Version management for ROPCO—a micro-incremental reuse environment

M K Zand*, M H Samadzadeh† and H Saiedian*

Version management is a pervasive issue in computer science. It is an important part of database theory, software engineering, distributed systems, etc. This paper focuses on version management for software being developed in a software development environment or a software factory. The main objective of the paper is to present a new approach on version management of reusable units of software by using modules/templates as the reusable units. This approach is adopted in the design of a software development environment called ROPCO. Reuse Of Persistent Code and Object code is an integrated collection of techniques, tools, and structures to facilitate the reusability of persistent code and object code.

reusable units, ROPCO, version management, persistent code, object code, software reuse

Productivity, adaptability, simplicity, maintainability, and reliability are among the main issues concerning large-scale software development. Reuse is an emerging practice in software development. The result of one survey shows that 40–60% of all code is reusable from one application to another, 60% of the design and code on all business application is reusable, and 75% of program functions is common to more than one program[1]. The survey also indicated that on the average only 15% of the code found in most programs is unique and novel to each specific application.

Several alternatives have been proposed for the level of software reuse. They include specification level, design level, program/subprogram level, module/template level, as well as code and object-code level[2,3,4]. The specific and design levels are at higher levels of abstraction than the other reuse levels, therefore their potential for accommodating reuse is greater and their adaptation to new applications can be simpler. However, the reuse process of the specification and design levels ultimately involves coding (either manual or system-generated), testing, and debugging[5,3,6,7]. On the other hand, reuse at the code level reduces the coding and overall testing efforts, hence it could be more economical where a large library of reusable code exists.

To make code (i.e., modules or segments of code) a feasible alternative as a reuse level, three main tasks should be addressed as the prime objectives in the design of software development environments.

(1) Identification of and providing access to a segment of code and object code based on user requirements *(locating)*. Given the availability of a large number of software components, like the 'Japanese software factory', one should ask: 'Does a part exist that preforms this function?'[1]. The goal is to provide an answer (or answers) to that query. This task would be simple to achieve if the components are designed and developed for the purpose of reuse and are organized in a library. Examples of successful code components are the SPSS statistical libraries and the IMSL math library. Relatively modern languages like Ada, Modula-2, and Smalltalk-80 not only provide features for developing reusable components, but also provide mechanisms to make a distinction between abstraction and implementation. Krueger believes that although in a few narrow areas this model has been successful, the applicability of code components in more broad areas is not yet clear[8]. However, according to Prieto-Diaz, software classification and identification is the kernel of successful reuse[9].
(2) Facilitating program modification at both coding and compiler levels (*recompilation*). Direct code editing is the main method to customize the code for new applications. This method has the problem of the possible impact of modifications on other components. Krueger considers 'work at a low level of abstraction' a drawback of this level of reuse[8]. This is not a drawback for all cases, we believe in some environments like ROPCO (Reuse Of Persistent Code and Object code) the low level of work adds to the capabilities of the environment.

Another approach for code customization is using parameterized components like the Ada generic package. The use of parameterized packages relieves the burden of validation of the modified components. The generic parameters of a parameterized package are declared in a way that the type of the instantiated elements must satisfy the parameter requirements[10].

Development environments for languages like Ada or Modula-2 facilitate 'macro-incremental' compilation, i.e., independent compilation of the

*Department of Computer Science, University of Nebraska at Omaha, Omaha, NE 68182, USA
†Computer Science Department, Oklahoma State University, Stillwater, OK 74078, USA

modified modules. This capability is utilized for reuse at the module level. In a code-level-reuse environment, code templates or other units of reusable code and the related object code must be relocatable, and each template's symbol table should be extracted from (or merged into) the program symbol table. These capabilities facilitate the reuse of portions of programs, hence the possibility of micro-incremental compilation[11].

(3) Providing facilities to store and retrieve all versions of a program efficiently *(persistency and version management).*

In this work we focus on the third issue. The first two issues are discussed in detail in an earlier paper[4]. The rest of this paper is organized as follows. The second section presents a discussion on different approaches to version and configuration management. In the third section, hierarchical and persistent data structures and the notions of *existence persistency* and *version persistency* are discussed. The fourth section gives an overview of the ROPCO environment. SCM, the kernel of ROPCO, is presented in the fifth section. In the sixth and seventh sections the methodology and procedures of the version management of objects in ROPCO are introduced, and the final section provides the summary and conclusions.

VERSION MANAGEMENT

A version management or configuration management system can be defined in different ways, depending on the design goals of a software system and the needs of the users of the software system. In general, a version management system can be defined[12] as a system for identifying the software architecture and its components by controlling changes in those components during their life-cycle and maintaining consistency among the components by controlling the effect of having multiple developers and allowing simultaneous work on the system components. Different concepts utilized to achieve these objectives include check-in/check-out, composition, change sets, and transaction.

Make-like systems augmented with an interface tool have been proposed for configuration management. The main weakness of such systems is that they are passive tools. In other words they act only when requested explicitly. If the structure of a system is not stable, there is a need for a new makefile for every change in the structure of the system. Make-type configuration management tools are based on file name suffixes and do not use implicit rules, so the user has to specify intermediate steps and target files. In these systems, it is rather difficult to maintain the integrity because of reliance on the user for keeping all dependency relations updated. Make assumes the success of each command, which can lead to obscure formats for the description file. Also, makefile may contain redundant information[12,13].

At times, file systems are promoted as version management systems. Characteristics that are specific to file systems should be considered during the design and construction of a configuration and/or version management system[14]. Typically, in a file system there are no file attributes that can be used to characterize the contents of the files. The only relation that can be used to navigate through files is the directories and the directory structure. There are in general no links between files to represent relations among these files, except naming conventions. Hence, it does not seem that file systems can provide all of the necessary operations for a version management system.

Another possibility for having a configuration/version management system is using a conventional database system. Typically, the transactions in conventional databases have short life times, and there are two ways to control concurrency[15], either by using a two-phase lock to avoid conflicts between the database records or by the use of a time-stamp to detect the conflicts. In a version management system, even minor logical changes such as enhancements or bug fixes may cause many conflicts.

Instead of using the conventional database technology to implement a version/configuration management system, the more recent object-oriented database technology can be used to implement an object-oriented version management system. Object-oriented databases such as ORION, ODE, Ires, O2, GemStone, and Vbase, have overcome many of the limitations of the relational and other record-oriented database management systems[16,17,18]. The following justification can be offered for the benefits that might be gained from combining the database technology with the object-oriented concepts (when dealing with version/configuration management). Classification of versions or configuration items depends on two important relationships: the hierarchical relationship and the syntactical relationship. The hierarchical relationship is the basis for the enumerative classification scheme. The faceted classification scheme, on the other hand, is constructed based on the syntactical relationship among the elements[19].

In this paper we mainly focus on the hierarchical relationships among versions of a program in an environment which facilitates micro-incremental reuse.

PERSISTENCY

Persistency has been defined and used with two different interpretations. The first type of persistency, which is called Eper in this paper, is defined as follows: *Eper, existence persistency, allows objects to exist as long as they are required and the life span of each object exceeds the life-cycle of its originator*[20,21]. In this type of persistency, only one version of each object is saved. Any change to an object results in a new object and the original one is lost.

The second type of persistency, which we call Vper, is defined as follows: *In Vper, version persistency, different versions of one object co-exist, and each version is marked with its time or instance of creation*[22,23,24,25]. In this type of persistency, the life span of a version may exceed the life-cycle of its originator.

Vper has been implemented using two different approaches. Tichy[24] used the incremental method for version control. In this approach all updates on the current version are recorded after each revision. To go back from the current version to one of the previous versions, say X, all recorded updates between the current version and version X are undone to reconstruct version X. This approach resembles the log or journal mechanism of database transactions[24]. This is an *ad hoc external* persistency. *Internal* techniques are those which use or modify the object structure to make it persistent. Cole[26], Sarnak and Tarjan[23], and Zand *et al.*[27] applied the internal Vper technique to ordered lists, red/black binary trees, and quad-trees, respectively.

Version persistency of hierarchical structures can be implemented in two different ways: the node-copy method and the path-copy method. We use binary trees to describe these methods.

In the first approach (i.e., the node-copying method), which is an internal technique, each node in the tree has *p* pointers in addition to its original two pointers. For each update in the tree, the key information and the last two pointers of the relevant node are copied into a new node; also a pointer is set in the node's parent pointer list that points to the new node. If all pointer fields in a parent's node are used up, the same operation is repeated on the path to the root till an ancestor node with an unused pointer field is reached, or the root is encountered. The auxiliary pointers in the nodes require a time stamp to indicate the time at which they are set. Also a root array is required to point to the root of the tree at a given time[23].

The second approach (i.e., the path-copy method), which is an internal technique, involves copying only the nodes in which changes are made. Any node containing a pointer to a node that is copied must itself be copied. This means copying a node causes a ripple effect of copying all the nodes on the path from the node to the root[28]. The path-copy method is quite versatile in the applications it supports (i.e., it has the capability to update any version of the tree), provided that an update is assumed to create an entirely new version. But the major drawback of this approach is its space usage. In a tree with *L* levels, for each update, *L* nodes must be copied. In the node-copy method, it is true that using extra space for the auxiliary pointers and time stamps results in a constant cost on spaces usage, but this space overhead in fact reduces the total cost of the copying operation. Zand and Fisher[25] devised an algorithm that utilized the node-copy method to implement PB-trees (persistent B-trees). That algorithm constitutes the base structure of the Software Control Mechanism (SCM) of the ROPCO environment.

Application of persistent structures in software reuse

The results of an experiment show that the optimal size of a reusable module is about 200 to 250 code lines[29]. Reuse of modules is usually incremental, i.e., it results in altering only one portion of the code of a software system. In most of the existing reuse methods, the older versions of the resulting module at each step either are not saved at all or are not stored efficiently. It turns out that there is no need to store the entire module under a new version number or recompile the whole module. The modified portion of the code is normally one or more of the constructs of the language (referred to here as a template). Consequently, it is natural to consider templates as units of reuse and store only those templates that are modified or are added to a module[4]. Code (i.e., templates/modules) stored in this fashion is called persistent code.

Persistency has been used in all levels of software reuse. Cheatham[30] used external incremental Vper at the abstract program level. Marzullo and Wiebe[31] applied incremental Vper at the module level in the Jasmine Modeling Facility. Neighbors[32] followed the same idea in the 'refinement history' at all levels of the DRACO System. Tichy[24] applied Vper to version control. At the time of this work, internal hierarchical Vper has not been utilized in software reuse. The merits of this technique include enhancing version management of software units through providing fast and direct access to the units of software, enabling dynamic changes within or among software units, and efficient storage of those units.

AN OVERVIEW OF THE ROPCO ENVIRONMENT

This section presents the subsystems and components of the ROPCO environment, relationships among its components, and a general overview of the system (for a complete explanation of ROPCO's components see Zand *et al.*[4]).

ROPCO is a software development environment that adopts a novel approach to reuse utilizing code and object code stored in persistent structures. ROPCO was designed to accomplish the following goals:

(1) Identification of and providing access to a segment of code and the related object code based on user specifications.
(2) Facilitating module modification both at the programmer and the compiler levels.
(3) Providing facilities to store efficiently and retrieve all versions of the module(s) under consideration.
(4) Facilitating module/template inter- and intra-connection[33].

Minimizing the user/programmer efforts spent on tasks other than programming was one of the main motivations in the design of the ROPCO environment. We emphasized on minimizing the compilation time and effort. ROPCO's minimal compilation strategy limits the propagation of micro-incremental compilation resulting from intra-procedure modifications, while consistency of variables and the intra-module flow graph remain intact. The problem of macro-incremental compilation resulting from inter-procedural modifications is addressed in ROPCO's design[11].

In ROPCO, templates/modules are chosen as units of reuse and the novel concept of a 'use network' is devised. Another design goal of ROPCO was to furnish the system with a pragmatic schema for storage of versions of a module, and maintaining direct and sequential access to the blocks of modules. Hierarchical and flat persistency methods were utilized to achieve this goal.

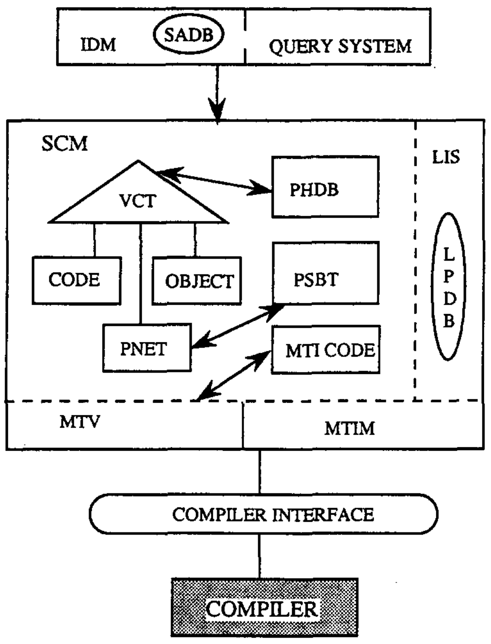
ROPCO consists of three main subsystems: IDentification Mechanism (IDM), Software Control Mechanism (SCM), and Interface. Figure 1 illustrates the main components of ROPCO and the relationships among them. The following is a brief description of IDM and Interface. SCM is described in some detail in the next section.

IDentification Mechanism (IDM)

IDM is designed to identify and select one or more modules/templates that meet user requirements. Each module is classified based on its function, environment, and implementation attributes. The attributes, which are stored in the Software Attribute DataBase (SADB), are used to identify the candidate module(s)/template(s).

Interface

The Interface is designed to facilitate micro-incremental compilation between ROPCO and the language-specific compilers. It is designed to prepare the templates/modules that are altered or otherwise affected by modification for compilation. As a post-compiling activity, the Interface transfers the address of the newly-created object code and symbol table to SCM.



IDM:	IDentification Mechanism	LIS:	LInkage System
LPDB:	LIS Persistent DataBase	MTIM:	Module/Template Inter-connection Mechanism
MTV:	Module/Template Verifier	PNET:	Persistent NETwork
PHDB:	Persistent History DataBase	PSBT:	Persistent SymBol Table
SADB:	Software Attribute DataBase	SCM:	Software Control Mechanism
VCT:	Version Control Tree		

Figure 1. Main components of ROPCO

SCM COMPONENTS

SCM, or the Software Control Mechanism, which is the kernel of ROPCO, is designed to facilitate program access, modification, storage, compilation, and interconnection. The objectives of the design of this subsystem are as follows:

(1) To store efficiently and to eliminate duplicate copies of code and object code for different versions of a program.
(2) To arrange access to different versions of a module/template.
(3) To coordinate compilation and recompilation (specifically, micro-incremental compilation).
(4) To maintain variable consistency throughout different versions of software components.
(5) To preform inter- and intra-connection operations.

The core of SCM is structural persistence that provides direct access to the different versions of a program and its constituent templates. A revised incremental persistence is utilized to assist by faster movement, back and forth, among the adjacent versions. ROPCO's persistent version management system that provides random and sequential access (in both directions), without the use of *ad hoc* structures, is a novel approach. SCM has five components:

(1) Program History DataBase (PHDB)
(2) Version Control Tree (VCT)
(3) Persistent NETwork (PNET)
(4) Persistent SymBol Table (PSBT)
(5) Module/Template Verifier (MTV)—MTV is designed to check for the compatibility of interconnection/substitution of modules/templates. This component of ROPCO employs an automatic verifier and utilizes a semantic specification model to verify functional consistency among the interconnecting components.
(6) Module/Template Interconnection Mechanism (MTIM)—MTIM is the component responsible for interconnection and substitution of modules/templates. MTIM utilizes PSBT, PNET, VCT, and MTV to preform its task.

The first four components of SCM are discussed briefly in the following subsections.

Persistent History DataBase (PHDB)

This component of SCM is designed to retain the information required to access the root of the family tree (version tree) of a module/program and to hold information on how to move from a specific version to the successor or predecessor version(s). To access the family tree of a module, the header pointing to the root of the version tree can be found in that module's record in PHDB. To move from one version to another, information on incremental changes to each version is also deposited in the module record of the working module. The structure of a program record in PHDB is given below.

```
PHDB-Program-Record
  Program-ID {same as Program-ID in SADB}
  Number-of-versions
  Persistent-symbol-table-location
  Root-header
    Version-root-pointer[n]
    version-ID {Start from a base number and
      increment by 1 after each modification}
    root-location
    non-technical-info {date, programmer, ...}
    Forward-construction
      template-modified[m]
        template-no.
        action-code
    Backward-construction
      template-modified[m]
        template-no.
        action-code
      replaced-template(s)
  Next-root-Header
```

Some information about various PHDB operations is provided later in the paper under 'Version management and access/retrieval in SCM'.

Version Control Tree (VCT)

We use a typical hierarchical Vper PB-tree structure for VCT[25]. The internal nodes of VCT serve as indices. The external nodes contain addresses of the clusters which hold code and a header for the chain of variables used in the template. Each internal node has *k* sets of child pointers and each pointer has a time stamp to indicate the version number. Each external node contains five address items related to the template represented by the external node. These addresses are: location of the template's code clusters, location of the interconnection specifications cluster, a header to a chain (in PNET) of names used in the template, and the location of the next external node. The external nodes are linked to maintain sequential access to the clusters. This is in addition to the direct access provided to the template clusters. Figure 2 shows examples of VCT and PNET. The access and update operations of VCT are given below under 'Persistent network and variable consistency'.

Persistent SymBol Table (PSBT)

The symbol table of a persistent programming environment exists as long as the program exists. The life-cycle of the symbol table of a program does not terminate at the end of program execution. The symbol table of a persistent program must be able to support all versions of the program. The main difference between this type of persistent symbol table and a conventional symbol

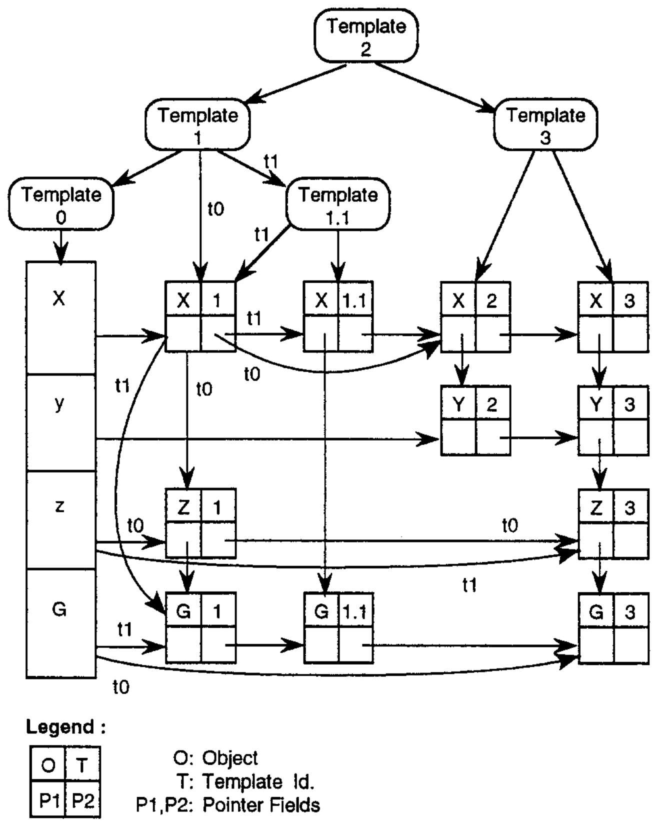


Figure 2. An example of a network of chains (PNET). In this example four objects are used in five templates. Template 0 corresponds to the declaration segment of the program

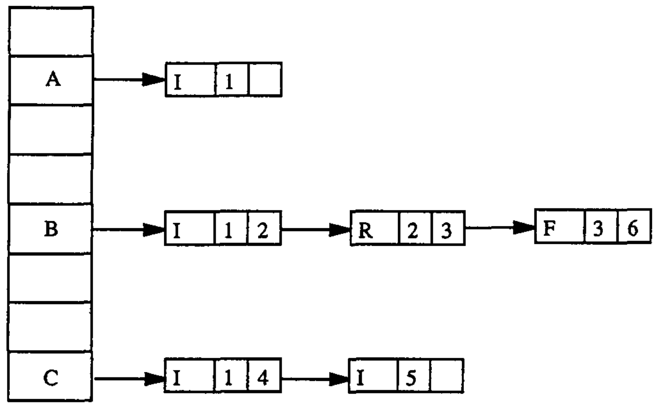
table is that in PSBT one variable could be defined in more than one context and can have different types. However, only one type of variable is allowed in each instance (version) of the table. Each variable could have one or more records in the table with the following structure:

```
VAR-NODE:
  VAR-NAME,
  TYPE,
  CREATION-INSTANCE,
  ALTERATION-INSTANCE,
  NEXT-VAR-NODE,
  ⟨other information⟩
```

Figure 3 illustrates a persistent symbol table for six versions of a hypothetical program with three variables. The next section presents the outline of the algorithms devised for storage and retrieval in PSBT.

MANAGEMENT OF OBJECT VERSIONS IN ROPCO

In this section, algorithms and procedures designed to preform the version management of objects (i.e., names and variables) are presented. The subsystems of SCM (VCT, PSBT, and PNET), which are designed to facilitate objects version management (among other tasks), simplify this process.



Sequence of change:
t1 - Var A, B, C : I;
t2 - Var A, C : I; B : R;
t3 - Var A, C : I; B : F;
t4 - Var A : I; B : F;
t5 - Var A, C : I; B : F;
t6 - Var A, C : I;

Legend: | T | td | tr |

T: Type
td: Time declared
tr: Time redeclared

Figure 3. The effect of type change/insertion or deletion of variables in the Persistent SymBol Table

Storage of objects in PSBT

Algorithm PSBT-store is devised to store a variable in the current version of PSBT. The description of the algorithm is given below.

```
PSBT-store (VAR-NAME, V-TYPE, and VID {version-ID})

(1) Use VAR-NAME as a hash value to find the
    location of the variable in the table.
(2) If there is no collision, allocate a new VAR-NODE
    and goto SET.
(3) If a VAR-NAME does exist, compare V-TYPE with
    TYPE:
    (a) if types are the same and if ALTERATION-
        INSTANCE is zero (is not set), then the variable
        is already defined (it is active);
    (b) else, compare the VID, say x, with the ALTER-
        ATION-INSTANCE, say y. If x > y, then the
        PSBT variable is active (attempt to update in a
        past version), Exit;
    (c) Otherwise, allocate a new VAR-NODE and goto
        SET.
(4) If the V-TYPE is not equal to TYPE, allocate a
    new VAR-NODE and goto SET. The ALTER-
    ATION-INSTANCE field of the resident variable is
    set to VID.

SET:
TYPE: = V-TYPE
CREATION-INSTANCE: = VID
NEXT-VAR-NODE: = NULL
ALTERATION-INSTANCE: = 0

End PSBT-store
```

Retrieval of objects from PSBT

Algorithm PSBT-access is designed to access a variable in the table. The required inputs to this algorithm are version-ID (VID), variable name (VAR-NAME), and variable type (V-TYPE).

```
PSBT-access (VAR-NAME, V-TYPE, VID),

(1) Use VAR-NAME as a hash value to locate the
    variable.
(2) If there is no collision, then the variable doesn't exist,
    Exit;
(3) If V-TYPE = TYPE and ALTERATION-IN-
    STANCE is zero or is larger than VID, then the
    variable is found, Exit;
(4) If ALTERATION-INSTANCE < = VID and the
    NEXT-VAR-NODE field is null, then the variable
    doesn't exist, Exit;
(5) Otherwise, follow the NEXT-VAR-NODE link and
    repeat Step 4.

End PSBT-access.
```

The search time of both algorithms is slightly higher than the search time of a non-persistent symbol table, but it remains proportional to n/m (n is the number of variables in the table and m is the size of the table). The slight increase in the length of search is related to searching the chain of NEXT-VAR-NODE to find the desired instance of the given variable (see Figure 3).

Persistent network and variable consistency

When a variable declaration is deleted or inserted in template zero of a program or when a new statement is added to a template, some tools are needed to enforce variable consistency. Fischer and Johnson[34] used the idea of one symbol table for each template of a program. In their scheme, every identifier in a symbol table uses a pointer to a chain of all uses of that identifier in the template. When a declaration is inserted in or deleted from a template, the use-chains are searched to access the relevant variables in the template.

In this research, the idea of a use-chain has been modified (and improved upon) and used as a tool to enforce variable consistency in a persistent environment. Also, the space overhead of using one symbol table for one template used by Cockshot *et al.*[21] is eliminated. Each unit of a program, which is allowed to declare variables, has one persistent symbol table. Each template of a program has a chain of pointers to the symbol table—one for each variable used in the template. Each variable in the symbol table has a chain of its uses in different templates. These two chains share one node at their intersection. Figure 2 depicts an example of a PNET with one NEXT-VARIABLE-LINK pointer per node.

Network node structure and operations

The general structure of the nodes of PNET is given below.

```
NETWORK-NODE
  POINTER-TO-SYMBOL-TABLE
  TIME-INSERTED
  NEXT-TEMPLATE-LINKS[n]
    NT-TIME-STAMP
    NT-LINK
  NEXT-VARIABLE-LINKS[n]
    NV-TIME-STAMP
    NV-LINK
```

Some of the possible operations on a chain network and the description of each operation are mentioned in the following subsection.

Operations on PNET

This section presents the primitive operations as well as other operations provided by PNET. ADD-VARIABLE (template_i. time_t, var_v)

Add variable var_v to the chain of variables used in template_i at time_t.
Add template_i to the chain of templates using var_v at time_t.

DELETE-VARIABLE (template_i, time_t, var_v)

Remove var_v from the chain of variables used in template_i at time_t.
Remove template_i from the chain of templates that uses var_v at time_t.

NEW-STATEMENT (template_i, time_t, var_1, var_2, . . . ,var_n)

Preform NEW-VARIABLE n times for the n variables in the new template.

DELETE-STATEMENT (template_i, time_t, var_1, var_2, var_n)

Perform DELETE-VARIABLE n times.

LOOK-FOR-Template-USE-VARIABLE (var_v, time_t)

Through the header node of var_v, find the chain of templates using var_v at time_t.
Traverse the chain to find templates using var_v (in each node follow the links with the largest time-stamp which is less than or equal to the search time_t).

LOOK-FOR-VAR-IN-Template (template_i, time_t)

Through the header node of template *i* (given in the external node in the template tree), traverse the chain at time_t to find all variables used in the template.

NEW-VARIABLE (var_v, time_t)

Create a header node for var_v and set it to null at time_t. This operation is needed when a new variable is declared.

DELETE-DECLARATION (var_v, time_t)

Perform LOOK-FOR-Template-USE-VARIABLE to find all templates using v at time_t.
Remove the node(s) for variable v from the chain of all those templates. Set variable v header's link to null at time_t.

DELETE-Template (template_b, time_t)

Through the header node of the chain (template_b, time_t), remove all nodes from this chain and all variable-chains intersecting the chain.

INSERT-Template (template_b, time_t, var_1, var_2, . . . ,var_n)

Preform ADD-VARIABLE n times for the n variables.

DELETE-VARIABLE(Var_v, time_t)

Through the header for var v at time_t, access chain (var_v, time_t). Remove all nodes on the chain from $\text{chain}_{(v,t)}$, and the chains of templates which intersect this chain. Set the header node for var v to null at time_t.

Analysis of time complexity of PNET operations

The average search time for all inquiries on one chain is $O(n)$, where n is the length of the chain. Every update needs two searches, one in the template chain and the other in the variable chain. For example, to remove variable v at time t† from all templates using it, we must traverse variable v's chain (via invoking the LOOK-FOR-Template-USE-VARIABLE operation) and find the templates using v. Then all of those templates must be traversed to remove v from their chains.

If m templates with the average chain length of n are involved, the processing time would be $O(m*n)$ This is a very expensive operation. Adding backward links to the nodes would eliminate the traversal of a template's chains. The use of backward links makes it possible to remove variables from the template chains during the removal of templates from the variable chains. The average processing time of such an update is search time $O(m)$, which is a substantial improvement over the first approach (singly-linked list) and the Fischer-Johnson method[34]. However, the trade-off for such an improvement is the additional space for the backward links and the required time-stamp for each link. Description of the operations described earlier need not be changed except for the DELETE-VARIABLE operation.

†Time and version numbers are used with the same meaning. They both refer to Version-ID.

VERSION MANAGEMENT AND ACCESS/ RETRIEVAL IN SCM

Access to modules/templates in SCM

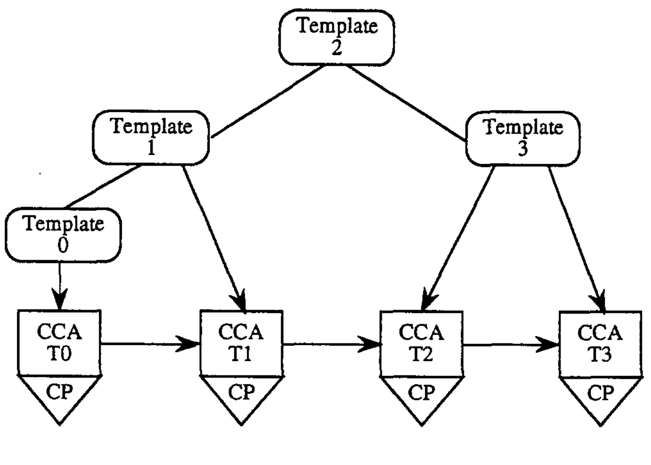
This section describes the method of access to a given version of a module/program in SCM. Using the Program-ID (given forth by IDM), SCM loads the program record from PHDB into memory. Information in the root-header facilitates access to any given version of the program. The *root-header* field in the *PHDB-program record* contains the location of the root(s) of VCT, which is (are) accessed and loaded into the primary memory. Using the *Persistent-Symbol-Table-Location* field of the program record, the persistent symbol table is also loaded into memory.

The user may start by browsing any version of a program, move from one version to another, or edit a specific template in a given version of the program. For example, if the user is interested in looking at all templates in version x of the program, first SCM locates the root of version x of the program tree through the *version-root-pointer*[x] field in the program record. Then SCM uses the first external node (which is found in the left-most node of the PB-tree and serves as header to the chained external nodes) to access all clusters related to version x, and displays the code alongside the template identifiers. To move from version x to version $x + 1$, only clusters of those templates which have been altered, moved, or inserted at instance $x + 1$ are fetched and replaced by the revised template (this information is recorded in the version history of $x + 1$).

Providing direct and sequential access to programs and templates is a substantial improvement over other approaches. This improvement is attributable to the utilization of both incremental and structural persistence in the system. Version history of incremental persistence, implemented by Cockshot *et al.*[21], eliminates the need to traverse the tree (to move from one version to another as needed) by using structurally persistent structures. However, their method only has direct access to the first or the last version (depending on the implementation) and needs forward (or backward) reconstruction to get to the other versions. Reps proposed using a 'shareable 2-3 tree' for the implementation of the symbol table for large attributes[28], but his proposed structure is not a persistent one. Reps has used this structure to implement incremental changes efficiently in a symbol table.

Update

Modification of a program could be divided into two broad categories of local and structural changes. Local modification may alter the structure of a program, the network of use-chains (PNET) of the program, or both. However, the flow graph of the program remains unchanged. Structural changes not only modify the network, but also alter the flow graph of the program. The possible template modification and the corresponding VCT's required actions are provided in the next section.



Legend:

CCA:	Code Cluster Address
CP:	Chain-pointer to PNET

Figure 4. A Version Control Tree (VCT) (in binary form) with four templates

In all cases, the inter-template modification results in updating the network of chains (PNET) and PSBT. The details of actions on each operation relevant to PNET and PSBT are provided above under 'Persistent network and variable consistency'.

Once a modification on a module/template is done, it is the task of the version manager to update the tree, record the modification history, and allocate clusters for new templates.

Operations on templates

In this section the list of possible operations (modifications) on templates and the actions taken by the relevant SCM components are presented. Examples of operations are shown on a binary tree to make the demonstration simpler. The original tree is given in Figure 4.

(1) **Intra-template modification**. This operation consists of modifying a portion of a template. If an intra-template modification does not have any effect on other templates, only compilation of the altered template is required. SCM dispatches the template and the working symbol table to the interface for compilation.
(2) **Template relocation**. If a template is relocated in a module, no compilation is required. The version manager (SCM) needs to update the tree by taking the following steps:

   (a) Remove the node related to the relocated template (described below);
   (b) Insert a new node for the relocated template (described below); and
   (c) Modify the pointers at the external level (Figure 5).

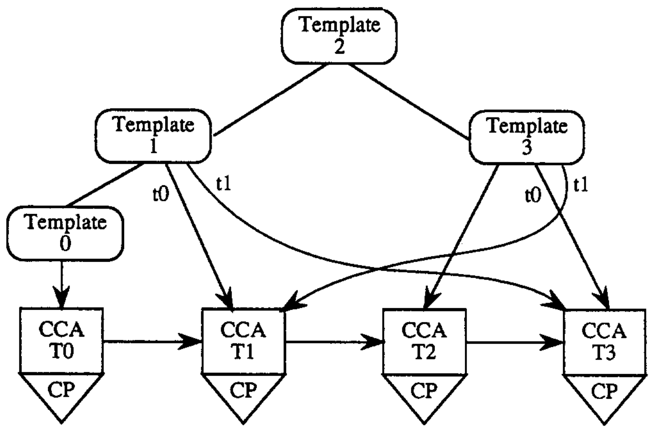


Figure 5. An example of template-move in VCT

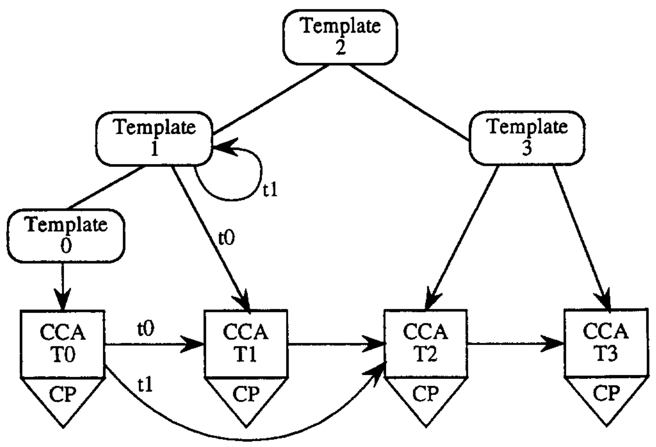


Figure 7. An example of template-removal (Template 1 is removed at t_l)

(3) **Insertion of a newly created template**. The template needs to be compiled and the version manager must update the tree. Figure 6 shows the tree after insertion of Template 1.5.

(4) **Template removal**. This operation removes the template node from the VCT program tree. The effect of this operation on the tree given in Figure 4 is shown in Figure 7. In this example, Template 1 is deleted at time t_1.

(5) **Template split**. This operation splits a template into two new templates. The new templates need to be compiled. SCM removes the split template from the old version and preforms the insert operation for the new templates. Figure 8 shows the VCT tree after splitting Template 1 into templates 1 and 1.5 at time t_1.

(6) **Consecutive-templates join or blend**. Two or more templates are combined/blended to create a new template. A compilation is required for the new template. Assuming templates $t_j, t_{j+1}, \ldots, t_{k-1}$, and t_k are to be joined, SCM preforms the template removal operation on t_j through t_k and the new template replaces the old templates. Figure 9 provides an example of the join operation. Templates 1 and 2 are joined to form the new Template 2 at time t_1.

SUMMARY AND CONCLUSIONS

The main objective of this paper was to present a new approach on managing the versions of reusable units of software. ROPCO is an integrated collection of techniques, tools, and structures to facilitate the reusability of persistent code and object code. As the kernel of ROPCO, SCM is an integrated collection of techniques and structures designed: (a) to assist reusability at the code and object-code levels; (b) to store efficiently versions of programs on persistent structures; (c) to facilitate micro-incremental compilation; and (d) to preform the interconnection process. SCM is also designed to provide direct and sequential access to the templates of a program, and to facilitate maintaining variable consistency across the versions of a program. Providing direct and sequential access to the versions of a program and its templates is a notable advantage of this design over similar reuse approaches.

The novel idea of a persistent network (PNET) and its structure, along with the related operations allowed on the PNET, are provided. The PNET network is designed to facilitate variable consistency throughout the versions of a program. It is shown that the PNET structure is

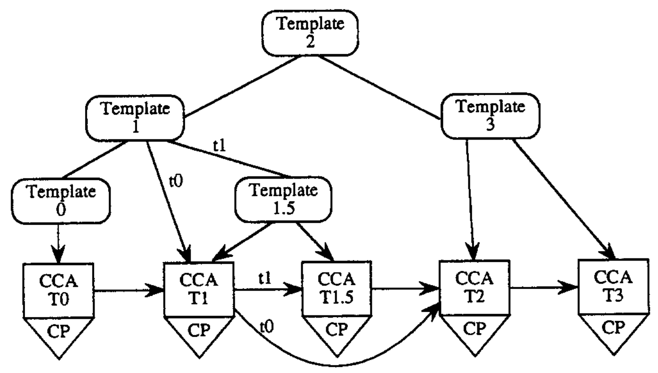


Figure 6. An example of template-insertion (Template 1.5 is inserted)

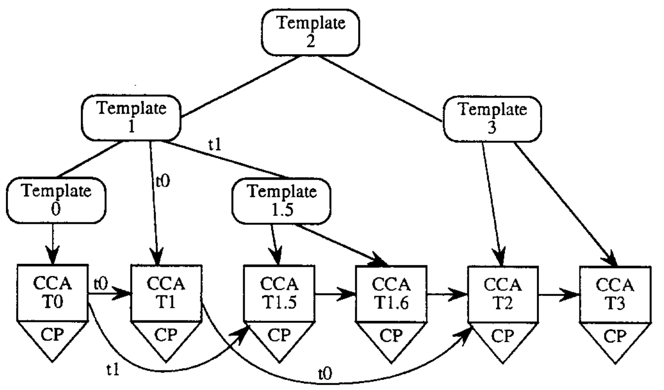


Figure 8. An example of template split (Template 1 splits into two templates at t_l)

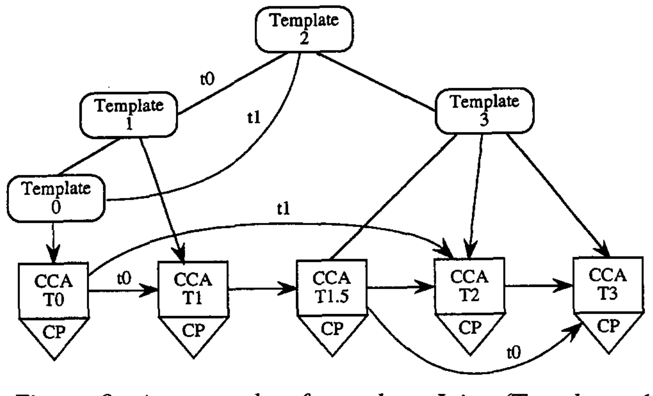


Figure 9. An example of template-Join. (Templates 1 and 2 are joined at t_1)

more efficient than the other similar methods on those type of operations that involve more than one variable and template. The idea of a persistent symbol table and its structure, as well as the access and update operations, is also presented.

SCM receives the program id(s) from IDM. Using these id(s), each of which is unique in the system, the user/programmer selects the desired version or may browse through the different versions of the program to make a selection. The principal contribution of this approach may be stated as its utility in the design of an environment to reuse existing software. The major contributions of this approach could be classified into four different categories: (a) micro-incremental reuse capabilities; (b) micro-incremental compilation; (c) version control and system management; and (d) a module/template interconnection language. What follows is a brief discussion of the contributions of this paper.

Selecting templates as the secondary units of reuse promotes micro-incremental reuse in a systematic way. Language-dependent templates are natural units of a programming language and simplify the process of syntax and semantic checking. Furthermore, devising a Persistent Network and a Persistent Symbol Table simplifies the micro-incremental compilation/recompilation process.

Application of the hierarchal Version-Persistent scheme in the version control structure provides an efficient and dynamic version control management system and supports direct and sequential access to the versions of a module/template.

The ROPCO environment is under implementation and testing. The Version Control Tree, Identification Mechanism, PNET, PSBT, and tools for reverse engineering are being implemented in the Department of Computer Science of the University of Nebraska at Omaha and the Computer Science Department of Oklahoma State University[19,35].

ACKNOWLEDGEMENTS

The authors acknowledge the contribution of D D Fisher and K M George in the development of basic ROPCO environment.

REFERENCES

1 **Tracz, W J** 'Software reuse: motivations and inhibitors' *Proc. COMPCON87* (February 1987)

2 **Biggerstaff, T J and Perlis, A J** *Software reusability* Vol I, ACM Press (1989)

3 **Kaiser, G E and Garlan, D** 'Systems from reusable building blocks' *IEEE Software* (July 1987) pp 17–24

4 **Zand, M K, Samadzadeh, M H and George, K M** 'ROPCO—an environment for micro-incremental reuse' *Proc. IEEE Int. Phoenix Conf. Computers and Communications*, Scottsdale, AZ (March 1990) pp 347–355

5 **Freeman, P** 'Reusable software engineering: a statement of long-range research objectives' *Technical Report 159* Depart. Inf. and Comp. Science, University of California, Irvine, CA (November 1980)

6 **Burton B A and Aragon, R W** 'The reusable software library' *IEEE Software* (July 1987) pp 25–33

7 **Lenz, M, Schmid, H A and Wolf, P F** 'Software reuse through building blocks' *IEEE Software* (July 1987) pp 34–42

8 **Krueger, C W** 'Models of reuse in software engineering' *Technical Report CMU-CS-89-188* Dept. Comp. Science, Carnegie Mellon University, Pittsburgh, PA (December 1989)

9 **Prieto-Diaz, R** 'Classification of reusable modules' in **Biggerstaff, T and Perlis, A (eds)** *Software reusability* ACM Press (1990) pp 99–123

10 **Goguen, J A** 'Reusing and interconnecting software components' *IEEE Computer* Vol 19 No 2 (February 1986)

11 **Zand, M K, Samadzadeh, M H and George, K M** 'Minimizing ripple recompilation in a persistent software environment' *Proc. ACM Comp. Science Conf.* Washington DC (February 1990) pp 166–172

12 **Dart, S A** 'The past, present, and future of configuration management' *Technical Report* Carnegie Mellon University, Pittsburgh, PA (July 1992)

13 **Baalbergen, E H, Verstoep, K and Tanenbaumn, A S** 'On the design of the amoeba configuration management' *Proc. 2nd Int. Workshop on Software Configuration Management*, Princeton, NJ (October 1989) pp 15–22

14 **Thomas, L,** 'Version and configuration management on a software engineering database' *Proc. 2nd Int. Workshop on Software Configuration Management* Princeton, NJ (October 1989) pp 23–25

15 **Crane, B M and Pal, A** 'Conflict management in a source version management system' *Proc. 2nd Int. Workshop on Software Configuration Mangement* Princeton, NJ (October 1989) pp 149–151

16 **Andrews, T, Harris, C, IBM Almaden Research Center, and Sinkel, K** Ontologic, Inc., 'ONTOS: a persistent databse for C++ *Object-oriented database with applications to CASE, networks, and CLSI CAD* Prentice-Hall (1990)

17 **Bacilhon, F, Barbedette, G and Benzaken, V** 'The design and implementation of O2, an object-oriented database system' *Proc. 2nd Int. Workshop on Object-Oriented Database Systems* Springer-Verlag (1988)

18 **Croft, W B and Turtle, H R** 'Retrieval of complex objects' *Proc. 3rd Int. Conf. Extending Database Technology*, Vienna, Austria (March 1992)

19 **Swanson, L E and Samadzadeh, M H** 'A reusable software catalog interface' *Proc. 1992 ACM/SIGAPP Symp. Applied Computing (SAC'92)* Kansas City, MO (March 1992) pp 1076–1083

20 **Atkinson, M P, Baily, P J and Chisholm, K J** 'An approach to persistent programming' *Computer J.* Vol 26 No 4 (1983) pp 360–365

21 **Cockshot, W P, Atkinson, M P and Chisholm, K J** 'Persistent object management system' *Software-Practice and Experience* Vol 14 (1984) pp 49–71

22 **Atkinson, M P, Chisholm, K J and Cockshott, W P** 'CMS—a chunk management system' *Software-Practice and Experience* Vol 13 (1983) pp 273–285

23 **Sarnak, N and Tarjan, R E** 'Planar point location using persistent search trees' *Comm. ACM* Vol 29 No 7 (July 1986) pp 669–679

24 **Tichy, W** 'RCS—a system for version control' *Software-Practice and Experience* Vol 15 No 7 (July 1985) pp 634–637

25 **Zand, M K and Fisher, D D** 'Space-efficient persistent B-trees' *Proc. 2nd Workshop Applied Computing* Stillwater, OK (March 1988) pp 295–318

26 **Cole, R** 'Searching and sorting similar lists' *J. Algorithms* Vol 7 (1986) pp 202–220

27 **Zand, M K, Saiedian, H and Farhat, H** 'A persistent quad-tree to store graphic images' *J. Congressus Numerantium* Vol 81 (1991) pp 173–182

28 **Reps T and Tentelbaum, T** 'Incremental context-dependent analysis for language-based editors', *ACM Trans. on Prog. Lang and Systems* Vol 5 No 3, pp 449–477 (July 1983)

29 **Teitelman, W and Mainster, L** 'The Interlisp programming environment' *Computer* Vol 14 No 4 (July 1985) pp 637–654

30 **Cheatham, T E Jr,** 'Reusability through program transformations' *IEEE Trans. Soft. Eng.* Vol 10 No 5 (September 1984) pp 589–594

31 **Marzullo K and Wiebe, D** 'Jasmine: a software system modeling facility' *Proc. ACM SIGSOFT/SIGPLAN Conf.* (November 1988) pp 28–30

32 **Neighbors, J M** 'The DRACO approach to constructing software from reusable components' *IEEE Trans. Soft. Eng.* Vol 10 No 5 (1984) pp 564–574

33 **Zand, M K, Samadzadeh, H, George, K M and Saiedian, H** 'An interconnection language for reuse at the template/module level' to appear in *J. Systems and Software* (1993)

34 **Fischer, G and Johnson, C** 'A meta-language and system for nonlocal incremental attribute evaluation in language-based editors' *Conf. Records 12th Ann. ACM Symp. on Prin. of Prog. Lang* New Orleans, LA (January 1985) pp 141–151

35 **Zand, M K and Hiesterkamp, D** 'A reverse-engineering tool for the ROPCO environment' (work in progress) Dept. of Math & Computer Science, UNO, Omaha, NE (1993)