# A Formal Model for Object-based Configuration Management


Yi-Jing Lin
IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598, USA
yjl@watson.ibm.com

Steven P. Reiss
Department of Computer Science
Box 1910, Brown University
Providence, RI 02912, USA
spr@cs.brown.edu



## Abstract

*This paper describes a formal model for object-based configuration management systems. We define the basic configuration management mechanisms with mathematical notations, and then derive a set of useful properties from them. These properties tell us what is guaranteed in systems based on this model and what is not. The process of developing these properties also helps us in extracting the essential elements in configuration management.*


## 1 Introduction

In this paper we present a formal model for object-based software configuration management systems. We define the required data types and data structures, model the configuration management activities with five basic operations, and then discuss how some useful properties can be derived from these basic definitions.

Our formal model differs from previous works in that we consider not only the versioning of software artifacts, but also *system building* - the generation of executable programs from source programs. Version control and system building are the most fundamental issues in configuration management. They interact with each other in various forms and cannot be discussed separately in a practical model.

Our model is object-based in the sense that it is based on compilation units that are represented as objects, However, it is not a full-fledged object-oriented model because we do not discuss the inheritance between classes.

The formal model described here was used as the basis of the POEM programming environment [11][12][13], which allows multiple programmers to develop C and C++ programs cooperatively over a local area network. Besides, all the properties discussed here can be proved. However, since most of the proofs are straightforward but tedious, we leave them out from this paper and only discuss their principles. The detailed of these proofs can be found in [11].

To give readers a more concrete feeling about this model, we will occasionally discuss how concepts in this model can be implemented in a C/C++ programming environment. C and C++ are quite popular and are good examples of the "conventional" languages. We also discuss a little about the handling of the Java programming language. We choose Java not just because it is quite popular recently, but also because it is a challenge to configuration management. Contrary to the assumption made by many existing configuration management systems, Java does not separate the interface and implementation of a class in most cases.

## 2 Basic Elements of the Model

In our model, we use five atomic data types and two data structures. The five atomic data types are *source code fragments*, *derived objects*, *identities*, *arguments*, and *Boolean values*. The two data structures are *sets* and *lists*.

A source code fragment is a piece of contiguous text. It can be implemented as a file in a file-based programming environment, or as an object in a object-based one. Our only requirement is that we be able to test the equivalence of two source code fragments. Besides, it is up to the implementation to define the actual meaning of equality between source code fragments. It can be based on lexical, syntactical, or semantic comparison.

A derived object is the result of applying translators, like compilers and linkers, to source code fragments or other derived objects. A derived object cannot be used in place for a source code fragment, nor vice versa. As with source code fragments, the only operation we need on derived objects is the testing of equivalence. Again, derived objects can be implemented as files in file systems or objects in an object-oriented database.

An identity is the property of an object that distinguishes it from all other objects [3]. The identity of an object should remain invariant across all possible modification of the object's value. The special identity value *NULL* is used to represent the absence of an object.

An *argument* is an option values for a compilation or linking command. In most existing systems, an argument is represented as a character string. For example, the argument "-g" means "generate debugging information" for the GNU C compiler. In our model, however, we do not need to know the format or meaning of the arguments. Again, the only operation we need on arguments is the testing of equivalence.

We use Boolean values and sets as they are conventionally defined. A list is an ordered collection. We use square brackets to represent lists. For example, "[x, *y*, *z*]" is a list with three elements; "[ ]" is an empty list. The operations we need on lists are concatenation, membership, indexing, and testing of equivalence. We use the operator "+" for the concatenation of lists, and the "$\in$" operator to test the membership of lists. The operator "*INDEX*" takes a list and a element and returns the position of the element in the list. For example, the result of *INDEX([x, y, z], y)* is 2. If the second argument is not a member of the first argument, than *INDEX* returns $\infty$. If an element appears more than once in a list, then *INDEX* returns the position of its first appearance. Two lists $x$ and $y$ are equal, denoted $x = y$, if and only if they have the same number of elements and all of their corresponding elements are equal.

In our model, we use sets and lists of atomic types only. We do not need sets of sets, sets of lists, lists of sets, or lists of lists. Also, we do not use sets or lists with elements of different types.

We denote the universe of source code fragments, that is, the set of all source code fragments, as $\Phi$. We also denote the universe of identities as $I$, the universe of derived objects as $O$, and the universe of arguments as $\mathcal{A}$.

## 2.1 Compilation Units

Our model is based on the notion of *compilation units*. A compilation unit represents a set of closely related functions, classes, or data structures. It has an *interface* that specifies the services it provides, and an *implementation* that contains the code to implement what is specified in the interface.

A compilation unit maps to a pair of ".h" file and ".c" file in a file-based C/C++ programming environments. We can treat ".h" files without matching ".c" files as compilation units with empty implementations, and treat ".c" files without matching ".h" files as compilation units with empty interfaces. In Java programming environments, a compilation unit maps to a ".java" file, which contains both interface and implementation. In actual implementation, we may extract the interface of a Java class on the fly by parsing the ".java" file.

Compilation units may depend on each other. If the *implementation* of a compilation unit X depends on the interface of another compilation unit Y, then we say that there is a $\alpha$ *dependency link* from X to Y. If the *interface* of X depends on the interface of Y, we say that there is an $\beta$ *dependency link* from X to Y. The $\beta$ link represents a stronger dependency between two compilation units. We assume that the implementation of a compilation unit always depends on its own interface, and thus a $\beta$ link from X to Y also implies that the implementation of X depends on the interface of Y.

In C and C++, the dependencies between compilation units are specified by the "#include" statements. A "#include" statement in a ".c" file represents a $\alpha$ dependency link; a "#include" statement in a ".h" file represents a $\beta$ dependency link. In Java, the dependencies between compilation units are specified by the "import" statements. Since the interface and implementation are not separated in Java, we have to assume that an "import" statement represents a stronger $\beta$ link unless additional mechanism is available. Furthermore, we assume that an "import" statement with the wild-card character "*" specifies $\beta$ links to all compilation units in a package.

A compilation unit has an unique *identity* that distinguish itself from others in a programming environment. It also has a *version predecessor pointer* and a *read-only flag*. If a compilation unit is created as a new version of another existing compilation unit, its version predecessor pointer points to its version predecessor. Otherwise its version predecessor pointer is null. The read-only flag is a boolean value that indicates the mutability of a compilation unit. If it is true, then the compilation unit is *read-only* and programmers cannot make any change on it. Otherwise, the compilation unit is *writable* and programmers can modify its interface, implementation, $\alpha$ links and $\beta$ links.

Summarizing all these properties, we can represent the data attributes of a compilation unit as a tuple *(ID, INF, IMP, A, B, VP, RO)*, where:

- *ID is the identity,*
- *INF is a source code segment, called the* ***interface****,*
- *IMP is a source code segment, called the* ***implementation****,*
- *A is a set of identities, called the* ***a dependency links****,*
- *B is a set of identities, called the* ***b dependency links****,*
- *VP is an identity, called the* ***version predecessor pointer****,*
- *RO is a Boolean value, called the* ***read-only flag****.*

We denote the data attributes of a compilation unit by the dot notation. For instance, if we have a compilation unit $x$, we denote its interface by *x.INF* and its $\alpha$ links by *x.A*. We say that two compilation units $x$ and $y$ are equal and denote it as $x = y$ if and only if they have the same value for each of their members.

In this representation, a link from a compilation unit $x$ to another compilation unit $y$ is represented by storing the identity of $y$ in $x$. For example, if *y.ID* is a member of *x.B*, then there is a $\beta$ link from $x$ to $y$.

Other than the data attributes represented by the tuple, a compilation unit also contains a set of methods. We do not include these methods into the tuple because their definition remained unchanged during the life-cycle of compilation units, and thus do not affect the state of a compilation unit. We will discuss the methods of compilation units in Section 3 and Section 4.

### 2.2 Compilation Unit Database

An important characteristic of object-oriented systems is the *complex states* of objects [21]. The state of an object depends not only on its own data attributes but also on the objects it refers to. It is usually meaningless to discuss an isolated object, or in our case, to discuss the links of a single compilation unit without considering others.

For clarity, we always specify a set of compilation units as the context of our discussion. This set of compilation units corresponds to all the compilation units in a programming environment and is called a *compilation unit database*. A compilation unit database *D* is *consistent* if and only if each member of *D* has a unique identity and every link in *D* points to another compilation unit in *D*. In other words, a consistent compilation unit database should satisfy the following two properties:

*- Unique identity:*

$$\forall x, y \in D \qquad (x.\mathrm{ID} = y.\mathrm{ID}) \Rightarrow (x = y)$$

*- No dangling links:*

$$\forall x \in D \ \ \forall i \in I\,(i \in \ (x.\mathrm{A} \cup x.\mathrm{B} \cup \{x.\mathrm{VP}\}) \Rightarrow \exists y \in D \ \ (y.\mathrm{ID} = i))$$

We denote the universe of compilation units as $\Sigma$ and the universe of consistent compilation unit databases as $\Delta$. Clearly, $\Delta$ is a subset of all possible sets of compilation units. That is:

$$\Delta \subset 2^{\Sigma} \tag{1}$$

### 2.3 System Building Subsystems

With compilation unit databases serving as the context of our discussion, we can find the sets of compilation units that are pointed to by the $\alpha$ and $\beta$ links of a compilation unit. Given a compilation unit x in a consistent compilation unit database D, we define *Alpha(D, x)* to be the set of compilation units to which x points a $\alpha$ link:

$$Alpha\,(D, x) \equiv \{y \in D | y.ID \in x.\mathrm{A}\} \tag{2}$$

And *Beta(D, x)* is the set of compilation units to which x points a $\beta$ link:

$$Beta\,(D, x) \equiv \{y \in D | y.ID \in x.\mathrm{B}\} \tag{3}$$

We can calculate the reflexive transitive closures of Alpha and Beta as following. They represent the sets of compilation units that can be reached from x by $\alpha$ links and $\beta$ links, respectively.

$$Alpha^*\,(D, x) \equiv \{y | \,(y = x) \vee \exists z\,(\,(y \in Alpha\,(D, z)\,) \wedge (z \in Alpha^*\,(D, x)\,)\,)\,\} \tag{4}$$

$$Beta^*\,(D, x) \equiv \{y | \,(y = x) \vee \exists z\,(\,(y \in Beta\,(D, z)\,) \wedge (z \in Beta^*\,(D, x)\,)\,)\,\} \tag{5}$$

And we can calculate the set of compilation units that can be reached via either $\alpha$ or $\beta$ links:

$$S\,(D, x) \equiv \{y | \,(y = x) \vee \exists z\,(\,(y \in Alpha\,(D, z)\,) \vee (y \in Beta\,(D, z)\,) \wedge (z \in S\,(D, x)\,)\,)\,\} \tag{6}$$

We call S(D, x) the *system building subsystem,* or simply the *subsystem,* designated by x. It contains all the software units x depends on. According to this definition, system-building subsystems may share compilation units and thus overlap each other. This is common and unavoidable in projects with reused compilation units.

## 3 System Building

In this section, we discuss how to find all the source code segments required in the compilation of a compilation unit, and how to model the compilation and link editing operations.

### 3.1 Effective Source Code

When compiling a compilation unit, we need not only its implementation but also all the interfaces it depends on. The *effective source code* of a compilation unit *x* is a list that contains all the source code segments required by the compilation of *x*.

In order to calculate the effective source code, first we have to calculate the *effective interfaces* of compilation units. The effective interface of a compilation unit *x*, denoted *EI(D, x)*, contains the interface of *x* itself and all the other interfaces that the interface of *x* depends on. We can calculate it using the following algorithm:

**Algorithm 1 Effective Interface**

```
FUNCTION EI(D: compilation unit database, x: compilation unit)
                        : list of source code fragments;
  VAR
    visited : set of compilation unit Identities;
    effective_interface : list of source code fragments;

  PROCEDURE EITraversal(x: compilation unit);
    VAR
      y: compilation unit;
    BEGIN
      visited := visited + {x.ID};
      for each y ∈ Beta(D, x)
        if (y.ID ∉ visited) {
          effective_interface := effective_interface + {x.INF};
          EITraversal(y);
        }
    END

  BEGIN
    IF (x ∉ D)
      RETURN( [ ] );
    visited := {};
    effective_interface := [];
    EITraversal(x);
    RETURN(effective_interface);
  END;
```

The effective interface of a compilation unit is determined by the $\beta$ links. The interface of a compilation unit *y* appears in the effective interface of *x* if and only if there is a path of $\beta$ links from *x* to *y*.

$$\forall D \in \Delta \ \ \forall x, y \in D \qquad y.\mathrm{INF} \in EI\,(D, x) \Leftrightarrow y \in Beta^*\,(D, x) \tag{7}$$

A $\beta$ link from *x* to *y* implies that the interface of *x* depends on the interface of *y*. Algorithm 1 guarantees that whenever the interface of *x* is included in an effective interface, then the interface of *y* is also included.

$$\forall D \in \Delta \ \ \forall x, y, z \in D \quad x.\mathrm{INF} \in EI(D, z) \wedge y \in Beta^*(D, x) \Rightarrow y.\mathrm{INF} \in EI(D, z) \tag{8}$$

Besides, the interface of *y* will appear before the interface of *x* if there is no cycles of β links.

$$\begin{aligned} &\forall D \in \Delta \ \ \forall x, y, z \in D \\ &(x.\mathrm{INF} \in EI(z) \wedge y \in Beta(D, x) \wedge x \notin Beta^*(D, y) \\ &\Rightarrow y.\mathrm{INF} \in EI(D, z) \\ &\wedge (INDEX(EI(D, z), y.\mathrm{INF}) < INDEX(EI(D, z), x.\mathrm{INF}))) \end{aligned} \tag{9}$$

Based on the effective interfaces, we can be calculated the effective source code *ES(D, x)* of a compilation unit *x* by Algorithm 2:

**Algorithm 2 Effective Source Code**

```
FUNCTION ES(D: compilation unit database, x: compilation unit):
                    LIST of source code fragments;
  VAR
     effective_source: list of source code fragments;
     s: compilation unit;
     t: source code fragment;
  BEGIN
     effective_source:= [];
     FOR EACH s ∈ S(D, x) DO
        FOR EACH t ∈ EI(D, s) DO
           if (t ∉ effective_source)
              effective_source:= effective_source + [t];
     effective_source:= effective_source + [x.INF, x.IMP];
  END;
```

The effective source code of a compilation unit *x* contains the implementation of *x* and a set of interfaces.It contains the interface of *y* if and only if the implementation of *x* depends on the interface of *y*. If we assume that the implementation of a compilation unit always depends on its own interface, which is usually the case in C and C++, then this dependency occurs in the following four cases:

- x and y are the same compilation unit.
- The implementation of x directly depends on the interface of y. In this case, there will be a α link from x to y.
- The implementation of x depends on the interface of z, and the interface of z depends on the interface of y. In this case, there will be a α link from x to z, and a path of β links from z to y.
- The interface of x depends on the interface of y. In this case, there will be a path of β links from x to y.

These four cases can be summarized in Theorem 1:

**Theorem 1 Membership of effective source code**

*Assume that the interfaces of all compilation units are distinct. The interface of y is in the effective source code of x if and only if x equals to y or x uses the interface of a compilation unit that has a path of* β *links to y.*

$$\begin{aligned} &\forall x, y \in D \quad y.\mathrm{INF} \in ES(D, x) \\ &\Leftrightarrow (x = y) \vee \exists z \in D\,[z \in S(D, x) \wedge y \in Beta^*(D, z)] \end{aligned} \tag{10}$$

A β link from *x* to *y* implies that the interface of *x* depends on the interface of *y*. Therefore, if the effective source code of some other compilation unit includes the interface of *x*, then the interface of *y* should also be included.

**Lemma 1 β links and membership of effective source code**

*If there is a path of* β *links from x to y, then x.INF appears in any ES(z) that contains y.INF.*

$$\forall x, y, z \in D\ (y.\mathrm{INF} \in ES(z) \wedge x \in Beta^*(D, y) \Rightarrow x.\mathrm{INF} \in ES(z)) \tag{11}$$

The ordering of elements in an effective source code is partly determined by the β links. Basically, if there is a β link from *x* to *y* and the interface of *x* appears in some effective source code *ES(z)*, then the interface of *y* will appear in front of the interface of *x* in *ES(z)*. This property is important because a β link from *x* to *y* implies that the interface of *x* depends on the interface of *y*. In this case, most programming languages require the interface of *y* appear before the interface of *x* in the text fed to the compiler.

**Theorem 2 Ordering in effective source code**

*If there is a* β *links from x to y, but not a path of* β *links from y back to x, then y.INF appears before x.INF in any ES(D, z) that contains x.INF.*

$$\begin{aligned} &(\forall D \in \Delta \ \ \forall x, y, z \in D) \\ &(x.\mathrm{INF} \in ES(D, z) \wedge y \in Beta(D, x) \wedge x \notin Beta^*(D, y) \\ &\Rightarrow y.\mathrm{INF} \in ES(D, z) \\ &\wedge (INDEX(ES(D, z), y.\mathrm{INF}) < INDEX(ES(D, z), x.\mathrm{INF}))) \end{aligned} \tag{12}$$

## 3.2 Compilation, Linking, and Derived Objects

Our model makes no assumptions about the formats of source code, arguments, and result of compilations in the system building process. The only requirement is that, given the same list of source code segments and arguments, the compilation should always return the same derived object. Therefore, we model the compilation of source code as a function. More precisely, a *compilation function,* denoted *Compile(S, A),* takes a list of source code segments *S* and a set of arguments *A* and returns a derived object.

Note that if multiple compilers are available, the specific compiler being used in a compilation should be modeled as an argument in the argument list. We do not use separate compilation functions for different compilers.

We call the result of compiling the effective source code of a compilation unit *x* the *local derived object* of *x*. More precisely, given a set of arguments *A*, the local derived object of a compilation unit *x* in a compilation unit database *D*, denoted *LocalDerived(D, x, A)*, is the result of compiling the effective source code of x with arguments A:

$$LocalDerived(D, x, A) \equiv Compile(ES(D, x), A) \tag{13}$$

Based on this, we can calculate he set of derived objects generated by compilation units in a system building subsystem. Given a set of arguments *A*, the derived objects of a subsystem *S(D, x)*, denoted as *Derived(D, x, A)*, is the set of derived objects generated by the members of *S(D, x)*:

$$Derived(D, x, A) \equiv \bigcup_{y \in S(D, x)} LocalDerived(D, y, A) \quad (14)$$

The set *Derived(D, x, A)* contains all the required derived objects to link *x* into an executable program. If a programming environment can calculate this value automatically, it will be easier for programmers to reuse *x* in a new project.

Similarly to the definition of the compilation function, we can use a function to model the linking of derived objects. Given a set of derived objects *O* and a set of arguments *A*, the *linking function,* denoted *Link(O, x, A),* returns another derived object which is the result of linking *O*.

Again, we make no assumptions about the format of derived objects and arguments used in a linking function. If multiple linkers are available, the specific linker used in a link editing is specified as an argument to the linking function.

Using the linking function, we can calculate the *executable* of a compilation unit *x* as following:

$$Exe(D, x, A) \equiv Link(Derived(D, x, A), A) \quad (15)$$

Notice that for compilation units which are not the main parts of programs, their executable will be NULL as the link editing will fail.

### 3.3 Implementing System-Building Methods

The functions *EI*, *ES*, *LocalDerived*, *Derived*, and *Exe* all take a compilation unit database and a compilation unit as their major arguments. This makes it rather straightforward to implement them as methods of a compilation unit. For example, the function *Derived(D, x, A)* can be implemented as a method *x.derived(A)* (using C++ syntax) of *x*. It is not necessary to put the compilation unit database *D* as an argument to the method because *D* represents all the compilation units in an environment. It is used only to set up the context of our formal discussion.

With the system-building functions implemented as methods of compilation units, we can get the system-building related data of a compilation unit rather easily. For example, we can invoke the *ES* method and the *Exe* method of a compilation unit to get its effective source code and executable code respectively.

## 4 Version Control

In our model, there are five basic operations that can be used to change the status of a compilation unit database. They are *new*, *edit*, *delete*, *snapshot* and *revise*. Each of them is atomic; that is, they either occur in their entirety or do not occur at all. We model each of these operations as a function that takes a compilation unit database as its major input and returns another compilation unit database as its result.

In actual implementations, there might be variants of these operations. For example, our prototype environment POEM supplies different interfaces for editing source code and editing links. They look very different to end users, but map to the same Edit operation in our formal model.

### 4.1 New

The New operation adds a new writable compilation unit to a compilation unit database. The new compilation unit contains empty interface and implementation, and has no links to other compilation units. It has an identity that is different from any existing compilation unit in the compilation unit database.

In order to choose a unique identity for the new compilation unit, we assume that we have a function, *NewID:* $2^S \to I$ , that takes a compilation unit database *D* and generates an identity that is different from the identity of any compilation unit in *D*. In other words, NewID satisfies the following proposition:

$$\forall D \in 2^{\Sigma} \quad \forall x \in D \qquad NewID(D) \neq x.\mathrm{ID} \quad (16)$$

Implementing such a function is not a problem in existing operating systems or databases. In ObjectStore [9], which is used to implement our prototype system POEM, each object in a database is assigned a unique identity when created.

**Algorithm 3 New:** $2^{\Sigma} \to 2^{\Sigma}$

```
FUNCTION New(D: compilation unit database) :compilation unit database;
   VAR
      new_cu: compilation unit;
   BEGIN
      new_cu:=(NewID(D),NULL,NULL,{ },{ },NULL, FALSE);
      RETURN(D U {new_cu} );
   END
```

It is obvious that applying the New operation on a consistent compilation unit database yields another consistent compilation unit database:

$$\forall D \in \Delta \qquad New(D) \in \Delta \quad (17)$$

And the New operation does not modify any read-only compilation unit in a compilation unit database.

$$\forall D \in \Delta \;\; \forall x \in D \qquad (x.\mathrm{RO} = TRUE) \Rightarrow \exists x' \in New(D)\ [x' = x] \quad (18)$$

### 4.2 Edit

The Edit operation is used to modify the interface, implementation, α links, or β links of a writable compilation unit. We model it as a function that takes a compilation unit database *D* and two compilation units *x* and *x′*. The compilation unit *x* should be a member of *D*, and *x′* is the new value for *x*. Edit replaces *x* with *x′* if all the following conditions are satisfied:

- *x* is in *D*.
- The identity of *x′* is the same as the identity of *x*.
- The version predecessor of x′ is the same as that of *x*.

- Both $x$ and $x'$ are writable.

If any one of these conditions is not true, then Edit becomes a null operation and returns $D$ without any modification.

**Algorithm 4 Edit:** $2^{\Sigma} \times \Sigma \times \Sigma \rightarrow 2^{\Sigma}$

```
FUNCTION Edit(D: compilation unit database, x: compilation unit,
                    x': compilation unit): compilation unit database;
  BEGIN
    IF ((x ∉ D) ∨ (x.RO = TRUE) ∨ (x'.RO = TRUE) ∨ (x.ID ≠ x'.ID)
         ∨ (x.VP ≠ x'.VP))
        RETURN(D);
    ELSE IF (there is an identity in (x'.A ∪ x'.B) that does not
              correspond to the identity of any compilation unit in D)
        RETURN(D);
    ELSE
        RETURN(D - {x'} U {x});
  END
```

If the Edit operation operates on a consistent compilation unit database, then the resulting compilation unit database is still consistent:

$$\forall D \in \Delta \ \ \forall x, x' \in \Sigma \qquad Edit(D, x, x') \in \Delta \tag{19}$$

The Edit operation does not modify any read-only compilation unit in a compilation unit database.

$$\forall D \in \Delta \ \ \forall y \in D \ \ \forall x, x' \in \Sigma \quad ((y.\mathrm{RO} = TRUE) \Rightarrow \exists y' \in Edit(D, x, x') \ [y' = y]) \tag{20}$$

### 4.3 Delete

The Delete operation is used to remove a writable compilation unit from a compilation unit database. If the compilation unit being removed is referred to by other compilation units, then the Delete operation becomes a null operation.

We model the Delete operation as a function that takes a compilation unit database $D$ and a compilation unit $x$ as its argument. It removes $x$ from $D$ if $x$ is a writable compilation unit in $D$ and $x$ is not referred to by any compilation unit in $D$.

**Algorithm 5 Delete:** $2^{\Sigma} \times \Sigma \rightarrow 2^{\Sigma}$

```
FUNCTION Delete(D: compilation unit database, x: compilation unit):
                       compilation unit database;
  BEGIN
    IF (x ∉ D)  ∨(x.RO ≠ FALSE)
        RETURN(D);
    ELSE IF there is a y in D - {x} such that x ∈ S(D,y)
        RETURN(D);
    ELSE
        RETURN(D - {x});
  END
```

Applying the Delete operation on a consistent software database unit yields another consistent compilation unit database:

$$\forall D \in \Delta \ \ \forall x \in \Sigma \qquad Delete(D, x) \in \Delta \tag{21}$$

The Delete operation does not modify any read-only compilation unit:

$$\forall D \in \Delta \ \ \forall y \in D \ \ \forall x \in \Sigma \quad (y.\mathrm{RO} = TRUE) \Rightarrow \exists y' \in Delete(D, x) \ [y' = y] \tag{22}$$

### 4.4 Snapshot

Like PCTE[1], we use Snapshot and Revise instead of check-in and check-out as the basic version control operations. The Snapshot operation creates a read-only version of a system-building subsystem as the version predecessor of an existing writable one. For example, in Figure 1we have a system building subsystem *S(D, x)* that contains two writable compilation units x and y, and a read-only compilation unit z. If we apply the Snapshot operation on x, we will get a subsystem *S(D, x')* that contains the read-only snapshots of *x* and *y*. Notice that since *z* is read-only, no snapshot of *z* is created and *z* is shared by both versions of *S(D, x)*.

We model Snapshot as a function that takes a compilation unit database *D* and a compilation unit *x* as its input, and returns a compilation unit database containing new compilation units that represent the snapshot of *S(D, x)*.

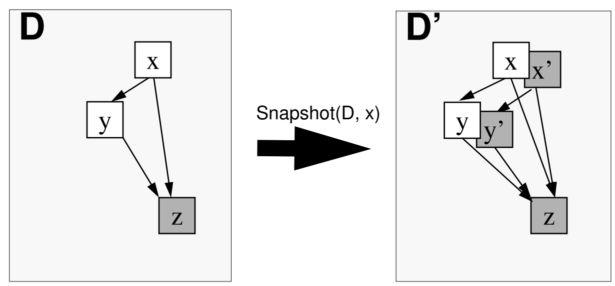


Figure 1:Snapshot operation

In the calculation of Snapshot, we use *VPred(D, x)* to denote the version predecessor of x in the compilation unit database *D*. That is, *VPred(D, x),* is the compilation unit in *D* whose identity matches *x.VP*. If there is no such compilation unit in *D*, then *VPred(D, x)* is NULL.

**Algorithm 6 Snapshot:** $2^{\Sigma} \times \Sigma \rightarrow 2^{\Sigma}$

```
FUNCTION Snapshot(D: compilation unit database, x: compilation unit)
                              : compilation unit database;
VAR
  D':      compilation unit database;
  mod_set: SET of compilation units;/* Modified compilation units. */
  mod_s_rts: SET of compilation units;/*roots of modified subsystems. */
  y, new_cu: compilation unit;

  FUNCTION Modified(D: compilation unit database, x: compilation unit)
                : BOOLEAN;
    BEGIN
      IF (x.RO = TRUE)
          RETURN FALSE;
      ELSE IF (x.VP = NULL)
          /* x is considered as modified if it has no version predecessor.*/
          return TRUE;
      ELSE IF (x.INF ≠ VPred(D, x).INF) ∨ (x.IMP ≠ VPred(D, x).IMP)
              ∨ (x.A ≠ VPred(D, x).A) ∨ (x.B ≠ VPred(D, x).B))
          RETURN TRUE;
      ELSE
```

```
21:             RETURN FALSE;
22:         END
23:
24:     FUNCTION ConnectSnapshots(D: compilation unit database,
25:                             y: compilation unit, L: SET of compilation units):
26:                     SET of identities;
27:         VAR
28:             result : SET of identities;
29:             z: compilation unit;
30:         BEGIN
31:             result := { };
32:             FOR EACH z∈ L
33:                 IF (z.RO = TRUE)
34:                     result := result ∪ {z.ID}
35:                 ELSE
36:                     result := result ∪ {VPred(D, z).ID}
37:         END
38:
39:BEGIN      /* Snapshot */
40:     IF (x.RO ∨ (x ∉ D))
41:         RETURN(D);
42:     D′ := D;
43:     mod_set := { };
44:     FOR EACH y ∈ S(D, x)
45:         IF Modified(D,y)
46:             mod_set := mod_set ∪ {y};
47:     mod_s_rts := { };
48:     FOR EACH y ∈ S(D, x)
49:         if (S(D, y) ∩ mod_set ≠ { })
50:             mod_s_rts := mod_s_rts + {y};
51:     FOR EACH y ∈ mod_s_rts /
52:         BEGIN      /* Create a snapshot for y. */
53:             new_cu := (NewID(D′), y.INF, y.IMP,{ }, { }, y.VP, TRUE);
54:             D′ := D′ + {new_cu};
55:             y.VP := new_cu.ID;
56:         END
57:     FOR EACH y ∈ mod_s_rts
58:         BEGIN
59:             VPred(D,y).A:=ConnectSnapshot(D,y,Alpha(D, y));
60:             VPred(D,y).B:=ConnectSnapshot(D,y,Beta(D, y));
61:         END
62:     RETURN(D′);
63:END.
```

The main part of Algorithm 6 can be divided into four parts. In the first part, lines 40 and 41, it tests the validity of its arguments. If $x$ is read-only or if $x$ is not a member $D$, then this is an invalid Snapshot operation. Algorithm 6 returns $D$ without making any modification.

In the second part, lines 42 through 50, it finds all the compilation units in *S(D, x)* that should be considered as modified and puts them in the set mod_s_rts. Basically, a compilation unit $y$ is considered modified if *S(D, y)* contains a modified compilation unit (lines 47–50).

In the third part, from line 51 to line 56, Algorithm 6 creates a new compilation unit for each compilation unit in mod_s_rts. The newly created compilation units are snapshots of the ones in mod_s_rts. They inherit their interface and implementation from the compilation units they are created from, but have no links pointing to other compilation units.

In the last part, from line 57 to line 63, Algorithm 6 sets the links of the compilation units created in the third part. This is done by calling the function ConnectSnapshots, which is defined as follow:

The last part of Algorithm 6 is a little complicated, but the basic idea is simple. Suppose we have a compilation unit $x'$ that is created as the snapshot of $x$. If $x$ has a link pointing to $y$ and there is also a newly created snapshot of $y$, then we set up a corresponding link from $x'$ to the snapshot of $y$. Otherwise, if there is no newly created snapshot of $y$, then we set up a link from $x'$ to $y$.

The Snapshot operation does not change the consistency of a compilation unit database. Applying the Snapshot operation on a consistent compilation unit database yields another consistent compilation unit database.

$$\forall D \in \Delta \;\; \forall x \in \Sigma \qquad Snapshot(D, x) \in \Delta \tag{23}$$

Also, the Snapshot operation does not modify any read-only compilation unit in a compilation unit database.

$$\forall D \in \Delta \;\; \forall y \in D \;\; \forall x \in \Sigma \quad (y.\mathrm{RO} = TRUE) \Rightarrow \exists y' \in Snapshot(D, x)\ [y' = y] \tag{24}$$

## 4.5 Revise

The Revise operation creates a writable version of a system-building subsystem as the version successor of a read-only one. As illustrated in Figure 2, applying the Revise operation on x creates a new system-building subsystem *S(D, x')* that contains the new versions of x and y. The Revise operation takes a set of compilation units $W$ as its third argument, and creates new versions only for the compilation units within $W$. $W$ corresponds to the *work area* of a programmer in a programming environment. When using the Revise operation, programmers usually do not want to create new versions for compilation units outside of their own workareas. By limiting the Revise operation with W, we also reduce the number of new versions created.

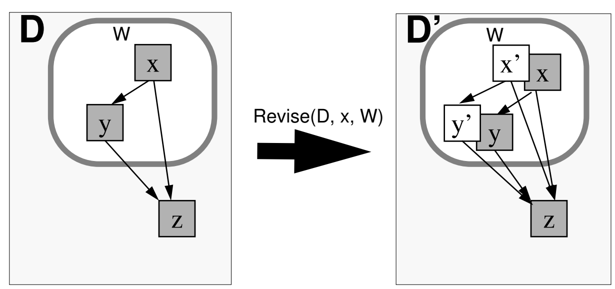


Figure 2: Revise operation

The Revise operation is defined by the following algorithm:

**Algorithm 7 Revise:** $2^{\Sigma} \times \Sigma \times 2^{\Sigma} \rightarrow 2^{\Sigma}$

```
FUNCTION Revise(D: compilation unit database, x: compilation unit,
                        W: set of compilation units): compilation unit database;
    VAR
        D′:     compilation unit database;
        y, new_cu: compilation unit;
        revise_set, new_cu_set: set of compilation units;

    FUNCTION ConnectRevisions(D: compilation unit database,
                x: compilation unit, L: SET of compilation units,
```

```
10:             N: SET of compilation units): SET of identities;
11:     VAR
12:         result : SET of identities;
13:         w, y: compilation unit;
14:     BEGIN
15:         result := { };
16:         FOR EACH w ∈ L
17:             IF (there is a z in N such that VPred(z) = w)
18:                 result := result ∪ {z.ID};
19:             ELSE
20:                 result := result ∪ {w.ID};
21:     END
22:
23: BEGIN /* Revise */
24:     IF (¬x.RO ∨ (x ∉ D))
25:         RETURN(D);
26:     revise_set := S(D, x) ∩ W;
27:     D′ := D;
28:     new_cu_set := { };
29:     FOR EACH y ∈ revise_set
30:         BEGIN
31:             new_cu := (NewID(D′), y.INF, y.IMP, { }, { }, y.ID, FALSE)
32:             new_cu_set := new_cu_set + {new_cu};
33:             D′ := D′ + {new_cu};
34:         END
35:     FOR EACH y ∈ new_cu_set;
36:         BEGIN
37:             y.A := ConnectRevisions(D, y, Alpha(D, y), new_cu_set);
38:             y.B := ConnectRevisions(D, y, Beta(D, y), new_cu_set);
39:         END
40:     RETURN(D′);
41: END;
```

The main function of Algorithm 7 can also be divided into four parts. The first part, lines 24 and 25, tests the validity of the arguments. If *x* is not read-only or if *x* is not a member of *D*, then the function returns *D* immediately without making any modification.

The second part, line 26, finds the set of compilation units that should be revised. This is simply the intersection of the subsystem designated by *x* and the containing workarea W.

The third part, lines 27 through 34, creates a new writable version for each compilation unit found in the second part. The new writable compilation units are created with their interface and implementation copied from the compilation units from which they are created.

The last part, lines 35 through 41, connects the newly created compilation units by calling the function ConnectRevisions. Again, this part is a little complicated, but the basic idea is rather simple. For each new compilation unit $x'$ created from an existing compilation unit $x$, if there is a link from $x$ to $y$ and a new version of $y$ is created, then we set up a link from $x'$ to the new version of $y$. If no new version is created for $y$, then we set up a link from $x'$ to $y$. In the first case, $y$ is a compilation unit in *W*; in the latter case, $y$ is a compilation unit outside of *W*.

Again, the Revise operation does not change the consistency of a compilation unit database. Applying the Revise operation on a consistent compilation unit database yields another consistent compilation unit database.

$$\forall D \in \Delta \;\; \forall x \in D \;\; \forall B \in 2^{\Sigma} \qquad Revise(D, x, B) \in \Delta \tag{25}$$

And the Revise operation does not modify any read-only compilation unit in a compilation unit database.

$$\forall D \in \Delta \;\; \forall y \in D \;\; \forall x \in \Sigma \;\; \forall B \in 2^{\Sigma} \quad (yRO = TRUE) \Rightarrow \exists y' \in Revise(D, x, B)\; [y' = y] \tag{26}$$

## 4.6 Successor of a Compilation Unit Database

If we apply any of the five basic operations to a compilation unit database *D*, we get another compilation unit database *D′*. We call *D′* an *immediate successor* of *D* and define *ISuccessors(D)* to be the set of all immediate successors of *D*.

$$\begin{aligned} ISuccessors(D) \equiv \{D' |\; & \exists x, y \in \Sigma \;\; \exists B \in 2^{\Sigma} \quad (D' = New(D, x)) \\ & \vee (D' = Edit(D, x, y)) \vee (D' = Delete(D, x)) \\ & \vee (D' = Snapshot(D, x)) \vee (D' = Revise(D, x, B)) \} \end{aligned} \tag{27}$$

From the properties derived in the previous sections, we know that the immediate successor of a consistent compilation unit database is still a consistent compilation unit database:

$$\forall D \in \Delta \quad \forall D' \in ISuccessors(D) \qquad D' \in \Delta \tag{28}$$

If we repeatedly apply operations on a compilation unit database *D*, we will still get a compilation unit base. We define *Successors(D)* to be the set of compilation unit databases that can be generated by repeatedly applying operations on *D*:

$$\begin{aligned} & Successors(D) \\ & \equiv \{X | (X = D) \vee \exists Y (X \in ISucessors(Y) \wedge (Y \in Successors(D)))\} \end{aligned} \tag{29}$$

From equation (28), we can infer that if *D* is a consistent compilation unit database, then all members of *Successors(D)* are also consistent.

$$\forall D \in \Delta \quad \forall D' \in Successors(D) \qquad D' \in \Delta \tag{30}$$

If we start with an empty compilation unit database in our programming environment and use only the five basic operation discussed in the previous section, we will always get a consistent compilation unit database:

$$\forall D' \in Successors(\{\ \}) \qquad D' \in \Delta \tag{31}$$

Compilation unit databases in *Successors({})* are of our special interest, because they represent all the compilation unit databases that can be possibly created by our basic operations.

Members of *Successors({})* have several useful properties. By induction on *D*, we can prove the following:

### Lemma 2

*For any compilation unit database D in Successors({}), the following properties hold:*

*1. If a compilation unit x is the version predecessor of another compilation unit y, then x is read-only.*

$$\forall D \in Successors(\{\ \}) \quad \forall x, y \in D \quad (y.\mathrm{VP} = x.\mathrm{ID}) \Rightarrow (x.\mathrm{RO} = TRUE) \tag{32}$$

*2. If a compilation unit x is used by a read-only compilation unit y, then x is also read-only.*

$$\forall D \in Successors(\{\ \}) \quad \forall x, y \in D \quad x \in S(D, y) \wedge (y.\mathrm{RO} = TRUE) \Rightarrow (x.\mathrm{RO} = TRUE) \tag{33}$$

### 4.7 Implementing Version-Control Methods

Like the system-building operations, the version-control operations Edit, Delete, Snapshot, and Revise all take a compilation unit database and a compilation unit as their major arguments. Again this makes it rather easy to implement them as methods of compilation units. For example, we can implement Snapshot(D, x) as a method *x.snapshot()* of *x*. By invoking this method, we can get a read-only version of the subsystem designated by *x*.

Since the New operation does not take any compilation unit as its argument, it can not be implemented as a method of compilation units. Instead, it should be a method of the database or a global function of the environment.

## 5 Properties of Read-only Compilation Units

Once created, a read-only compilation unit should not be changed, and the derived objects and executable it generates should remain constant over time. From equations (18), (20), (22), (24) and (26), we know that for any read-only compilation unit x in a compilation unit database D ∈ Successors({}), x remains unchanged in the immediate successors of D.

$$\forall D \in Successors(\{\ \})\ \ \forall x \in D\ \ D' \in ISuccessors(D)\quad (x.\mathrm{RO} = TRUE) \Rightarrow \exists x' \in D'\,[x' = x] \tag{34}$$

With equation (34), we can show that the effective source code of a read-only compilation unit remains constant in the successors of *{}*:

$$\forall D \in Successors(\{\ \})\ \ \forall x \in D\ \ \forall D' \in ISuccessors(D)\quad ((x.\mathrm{RO} = TRUE) \Rightarrow \exists x' \in D'\,[(x' = x) \wedge (ES(D', x') = ES(D, x))]) \tag{35}$$

And then we can infer that the local derived objects of a read-only compilation unit remain constant in the successors of *{}*:

$$\forall D \in Successor(\{\ \})\ \forall x \in D\ \forall D' \in ISuccessor(D)\ \forall a \in \mathcal{A}\quad (x.\mathrm{RO} = TRUE) \Rightarrow \exists x' \in D'\ [(x' = x) \wedge LocalDerived(D', x', a) = LocalDerived(D, x, a)] \tag{36}$$

Based on Lemma 2 and equation (36), we can prove that if the root compilation unit of a subsystem is read-only, then the derived object it generates will not change over time:

**Lemma 3 The derived objects of a r ead-only compilation unit are constant.**
*For any compilation unit x in a compilation unit database D ∈ Successors({}), the derived objects of x remain unchanged in the immediate successors of D:*

$$\forall D \in Sucessor(\{\ \})\ \forall x \in D\ \ \forall D' \in ISuccessor(D)\ \ \forall a \in \mathcal{A}\quad (x.\mathrm{RO} = TRUE) \Rightarrow \exists x' \in D'\,[(x' = x) \wedge (Derived(D', x', a) = Derived(D, x, a))] \tag{37}$$

And it is not hard to show that if the root compilation unit of a program is read-only, then the executable it generates will remain constant:

**Theorem 3 The executable of a read-only compilation unit is constant.**
*For any read-only compilation unit x in a compilation unit database D ∈ Successors({}), the executable of x remains unchanged in the immediate successors of D:*

$$\forall D \in Sucessor(\{\ \})\ \ \forall x \in D\ \ D' \in ISucessor(D)\ \ \forall a \in \mathcal{A}\quad ((x.\mathrm{RO} = TRUE) \Rightarrow \exists x' \in D'\,[Exe(D, x, a) = Exe(D', x, a)]) \tag{38}$$

## 6 Versional Consistency

In a programming environment, there may be multiple versions of the same source code segment. But when building the executable of a program, we should use at most one version of each source code segment. If multiple versions of the same source code segment are used, the result is usually erroneous - either the linker will complain about the redundancy, or a version of the source code segment is selected randomly.

We say that the configuration of a program is *versionally consistent* if it contains at most one version of each source code segment. In this section, we discuss how versional consistency can be defined in our model and how it is affected by different operations.

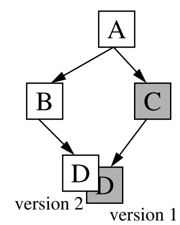


Figure 3: A versionally inconsistent subsystem.

To discuss the versional consistency of subsystems, we need to define some terms. A compilation unit *y* is a *version ancestor* of another compilation unit *x* if there is a path of version predecessor links from *x* to *y*. We define *VA(D, x)* as the set of *x*'s version ancestors:

$$VA(D, x) \equiv \{y \mid (VPred(D, x) = y) \vee \exists z \in D\,[(VPred(D, x) = z) \wedge y \in VA(D, z)]\} \tag{39}$$

We say that two compilation units are of the same *version family* if they can be traced back to a common version ancestor. They are both versions of the same "logical compilation unit." We denote this property with the predicate *SVF. SVF(x, y)* is true if and only if x is equal to y or x and y have a common version ancestor:

$$SVF(x, y) \equiv (x = y) \vee \exists z\,[z \in VA(D, x) \wedge z \in VA(D, y)] \tag{40}$$

Version family is a notion in our model, not a unit of management. In some configuration management systems, like Adele [4], system models are specified in terms of version families, and version families are first-class entities that can be directly operated by users. We do not support operations to handle version families because our system models are effectively specified by the links between individual versions of compilation units.

Then, we say that *S(D, x)* is *versionally consistent* if and only if every element of *S(D, x)* belongs to a different version family:

$$VConsistent(S(D,x)) \equiv \forall x, y \in S(D,x) \quad [(x = y) \vee \neg SVF(x,y)]$$

The versional consistency of a system-building subsystem may be changed by Edit and Revise, but not New, Delete, and Snapshot. The New operation and the Delete operation cannot change the versional consistency of a subsystem because the compilation units they create or delete cannot be members of other existing subsystems.

Based on Lemma 2, we can also prove that the Snapshot operation cannot change the versional consistency of a subsystem:

**Theorem 4 Snapshot retains versional consistency.**
*In any compilation unit database D ∈ Successors({}), if a subsystem is versionally consistent, then its snapshot is also versionally consistent.*

$$\begin{aligned}&\forall D \in Sucessor(\{\ \}) \quad \forall x \in D\\ &\qquad VConsistent(S(D,x)) \wedge (D' = Snapshot(D,x))\\ &\Rightarrow VConsistent(S(D', Pred(D',x)))\end{aligned}$$

The Edit operation may change the versional consistency of a subsystem because it allows users to change the links of writable compilation units arbitrarily. For example, we can create a versionally inconsistent subsystem as illustrated in Figure 3 by Edit operations.

One way to avoid this problem is to modify the algorithm of Edit so that it disallows any action that results in versionally inconsistent. But this may be expensive. For example, to detect that adding the link from *B* to version 2 of *D* in Figure 3 causes a versionally inconsistent subsystem, we have to know that *B* is used by *A* and *A* is indirectly using another version of *D*.

The Revise operation may also create versions of subsystems that are not versionally consistent. Figure 4 shows an example. Here although the subsystem *S(D, x)* is versionally consistent, the new version of *S(D, x)* generated by *Revise(D, x, {x, y})* is not. The reason for this problem is that the Revise operation does not create new versions for compilation units outside of *W*. Although *z* is a member of the subsystem designated by *x*, no new version is created for it.

Applying a Revise operation on a subsystem, however, does not always cause a versional consistency problem even when the subsystem spans several workareas. We observe that the problem occurs only when there are paths of links leading out and back to a workarea. For example, in Figure 4 the link from *x* to *z* leads out of workarea *W*, then the link from *z* to *y* leads back to workarea *W* again.

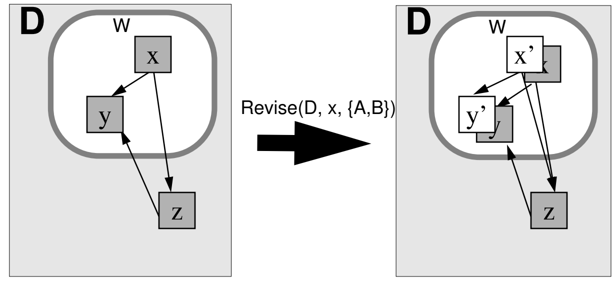


Figure 4: Version inconsistency caused by Revise

To state this property more precisely, we bring in the notion of *endpoint enforced* sets. We say that a set of compilation units *C* is endpoint enforced if there is no path of links leading out and then back to the set again:

$$\begin{aligned}&\forall D \in \Delta \quad \forall x, y, z \in D\\ &[(x \in C) \wedge (y \in C) \wedge (z \in S(D,x)) \wedge (y \in S(D,z)) \Rightarrow z \in S]\end{aligned} \quad (41)$$

Based on this definition, we can prove that *Revise(D, x, W)* will not cause versional consistency problem if the intersection of *S(D, x)* and *W* is *endpoint enforced*.

**Theorem 5 Revise retains versional consistency on endpoint-enforced subsystems.**
*If a subsystem S(D, x) is versionally consistent and S(D, x) ∩ W is endpoint enforced, then the new revision of S(D, x) in Revise(D, x, W) is also versionally consistent.*

## 7 Related Works

Feiler classified the models of configuration management systems into checkout/checkin model, composition model, long transaction model, and change set model [6]. According to Feiler's classification, our model is closer to the composition model, but with some significant differences. Zeller [22] proposed a unified version model based on Feiler's work. NUCM [19] provides a generic model of a distributed repository and an associated programmable interface.

There are a lot of research and commercial configuration management systems developed in recent years, but not many formal model has been proposed. Borison [2] described a model of system building for programming environments based on database. Rochkind [14]and Ticky [18] discussed the calcuation of deltas in version control. All these discussions are rather low-level. Wiebe [20] proposed a generic mathematical model for software configuration management and implemented a generic system called JASON. The JASON model is much more complete and rigid than the earlier work. But although intended to be generic, the JASON model is somewhat biased toward the compositional model. It directly supports mechanisms that are fundamental in the compositional model, but not those for the long-transaction model and change set model.

Our framework is based on the object-oriented paradigm. Version control in terms of objects is studied in the area of software development environments as well as in the area of computer-aided design (CAD). Zdonik describes an object-oriented database system that includes a built-in version control mechanism [21]. PCTE [1] supports an versioned object base for constructing software engineering environments. Our definitions of Snapshot and Revise operations are based on PCTE. Katz surveys version modeling in engineering databases and proposes a unified framework [8]. Like our framework, Adele [4][5] defines version control operations on top of interconnected objects that represent software.

## 8 Conclusion

We showed it is possible to model an object-based configuration management system with a set of simple data structures and basic operations. We also derived a set of properties from these basic definitions.

The properties we derived tell us what is guaranteed in our configuration management system and what is not. Theorem 1 tells us which source code segments will be fed to a compiler when we compile a function or a class. Theorem 2 tells us how the ordering of the interfaces fed to a compiler is decided by the dependencies we specified between functions and classes. Theorem 3 insures that the executable generated from a read-only program will be the same if the same arguments are given. Theorem 4 and Theorem 5 tells how version control operations can affect the versional consistency of a system.

These properties can help us in locating problems during the configuration management process. The process of developing them also helps us in understanding the essential elements of configuration management.

## References


[1] Gerard Boudier, Ferdinando Gallo, Regis Minot, and Ian Thomas, "An Overview of PCTE and PCTE+," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (Peter Henderson, ed.), pp. 248-257, November 1988. Published as ACM SIGSOFT Software Engineering Notes, Vol 13, No 5, November 1988, and ACM SIGPLAN Notices, Vol 24, No 2, February 1989.

[2] Ellen Borison, "A Model of Software Manufacture," in *Advanced Programming Environments — Proceedings of an International Workshop*, Springer-Verlag, pp. 197-220, 1986.

[3] Setrag N. Khoshafian and George P. Copeland, "Object Identity," *SIGPLAN Notices*, Vol. 21, No. 11, pp. 406-415, November 1986.

[4] Jacky Estublier, "A Configuration Manager: The Adele data base of programs," in *Workshop on software engineering environments for programming-in-the-large*, June 1985.

[5] Jacky Estublier and Rubby Casallas, "The Adele Configuration Manager," in *Configuration Management* (W. F. Tichy, ed.), John Wiley & Son Ltd., 1994

[6] Peter H. Feiler, "Configuration Management Models in Commercial Environments," Technical Report CMU/SEI-91-TR-7, Software Engineering Institute, Carnegie Mellon University, March 1991.

[7] Ferdinando Gallo, Regis Minot, and Ian Thomas, "The Object Management System of PCTE as a Software Engineering Database Management System," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (Peter Henderson, ed.), pp. 12-15, December 1986.

[8] Randy H. Katz, "Toward a Unified Framework for Version Modeling in Engineering Databases," *ACM Computing Surveys*, Vol 22, No 4, pp. 375-408, December 1990.

[9] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinred, "The ObjectStore Database System," in *Communications of the ACM*, Vol. 34, No. 10, pp. 50-63, October 1991.

[10] Roy Levin and Paul R. McJones, "The Vesta Approach to Precise Configuration of Large Software Systems," DEC System Research Center Research Report No. 105, June 1993.

[11] Yi-Jing Lin, "Configuration Management in Terms of Logical Structures," Ph.D. dissertation, Department of Computer Science, Brown University, Rhode Island, May 1996.

[12] Yi-Jing Lin and Steven P. Reiss, "Configuration Management in terms of Modules," in *Proceedings of the Fifth International Workshop on Software Configuration Management*, April 1995.

[13] Yi-Jing Lin and Steven P. Reiss, "Configuration Management with Logical Structures," in *Proceedings of the 18th International Conference on Software Engineering*, pp. 298-307, March 1996.

[14] Marc J. Rochkind, "The Source Code Control System," in *IEEE Transactions on Software Engineering*, pp. 364-370, Vol 1 No 4, December 1975.

[15] James Rumbaugh, "Controlling Propagation of Operations using Attributes on Relations," *ACM OOPSLA '88 Proceedings*, pp. 285-296, September 1988.

[16] SunPro, "SPARCworks/TeamWare Solution Guide, Sun Microsystems, Inc., California, USA, 1994.

[17] Warren Teitelman, "A tour through Cedar," in *IEEE Software*, pp. 44-73, Apr 1984.

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

[19] Andre van der Hoek, Dennis Hembigner, and Alexander L. Wolf, "A Generic, Peer-to-Peer Repository for Distributed Configuration Management," in *Proceedings of the 18th International Conference on Software Engineering*, pp. 308-317, March 1996.

[20] Douglas Wiebe, "Generic Software Configuration Management: Theory and Design," Technical Report 90-07-03, Computer Science Department, University of Washington, Seattle, WA 98195, 1990.

[21] Stanley B. Zdonik, "Version Management in an Object-Oriented database," in *Advanced Programming Environments*, (R. Conradi, T. M. Didriksen, and D. H. Wanvik, ed.), pp. 405-422, 1986.Stanley B. Zdonik and David Maier (eds). *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, Inc., Palo Alto, CA, 1990.

[22] Andreas Zeller, "A Unified Version Model for Configuration Management," in *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering,* (Gail Kaiser, ed), pp. 151-160, October, 1995.