Temporal Aspects in Reuse of Requirement Specifications

V. De Antonellis, L. Vandoni

Politecnico di Milano

Abstract. This paper presents a methodological approach to support reuse of requirements specification based on the use of a model for handling temporal information associated to complex objects. The model is used to maintain, in a repository, project histories and reuse histories in different applications. The approach is intended to support an application engineer in determining candidate components for reuse and in restructuring existing reusable components in the repository.

1. Introduction

Reusability of project components, either at the code level or at the conceptual specification level, is considered a fundamental aspect in the application development process [1, 4, 13, 14]. In fact, it facilitates both effective development of good quality applications by exploiting existing validated components, and easy re-development of applications by stressing and promoting a modular design approach. Reusability can be considered from two different perspectives. According to a *design-by-reuse* perspective, an application can be obtained by tailoring existing components extracted from an available repository. According to a *design-for-reuse* perspective, a repository must be populated with reusable components, properly defined to be exploitable for designing new application not from scratch. Models and methods to support the development of reusable conceptual specifications, according to a *design-for-reuse* strategy, have been defined in [5, 6, 7]. The overall process of reusable specifications design can be articulated in the following methodological phases:

1• candidature of project components for reuse;
2• classification of the candidate project components;
3• design of reusable components;
4• restructuring of reusable components.

While phases 2 and 3 have been investigated in [5, 6, 7], in this paper we focus on phases 1 and 4. Specifically, models and methods for analyzing project histories with the aim of

identifying "good" candidates for reuse (phase 1) and for analyzing reuse histories with the aim of restructuring existing reusable components (phase 4) are presented.

We base our approach on the use of a repository in which historical information relative to both projects and reusable components are stored. Most proposals of software engineering environments are based on the use of a knowledge base or repository, which stores every kind of information that can result useful to the application developer. This information may refer to the current projects, to the methodology used in the development process and to previously developed applications [12, 18, 27, 30]. A common characteristics of most proposals is the representation of the project history as a set of decisions which transform the state of the project evolution. In DAIDA, for instance, a project history is modeled by a sequence of decisions, carried out with the help of some associated tool, that allow the transformation of design objects into other design objects [18]. CARE allows the designer to give a high level description of the knowledge of interest by means of a simple "real-world description" based on the concepts of actor, entity and event. This description is then compared against a set of predifined "patterns", describing common situations in software development. Each pattern is associated to a set of triplets in the form <situation, decision, action>. The situation is the current pattern, the decision is something that the designer will be asked for, and the action is the schema transformation that is triggered by the designer's choice [12]. In KBMS, the Software Engineering Knowledge (SEK) is maintained in a knowledge base and is represented as a set of procedures that guide the elicitation process [30]. Scacchi and Mi propose a meta-model for representing software development processes, based on the concepts of agent, task and resource. A development process can be evaluated through simulation, and by means of a ad-hoc query mechanism [27]. Souquieres and Levy separate the description of the development process, whioh is modelled as a set of interrelated tasks, from the description of the product being built, connecting tasks and products via links [26]. In our approach, for proper management of project histories, temporal information is used to keep track of the project evolution and of alternative design decisions. The concept of version is introduced to model different snaphots of a project's evolution.

This paper is organized as follows: in Sect. 2 we provide the basic concepts underlying our approach; in Sect. 3, the conceptual model proposed by Snodgrass and Jensen for retaining the semantics of relational temporal databases is generalized to a framework for complex objects evolving in time; in Sect. 4, the object-oriented model used for representing projects and reusable components is presented; Sect. 5 presents the methodological approach; in Sect. 6 concluding remarks are given.

2. Temporal Information for Reusable Components

In this paper, we focus on the design-for-reuse perspective at the conceptual design level. In this framework, the role of application engineer is defined as responsible for: (i) the definition of the reusable components, extracting generic conceptual specifications from existing application specifications and defining corresponding guidelines for reuse; (ii) organization of the reusable components in a repository according to a classification model, to support their search and retrieval by the application developer. Models and methods to support the development of reusable conceptual specifications, according to a *design-for-reuse* strategy, have been defined in [5, 6, 7]. Specifically, in the proposed approach, reusable components, at the conceptual design level, are defined as Generic Conceptual Units and associated Meta - Conceptual Units containing guidelines supporting the reuse. A Conceptual Unit (CU) is the description of a real-world object within a conceptual schema (e.g. an entity type or an object class). A Generic Conceptual Unit (GCU) is a "minimal" CU which encapsulates the set of common structural and behavioral properties of a number of similar CUs. Guidelines associated to a generic conceptual unit provide a set of design suggestions about ways to incorporate that unit in an application by means of possible adaptations and transformations.

In this paper we focus on methodological phases devoted to the identification of candidate components for reuse, by the analysis of project histories, and the restructuring of reusable components existing in the repository, by the analysis of reuse histories.

A *Project History* is a tree of conceptual schemas which represent the evolution of a project in terms of the ways in which it has been incrementally defined. A *Reuse History* is a tree whose root is a reusable component, whose intermediate nodes are its different versions, and whose leaves are CUs representing effective ways in which the component has been reused in different applications. Proper management of project and reuse histories requires capabilities of temporal information handling. In project histories, temporal information is used to keep track of the project evolution and of alternative design decisions; in reuse histories, temporal information is used to keep track of the actual usage of the reusable component. The concept of version is introduced. Versions are used to model different snaphots of a project's evolution and different ways in which a generic component has been reused in distinct applications. A *Version* is a snapshot of the state of a complex object with associated temporal information which identifies the version.

Versions of a reusable component model the evolution of a GCU in terms of the ways it has been reused in successive applications; the associated temporal information identifies a version in a rooted*Version Graph*, representing the reuse history of the reusable component. A directed

arc between two nodes means that the latter version has been derived from the former through a transformation primitive.

Versions of a project represent the project evolution at a particular moment in time; the associated temporal information identifies a version in a rooted version graph, representing the project history.

3. A Conceptual Framework for Modeling Temporal Complex Objects

In this Section features of a temporal model for representing project and reuse histories are discussed. Several models for temporal databases have been defined to provide application indipendent support for the management of time-dependent data, to answer historical queries and perform trend analysis [19, 22, 25, 28]. Recently, a unifying conceptual model has been proposed for retaining the semantics of temporal information, together with the suggestion that implementation or presentation models, used for storing data or presenting them to the user, should be made equivalent to such model [20]. This model, named Bitemporal Conceptual Data Model (BCDM), associates relational tuples with time values from two orthogonal time domains, *Valid Time* and *Transaction Time*.

Valid time refers to the time in which a fact is considered true in reality, whereas transaction time is referred to the presence of the fact in the database as stored data. The bitemporal chronons associated to each tuple identify a subset of the bitemporal domain in which the relation attributes assume those particular values. A bitemporal relation instance can be graphically represented as in Fig. 3.1. In this example, reported from [20], the tuple <Jake, Ship> is considered true at valid time 5, according to transaction times 10, 11 and 12, while the tuple <Jake, Load> is considered true at valid time 5, according to transaction times 13, 14 and 15. A similar situation may arise if we update the database with the insertion of a new fact (employee Jake works in the Shipping Department at valid time 5) which is later recognized as false (Jake's Department is the Load Department) and consequently corrected.

Update operations in this model, instead of substituting the old value with the new value, create a new tuple holding the new value and properly adjust the associated set of chronons. In this way, information is never deleted, and one can ask about past database states (*rollback* queries: e.g., in which Department is Jake working, according to the information avaliable two months ago?) and about facts which once held in reality (*historical* queries: e.g., in which Department was Jake working three years ago, according to the information avaliable now?).

Emp	Dept	Bitemporal chronons
Jake	Ship	(5,10) (5,11) (5,12)
Jake	Load	(5,13) (5,14) (5,15)
Kate	Ship	(5,10) (5,11) (5,12) (5,13) (5,14) (5,15)

Fig.3.1. A bitemporal relation in BCDM

Since the objects we consider - reusable components and project schemas - have complex structure, we need a generalization of BCDM that considers objects as stored in an object-oriented database [2, 10, 24].

As we will show in the following, the modelling cababilities of object-oriented languages are suitable to be exploited for: (i) better representation of temporal information associated to an object, and (ii) definition of new temporal domains besides "valid time and transaction time" domains [31]. For point (i), note that in object-oriented systems there is a difference between an object's state, that is the value of its properties, and the object's identity: two objects may share the same value for all of their properties and neverthless be different objects. Thus, we must distinguish between temporal information associated to the object itself and temporal information associated to the object state. As regards point (ii), note that a temporal database associates information from a temporal domain to ordinary data and provides facilities for their management. Proposals for several time domains can be found in the literature which differ in the primitive entities (time points vs. time intervals), in the topology (linear vs. branching), in the structure (continuous vs. discrete), in the metric and in the number of dimensions [28, 29]. A temporal domain can result more appropriate than others for a particular application; for instance, a branching domain, that is a temporal domain in which several possible future states are considered, can be useful for Decision Support Systems and versioning mechanisms [15]. For this reason, we believe that considering only a predefined bitemporal domain can sometimes be too restrictive and prevent the applicability of the model in some application domain.

The generalized modeling framework is presented in the following by, first, defining the objects to which temporal information will be associated, second, defining a generic n-dimensional temporal domain, and last, describing how time domains are associated to objects.

A *class* <cn,P,M> is defined by a *classname* cn, a set P of *properties* and a set M of *methods*. An object is defined as an *instance* of a class and is provided with a unique *identifier*. The

object's *state* is given by the values of its properties. *Inheritance* and *overriding* of properties and methods are defined in the usual way.

A *time domain* is a class. There is one predefined time domain T whose objects are linearly ordered time points. Other time domains can be defined by means of this domain. For instance, a class time interval can be defined whose properties are two time points and whose methods implement the common relationships holding between intervals.

A *temporal class* is a class with an associated time domain. Instances of the time domain are associated to the object identifier, representing its lifespan, and to each object state. An instance of a temporal class can be graphically represented as in Fig. 3.2.

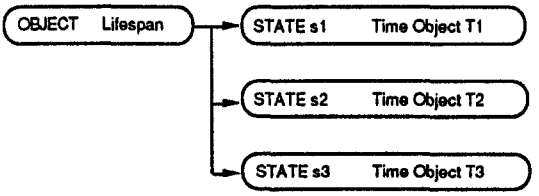

Fig.3.2. Instance of a temporal class

Fig 3.3 shows a simple example, in which objects from class Emp are associated with a transaction time domain. In the example, the object identified by X has a lifespan represented by six temporal objects; this object, during its lifespan, has assumed two different states, which are in turn associated with a set of temporal objects. Predefined methods for creation and deletion of objects and for updating their properties are overriden in temporal classes, in that old values and not discarded but associated to appropriate instances of the given time domain.

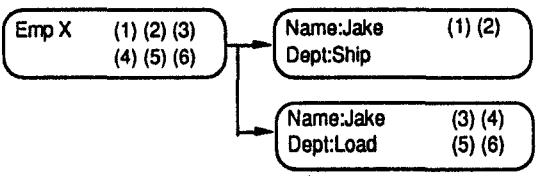

Fig.3.3. Instance of temporal class Emp

4. Temporal Information Handling in an Object-Oriented Environment

In this Section we present the object-oriented temporal data model, used for representing project and reuse histories in the repository. The model is based on the conceptual modeling framework discussed in Sect. 3 and will be identified in the sequel with the acronym OOtDB. The concepts of time domain and temporal class are in this model represented by means of predefined classes, while user-defined subclasses of these can be used to construct actual time domains and temporal classes.

There are four predifined classes, namely *Top-Class*, *Time-Domain*, *Version* and *Temporal-Class*, which are defined as follows:

<Top-Class,
{},
{new, get, change, delete}>

is a built-in class, on top of the class specialization hierarchy: it has no properties, and has four methods, which are inherited by all subclasses and serve to create, delete and manipulate object properties.

<Time-Domain,
{},
{After: [Time-Domain x Time-Domain] -> Boolean}>

is a time domain with no properties and an ordering relationship between its instances. It is defined as a subclass of Top-Class. Every subclass of Time-Domain is a time domain.

<Version,
{Value:Top-Class, Id:Time-Domain},
{}>

is a class with two properties, Value constituted by a snaphot of the state of a complex object, and Id constituted by a temporal object which univocally identifies the version among others. Class Version is defined as a subclass of Top-Class.

<Temporal-Class,
{Lifespan:Time-Domain, Versions:list-of(Version)},
{new, get, change, delete}>

is a temporal class with two properties, Lifespan constituted by a temporal object associated to the Temporal-Class instance as a whole, representing a subset of the time domain in which that instance was "alive", and Versions constituted by a list of snapshots of the instance state, each with associated temporal objects.

Temporal-Class is also defined as a subclass of Top-Class, but the four methods defined in the superclass are overridden. While in Top-Class the change and delete methods cause information to be definitively lost, in Temporal-Class they allow the creation of versions. In order to define the behavior of the Temporal-Class methods, we must introduce three more predefined entities: class *Time-Point* (which corresponds to the predefined time domain T of the modelling framework), defined as a subclass of Time-Domain; object *Now*, defined as an instance of class Time-Point; and object *Current*, defined as an istance of class Time-Domain. Class Time-Point is used as a basic class upon which other time domains can be built. Object Now is a particular instance of class Time-Point which is associated to the current moment in time, or the current computer clock time. One object labelled Current is defined as an instance of each of the defined time domains. When dealing with time points, we set Current=Now. We can choose an appropriate value for Current in other time domains: for instance, when dealing with time intervals, we can set Current either equal to the endless open interval starting from Now or to the minimum interval starting from Now and ending at Now+1. The precise behavior of the Temporal-Class methods, in terms of the above defined concepts, is given, in pseudo-code, in Figure 4.1.

```
/* Method new creates a new object with an empty list of versions and sets the object lifespan to the appropriate
value. The object is created invoking the Top-Class method new. */

new =          self.^new;
               Lifespan:=Current;
               Versions:=[].

/* Method get returns an attribute value. It requires two arguments: the first indicates the attribute to be returned;
the second is a boolean expression used to identify a particular version of the object, in which both temporal
attributes and ordinary ones may appear. A full version may be returned specifying version as first argument.
Instances of the associated time domain may be returned specifying time as first argument. The list of versions
is searched starting from its bottom until one is found that is consistent with the given expression. */

x.get(attrname,expr) =    go to the bottom of Versions;
                          repeat   found:=evaluate(expr)
                          until found;
                          return(attrname).
```

Fig.4.1. Temporal-Class methods

```
/* Method change creates a new version of the object with an attached instance of the associated time-domain.
Notice that, because of the behavior of the get method for retrieving information, there is no need of modifying
the time entities associated to other versions */

x.change(attrvalues) =    State.new;
                          State.Value:=attrvalues;
                          State.Id:=Current;
                          Versions.append(State).

/* Method delete deletes logically the object. The object lifespan is modified in order to finish at the moment in
which it is logically deleted */

x.delete =        Lifespan:=Lifespan-Current.
```

Fig.4.1 *(continued).* Temporal-Class methods

We will now represent the concepts defined in Sect. 2, namely Project History, Reuse History, Conceptual Schema, Conceptual Unit and Generic Conceptual Unit, according to the OOtDB model. This is achieved through the definition of appropriate subclasses of the four predefined classes introduced above, each used to represent a distinct concept. We will proceed in a top-down fashion, starting from temporal classes and then detailing their components.

A *project history* is a tree of conceptual schemas, and a *reuse history* is a tree of conceptual units, whose non-leaf nodes are generic conceptual units. Their are both defined as temporal classes, and consequently represented by subclasses of Temporal-Class, which override the domains of properties Lifespan and Versions and inherit the four methods new, get, change and delete.

<Project-History,
{Lifespan:Project-Version-Time, Versions:list-of(Schema_Version)},
{}>

<Reuse-History,
{Lifespan:Reuse-Version-Time, Versions:list-of(CU_version)},
{}>

Versions of both project and reuse histories are organized in a version graph, which imposes a partial ordering among versions. Since versions are maintained in a list and identified by their associated temporal object, the time domain associated to versions is non-linear. Non-linear

time domais are called branching time domains, and are characterized by a partial ordering relation defined among their elements [22, 28, 29]. Two *time domains* are defined as subclasses of Time_domain. Their properties are useful to identify versions of a project or of a reusable component in the version tree: Transaction is the time point in which the information has been entered in the database; From is the parent node in the version tree; Through is the decision, applied to the parent node, to produce the child. The "after" relationship is overriden to become a partial ordering relationship between nodes in the version tree: node A is after node B iff B is an ancestor of A.

<Project-Version-Time,
{Transaction:Time-Point, From:Schema_Version, Through:Project-Decision),
{After:[Project-Version-Time x Project-Version-Time] -> Boolean}>

<Reuse-Version-Time,
{Transaction:Time-Point, From:GCU_Version, Through:Reuse-Decision),
{After:[Reuse-Version-Time x Reuse-Version-Time] -> Boolean}>

A *conceptual schema* is a set of inter-related conceptual units, and is also a version of a project history. It is thus represented by a subclass of class Version, whose associated time domain in the project version time.

<Schema_Version,
{Value:set of CU, Id:Project-Version-Time},
{}>

A *conceptual unit* is the construct used to describe real-world objects within a schema, and is also a version of a reuse history. Generic conceptual units encapsulate a set of common structural and behavioral properties of a set of similar CUs. Class CU_Version is defined as a subclass of Version, whose associated time domain is the reuse version time. GCU_Version is defined as a subclass of CU_Version representing generic conceptual units. CU is the class which contains the information to describe real world objects, with attributes Name, Interface and Implementation.

<CU_Version,
{Value:CU, Id:Reuse-Version-Time},
{}>

<GCU_Version,
{},
{}>

<CU,
{Name:string, Interface:set-of(Function),
Implementation:Code},
{}>

Project-Decision and Reuse-Decision represent decisions taken by the designer to transform versions of schemas and generic conceptual units into new versions. In case of project decisions, the operation performed is defined as an instance of class Transformation, and the property Applied-to identifies the CU belonging to the parent node to which the operation is applied. In case of reuse decisions, the operation performed is defined as an instance of class Specialization.

<Project-Decision,
{Applied-to:CU_Version, Operation:Transformation},
{}>

<Reuse-Decision,
{Operation:Specialization},
{}>

Two subclasses of class Transformation are defined, to represent two different kinds of operations associated to a project decision: "refinement" decisions, that consist in exploding the source CU_Version instance in a set of CU_Version instances describing the same concept at a lower level of detail, and "tailoring" decisions that consist in selecting a reusable component and possibly a version of it, and then adapting them to particular needs applying the guidelines provided by the meta-CU.

5. Candidature for Reuse and Restructuring of Reusable Components

The present methodology is intended to help the Application Engineer in identifying candidate reusable schemas from the analysis of existing project histories and in restructuring reusable components together with their reuse history and reuse guidelines. The overall intended behavior is outlined in Figure 5.1.

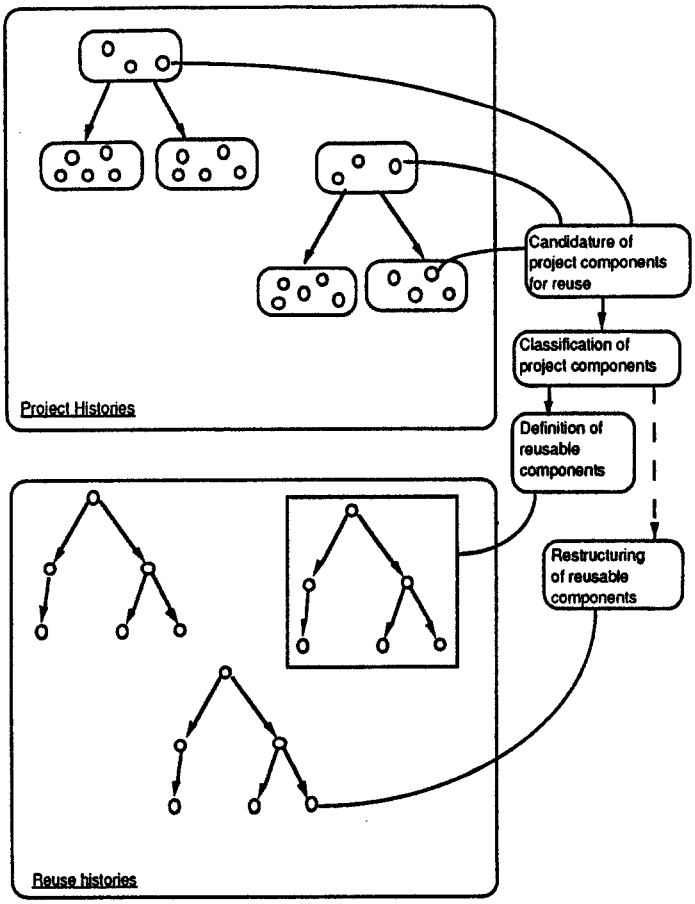

Fig.5.1. Methodological phases

Two methodological phases are defined (phase 1 and phase 4 of the overall methodology). The first is concerned with the analysis of project histories, while the second is concerned with the updating of reuse histories. Original contribution of this paper concerns the use of the OOtDB model for better inspecting the project histories and for continuously upgrading the reuse histories including new reuse experiences.

5.1. Candidature of Project Components for Reuse

The goal of the first phase is to identify project components that prove to be good candidates for reuse and provide them as inputs for the second and third phases as defined in [5, 6, 7]. Criteria, with associated metrics, to test whether a given component can be considered a good candidate for reuse have been proposed in the literature [21, 8]. According to these criteria, a good candidate must be loosely coupled with other conceptual units, have a simple interface,

and appear at a high level in the specializion hierarchy. In essence, an easy-to-reuse component must be meaningful, simple and not too application dependent. Adopting these considerations, and exploiting the capabilities of analyzing the evolution of a project in time, we can greatly improve the chance of finding "good" components.

The advantage of inspecting different versions of the same project lies in the fact that the "most reusable" version of a project component can be found. Whenever the same component (we assume the existence of a data dictionary including definitions of all conceptual units managed in the project, so that the expression "the same component" has a precise and not intuitive meaning, [3, 9]) appears in different versions of a project, it may be the case that it has been slightly modified, e.g., with the addition of a new property, from one version to another. If this happens, the version of the component which shows the greatest degree of reusability according to the defined criteria, is considered.

The availability of a full project history allows also the definition of new criteria for reusability which involve the temporal properties of a component. The three criteria we propose also aim at identifying simple and not too application dependent components. They can be enunciated as follows. First, it would be better for the candidate to appear at a high level in the project history, because this means that the component has been soon identified as a meaningful one in the project being developed. Second, it would be better for the candidate to appear in many versions of the project, because this means that it is not specific to a particular version. Third, it would be better for the candidate to have been refined and transformed, at lower levels in the project history, into many and more specific conceptual units, because this means that it can be considered as a high level view for more specific concepts.

The association of metrics to these criteria is quite straightforward. As to the first one, a value is associated to the root version, say 100, and, if the longest path in the tree is N steps long, the value 100-K(100/N) is associated to each node K steps far from the root. As to the second one, the value is the percentage of versions in which the component appears. As to the third, a positive value is given to components which have been refined, and a negative value is given to those which have been not. All information needed to compute this values is recorded in the OOtDB instances of class Project-History, and can be accessed via simple get messages.

We associate with each project component a *reusability value*, consisting of a weighted sum of all the single values for each of the defined criteria. All project components whose reusability value exceeds a given threshold are stored as inputs for the second phase. The threshold is computed in such a way to have a desired number of candidates to evaluate in the second phase: if we wish to evaluate only a small number of candidates, a high threshold is needed, while a

low threshold is preferable if we wish a large amount of candidates. A threshold value of 2/3 the sum of the maximum and minimum reusability values seems to be an appropriate one.

5.2 Restructuring of Reusable Components

A first goal of this phase is to find CUs which have been derived by tailoring existing reusable components. In OOtDB, the operation of retrieving components out of the repository is recorded as a project decision in the project history. Specifically, it is recorded as an instance of a particular subclass of class Transformation. To retrieve the required CUs, we send a message to a project history, requiring the selection of all versions which have been derived through a "reuse" decision; then we send a message to such versions, which are instances of class Schema_Version, to select the CU which has been derived by tailoring the selected reusable component. Syntax of the two messages is shown below.

```
Version:= Project.get(version,Time.Through.Project-Decision="reuse");
Comp:=Version.get(Id.Through.Applied-to);
```

Not every selected CU will be used to enrich reuse histories, but only those of them which appear to be "easy-to-reuse", according to the criteria discussed in the previous subsection. A reusability value is thus computed for these CUs, and only the ones exceeding a given threshold are stored, together with the reusable component version they have been derived from. A reusable component is structured as in Fig.5.2. A meta-CU is associated to the tree root and contains guidelines for tailoring the reusable component according to particular needs. The root and other non-leaf nodes represent GCUs, while leaf nodes are CUs which have actually been used in some developed project.

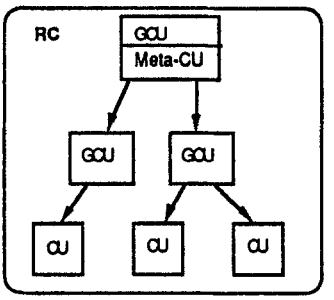

Fig.5.2. A reusable component

Selected conceptual units, obtained through a "reuse" decision, may be used to improve the information associated to a reusable component. The improval consists in augmenting the version graph and in adding some information to the meta-CU. The version graph is augmented in that a new leaf, corresponding to the new CU, is added, and it is connected via an arc to the particular reusable component version it has been derived from. Information to the meta-CU is added to record the fact that the component has been reused in that particular process. This information is represented in a meta-CU by a rule in the form "if you work at process P and want to reuse this component, then it is better for you to select version V1, and possibly tailor it to version V2" [6].

6. Concluding Remarks

We have presented a methodological approach that exploits an object-oriented temporal data model, to support application engineers in the analysis of project and reuse histories for the definition and maintenance of reusable components.

The concepts of Project History, Reuse History, Version and Version Graph have been defined and a model for representing this information, according to both temporal databases and object-oriented databases concepts, has been introduced. The model, named OOtDB, is actually constituted by a set of classes in object-oriented style, which serve to represent the concepts of time domain and temporal class. A time domain is defined by a class provided with an ordering relationship among its elements. A temporal class is defined by a list of versions identified by an associate instance of a time domain. OOtDB is used for modelling project and reuse histories. Project histories and reuse histories are defined as particular temporal classes, and suitable time domains are designed to retain the semantics of project evolution and component reuse in different projects.

Major advantages of the proposed approach consist in the possibility of choosing the most reusable version of a project component (as a consequence of the capability of version handling) and of using suitable criteria for reusability based on temporal properties of components.

Acknowledgements
Part of this work has been supported by the F^3 Esprit Project N.6612, by the Italian National Research Council Project "Sistemi Informatici e Calcolo Parallelo", L.R.C. INFOKIT, and by MURST 40%.

References

1. ACM Issue on Object-Oriented Design, September 1990

2. M.Atkinson, F.Bancilhon, D.DeWitt, K.Dittrich, D.Maier, S.Zdonick - The object-oriented database system manifesto - in W.Kim, J.M.Nicholas, S.Nishio (eds.) "Deductive and object-oriented databases", Elsevere Science Publishers, Amsterdam, the Netherlands, 1990

3. C.Batini, G.Di Battista, G.Santucci - A formal framework for multilevel schema documentation in a data dictionary - Information System Concepts: Improving the understanding (E.D.Falkenberg, C.Rolland, E.N.El-Sayed Eds.), Elsevier Science Publishers 1992

4. T.J. Biggerstaff, A.J. Perlins (eds.), "Software Reusability- Concepts and Models", vol.I, ACM Press, Addison-Wesley, 1990

5. S.Castano, V. DeAntonellis, B. Zonta, "Classifying and reusing conceptual schemas", Entity-Relationship Int. Conf., Karlsruhe, October 1992

6. S.Castano, V. DeAntonellis, "Reuse of Conceptual Requirement Specifications", accepted to RE '93, ACM/IEEE Int. Conf. on Requirements Engineering, San Diego, CA, January 1993

7. S.Castano, V. DeAntonellis, "A Constructive Approach to Reuse of Conceptual Components", accepted to ACM/IEEE Second Workshop on Software Reusability, Lucca, March 1993

8. S.R.Chidamber, C.F.Kemerer - Towards a metrics suite for object-oriented design - ACM SIGPLAN Notices, vol.26, no.11, November 1991

9. De Antonellis V., Zonta B, "A disciplined Approach to Office Analysis", IEEE TSE, Vol. 16, No. 8, 1990, pp. 822-828

10. O.Deux et al. - The O2 system - Communications of the ACM, vol.34, no.10, October 91

11. K.R.Dittrich, R.A.Lorie - Version support for engineering database systems - IEEE Transactions on Software Engineering, vol.14, no.4, 1988

12. G.Grosz, C.Rolland - Why and how should we hide conceptual models? - Third Int. Conf. on Software Enginnering and Knowledge Engineering, USA, 1991

13. S.Gibbes, O.Tsichritzis, E.Casais, O.Nierstrazs, X.Pintado - Class management for software communities - Communications of the ACM, vol.33, no.9, September 1990

14. J.W.Hooper, R.O.Chester - Software reuse: guidelines and methods - Plenum Press, New York 1991

15. W.Kafer, H.Schoning - Mapping a version model to a complex object data model - Proc. Int. Conf. Data Engineering, 1992

16. R.H.Katz, E.Chang, R.Bhateja - Version modelling concepts for CAD databases - Proc. ACM SIGMOD Int.Conf. Data Management ,1986

17. R.Kowalski - Database updates in the event calculus - Journal of Logic Programming, vol.12, 1992

18. M.Jarke, T.Rose - Managing knowledge about information system evolution - Proc. ACM SIGMOD, June 1988

19. C.S.Jensen, J. Clifford et Al. - A glossary of temporal database concepts - SIGMOD RECORD, vol.21, no.3, September 1992

20. C.S.Jensen, R.Snodgrass - Proposal for a data model for the temporal structured query language - TEMPIS tech.rep. no.37, 1992

21. R.E.Johnson, B.Foote - Designing reusable classes - Journal of Object-Oriented Programming, June-July 1988

22. R.Maiocchi, B.Pernici - Temporal data management systems: a comparative view - IEEE Transactions on Knowledge and Data Engineering, vol.3, no.4, 1991

23. J.Rothemberg - Prototyping as modeling: what is being modeled? - Int. Work.Conf. on Dynamic Modelling of Information Systems, April 1990

24. L.A.Rowe, M.R.Stonebraker - The POSTGRES Data Model - Int. Conf. on Very Large Databases, 1990

25. F.Schreiber - Is time a real time? An overview of time ontology in informatics - NATO Adv.Study on Real Time Computing, October 92

26. J.Souquieres, N.Levy - Description of a specification development model - Int. Conf. on Requirements Engineering, 1993

27. W.Scacchi, P.Mi - A knowledge-based environment for modeling and simulating software engineering processes - IEEE Transactions on Software and Data Engineering, September 1990

28. R.Snodgrass - Temporal databases: status and research directions - SIGMOD RECORD, vol.19 no.4, December1990

29. R.Snodgrass - Temporal databases - in Proc. Int. Conf. GIS Sept. 1992

30. K.Zeroual, P.N.Robillard - KBMS: a knowledge-based system for modelling software system specifications - IEEE Transactions on Knowledge and Data Engineering, vol.4, no.3, 1992

31. G.Wuu, U.Dayal - A uniform model for temporal object-oriented databases - Proc. Int. Conf. Data Engineering, 1992