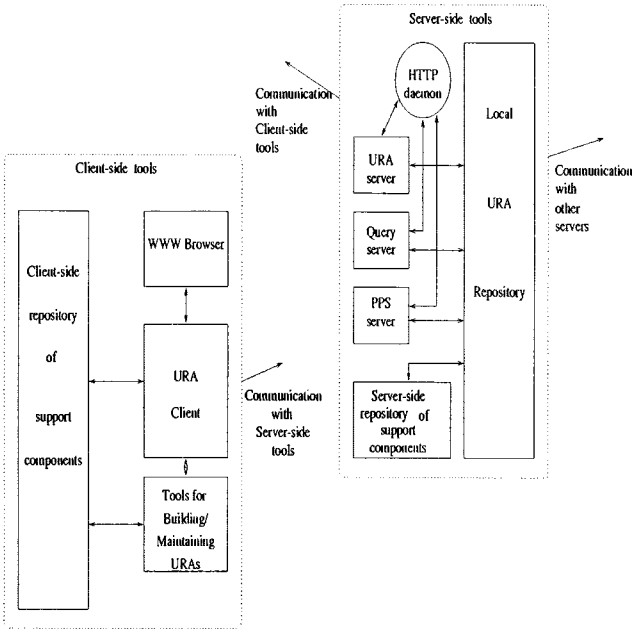


Fig. 8. Architecture for URA tools.

For example, if the artifact in question is a class contained in an archived library, its server-side support component enables the user to extract the given class from the library. Correspondingly, its client-side support component provides means by which the class can be archived into the local library. Hence a set containing a server-side component and its corresponding client-side component caters to a specific class of artifacts. The scope of the kinds of artifacts that may be represented by URA's may be increased by building more support components.

The server-side tools are implemented as common gateway interface (CGI) scripts for the hypertext transfer protocol (http). The user may browse through URA's using any WWW browser. However the present design is directed toward using the editor XEmacs as an integrated client side environment. The user browses through URA's using the elisp package **w3**, which can be integrated into an existing XEmacs editor.

Each URA is present as a pair of files. A *uraname*.html file forms the anchor for the URA. This is returned to the WWW browser whenever it requests access to a URA. A *uraname*.ura file consists of a description of each URA component. Every support component that the URA uses is specified in this file. Whenever the user requests access to any URA component, the server-side tool reads this file and responds accordingly. The description language used by the *uraname*.ura file is called *URA markup language* (URAML).

Fig. 8 shows a set of three server-side tools. The "URA-server" is a tool that is used to access URA's and their components. The "query-server" is a tool that matches client queries to URA's which are present locally. The "PPS-server" is a tool which provides services like positioning and schema visualization.

2) Tools for Change Propagation: Tools to handle change, constraint and inconsistency propagation have been built on top of the URA tools. One of the main issues in designing tools for change propagation is to address the distributed nature of a URA repository.

URA's are present on different machines, each catered by a set of server-side tools. Propagating changes among URA's which are managed by the same server is straight forward; however propagating changes across servers involves a number of overheads. Hence the design of the change propagation tools is directed toward minimizing the number of inter-server propagation.

In addition to the above, a version change recommendation has a greater precedence over an equivalent change recommendation. Hence version change recommendations have to be addressed before equivalent change recommendations in order to prevent revisiting of nodes.

The overall design semantics of the change propagation tools which address the above issues, is as follows.

- Propagate version recommendations in the present server in a breadth-first fashion.
- Propagate equivalent recommendations in the present server in a breadth-first fashion.
- Transfer control to the next server in a breadth-first fashion to continue with the propagation.

The design semantics is achieved by maintaining a data structure called *token* consisting of a system of queues and a hash table. The server which holds the token carries out the change propagation among the URA's managed by it. During the propagation, target nodes are placed in appropriate queues. When the propagation in the present node is complete, the server passes the token to the next server. The propagation continues in this fashion till all the queues in the token become empty.

Fig. 10 depicts the structure of the token. The token consists of a system of queues called *servQ* and a hash table. The servQ is a queue in which each element is in turn a pointer to a queue structure. Each server that is encountered during propagation is made an entry in the servQ. A entry for a server consists of two elements—a pointer to a version queue, and a pointer to an equivalent queue. Hence, assuming that the index begins with 0, every even numbered index in the servQ contains a pointer to a version queue of a specific server and each odd numbered index contains a pointer to an equivalent queue of a specific server.

The propagation begins by making an entry for the server which initiates the change propagation. The initiator node is placed in either the version or equivalent queue based on the type of change. The rest of the algorithm involves taking out the first element from the topmost queue and placing all the nodes connected by it on the appropriate queues. Before placing a node in a queue, a check is made with the hash table to ensure that the node is not being revisited. A hash table entry of "e" allows an entry of type "v" to proceed, while a hash table entry of "v" does not allow the node to be placed in the servQ.

When the version and equivalent queues of the present server have been exhausted, the token is sent to the next server in servQ. This server would now be at the top of the queue and the propagation proceeds as above. After all the queues become empty, the result would be available in the hash table which depicts all the nodes visited and the type of change recommended.

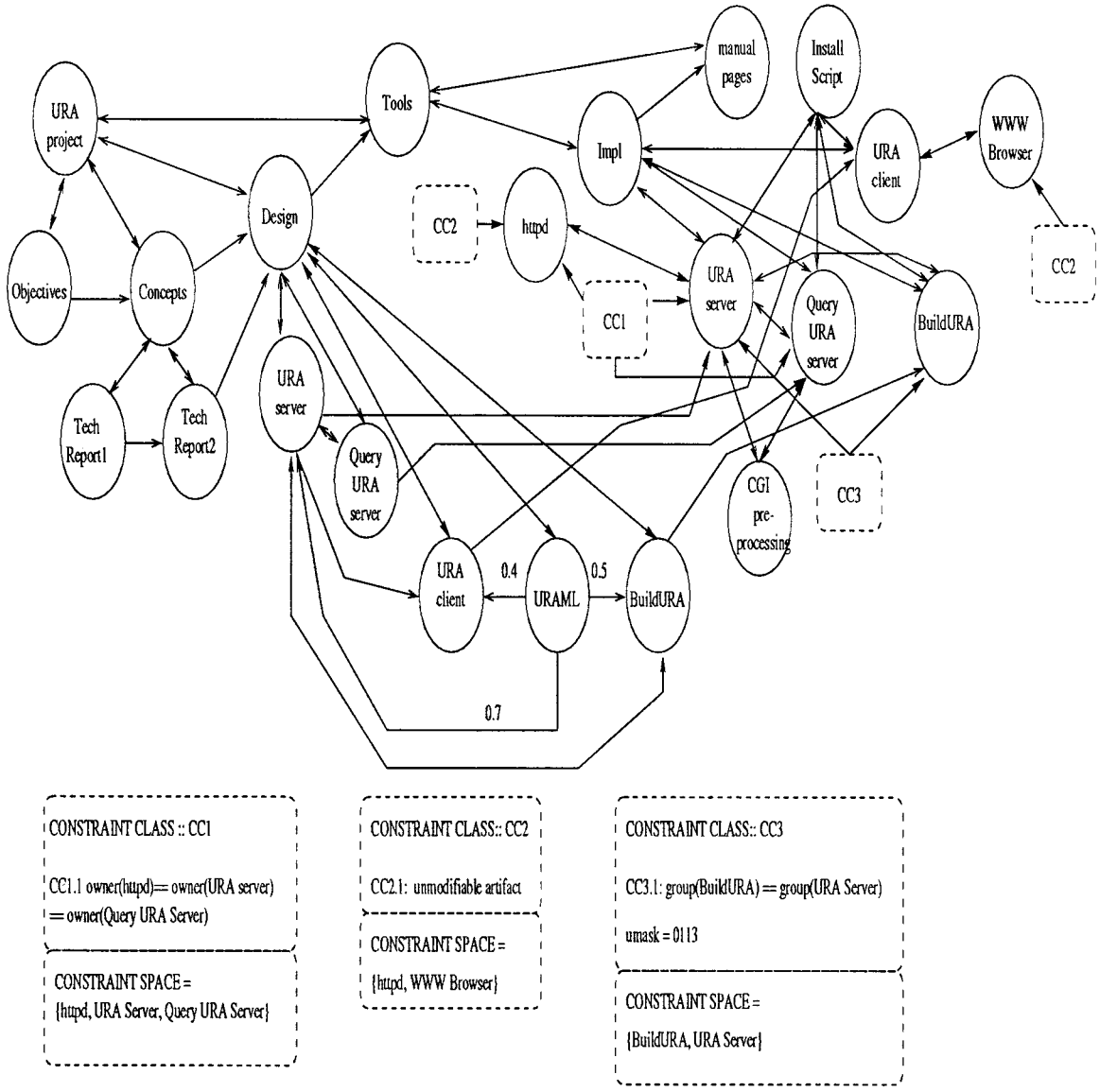


Fig. 9. Partial graph for URA tools.

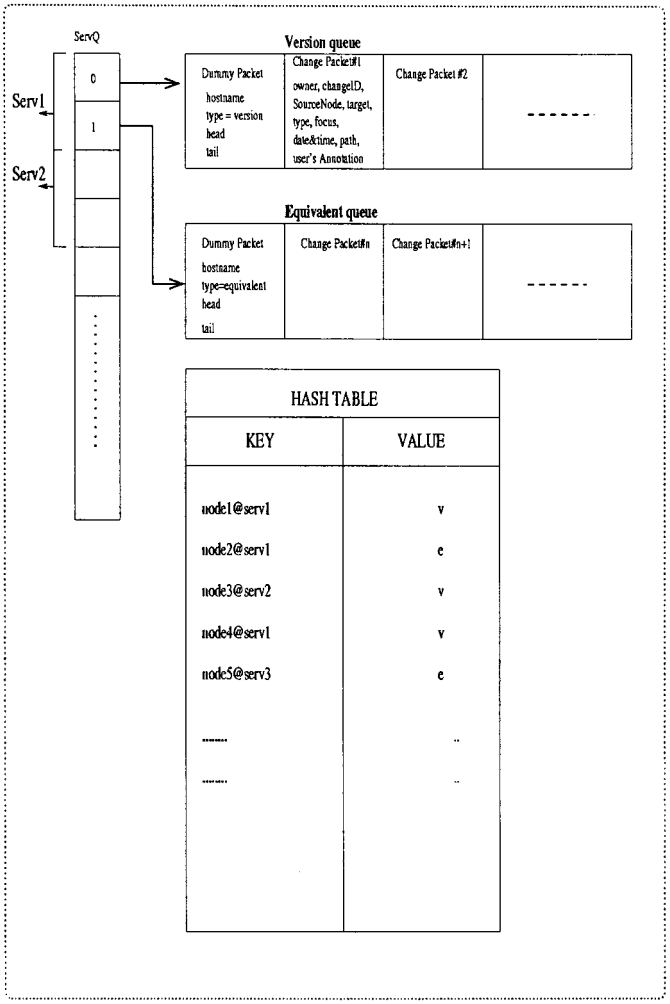


Fig. 10. Token structure.

The contents of the hash table are sent back to the initiator of the propagation. Intimation of change and result is done by sending mails to the owners of the appropriate URA's.

Fig. 11 shows a schematic description of the activities of the change propagation server.

3) Incorporating New Tools: Fig. 9 shows a partial URA graph for the tools over which change propagation tools were built. The figure also shows URA relationships and the "degree of cohesion" for some of the links.

Since the URAML did not incorporate version and equivalent information as part of its syntax, one of the main requirements for the incorporation of change propagation involved changes to the URAML syntax. As shown in Fig. 9, we can estimate the extent of change required from a change occurring in the URAML node.

The accuracy of estimates may be improved by judiciously assigning "degree of cohesion" for links. In the above graph, URAML contains three types of descriptors—component descriptor, link descriptor and feature descriptor. The BuildURA program needs to know the syntax of link and feature descriptors as part of its operations. In addition, it also needs to know the names of all the components described by URAML. The "degree of cohesion" between URAML and BuildURA can hence be calculated as—

$$cdegree = \frac{\Sigma r_i}{\Sigma w_i} \tag{1}$$

where r_i is the weightage given for the extent to which the ith descriptor is used by BuildURA and w_i is the weightage for the ith descriptor ($r_i \leq w_i$). For the URAML node, all descriptors were given the same weightage, namely 1. For calculating the $cdegree$ between URAML and BuildURA, component descriptors were given a weightage of 0.2 and link and feature descriptors were given a weightage of 1. This is because BuildURA needs to know the complete syntax for feature and link descriptors, while it only needs to know how to extract the names of each of the components from the component descriptors. A URAML template consists of three component descriptors, one link descriptor and one feature descriptor. Using this, the resultant value of $cdegree$ would be 0.52 which is approximated to 0.5.

By assigning values for other links, and tracing the change, we find that a total of 16 URA's get affected and version change is recommended for 8 URA's. This indeed provides a rough estimate, as verified by the large amount of changes that had to be made in URA-server, BuildURA, manual pages, etc.

Fig. 9 also shows a set of constraint nodes CC1, CC2 and CC3. These constraints became explicit during the course of the project. For example, constraint class CC1 imposes the constraint that the http daemon, URA-server and the query-server be owned by the same user. This constraint was detected when URA-server and query-server failed to run as CGI programs even though they executed correctly when tested individually. This is because, URA-server and query-server refer to initialization scripts for maintaining a local URA database. When they were invoked as CGI scripts, they searched for the initialization scripts in the home directory of the owner of httpd instead of searching in the home directory of the user who has installed the server side tools. This constraint resulted in a new version of the installation script to ensure that URA-server, query-server and httpd is owned by the same user.

Similarly, CC2 imposes the constraint that the artifacts in its constraint space are unmodifiable. The existence of this constraint was realized when an attempt was made to run URA-client in an interactive mode. The URA-client runs as a "helper" application for the WWW browser. This is the only way in which a WWW browser can call another program at the client site. Ideally, the browser itself can be used as the user interface for URA-client. However, this is not possible to implement because the WWW browser is a publicly available software and was not built as part of this project and hence cannot be modified. The only way to make URA client run in an interactive mode is to have it support its own GUI. Once this constraint became explicit, it was apparent that the same constraint also holds on httpd, since httpd was not developed as part of the project.

The third constraint CC3 was detected after a conflict arose when trying to incorporate tools for change propagation. Change packets are routed to URA's through URA-servers.

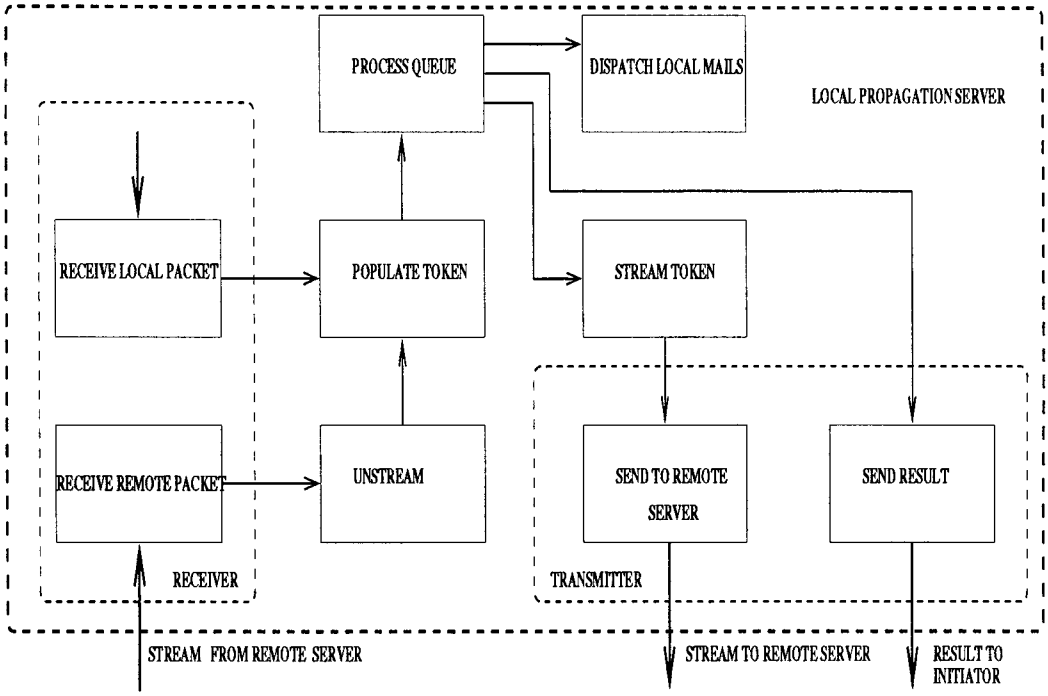


Fig. 11. Change propagation server.

When a URA-server receives a change packet it saves it in a local repository and sends a URAML tag to all the URA's which have to be affected. However, since each URA is owned by its respective owner it is not possible for URA-server to write the URAML tag into the URA files. This conflict is now resolved by placing a constraint that BuildURA, URA-server and all the URA's built should belong to a common group, and URA's should be group writable. This constraint resulted in a new version of BuildURA which managed the permissions of each URA built by it.[2]

An inconsistency resulting from constraint violations was detected due to a change in the installation script. Once constraint CC1 became apparent, the installation script was modified to accommodate this constraint. The changed installation script now determined the owner of httpd running on the system and changed the owner tag for all the CGI scripts that it installs. However if we trace this change from the node for installation scripts, back to the httpd node, we see that it causes a constraint violation. This is because if the owner of httpd is *root*—the system administrator in Unix systems, attempting to change the owner of a file to *root* may fail in some systems. It was hence decided to change the installation scripts to prompt the user to provide the owner of httpd. If a machine has a running httpd managed by *root*, the modified scheme now requires the user to run another httpd on a different port number.

E. Summary

In this section, we looked at how the proposed model for CM can be applied to software projects. One of the main challenges in dealing with software projects is that of hidden complexity. Often changes are sporadic and arbitrary, and if the project had poor analysis and design phases, a seemingly small change can overwhelm a major part of the project.

Representing the project as a directed graph enables us to track changes in the project across different levels of granularity. The proposed CM model provides a means for capturing the effect of changes on the project in the form of semantic versions.

A suite of tools which support the URA paradigm and semantics-based versioning has been designed and implemented to work in an intranet environment. More details about the tools for the URA paradigm may be found in [20] and that of change management may be found in [13].

VI. Conclusions

This paper proposed a generic model for semantics-based configuration management in projects. The semantics of version management are captured in terms of changes in design attributes as versions and changes in nondesign attributes as design equivalents. However, our model does not include other semantic information such as the engineering reasons for creation of version, etc. Our model is generic and has been applied to CAD and SDP.

Acknowledgment

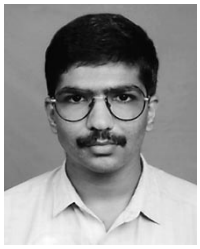
The authors wish to thank the anonymous reviewers for their valuable comments which greatly helped in improving the quality of the paper.

References

[1] D. Beech and B. Mahbood, "Generalized version control in an object-oriented database," in *Proc. IEEE 4th Int. Conf. Data Engg.*, 1988.

[2]This constraint has now been made obsolete by a decision to design a mail filter which receives incoming mails regarding change propagation and updates the URA files accordingly.

[2] C. Burrows, "Configuration management—A white paper,", http://www.ovum.com/evaluate/cm3/cm3wp.html.
[3] H. T. Chou and W. Kim, "A unifying framework for version control in a CAD environment," in *Proc. VLDB'88*, Japan, 1988.
[4] ——, "Versions and change notification in an object oriented database system," in *Proc. 25th Design Automat. Conf.*, 1988.
[5] S. A. Dart, "Parallels in computer-aided design framework and software development environment efforts," Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-92-TR-9, May 1992.
[6] ——, "The past, present, and future of configuration management," Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-92-TR-8.
[7] ——, "Achieving the best possible configuration management solution," in *8th Annu. Software Technology Conf.*, Salt Lake City, UT, USA, Apr. 1996, http://stscols.hill.af.mil/CrossTalk/1996/sep/achievin.html.
[8] *ANSI/IEEE Standard 1042-1987*, 1988.
[9] *Jane's All The World's Aircrafts*: Jane's Information Group, 1995.
[10] R. H. Katz and E. Chary, "Managing change in a CAD database," in *Proc. 13th VLDB'87*, Brighton, U.K., 1987, pp. 455–462.
[11] R. H. Katz, "Toward a unified framework for version modeling in engineering databases," *ACM Comput. Surv.*, vol. 22, no. 4, 1990.
[12] S. C. Keshu and K. K. Ganapathy, *Aircraft Production Technology and Management*. Bangalore, India: Interline, 1993.
[13] S. Lal, "A model for change management in software projects," M.Tech. thesis, Ind. Inst. Technol., Chennai, India, Jan. 1998.
[14] S. Moser, "The effect of object oriented frameworks on developer productivity," *IEEE Trans. Comput.*, pp. 45–51, Sept. 1996.
[15] K. Pillai, "The fountain model and its impact on project schedule," *ACM Softw. Eng. Notes*, pp. 32–38, Mar. 1996.
[16] R. Ramakrishnan and D. J. Ram, "Modeling design versions," in *Proc. 22nd Int. Conf. Very Large Databases (VLDB'96)*, Mumbai, India, Sept. 1996, pp. 556–566.
[17] M. J. Rochking, "The source code control system," *IEEE Trans. Softw. Eng.*, vol. SE-1, pp. 364–370, Dec. 1975.
[18] E. Sciore, "Versioning and configuration management in an object-oriented data model," *VLDB J.*, no. 3, pp. 77–106, 1994.
[19] A. H. Skarra and S. B. Zdonik, "The management of changing types in an object oriented database," in *Proc. OOPSLA*, 1986.
[20] S. Srinath, "URA: A paradigm for context sensitive reuse," M.S. thesis, Indian Institute of Technology, Chennai, India, 1998.
[21] S. Srinath, K. Venkatesh, and D. J. Ram, "An integrated solution based approach to software development using unified reuse artifacts," *ACM Software Eng. Notes*, pp. 56–60, July 1997.
[22] W. F. Tichy, "RCS—A system for version control," *Softw.— Pract. Exp.*, pp. 637–654, July 1985.
[23] J. D. Williams, "Managing iteration in OO projects," *IEEE Trans. Comput.*, vol. 29, pp. 39–43, Sept. 1996.
[24] A. Zeller, "A unified version model for configuration management," *ACM SIGSOFT*, pp. 151–160, Feb. 1995.



S. Srinath received the B.E. degree from the National Institute of Engineering, Mysore, India, and the M.S. degree from the Indian Institute of Technology, Madras, India, in 1999. He is currently pursuing the Ph.D. degree at the Berlin-Brandenburg Graduate School for Distributed Information Systems, Germany. His research interests include modeling interactive behavior, design issues of open information systems, change management in software engineering, and knowledge discovery.

R. Ramakrishnan received the B.Tech. and M.S. degrees from the Indian Institute of Technology (IIT), Madras, India, in 1997. He was a M.S. Research Scholar in the Distributed and Object Systems Lab. of the Department of Computer Science and Engineering at the IIT.

He is currently a Software Engineer in Honeywell India Software Operations, Bangalore, India. His research interests include distributed and object oriented databases, CAD/CAM, and software engineering.

D. Janaki Ram (M'95) received the Ph.D. degree from the Indian Institute of Technology (IIT), Delhi, India, in 1989.

He is currently with the Department of Computer Science and Engineering, IIT, Madras, Chennai, India, where he heads the Distributed and Object Systems Laboratory. He has taught several courses on distributed systems, object oriented software development, operating systems, programming languages, and artificial intelligence at graduate and undergraduate levels. He has also to his credit several publications in leading international journals and conferences. His research interests include software engineering, distributed and heterogenous computing, distributed object databases, and object technology. He was program chair for the Eighth International Conference on Management of Data (COMAD'97) and the First National Conference on Object Oriented Technology (NCOOT'97). He has served as a program committee member on several international conferences. He is also a Consulting Engineer in the area of object-oriented software development, organizational reuse for several software industries.

Dr. Ram is a member of ACM and the IEEE Computer Society.