Using the Temporal Versions Model in a Software Configuration Management Environment

Fabrício Ávila da Silva, Raquel Vieira Coelho Costa,
Nina Edelweiss, Clesio Saraiva dos Santos

Instituto de Informática – Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500 – Bloco IV – Porto Alegre, RS, Brazil
CEP 91501-970 – Caixa Postal: 15064
{favila, raquelc, nina, clesio}@inf.ufrgs.br

Abstract. Temporal Versions Model (TVM) is an object-oriented data model with versioning facilities, allowing revisions and project alternatives, and temporal features, used to maintain the history of the system evolution. This paper presents TVM under a Software Configuration Management (SCM) perspective, compares this model with available SCM tools, and describes why TVM is adequate to be used as the basis of an SCM environment. A modeling example and the implementation of TVM on a commercial Database Management System (DBMS), which is an ongoing work, are also presented.

Key words. Software Configuration Management, Temporal Versions Model, object-oriented modeling.

1 Introduction

Versioning of software components have always been one the most important issues addressed by Software Configuration Management (SCM) tools and researchers. In the last decades, the basic concepts about it were defined and well formalized, although some new approaches and techniques have been proposed recently.

Temporal Versions Model is an object-oriented data model with versioning facilities, implementing the most traditional and natural strategies for entities versioning. The model also presents temporal features, which have not been implemented before in SCM environments. The union of these characteristics constitutes a model with unique qualities, allowing versioning of objects and, within each version, the keeping of all modifications done on its dynamic attributes and relationships.

The paper is structured as follows. Section 2 presents the reasons TVM is being proposed to be used in an SCM environment. The model is introduced in section 3, according to configuration management concepts. Section 4 briefly compares TVM to version models found in available SCM tools. A modeling example is shown in Section 5. Some details about TVM implementation on a commercial DBMS are presented in Section 6, and in Section 7 there are some concluding remarks.

2 Motivation

Software Configuration Management (SCM) is the software engineering area responsible for controlling the evolution of complex systems. Formally, it is the discipline based on which developers can maintain under control the evolution of large and complex software systems [1]. A software configuration management system represents one of the most important aspects involved in any software development process. Currently, a typical SCM system should provide services in the following areas [2]:

Managing a repository of components: the system must store several types of software components, in a safe and efficient way. This area includes versioning, product modeling and complex objects management.

Help engineers in their usual activities: SCM systems must provide engineers with right objects in the right place. This item refers to workspace control, besides also considering compilation and derived objects control.

Process control and support: defines what has to be done on which object (a process model), and the utilized mechanisms to help or force the usage of this model.

This paper is focused on the first case, where the concepts of versioning, product space, and their integration are important issues to be considered.

The first configuration manager systems, like RCS [3] or its successor CVS [4], applied the versioning concepts on single files, and used different techniques for building composite components and configurations (like selecting determined files based on tags recorded with them). Files are individually versioned, making very difficult the building of complex objects. This approach is not adequate to completely support software development, because it does not consider system components in a proper manner nor the other artifacts involved in the development process, like test reports and software documentation. Even today, commercial SCM systems only capture the files and directories that represent a software product, barely storing relationships and dependencies between them [5, 6]. A weak data model raises lots of problems, like the treatment of configurations and versions as “something” special and not first-class entities, making impossible for them to play roles directly in relationships [2].

These issues fostered a number of works about advanced version and data models, including [7, 8, 9, 10, 11, 12, 13]. Some approaches define methods for versioning every entity stored in the repository, including attributes, relationships and configurations. These works, among others, proposed interesting solutions, but from a practitioner point of view, they are too complicated or inefficient, providing more power than actually needed [5].

It has been pointed out in the last years that a not so complex data model should be used, supporting typed objects, relationships, attributes, and a uniform manner of identifying uniquely components and its versions in the repository [14, 15]. In the Component-Based Software Development paradigm (CBSD), for example, users see each component as a primitive item, which may be implemented as a set of files. Management of composite components and relationships between them is a basic task for an SCM environment tailored for CBSD [16].

So far there is no commercial database able to support such advanced models [5]. TVM is a conceptual model considered as a solution for the problems presented above, and its implementation within a commercial database system can provide the basis for the development of an efficient SCM environment.

3 Temporal Versions Model

TVM (Temporal Versions Model) [17, 18] is an extension of Golendziner's Versions Model [19], incorporating temporal features. Its main characteristics are presented here. It is an object-oriented model with versioning and temporal characteristics, allowing users to store project alternatives (object versions) and, for each one of these alternatives, the history of its dynamic attributes and relationships.

Various versioned object-oriented models have been proposed, as well as temporal ones, for example [11] and [20], respectively. The union of both properties provides the user a more flexible and powerful data model, which can be used in an SCM system to represent software product, project workers and all the software artifacts, like documentation and system requirements.

The main concepts of data and version models for configuration systems have been formalized in several works [2, 21, 12, 22]. TVM will be described concerning these topics, facilitating the comparison between TVM and some available SCM tools presented in Section 4.

3.1 Product Space

The product space describes the software under development with its structure and components, not taking into account if they are versioned or not. TVM can be considered a *domain-independent model*, since its object-oriented features allow the modeling of any kind of software artifact – documentation, source code, project plan, etc.

The software product can be visualized as the classes' instances and its relationships, which can be of two types: association and aggregation. Dependencies between modules (objects) are represented by association relationships, whose names and cardinalities can be defined by the user to customize the software representation. The aggregation relationship allows the construction of composite objects. Figure 1 illustrates a software product represented in TVM.

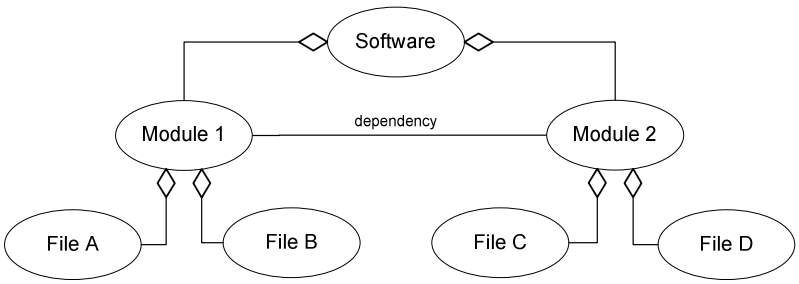

Fig. 1. A sample of a software product in TVM.

3.2 Version Model

The version model defines the software objects that can be versioned, the way versioning is implemented and the manner the versions are arranged.

TVM supports *state-based versioning*, where each version represents a state of the versioned object. The model does not define differences between two versions, at least conceptually. TVM provides *extensional versioning*, where all versions are explicitly stored and can be retrieved at any time.

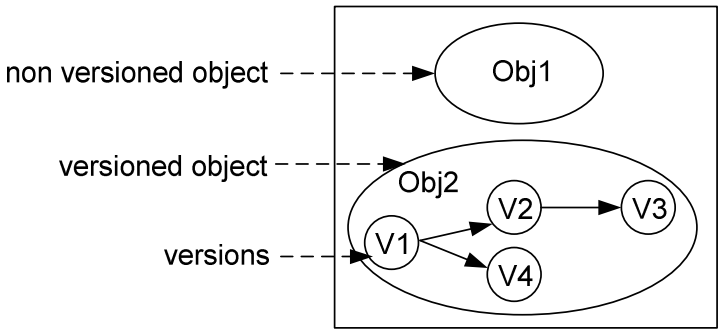

Fig. 2. Versions graph in TVM.

The version space is defined by a version graph, where each node is a version and the edges represent the derivation relationship. It is, in fact, a directed acyclic graph, since TVM supports the *merge* operation (the creation of a new version derived from two predecessors). *Revisions* (a version created in order to supersede its predecessor) and *variants* (project alternatives or collaborative work supporters, which create branches) are treated uniformly. Versions of the same instance of a class are kept together in a *versioned object* that holds common properties and information about its associated versions. The creation of a version from an object yet without versions causes this object to become the first version and the new one is derived from it. Figure 2 presents a graphical example of the possible instances in a TVM application. There are a non-versioned object and a versioned object, with its respective versions.

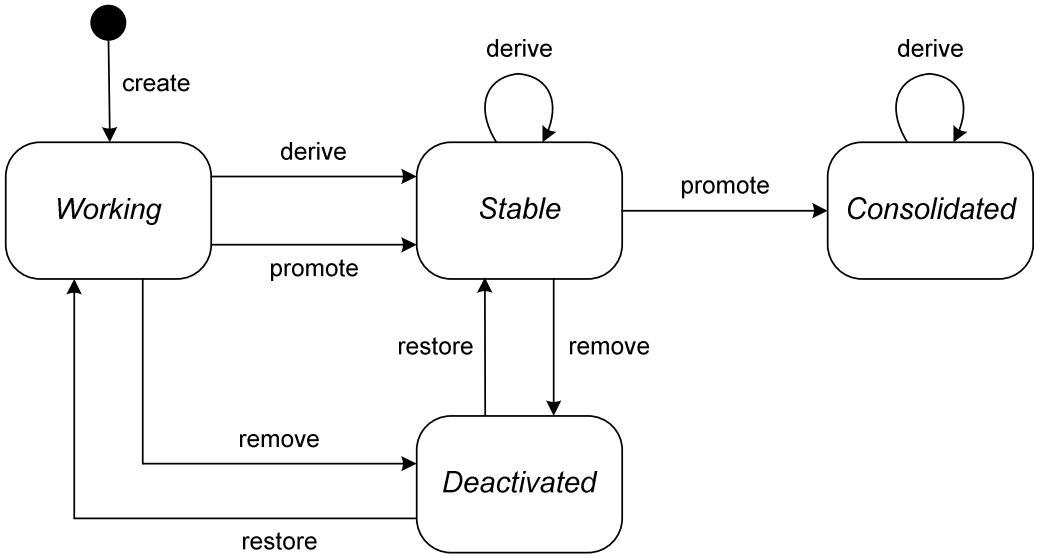

Fig. 3. TVM statuses diagram.

During its lifetime, a version can assume different statuses (*working*, *stable*, *consolidated* and *deactivated*). Transitions between them and the respective events are presented in Figure 3. The versions' immutability property is guaranteed by constraints associated to each one of these statuses. When created, a version assumes the status *working*, meaning that it can be modified, queried, removed, and derived. A derivation creates a new *working* version and automatically promotes its predecessor to *stable*, in order to avoid modifications that would compromise the history of the object. In the *stable* status, the version can be derived,

promoted to *consolidated*, queried, and removed (if there is no successor), but cannot be modified. Once *consolidated*, a version can be queried and derived, but cannot be modified nor removed. There are no physical removals predicted in TVM; the remove operation moves the version to the *deactivated* status and finishes its lifetime. In this status, the version can only be queried or restored.

3.3 Version Model and Product Space Integration

Non-versioned objects and versioned ones can coexist in the same database. Figure 4 presents the TVM class hierarchy. The model allows the definition of two application class types:

- *Non temporal nor versionable application class*, defined as subclass of *Object*; used to model classes in which time and version concepts are not necessary. It also allows the integration with other models that do not present versioning and temporal features.
- *Temporal and versionable application class*, defined as subclass of *TemporalVersion*. Its instances are versions and its attributes and relationships can be defined as *static* or *temporal*. The temporal aspects will be discussed in a forward section.

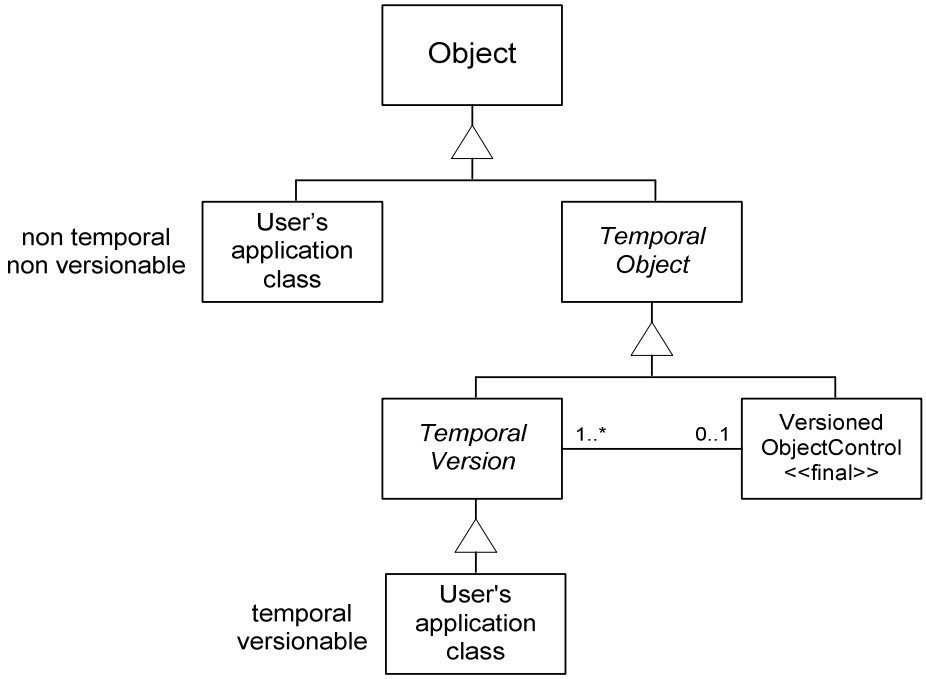

Fig. 4. TVM classes hierarchy.

Non-versioned objects, versioned ones and versions themselves can be directly manipulated or queried, since the model identifies them in a uniform manner. The OID (Object Identifier) is a structure that contains the following information:

```
< Entity identifier, Class identifier, Version number >
```

This structure allows the existence of several objects composing the same entity, since the *class identifier* and *version number* guarantee the OID uniqueness. This approach allows the construction of relationships between non-versioned objects and versions directly (*static reference*) or, yet, versioned objects (*dynamic reference*, where the version to be used is defined as the current one).

TVM implements *total versioning*, admitting that an object at any level of the composition hierarchy be versioned (in contrast to *component versioning*, where only atomic objects can be versioned).

TVM supports two kinds of inheritance: inheritance by refinement and inheritance by extension, presented in [23]. The former is the traditional one, corresponding to the *is-a* relationship between objects. The latter allows the description of an entity in several levels of the hierarchy, and an object at any of these levels can be versioned. The union of the objects of these different levels represents the complete modeled entity.

Figure 5 presents an example that evidences the differences between the different inheritance types in TVM. Inheritance by refinement (a) consists in deriving an object from the leaf element of the modeling hierarchy. This object stores the properties declared in its own class, as well as the ones defined in its ascendant class. On the other hand, inheritance by extension (b) determines that all attributes' values will be stored at the level in which they were declared, being these properties shared by every descendent objects that might exist.

In this example, imagine that an employee is hired and, some time after that, he is promoted to manager. The utilization of inheritance by refinement implies in deleting the employee and creating a new object to represent him, this time an instance of the class *manager*. This is not a good solution, since the employee existed previously in the base and had its own identifier, generated automatically by the system. If inheritance by extension had been used from the beginning, the new object of the class *manager* would simply be linked to its ascendant, created from the class *employee*. The union of the objects from both classes represents the complete entity, in this case an employee who is also a manager.

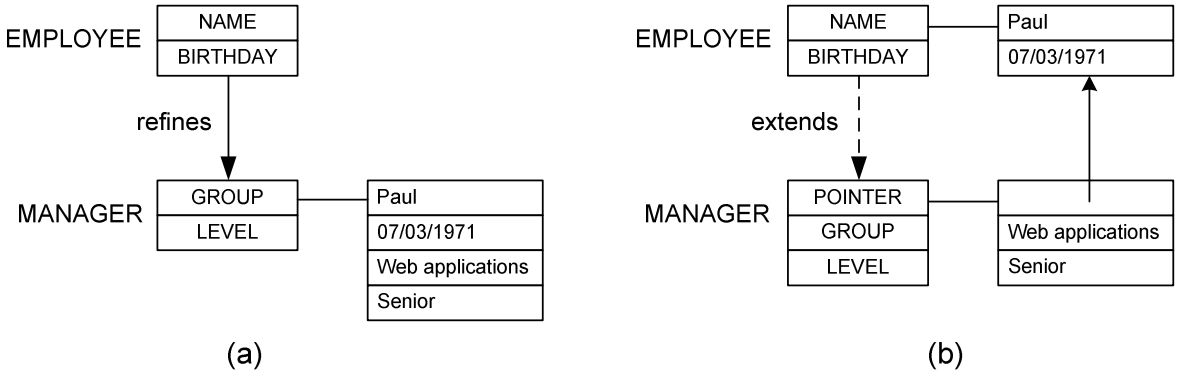

Fig. 5. Inheritance by refinement (a) and inheritance by extension (b).

A configuration is a set of different objects' versions that together make up a complex object. For instance, one can combine a specific version of each system module in order to make up a complete software product. In TVM, the user executes the operation *getConfiguration* on a version (called *base version*) to produce a configured version for it. Then, one version for each existing ascendant in the inheritance by extension hierarchy is selected (by expressions or pre-defined criteria), as well as one version for each object in the aggregation hierarchy. For every chosen version, a new one is derived, called a *configured version*, with its own OID. Thus, versions and configurations are treated uniformly in TVM, since a configuration is nothing else but a set of versions connected by relationships. The complete software product (or a component, depending on the abstraction level the configuration was created) can be retrieved by means of applying the method *getCompleteObject* on the configured version. Figure 6 presents an example of a configuration, before (a) and after (b) its creation. Supposing the user solicits the construction of a configuration using the version *a1*, an instance of class *A*, he has to choose one of the

versions from the aggregate class, *B*. In this case, *b1* was chosen, then two new versions, *c1* and *c2*, are derived from the selected ones.

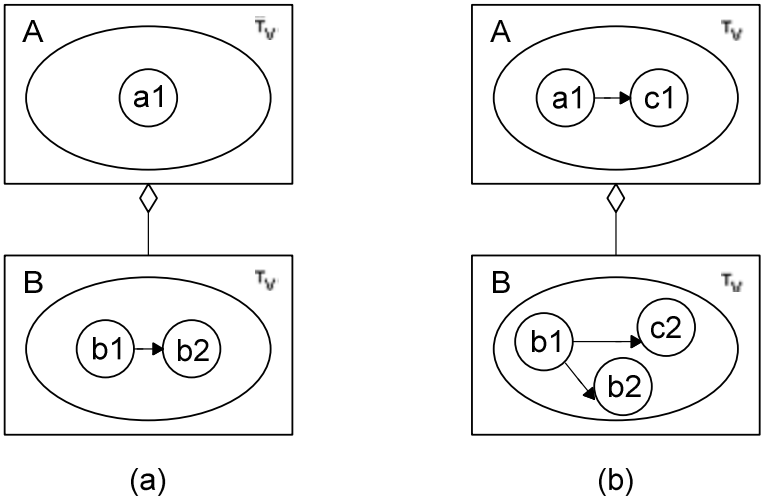

Fig. 6. A configuration, before (a) and after (b) its creation.

3.4 Temporal Features

Temporal aspects have not been yet well explored by currently available SCM systems, and that is one of the main contributions in using TVM within a software configuration management environment.

In TVM, time is associated to objects, versions, attributes and relationships, allowing the storage of all components evolution in a system, whether they are versioned or not. Each version has its own timeline, so that an object can have several valid versions at the same time. This feature is called branched time, because in the same object there can be several lines of evolution.

An attribute or relationship in a *TemporalVersion* class must be defined as static or temporal. Static attributes and relationships behave as traditional ones, which means that any update in their contents will overwrite the previous values. Modifications are only stored by means of versioning. On the other hand, updates made on temporal attributes and relationships of a single version are stored, allowing the system to keep all the modifications performed on the object. The definition of attributes and relationships as static or temporal is under user responsibility, during the modeling phase. A class may hold attributes and relationships of both types.

The temporal variation is discrete, and temporality is represented through intervals (sets of consecutive equidistant time instants). TVM supports two time dimensions – to every temporal data stored is associated a bitemporal timestamp, which contains:

- *Valid time*: defining when the described fact becomes effective in reality;
- *Transaction time*: informing when the new value was defined in the database.

There are some temporal integrity rules considered by the model, concerning the objects lifetimes and the operations executed on temporal information (insert and update).

TVM does not define rules for object physical exclusion, since it aims to keep the whole application history. The model allows only logical exclusion, when any open valid or transaction time receives the same final time value defined for the object.

A deeper discussion on temporal issues can be found in [24] and [25].

3.5 Query Language

In order to extract data from a TVM base, a query language called TVQL – Temporal Versioned Query Language – was defined [26]. It is based on SQL, in a way that static elements and temporal versioned ones may be treated uniformly. A query whose contents does not hold any restriction or clause concerning a temporal or versioning characteristic will return, by default, the current values stored in the last versions of each object, behaving, therefore, as a traditional SQL query. Just as in SQL, a common TVQL query is composed of the basic structure: `SELECT <list of rows> FROM <list of tables> WHERE <conditions>`. Of course all the SQL operators (+, –, *in*, *between*), aggregation clauses (*count*, *sum*), and other reserved words (*group by*, *order by*) can also be used in TVQL statements.

In order to return temporal values, the special clauses EVER and PRESENT may be used. EVER can appear after SELECT (to return as the query results the complete history of modifications made on the selected attributes) or WHERE (to consider not only current values, but everything stored in the base). A query like the following: SELECT EVER (…) WHERE PRESENT (…), returns historic values according to current values. Each temporal attribute (or relationship) has some temporal properties associated to it, for example *viInstant* and *vInterval* (Valid Initial Instant and Valid Interval). These properties can be referenced in WHERE clause to get valid values at specific instants or intervals, as well as they can appear in SELECT clause to return the period during which some information were valid.

If the objective is considering all the versions stored in the base, the key word VERSIONS must appear after the correspondent table name. The user can, yet, use several properties in order to reference specific versions or to get other information. For example, the property *currentVersion* returns the version defined as the current one, as well as *versionCount* provides the quantity of versions stored in the referenced object.

Naturally, temporal and versioning characteristics can be combined in the same statement, building more complex queries. Some examples will be presented in the section 5.

4 TVM and SCM Tools

An extensive study about version models in SCM environments and a correspondent taxonomy for classifying them is presented in [21]. We present here the TVM features concerning this taxonomy, comparing this model with some of the available SCM tools analyzed in that paper. Some concepts described in [21] are not taken into account in our analysis because they do not apply to TVM, as, for example, the *intensional versioning* aspects. As a conceptual model, TVM does not specify anything about the implementation of delta algorithms.

The following features are used to compare TVM to SCM tools.

Object Management. Most SCM systems use a file system to manage the software objects (RCS, Aide de Camp, DSEE, ClearCase). An SCM tool implemented on TVM will obviously use a database system.

Product Space Domain. TVM applies to a *general domain*, since the model supports the representation of any kind of object. Some SCM environments deal only with specific types of software objects, like Gandalf and POEM.

Product Space Granularity. TVM, just as almost all the analyzed systems, deals with both granularity levels, coarse and fine.

Relationships. According to the taxonomy, relationships can be of two types: composition and dependency. Every tool cited in [21] supports compositions, but some of them do not take dependencies into account, like RCS and PCL. TVM, as an object-oriented data model, provides the aggregation relationship explicitly. Dependencies are represented by associations with names and cardinalities customized by the user, so that any kind of relationships can be modeled.

Version Space Structure. The version space can be represented by a version graph or a grid. Some tools support both, like Adele, but most of them uphold just one. TVM uses a version graph.

Version Set. TVM utilizes the most traditional approach, supported by almost every tool, called *extensional versioning*. The *intensional versioning* approach is also used by several systems, for example ICE and ClearCase.

Version Specification. The *state-based* strategy is found in almost every tool, as well as in TVM. A few systems implement the *change-based* approach, for instance Asgard and Aide de Camp.

Granularity of Versioning. *Component versioning* means that only atomic objects can be versioned (RCS and Inscape). TVM supports *total versioning*, where all levels of the composition hierarchy can be put under version control. A few tools use the *product versioning* approach, for example PIE and COV.

TVM consolidates twenty years of research and practice by supporting the basic principles of a version model, like state-based versioning and extensional versioning. Its version space is represented by a version graph and its domain is not specific for determined types of software components. Composition and dependencies between objects are achieved through the use of aggregation and relationships, respectively.

Instead of classical systems, like RCS [3], based on the check-out/check-in model, the versions in TVM are created explicitly by the user, through the operation *derive*. The statuses assumed by a version during its lifetime provides immutability in a different approach; while in the *working* status, a version can be freely modified, and TVM will not create a new version until the user commands it. Once *stable*, a version can no longer have its attributes modified, guaranteeing the immutability property.

In TVM, the version model is built into the data model, like in DAMOKLES [8] and Adele [7]. A commercial DBMS is being extended to support the model characteristics, providing all the benefits a database can provide to an SCM environment, like durability of changes and transaction facilities.

The control of temporal dimension in SCM systems is usually underestimated [7]. In TVM, the complete history of changes made on an attribute (or relationship) can be stored by defining it as a *temporal* one. The possibility of recording the whole history of modifications

done on a single version during its lifetime is a facility not offered by traditional SCM systems. Using TVM, the user do not have to add a version when he only wants to keep track of his modifications. A developer can work on a single version for a long time and derive a new revision from it when its code is stable (this approach is supported by the statuses diagram). His work will be stored by the system and any past state of the version can be retrieved using the temporal facilities offered by the model. These features allow the aggregation of conceptual meaning to each revision created in the versions graph. For example, each revision in a project may represent the work of a developer (considering that two or more professionals may work in the same artifact), or each revision may represent a new functionality added to the object. In traditional systems, a functional update made on a software element is usually represented as several revisions in order to maintain the object evolution during its development. Using TVM, only one version will be stored by the system and the history of modifications is held through temporal facilities associated to object's attributes and relationships. With this approach in mind, a developer can construct a versions graph clean and well organized, creating revisions and variants only when there is a logical reason for doing so.

The utilization of versions and time in a single project modeling is exemplified in the next section. Besides that, TVM automatically associates temporal information to every version and versioned object created.

5 A Modeling Example

Table 1 presents the symbols defined to represent graphically, in a class diagram, the *TemporalVersioned* classes and its temporal attributes and relationships.

Table 1. Symbols defined for graphical representation.

Symbol	Meaning
T_V	The associated class is temporal and versionable
<<Temporal>>	The associated relationship is temporal
<<T>>	The associated attribute is temporal
<<extension>>	Inheritance by extension

Figure 7 presents an example of an SCM project modeled in TVM. The software product under development is represented by the class *SoftwareProduct*, which is an aggregation of instances of the *Module* class. A component is composed by several files (class *File*), and a set of components (class *Component*) makes up a module. The *Documentation* class holds information about the other artifacts involved in the software development, like system requirements and product manual. This example does not specify anything about the documentation components in order to not make it too complex. These classes are naturally *TemporalVersioned*, because their instances will be versioned during the software development process.

The classes *ProjectManager*, *Programmer* and *Analyst* are subclasses of *Employee*, which is *TemporalVersioned* because some of its attributes may be temporal ones, for instance *email*. Besides that, its relationships are defined as *Temporal*, in order to maintain the history of changes; for example, one programmer can begin the construction of a module and another one may substitute him later. The project manager is responsible for the software product (the highest level in the software hierarchy), and the analyst formalizes the documentation.

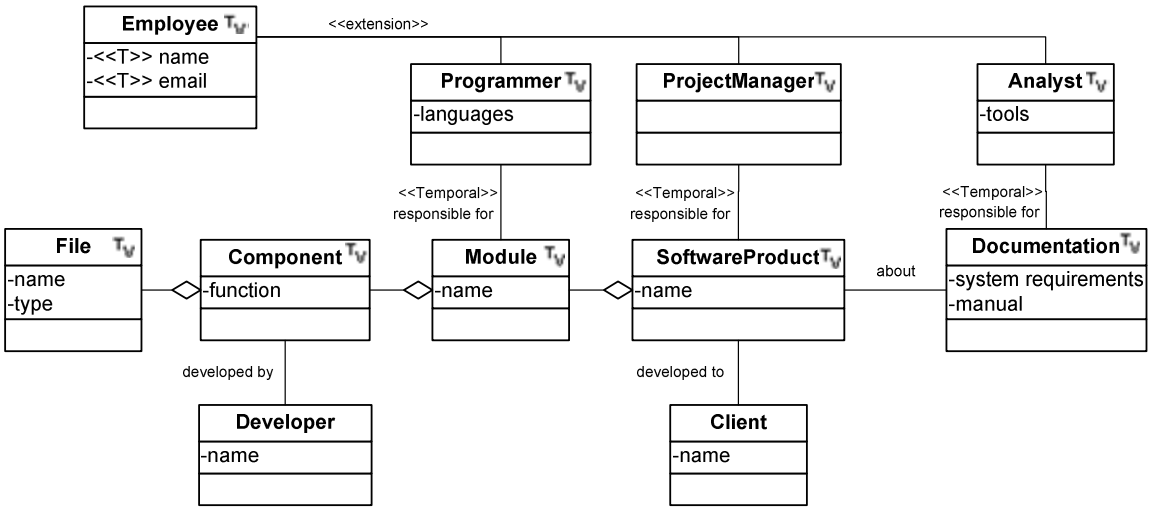

Fig. 7. Example of a software project in TVM.

Most of the classes' attributes are not shown in the figure to simplify the visualization of the classes and their relationships. The class *Developer* holds information about other companies whose components are used in the software product, supporting the reuse of elements (a basic requirement for component-based software development). The class *Client* has a relationship with *SoftwareProduct*, indicating who ordered the software development. *Developer* and *Client* are regular classes (non temporal nor versionable) because there is no need to store modifications made on its instances, since they represent real world entities that do not belong to the software project.

In this example, the files are represented as first-class objects, and a component is composed of a set of files. The same result could be obtained (from a conceptual point of view) if the files were defined as attributes of the *Component* class. This situation shows that TVM is a generic model; an SCM environment can personalize the way of representing software and other system objects.

TVM does not specify any mechanism for controlling the access to specific objects in the base. However, an SCM environment built on top of the model could use information about employees to administrate the system users and their rights.

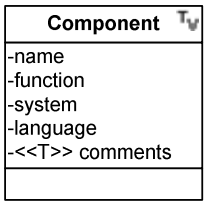

Fig. 8. *Component* class.

An example concerning instances evolution from this classes diagram can demonstrate some of the TVM features when applied to software development. Figure 8 presents the *Component* class with some attributes that were hidden in the previous classes diagram. The properties *name*, *function*, *system*, and *language* behave as in the traditional way. Considering an object whose status is *working*, these attributes can be freely modified without storing their historic values. The property *comments*, on the other hand, is defined as *temporal*. This way, every update will be kept by the system, recording all the comments made by the programmers during the object's life time.

Figure 9 presents *Component* class' three instances, and all versions of the *InterfaceHM* (interface human-machine) object. This component was developed to be reused in several software projects, so that it holds a version for each operational system. The *comment* attribute is not shown in the figure because its value varies within a single version.

The version number 2 (OID = 3,6,2) was derived from the first one, whose attributes *system* and *language* hold the null value (since this version is only a base for development), and implements the component in C++ for the Windows 2000 operational system. For Windows XP, two different versions were developed: one in C++ e the other one in Java, both derived from Windows 2000 version because they share some characteristics. After a while, the version in Java received an important improvement, consequently creating a new version from it. The seventh version was developed under the same circumstances, but for the Linux operational system.

The utilization of TVQL provides the user a simple way of getting the desired version of a component, without worrying about the version's number or its position in the version graph. For example, the following query returns the component's version developed for Windows 2000:

```
SELECT *
FROM Component.versions
WHERE system = 'Windows2000'
```

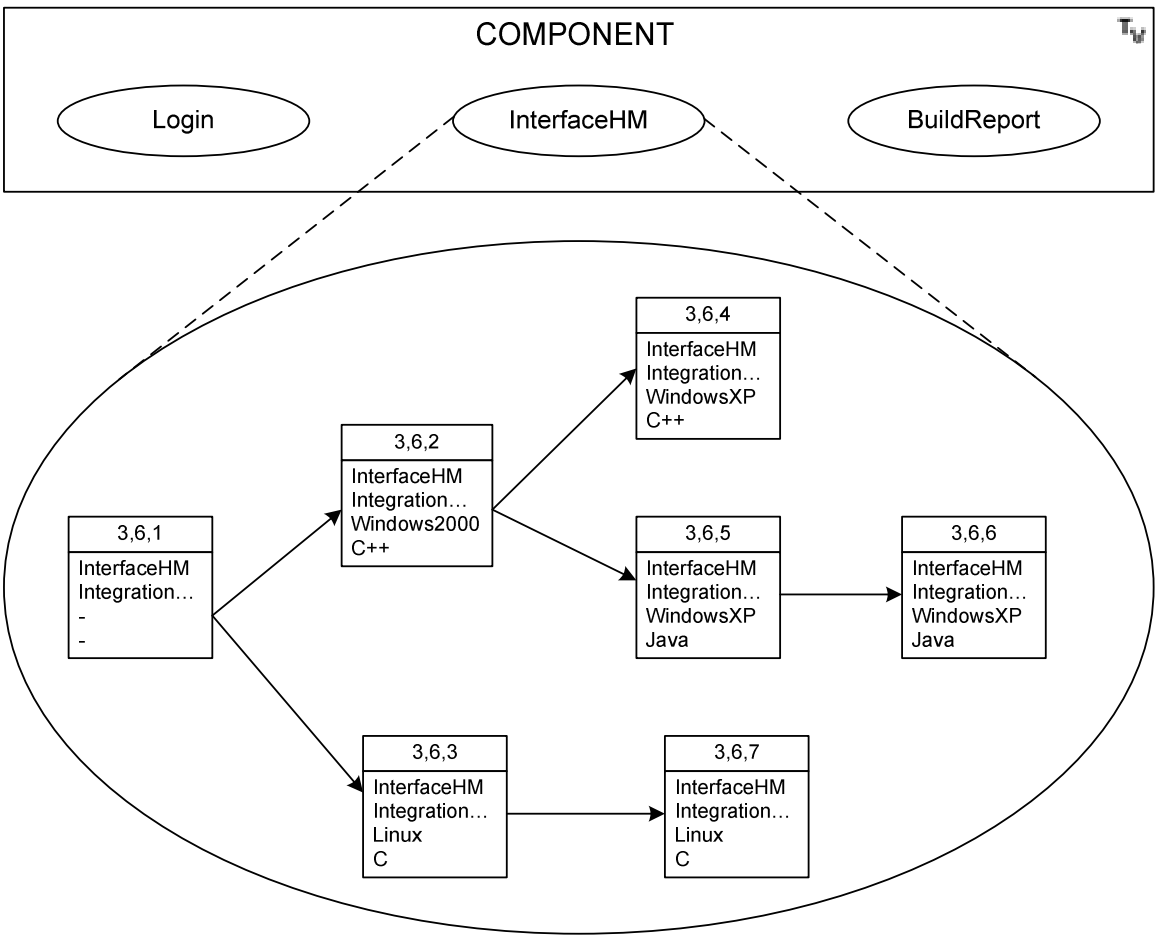

Fig. 9. *InterfaceHM* object's versions

Two versions were developed for the Windows XP operational system, each one in a different programming language. In order to obtain the version in C++, the following query must be executed:

```
SELECT *
FROM Component.versions
WHERE system = 'WindowsXP' AND language = 'C++'
```

The query below obtains all the versions currently under development. In this situation, it would return the versions 6 (3,6,6) and 7 (3,6,7), since they were not promoted yet to state *stable*.

```
SELECT *
FROM Component.versions
WHERE isWorking
```

To get the whole history of stored values in the *comments* attribute of any version, the following query might be used:

```
SELECT EVER comments
FROM Component.versions
```

Obviously, this query would not be very useful, since it returns every registered value in any component developed at any time. The user should, then, combine temporal and versioning features, building a query like the one below. It returns the history of modifications made on the *comments* attribute during the development of the component for Linux:

```
SELECT EVER comments
FROM Component.versions
WHERE system = 'Linux'
```

These examples show that temporal characteristics and traditional versioning techniques can be combined to provide a new dimension to software development, simplifying visualization and understanding of the development process.

6 Implementation of TVM on a Commercial Database

TVM is defined as a conceptual model, allowing future adaptation to different database systems. To validate this approach, a prototype on IBM DB2 is being implemented, which allows the main operation of TVM. This project is based on some works concerning temporal databases implementation, for example [27].

One of the reasons that contributed for choosing this DBMS is the fact that it is the most similar to the SQL-92 pattern concerning its support for temporal data types. Besides, another strong characteristic of DB2 is its extensible architecture, through the extenders facilities. These extenders explore the object-relational features of DB2, including new functionalities for various application domains. Each extender defines new data types, offering functions to create, update and delete data. The extension of TVM for DB2 is called TVM Extender and consists of a set of mechanisms (UDTs – *User Defined Types*, UDFs – *User Defined Functions*, triggers, stored procedures and constraints) and metadata that maps the model hierarchy and manages the time and version aspects of data.

The TVM Extender supports the model's main features. In a first version of this extender, configuration concepts are not included, as well as inheritance by extension, for not making it

too complex. As these aspects are independent of the model core, their future insertion will not damnify the basic operation of the system.

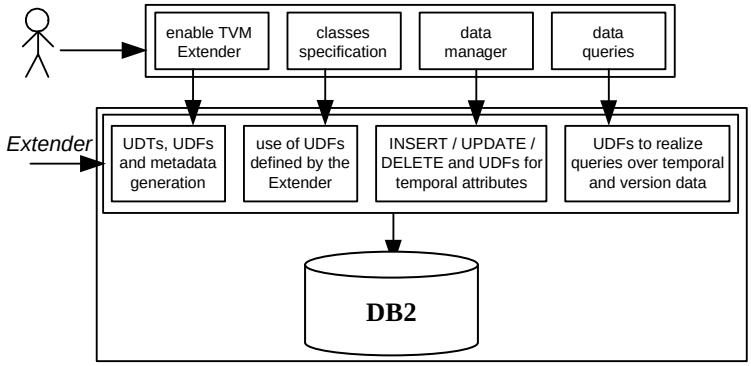

Fig. 10. TVM Extender and its interactions with the user.

As DB2 just allows extending SQL by user-defined functions, it is not possible to modify the SELECT syntax itself, as in TVQL. It is necessary to define UDFs that return values regarding temporal and version concepts, that can be used in SELECT, FROM or WHERE clauses.

Figure 10 illustrates TVM Extender functions and their respective interactions with the user. Although it executes actions on DB2, the extender seems to work inside the database from the user's point of view.

7 Concluding Remarks

Temporal Versions Model is an object-oriented model with temporal and versioning features. TVM consolidates the research developed about versioning in the last twenty years, with a simple understanding and very flexible model. The temporal features allow the control of modifications made on a single version, keeping the whole history of changes on dynamic attributes and relationships, independent from the versioning aspects.

A Software Configuration Management system built on top of TVM can use the resources provided by the model and personalize objects and relationships to represent the development of software products in a simple and efficient way. The facilities introduced by the temporal characteristics have never been properly explored by SCM tools before; they offer very interesting features concerning the storage of modifications without creating explicit new revisions of a software component.

This paper presented the main features of TVM based on configuration management concepts, facilitating a comparison of the model with available SCM tools. A modeling example has also been shown.

A prototype is being developed within a commercial DBMS in order to test the main characteristics of the model. A complete environment with class specification facilities and an interface to TVQL (TVM's query language) is also under development.

References

[1] Tichy, W. F.: Tools for software configuration management. In: Proc. of the Int. Workshop on Software Version and Configuration Control, Grassau, January (1988).

[2] Estublier, J.: Software Configuration Management: A Road Map. In: Finkelstein, A. (ed.): The Future of Software Engineering (supplementary Proc. for 22nd Int. Conf. on Software Engineering), Limerick, Ireland, ACM Press (2000) 279-289.

[3] Tichy, W. F.: RCS – A System for Version Control. Software – Practice and Experience, 15 (1985) 637-654.

[4] Berliner, B.: CVS II: Parallelizing Software Development. In: Proc. of the 1990 Winter USENIX Technical Conference. Washington, DC (1990).

[5] Estublier, J.: Impact of the Research Community On the Field of Software Configuration Management. Software Engineering Notes vol. 27 no. 5. ACM Press. New York, NY (2002) 31-39.

[6] Frühauf, K., Zeller, A.: Software Configuration Management: State of the Art, State of the Practice. In: Estublier, J. (ed.): Proc. of the 9th Int. Symposium on System Configuration Management, SCM-9. Toulouse, France (1999).

[7] Estublier, J., Casallas, R.: The Adele Configuration Manager. In: Tichy, W. F. (ed.): Configuration Management, Trends in Software vol. 2. Wiley. New York, NY (1994) 99-134.

[8] Dittrich, K., Gotthard, W., Lockemann, P.: DAMOKLES, a Database System for Software Engineering Environments. In: Conradi, R., Didriksen, T. M., Wanvik, D. H. (eds.): Proc. of the Int. Workshop on Advanced Programming Environments. LNCS 244, Springer-Verlag (1986) 353-371.

[9] Boudier, G., Gallo, F., Minot, R., Thomas, I.: An Overview of PCTE and PCTE+. In: Proc. ACM/SIGSOFT Software Engineering Symposium on Practical Software Development Environments. Boston (1988) 248-257.

[10] Munch, B.P.: Versioning in a Software Engineering Database – The Change Oriented Way. Ph.D. Thesis. NTNU Trondheim. Norway (1993).

[11] Lamb, C., Landis, G., Orenstein, J., Weinreb, D.: The ObjectStore Database System. Comm. of the ACM, 34(10) (1991) 50-63.

[12] Conradi, R., Westfechtel, B.: Towards a Uniform Version Model for Software Configuration Management. In: SCM-7 Workshop. Springer LNCS 1235 (1997) 1-17.

[13] Zeller, A., Snelting, G.: Unified Versioning through Feature Logic. ACM Transactions on Software Engineering and Methodology, 6(4) (1997) 397-440.

[14] Conradi, R., Westfechtel, B.: SCM: Status and Future Challenges. In: Estublier, J. (ed.): Proc. of the 9th Int. Symposium on System Configuration Management, SCM-9. Toulouse, France (1999).

[15] Weber, D.W.: Requirements for an SCM Architecture to Enable Component-Based Development. 10th Int. Workshop on Software Configuration Management, SCM-10. Toronto, Canada (2001).

[16] Mei, H., Zhang, L., Yang, F.: A Software Configuration Management Model for Supporting Component-Based Software Development. Software Engineering Notes vol. 26 no. 2. ACM Press. New York, NY (2001) 53-58.

[17] Moro, M.M., Saggiorato, S.M., Edelweiss, N., Santos, C.S.: Adding Time to an Object-Oriented Versions Model. In: Proc. of 12th Int. Conf. on Database and Expert Systems Applications – DEXA 2001. Lecture Notes in Computer Science, vol. 2113. Springer-Verlag. Berlin (2001) 805-814.

[18] Moro, M.M., Saggiorato, S.M., Edelweiss, N., Santos, C.S.: A Temporal Versions Model for Time-Evolving Systems Specification. In: Proc. of the 13th Int. Conf. on Software Engineering & Knowledge Engineering – SEKE01. Buenos Aires, Argentina (2001) 252-259.

[19] Golendziner, L.G., Santos, C.S.: Versions and Configurations in Object-Oriented Database Systems: A Uniform Treatment. In: Proc. of the 7th Int. Conf. Manag. of Data. Pune, India (1995) 18-37.

[20] Kakoudakis, I., Theodoulidis, B.: The Tau Temporal Object Model. Timelab Technical Report, Department of Computation. UMIST, UK (1996).

[21] Conradi, R., Westfechtel, B.: Version Models for Software Configuration Management. ACM Computing Surveys – CSUR, vol. 30. ACM Press. New York, NY (1998) 232-282.

[22] Dart, S.: Concepts in Configuration Management Systems. In: Proc. of 3rd Int. Workshop on Software Configuration Management. Trondheim, Norway. ACM Press. New York, NY (1991) 1-18.

[23] Biliris, A.: Modeling Design Object Relationships in PEGASUS. In: Proc. Data Engineering. Los Angeles, USA (1990) 228-236.

[24] Tansel, C.G.: Temporal Databases – Theory, Design and Implementation. Benjamin/Cummings, Redwood City (1993).

[25] Zaniolo, C.: Advanced Database Systems. Morgan Kaufmann Publishers. San Francisco (1997).

[26] Moro, M.M., Zaupa, A.P., Edelweiss, N., Santos, C.S.: TVQL – Temporal Versioned Query Language. In: Proc. of the 13th Int. Conf. on Database and Expert Systems Applications – DEXA 2002. Aix en Provence, France. LNCS 2453 (2002) 618-627.

[27] Snodgrass, T.: Developing Time-Oriented Database Applications in SQL. Morgan Kaufmann (2000).