Towards a Versioning Model for Component-based Software Assembly

Jaroslav Gergic
Charles University, Faculty of Mathematics and Physics
Department of Software Engineering
Malostranske nam. 25, 118 00 Prague, Czech Republic
gergic@nenya.ms.mff.cuni.cz, http://nenya.ms.mff.cuni.cz

Abstract

The world of software development has rapidly changed in the last few years due to the adoption of component-based technologies. The classical software configuration management, which deals with source code versioning, becomes insufficient in the world where most components are distributed in a binary form. This paper focuses on solving versioning issues in the SOFA/DCUP component environment, however, many ideas introduced in this paper are applicable in other component-based environments. The paper incorporates the following versioning issues: component and type identification, component and type evolution, component version description model and component retrieving based on the version description model. It also outlines a prototype repository designed as a proof of concept.

1. Introduction

1.1. Software configuration management

Software configuration management (SCM) is a very important constituent of software engineering. SCM deals with development, assembly, configuration, updating and maintenance of software systems. SCM covers all the stages of software evolution: project management, software development, application assembly/configuration and software systems maintenance [6]. Majority of the SCM tools focus on the development stage. These tools are typically file-oriented, because they deal with the source code versioning. Common examples of the file oriented tools are the Revision Control System (RCS) [22] and its network-enabled variant, the Concurrent Version System (CVS). Both tools are widely accepted by the open-source community. Several SCM tools evolved from the basic file-versioning utilities in different directions: some include support for process management and workflow, other support long and nested transactions required for staged software development. This section provides a brief introduction into basic SCM-related terms used in this paper. For more detailed overview of recent status in SCM, we refer the reader to [6] and [2]. We briefly evaluate the existing SCM and other related technologies in the *Goals and Motivations* section.

Versioned entity. Versioned entity is a mutable item (file, object, component, etc.) which evolves in time and space and can exist in multiple incarnations. During the development stage, typical versioned entities are source code files, exported data structures and public interface/component declarations. During the software assembly stage we need to track component binaries, distribution packages or assemblies. During the software system maintenance and upgrading we have to maintain the deployment descriptors, configuration and persistent state of the components in a particular software environment. The versioned entities change their nature during the different stages of a software system life-cycle. Each phase deals with different kind, granularity and quantity of versioned entities and thus imposes quite different requirements on a version model.

Version, revision, variant. Revisions are versions of the same entity which are created by repetitive modifications of the original entity in time. The revisions can be easily ordered along the time axis. The successor/predecessor relation is implicit to the entity revisions. Variants are versions of the same entity which can occur in parallel with the respect to the time axis. The successor/predecessor relation typically does not have sense when referring to the variants. Both the revisions and variants of the same entity are called versions. Versions are immutable items: a new version is created when modifying any existing version.

Version group, version graph, version grid. Version group is a set of immutable item versions which belong to the same versioned entity. The version group of a particular entity can be organized into a directed graph. Particular versions are nodes of the graph and edges are labeled with the name of the relation which has been used to derive one

version from another (revision, branch, merge, etc.). An alternative approach to the version group organization uses a set of named attributes attached to each entity version describing its properties. The version space then maps to an n-dimensional grid having one dimension for each named attribute.

Extensional versioning, intentional versioning. Extensional versioning allows only retrieval of pre-existing versions identified by their unique version identification. The intentional versioning allows to retrieve versions by specifying a logic constraint, so the versions can be either retrieved from the version repository or constructed on-the-fly according to the specified criteria. The most common example on the intentional versioning is probably conditional compilation directed by C preprocessor.

1.2. Software components

The world of software development has rapidly changed in the last few years due to the adoption of component-based technologies. This process induces changes in the area of software configuration management. The majority of software today is not developed from scratch. A typical up-to-date software system is composed from a set of software components, these components are partially designed by a development team and partially obtained from third parties. A precise definition of the term *software component* has not been established yet, however, there is a common sense of what properties are typical for a piece of software in order to get it labeled as a software component. Those typical properties are:

- *autonomy* (a software component is the unit of deployment/configuration)
- *component provisions* (a formally declared set of services a given component provides)
- *component requirements* (a formally or informally declared set of assumptions and constraints imposed on a component deployment/runtime environment)
- *abstraction* and *information hiding* (a component hides its implementation details and describes its behavior in terms of its provisions and requirements)

1.3. SOFA/DCUP

SOFA (SOFtware Appliances) is a research project which tries to analyze all the requirements for a modern component-based technology (trading, licensing, billing, versioning, security, etc.) and design a solution for these requirements. The DCUP (Dynamic Component Updating) project extends SOFA technology with the capability to change components at software system runtime. This feature enables such options as silent updating or automated software re-configuration. We briefly describe important parts of the SOFA/DCUP architecture with the respect to the versioning issue. For more detailed information we refer the reader to previously published work about SOFA/DCUP. [15], [16]

Interfaces. SOFA/DCUP architecture uses CDL (Component Definition Language [8]) to specify interfaces, component frames and architectures. CDL is an extension of OMG IDL. A basic building block of component specification is an interface. The SOFA interface has syntax and semantics very similar to the OMG IDL interface element. The only extension is that the CDL interface also contains protocol specification. A protocol is a regular-like expression used to partially express an interface semantics [17]. Such a protocol specification is considered to be a part of an interface type, which means a protocol can have an influence on the substitutability of interfaces.

Component frames. a Component frame is a collection of *provides* interfaces and *requires* interfaces bound with a frame protocol. Services a component provides are specified in the provides section of its definition and services it requires from its environment are specified in the requires section. This feature allows flexible and automated tracking of type dependencies in a software system. The frame protocol specifies a component frame behavior. It can be generated from provides and requires interface specifications or it can be written by hand, however, it only can tighten the contract imposed by the interface protocols of provides and requires interfaces. When a component frame is used in a software system, all the requirements should be satisfied and all the provisions have to be bound to requires interfaces of other components. It means that SOFA components are (like COM components) bound at the CDL interface level. a component frame together with interfaces specification provides a *black-box view* of a SOFA component.

Architectures and implementations. SOFA supports component nesting and compound components. A SOFA component can use other services in a system using the client-server relationship (binding *requires* interfaces to corresponding *provides* interfaces) or by aggregating other components. SOFA differentiates between primitive components and compound components. Only primitive components can contain real functionality. Compound components are provided to bind several smaller components together. A parent component delegates incoming requests to its nested members and subsumes outgoing requests through its own requires interfaces. This mechanism has a great impact on the architectural scalability of applications based on SOFA architecture and thus allows the building of software systems from fine-grained components with the power of both abstraction and refinement. Bindings between a parent component and its direct sub-components

are also specified in CDL. This is called *architecture specification*. Architecture specification also contains a protocol specification - the architecture protocol. The architecture protocol is usually generated from frame protocols of nested components and is necessary when checking the conformity of architecture against the component frame of a parent component. This feature allows for static checking whether designed architecture violates component frame specification or not. Architecture specification at the CDL level provides a *gray-box view* of a SOFA component. Because nested components are identified only using their component frames in an architecture specification, each architecture can have multiple *implementations* (instantiations) depending on a substitution of particular nested components.

1.4. Structure of the paper

The following section summarizes our goals and motivations. It also briefly evaluates the existing solutions and related work. The section 3 describes all the aspects of the proposed version model, including identification, description and retrieval of versioned entities. It also describes the prototype implementation. The section 4 summarizes this paper and outlines the current status and possible future work in the research area.

2. Goals and motivations

2.1. Motivations

During the design and development of the SOFA/DCUP architecture a requirement for a component-based versioning model arose. There was a need to provide the developers designing their applications in SOFA/DCUP with a tool allowing them to deal with different component versions.

A typical component-based application is composed of several pre-existing components while some other (application specific) components are developed by the application developer. Every component used in a particular application can evolve during time. Moreover, sometimes multiple different components provide similar or identical functionality and some of them can be substituted at a given place in the application architecture.

The component selection and substitution into the application architecture we call an *application assembly process*. The application assembly can be driven manually by the application developer at the design time. It can be postponed to the application build phase, where the assembly process is driven by predefined composition rules, or it can dynamically proceed even at runtime. In all the above cases, there is a need to provide version management tools to help the application developer to manage this complex task.

2.2. Related work

We started to evaluate the existing solutions to software versioning in order to find a suitable version model, which can aid and partially automate the software assembly process in the SOFA/DCUP environment. The results of our investigations were quite unsatisfactory. The most popular tools used for software versioning (CVS, RCS) are based on the source-code versioning [22]. These systems do not have a notion of a software component – a versioned entity is a text file. The source-code version systems are mainly focused to revisioning. Support for variants is limited to branching and typically the variants are constructed using conditional compilation, i.e., variants are not captured the its version repository. Retrieval capabilities of these systems are also quite limited: *get-latest*, *get-by-date*, *get-by-a-version-number*, *get-by-a-symbolic-name*. There are also other file-oriented version models, e.g. *change-set model* (*Apollo DSEE*) which makes up a configuration using base configuration and several change-sets, or the *long transaction* model (*Sun NSE*) which extends a change-set model with the support for nested transactions [6]. However, these models strongly reflect the requirements of a development process instead of enabling mechanisms for process independent version retrieval.

We also examined a representative of the process-oriented versioning tools, the Adele Configuration Manager [5]. We realized that such a tool can hopefully satisfy our requirements, but the cost of adoption is very high. The tool is very general, because it tries to cover all of the development stages including development process and not just the software assembly task. The user of such a system is forced develop its own versioning model using scripting, triggers, events and other *imperative* programming tasks on the top of the existing general-purpose versioning engine.

To our knowledge, there is no general component-based versioning model. We evaluated the following systems with the respect to versioning support: OSF/RPC, MS COM and CORBA [11], [18], [13], [14]. None of these technologies provide a versioning model sufficient for a modern component based system. Their versioning models are tightly bound to the underlying technology and are not flexible enough to be adapted to a completely different component architecture like SOFA. One of the common properties of those version models is their versioning units are very small – individual component interfaces and data structures. The application developer deals with versioning on many places in the code which introduces significant overhead. The result is, the software producers often avoid using even the elementary available versioning features in their component-based products in order to reduce development cost.

We also evaluated popular distribution package managers like RedHat Package Manager (RPM) [19], De-

bian Package Manager and assembly versioning support in MS .NET [10]. These technologies feature some interesting concepts not present in many component versioning models. For example: package dependency tracking and multiple variants support. On the other hand, those systems have no notion of software components or data types, so they lack many features required for semantically rich version description.

We are aware of few research projects related to the topic of this paper. For example *Source Tree Composition* [4] deals with software package assembly, but is assumes the software components are available in source code. The composition process builds the compound packages by hierarchically nesting the build process tasks of their subcomponents. The project does not further explore the area of the version attributes and their type system and/or value taxonomies.

The work described in [9] deals with software component retrieval based on a *concept lattice* of formal component specifications. The approach based on a formal specification language works well applied to the model examples, but it is very hard to imagine to scale this approach to description of real software components. Taking the complexity of the today mainstream software into account, a complete formal specification seems to be impossible. The idea of using concept lattices for knowledge representation is also described in the following publications: [20], [12].

You will see in the section 3, that we are using partial ordered sets to describe attribute value taxonomies. This idea has been influenced by the concept lattices, however, we realized we can not guarantee every attribute value taxonomy holds all the properties necessary to form a lattice structure [3].

2.3. Version model requirements

The results of our version models evaluation study, led us to the decision to design a new version model for the SOFA/DCUP from scratch. The following features that the version model must provide were identified:

1. the model focuses on a single task – software assembly
2. the model allows to define components, interfaces and other high-level concepts as first-class entities
3. the model provides unique identification of versioned entities in time and space
4. the model supports revisions as well as variants for every versioned entity
5. the model supports encapsulation in order to scale to large software systems
6. the model provides powerful retrieval capabilities usable during design, assembly and runtime

The lessons we took during the existing systems evaluation were important also in one additional aspect: It is not a good design to tie a versioning model to an underlying component model. Such a tightly bound version model is hard to evolve along the underlying component model. It is almost impossible to integrate several component technologies into a single system while each technology uses its own versioning model and different terminology.

3. A version model for component assembly

Considering the results of the versioning models evaluation and given the requirements above, we designed our versioning model as a flexible *meta-model* allowing to instantiate a *concrete versioning model* for a particular component model or a software architecture. Such a meta-modeling capability can also serve as an abstraction layer for a conceptual integration of different component technologies. On the other hand we limit ourselves to the software component version description and retrieval. We do not cover other aspects of an end-to-end versioning system like development process support (locking, triggers, scripting, etc.). Our meta-modeling is strictly declarative, it is similar to a relational schema declaration. Our prototype repository comes with pre-defined entity and relationship types for SOFA/DCUP architecture.

The structure of this section follows all the important elements of the described versioning model framework. We discuss the issue of establishing a naming convention for version identification first (3.1). a well defined identification schema is an essential building block for any scalable version model. Once having the identification schema established, we discuss how to describe the versioned entities using version attributes and how to capture the relations between the versioned entities (3.2). Since every versioned entity (and its respective versions) are uniquely and uniformly identified, their "description" is accomplished in a generic way by attaching attribute values and relation bindings to their unique identifications. The subsection 3.3 enumerates viable options and describes a chosen solution for storing the versioned entity descriptions. The last subsection (3.4) describes the query language we define to retrieve the information about the versioned entities. The language allows to specify a set of constraints (a *query*) and the versioning engine returns a set of versioned entities which conform to the given constraints. A special query language was needed in order to express the unique features provided in our versioning model.

3.1. Version identification

The identification of versioned entities is an issue whenever a technology should support the integration of com-

ponents provided by different vendors during an unconstrained time period. According to the terms defined above we can view the identification of a particular entity version as a two-step process: (1) we have to identify the entity (which evaluates to a version group), (2) and identify a particular version in a given version group. Another approach can use single-step version identification and assign a globally unique identification directly to a particular entity version - using a flat-space identification domain. Regardless of the identification assignment process, we are using the term *Universally Unique Identification* (UUID) for entity version identification.

Other criteria for version identification is whether the UUIDs are human-readable or not. Both approaches are used in the real world. Human-readable version identification is typically used together with a two-step identification process, so given a UUID the user can recognize also a version group. As an example of this approach we consider the Red Hat Package Manager (RPM) [19] naming schema used in many Linux distributions. For example, given the following RPM package UUID: `apache-1.3.9-8mdk-i586.rpm`, the user can detect the version group (`apache`), the revision number (`1.3.9`) and other attributes used to denote a particular package version. A typical example of a flat-space identification schema using human-unreadable UUIDs is Microsoft COM [18] or DCE RPC [11] which use 128-bit hexadecimal UUIDs generated by software development tools: `148BD520-A2AB-11CE-B11F-00AA00530503`.

Scope of versioning. Majority of current component technologies attaches the version identification to individual interfaces and components. The versioning model maintains and handles relations between individual interfaces (classes, components). Many of these technologies are inspired by the original DCE-RPC standard. These old models are suitable for simple software systems with a small number of versioned interfaces. The complexity and maintenance overhead is growing very rapidly when increasing the number of versioned entities - interfaces, classes, components. We can say, from the point of view of our today knowledge, that an interface/component based versioning models became insufficient for current complex software systems. Our solution reflects the real-world practice used in many software environments. We moved versioning to the scope of *distribution packages*. All the versioned entities (types, interfaces, classes, components) are identified in the scope of a distribution package they are bundled in. A typical software package is developed as a unit, so that the developer does not need to care about versioning when referring to the entities inside the package. Version identification issue occurs only when referring to external entities in other distribution packages. This approach reduces versioning complexity and overhead for the developers.

We propose to use human readable UUIDs to identify distribution packages. Following the two-step process described above, a distribution package UUID has two components. The first part of the UUID is a globally unique package name. We prefer reversed Internet-domain names used in the Java Platform, because they provide safe and well-established convention resulting in human-readable identification strings. The second part of the UUID is version identification – an arbitrary string which uniquely identifies a particular distribution package version within its version group (a version group is identified by the distribution package name). We specify neither format nor semantics of the version identification string. The information about properties of a given distribution package is decoupled from the version identification. We prefer to use version attributes and relations as a more flexible and powerful way of describing properties of distribution packages. The following two examples illustrate the proposed naming convention, please note the package UUID in **`bold`**:

- **`com.sun.JDK/1.2.2`**`/java.util.Map`
- **`org.w3c.DOM/Level1`**`/org.w3c.dom.Node`

3.2. Version attributes and relations

Version attributes are properties describing a particular entity version, e.g. a supported operating system, an optimization level, run time libraries and many other. It may seem that the version attributes apply to the version models using *grid version space* only. When investigating the issue more deeply, we note that version attributes also occur in the version models based on the version graph space concept. In these models the version attributes are often encoded directly into entity version UUIDs (RPM packages) or they are attached as labels to the edges of a particular entity version graph (RCS).

Almost all the version models support some kind of version attributes. The major difference between the grid-space and the graph-space version models is in the flexibility of using version attributes. In the case of grid-space models, an entity can have possibly unlimited number of version attributes attached. In the case of graph-space models, a number of attributes is determined (and limited) by the number of UUID components (RPM) or a version graph topology and its branch-naming convention (RCS, CVS). We consider the grid-space concept as a generalization of graph-space (naming-convention) based version models. For example an RPM package UUID `apache-1.3.9-8mdk-i586.rpm` can be rewritten as a list of attributes: `entity=apache`, `revision=1.3.9`, `release=8mdk`, `processor=i586`.

Version relations are used to describe connections between versioned entities. A relation can be defined between entities (version groups), so that it relates sets of objects,

or a relation can be defined between concrete entity versions (particular objects), for example: `ftpd` *requires* `libc` versus `ftpd2.5` *requires* `libc5`. The dependency relation (*requires*) is one of the most common relations the developers are dealing with. It is quite often expressed in terms of a programming language (C/C++ `include`, Java `import`) rather then in terms of a versioning model. Considering a relation-aware versioning model, such programming-language expressed relations can be relatively easily captured by an automated tool and represented in the version model together with other explicitly constructed relations.

Relations versus attributes. Attributes can be considered as a special case of relations. E.g. an attribute `OS="Linux"` can be easily translated into *requires* relation. The dividing edge between version attributes and version relations can be based on the fact whether the "related" object is a *versioned entity* recognized by a given versioning model or it is an *external entity*. If the "related" object is a versioned entity, a version relation should be established. If the "related" object is an external entity to the versioning model, the "fact" should be modeled as a version attribute.

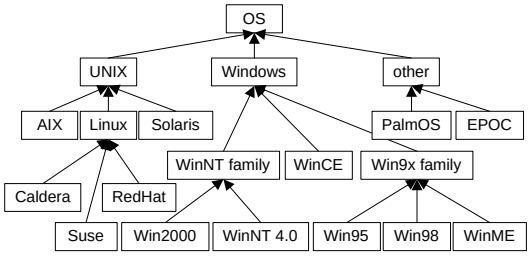

Figure 1. An operating systems tree.

Attribute value taxonomies. Other versioning models typically state that the versioned entities can have named attributes. Some versioning models also allow to specify attribute types, but majority of systems uses an arbitrary string as the attribute value. Simply speaking: version attributes are name-value pairs in a typical case. Our versioning model distinguishes two types of attributes: *descriptive* and *taxonomic*. The descriptive attributes correspond to generic entity attributes as used by other versioning models. These attributes are useful for unordered information pieces like company name, URL, author name and similar attributes. The taxonomic attributes provide more information about their value domains: enumeration of legal values and a taxonomy of the values. Taxonomic attributes are useful especially for classification of versioned entities. Let us explain the concept of taxonomic attributes on several examples.

Consider a version attribute *operating system* (`OS`) identifying the operating system required by a particular entity version. You can leave the value domain of the attribute `OS` unspecified, however, this significantly reduces the retrieval capabilities - in general you can test the entity versions for equality only: `OS="Windows 95"`. You are unable to issue queries like: *Get me all components for any Win9x compatible OS*. or *Get me components for any UNIX operating system.* The simplest possible solution to this problem is to create a category tree for the `OS` attribute. See Figure 1 for a fragment of such a category tree.

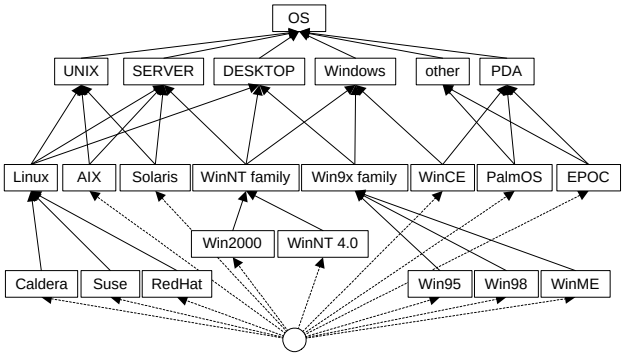

Figure 2. An operating systems poset.

The category tree on Figure 1 can easily solve the questions mentioned in the previous paragraph, but it can be very constraining to try to construct a category tree in every situation. Moreover, sometimes it is impossible to create a tree because some nodes belong to more than one branch – this problem typically occurs when trying to mix and combine several criteria into one taxonomy. Considering our OS example: some users can prefer the classification based on OS families (UNIX, Windows, ...) while others can prefer categories based on the type of usage of operating systems (server, desktop, PDA, etc.). The result of this analysis is obvious: the tree-structure is not sufficient to represent the knowledge about version attribute value taxonomy in a general case. We have to look for a more general structure to represent version attribute value taxonomies. Such a structure should support multi-criterial classification of elements. The simplest algebraic structure which fulfills our requirements is the partially ordered set (*poset*) structure [3]. Figure 2 shows a poset covering the knowledge represented by both the classification criteria mentioned above applied the `OS` domain. The true power of the poset representation becomes more evident when thinking queries like: *"Get me UNIX operating systems usable for desktop."* or *"Are there any Windows-based operating systems for PDA?"*

Posets are used in our versioning model to structure values of taxonomic attributes. Each taxonomic attribute provides at least one poset taxonomy covering whole attribute

value domain. Any taxonomic attribute can also provide unlimited number of additional taxonomies describing other aspects of the attribute. The additional taxonomies can cover only a subset of the attribute value domain and provide any *partial order* of these values. Using the taxonomic attribute `OS` (see above) we can easily define for example relation *successor/predecessor* which will describe evolution of operating systems. This feature enables queries like: *Get components of type T for Win95 or newer.* Figure 3 demonstrates a fragment of such an optional taxonomy.

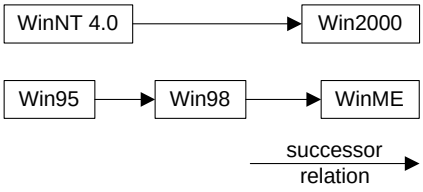

Figure 3. A successor relation fragment.

The optional taxonomies are not limited to a single attribute. In general case – following the mathematical definition of the term *binary relation* – a relation can be defined between any pair of the taxonomic attributes. This enables us to define *constraints*. For example having the taxonomic attributes `OS` and `PROCESSOR`, we can define the `runs_on` relation describing the matrix which operating systems are implemented for a particular processor and vice-versa.

Summary. Taxonomic attributes with their mandatory poset taxonomies and additional optional taxonomies bring a new power to describing versioned entities using version attributes. When using taxonomies, the user knows not only the value of a particular attribute but also the "meaning" of the value. This knowledge enables a new category of advanced queries allowing to extract more information from the version repository.

3.3. Version repository

There are two possible approaches to version attribute and relation handling. One possible approach is to append a special meta-information section to each distribution package. Such a meta-information section contains attributes and relations information associated with a particular distribution package. In case a given package references other distribution packages, the user is forced to download the referenced packages recursively in order to access their meta-information. This approach leads to a distributed version model without any central point of control. Other possible solution is based on the *version repository* concept. A version repository contains version-related information for all versioned entities maintained in a particular software system. Such a version repository (in contrary to the first solution) can also store pieces of information which do not belong to any particular distribution package - system-wide version information (such as attribute value taxonomies).

We chose the repository-based approach with separate version and component repositories. The version repository serves as an search engine which allows the user to query the repository and obtain a set of UUIDs of a particular entity type as a result. The UUIDs then can be used to retrieve the physical components from the component repository. The chosen approach clearly de-couples versioning and a particular component technology and therefore allows us to study also the software systems that offer limited or no versioning support. The system-wide version repository also adds possibilities for global analysis of version information across the whole modeled system.

We use the concept of object oriented database to model our version repository. There is a clean mapping between versioned entities, entity attributes and entity relations on the versioning side and objects, object attributes and object relations on the database side. The versioned entity UUIDs are mapped directly to the persistent object references - often referred to as *object identifications* (OID) using object database terminology. Using the object-database paradigm we can easily characterize potentially any component/versioning model.

The proposed versioning model goes a step beyond a generic object database concept by emphasizing the versioning-related aspects. It allows to define both the entity relations and attribute taxonomies, including their relational properties (*reflexivity*, *symmetry*, *transitivity*), in a declarative way. For example, the `successor` taxonomy on Figure 3 is defined as transitive, so the user is required to enter only the immediate successor for each value. The transitive closure is built by the versioning engine. The same holds for the poset taxonomies like the `OS` taxonomy on Figure 2. The user only defines the properties (reflexivity and transitivity in this particular case) and enters the information about the *arcs* between the nodes. The reflexive and transitive closures are built by the versioning engine, so that the query evaluation can accommodate not only the arcs but also the *paths* visiting multiple nodes of the poset.

M-cube is the code-name of the prototype version repository implementation. We made this prototype to be our testing environment which enables us to test the version model described in the previous sections. M-cube is implemented on top of a relational database - it uses a relational database as the underlying data store. The business logic of the repository is implemented using the PHP scripting language. Any HTML 3.2 compliant web browser can be used to access and manipulate the repository remotely.

The version repository splits into two logical parts: the meta-repository and the repository instance. The meta-

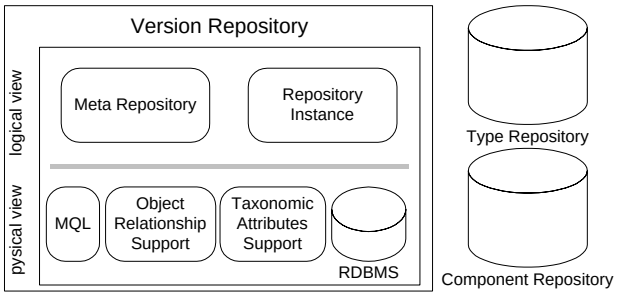

Figure 4. Version repository and its relation to type and component repositories.

repository contains definition of entity types, their relations and taxonomies. The meta-repository definition process is similar to creating a database schema in a relational database. The fact our version repository prototype is not bound to a specific component model allows us to use the prototype repository to model our SOFA/DCUP components system as well as other software component frameworks like EJB or COM. Having defined a meta-repository, the user can enter information about a particular software system into the version repository instance and then use the tools provided by the versioning engine to track the properties and relations of the described components. The M-cube query language can be used to formulate advanced queries to the version repository instance.

3.4. The M-cube query language

The M-cube query language (MQL) is designed to correspond with our version repository object model. It is a special-purpose query language optimized to be syntactically effective for complex queries over the version repository. It uses syntax similar to SQL, but the M-cube query language is more compact and uses only the elements required for the version repository retrieval tasks. These features make the language easy-to-learn for anyone familiar with SQL and at the same time they allow construction of a relatively simple parser for the language. Some features, like constraining a descriptive attribute or navigating through a relation, can be achieved by standard query languages like SQL. Some features like constraining a taxonomic attribute or navigating through a possibly recursive relation go beyond the possibilities of SQL which relies on the relational algebra.

MQL allows the retrieval of a set of entities of a particular entity type defined in the version repository. It does not support, unlike SQL, the more detailed specification of the result set content, e.g., the specification of a subset of elements (in SQL context called columns or fields) constituting a given object type. Given an entity type (a *domain constraint*), it simply returns a list of UUIDs conforming to the provided selection criteria. The example below shows the most simple M-cube language query and its SQL semantic "equivalent":

```
M-cube: SELECT component_frames
SQL:    SELECT id FROM component_frames
```

MQL is able (like SQL) to constrain a result set using selection criteria applied to attributes of a given object type. MQL uses a bit different syntax for this kind of constraint. The main reason for doing so is the fact M-cube directly supports the taxonomic attributes which imposes additional syntax requirements. The second example shows the ability to constrain a selection to a specific value of a descriptive attribute (`ProductURL` in this case).

```
SELECT component_implementations HAVING
 (ProductURL="http://abc.com/product") OR
 (ProductURL="http://abc.com/other/product")
```

The following example demonstrates constraining a *taxonomic* attribute (`OS`). The query returns all of the `component_implementations` having the attribute `OS` equal-to or related-to the value `Linux` in terms of the *`base`* taxonomy bound to the attribute `OS`. Considering the sample taxonomy on Figure 2, the query returns the `component_implementations` having `OS` in the following set: `Linux`, `Caldera`, `Suse`, `Red Hat`.

```
SELECT component_implementations
HAVING (OS base "Linux")
```

The following query employs an optional taxonomy *successor* defined for the taxonomic attribute `OS`. It retrieves all of the `component_implementations` claiming to require the `"Win95"` operating system or newer - in terms of the *`successor`* taxonomy (see Figure 3).

```
SELECT component_implementations
HAVING (OS successor "Win95")
```

MQL features direct support for constraining a query using relations. The SQL contains support for a similar feature from version SQL92 using keyword `JOIN` and its variants. Former versions of SQL used equality constraint on corresponding columns of joined relations. The M-cube query language uses three keywords in the context of relations: `JOIN`, `WITH` and `THROUGH`. `JOIN` begins the relation restriction part of the query. `WITH` is used to specify a target object type and `THROUGH` defines the relation.

```
SELECT component_frames
JOIN (WITH Interfaces THROUGH provides
      HAVING (Name="java.io.DataInput")
)
```

The query above returns the `component_frames` providing the `"java.io.DataInput"` interface. The query demonstrates direct support for navigating through relations, which is a very important feature of the M-cube query language. The last example demonstrates a more complicated relation chain. The query retrieves the `component_implementations` implementing the `"java.io.DataInput"` interface. Please note the example uses the SOFA/DCUP component model so the terms like `architecture` or `component_frame` are defined in the version repository (meta repository part) and they are used to express the exact component's properties:

```
SELECT component_implementations
JOIN (WITH architectures THROUGH implements
 JOIN (WITH component_frames THROUGH refines
  JOIN (WITH interfaces THROUGH provides
  HAVING (Name = "java.io.DataInput"))))
```

The examples above were chosen to show all of the features of the language, while remaining relatively simple, in order to allow the reader comprehend the basic language principles. For a complete M-cube Query Language reference and grammar specification see [7].

4. Conclusion

The proposed approach to managing versioning in the context of component software assembly process fulfills our requirements as stated in the section 2. The proposed versioning model is focused on identification and description of versioned entities which occur during the component software assembly and configuration phase. (1)

The model does not impose usage of any specific versioning system during the development phase. The meta-modeling capability allows the version repository user to define new first-class entities if required so in a given component model. The SOFA/DCUP entities are pre-defined in the prototype repository instance. (2)

We propose a reliable naming convention which allows to uniquely identify the versioned entities in time and space. We prefer usage of a human readable UUIDs assigned in a two-step process, however, we do not claim this is the only eventuality. (3)

Our version model supports evolution of versioned entities (revisions) employing the concept of the user-defined attribute taxonomies and entity relations (e.g. a user-defined successor/predecessor taxonomy and/or relation). Our approach to the definition of revisions is much more flexible than the typical approach based on revision numbers. The proposed version model also directly supports variants (parallel versions) which are implemented using the entity attributes. Special attention is paid to the taxonomic attributes which are very powerful as they directly encode attribute value relations and value domain structure. (4)

By moving the scope of version identification from the individual versioned entities (interfaces, components) to distribution packages, we reduce the versioning issues when developing particular distribution packages. The developer needs to deal with a version identification only when integrating distribution packages but not when developing the packages themselves. This feature reduces the overhead of versioning for the developer and allows the version model to scale to the enterprise level. (5)

For the prototype implementation we have developed a special query language (M-cube Query Language) which seamlessly incorporates the version model and allows for flexible and powerful version information retrieval. The language can be used during the design time as well as during build time or run time to query the repository and select proper components and configurations involved in the system assembly process. The query language also enables the intentional versioning approach in the context of software assembly and configuration. (6)

The prototype implementation of the version repository allows us to evaluate the version model and helps us to evolve the proposal towards a practical application in the SOFA/DCUP architecture development.

4.1. Acknowledgements

Many colleagues in the Department of Software Engineering and other people contributed to this paper by their advice and expertise. I would like to thank namely to the following people for their outstanding support: Frantisek Plasil, Petr Tuma, Vladimir Mencl, Premysl Brada and Tomas Macek. This work has been supported by the Grant Agency of the Czech Republic (project #102/03/0672).

References

[1] G. Aschemann and R. Kehr. Towards a requirements-based information model for configuration management. In *Proceedings: ICCDS'98*, May 1998.

[2] R. Conradi and B. Westfechtel. Proceedings: Towards a uniform version model for software configuration management. In *Proceedings: ICSE'97 SCM-7 Workshop*, May 1997.

[3] B. A. Davey and H. A. Priestley:. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[4] M. de Jonge. Source tree composition. In C. Gacek, editor, *Proceedings: Seventh International Conference on Software Reuse*, volume 2319 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, April 2002.

[5] J. Estublier and R. Casallas. The Adele configuration manager. In *Configuration Management*, number 2 in Trends In Software. J. Wiley and Sons, 1994.

[6] P. H. Feiler. Configuration management models in commercial software development environments. Technical Report CMU/SEI-91-TR-7, SEI, March 1991.
[7] J. Gergic. The M-cube Query Language syntax reference. http://nenya.ms.mff.cuni.cz/∼gergic/MQL.html.
[8] V. Mencl. Component Definition Language. Master's thesis, Dept. of SW Engineering, Charles University, 1998.
[9] R. Mili, A. Mili, and R. T. Mittermeir. Storing and retrieving software components: A refinement based system. *IEEE Transactions on Software Engineering*, 23(7), July 1997.
[10] Microsoft .NET. http://www.microsoft.com/net/.
[11] DCE 1.1: Remote procedure call. The Open Group, 1997. http://www.opengroup.org/dce/.
[12] F. J. Oles. An application of lattice theory to knowledge representation. Technical report, IBM T.J. Watson Research Center, October 1996.
[13] Object Management Group: Trading Object Service Specification. http://www.omg.org/, 1997.
[14] Object Management Group: CORBA V2.2 - The Interface Repository. http://www.omg.org/, 1997.
[15] F. Plasil, D. Balek, and R. Janecek. DCUP: Dynamic component updating in Java/CORBA environment. Technical Report 97/10, Dept. of SW Engineering, Charles University, Prague, 1997.
[16] F. Plasil, D. Balek, and R. Janecek. SOFA/DCUP: Architecture for component trading and dynamic updating. In *Proceedings: ICCDS'98*, Annapolis, Maryland, USA, May 1998. IEEE CS Press.
[17] F. Plasil, S. Visnovsky, and M. Besta. Bounding component behavior via protocols. In *TOOLS USA '99*, volume 30 of *TOOLS*, pages 387–398. CS IEEE, August 1999.
[18] D. Rogerson. *Inside COM*. Microsoft Press, 1997.
[19] RedHat Package Manager. http://www.rpm.org/.
[20] G. Snelting. Concept analysis—a new framework for program understanding. In *Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*, pages 1–10, July 1998.
[21] ISO/IEC 9075:1992. (SQL92).
[22] W. F. Tichy. RCS – a system for version control. *Software – Practice & Experience*, 15(7):637–654, July 1985.
[23] A. Zeller and G. Snelting. Unified versioning through feature logic. *ACM Transactions on Software Engineering and Methodology*, 6(4):398–441, October 1997.