Managing Evolution of Component Specifications using a Federation of Repositories[⋆]

Vladimír Mencl and Petr Hnětynka

{mencl,hnetynka}@nenya.ms.mff.cuni.cz
Department of Software Engineering
Faculty of Mathematics and Physics
Charles University in Prague, Czech Republic

Abstract. When developing software components in a distributed environment, certain issues regarding the component specifications must be handled: 1. specifications need to consistently reference concrete versions of other specifications, while easy transition to refer to the new versions in either a specification or an implementation is demanded; 2. a hierarchy of description languages exists; specifications declared in additional description languages need to reference specifications already declared in preceding languages; 3. specifications are expected to evolve (potentially in multiple lines of development) while being at the same time used on other, potentially disconnected nodes; 4. a *successor* relation among the versions is desired.

In the common solutions available, these issues are usually addressed only by a simple unique version identifier. We address the issue using a concept of a repository holding the specifications, assuming existence of a federation of the repositories. Besides generic reasoning, we propose a concrete realization of the concept, the Type Information Repository (TIR), integrated into the SOFA component model. Meanwhile, we also derive general requirements for a version model to be used in such a repository and we also describe the concrete version model used in TIR; we show its integration with the SOFA Component Definition Language (CDL) and rules to be imposed on combining different versions of related specifications in CDL declarations. The integration also includes rules for automatic selection of the appropriate version (*profiles* are used for this).

The concept of a repository has been proven as realistic by implementing TIR and a CDL compiler storing compiled specifications in the repository. Furthermore, a graphical interactive interface to both the repository and the CDL compiler has been developed as a module for the Forte environment, allowing to browse the repository and interactively compile CDL source files.

⋆ This work was partially supported by the Grant Agency of the Czech Republic – project 201/99/0244.

1 Introduction

When developing software components in a distributed environment, certain issues arise with managing component specifications. Section 2.5 elaborates the issues in detail, here we briefly present them and lay out the goals of the paper.

Component specifications need to consistently reference concrete versions of other specifications. Also, a hierarchy of description languages exists, parts of specifications declared in additional description languages need to reference specifications already declared in preceding languages. Furthermore, specifications are expected to (potentially concurrently) evolve; consistency of references among different versions must be maintained, while providing support for easy transition to the new specifications is desired, as well as establishing a *successor* relation among the versions. In the common solutions available, these issues are usually addressed only by a simple unique version identifier; the issue of the successor relation is usually not addressed, and neither is the issue of providing a language mapping invariant with the version identifier.

We believe that component specifications must evolve and that multiple versions of a specification must be available; this can be achieved by holding them in a repository. We introduce extensions to the Component Definition Language (CDL) to allow for version selection, together with providing clear rules for automatic selection of the appropriate version (we use *profiles* for this). In this paper, we describe our version model, its integration into the CDL language and rules imposed on combining different versions of related specifications in CDL declarations.

We address the issue by proposing the concept of a federation of repositories holding the specifications, specifying criteria for such a repository, independent on a concrete component model. We follow by specifying a repository for the SOFA component model [2] and later by describing a proof-of-the-concept implementation of this repository.

2 Background

2.1 SOFA/DCUP

SOFA [1, 2] is a project aiming to provide a platform for software components; in SOFA, applications are created by composing software components together. Software components can themselves be composed from other software components, in this case, we speak about *compound components*, or they can be the basic building elements already containing directly the implementation, we call them *primitive components*.

2.2 Component Definition Language – CDL

The *Component Definition Language (CDL)* [3], is used to describe interfaces and internal structure *(architecture)* of SOFA components. The language is syntactically derived from CORBA IDL [7], the language extends the feature of IDL by making a shift from descriptions of *interfaces* of services to specifications of software components. (The extensions to IDL proposed in the Corba Component Model [9] specification are similar to some aspects of the CDL.)

In CDL, SOFA components are described at several levels. Besides elementary declarations of *types* and *interfaces* (which have the same semantics as in the IDL language), CDL describes the black-box view of a component – the component *frame* and gray-box view of a compound component, the component *architecture*.

Component frame lists the services provided and required by the component, eventually parameterizes the component by declaring *properties* of the component.

Component architecture describes the structure of the component at the first level of nesting in the component hierarchy, only frames of subcomponents are specified, the decision of selecting the appropriate architectures implementing the respective frames has been deferred. Ties have to be established to link interfaces provided by the component and services required by subcomponents to services required by the component (obtained from the outer environment) and services provided by subcomponents. These ties can take either the form of a *binding* (at the same level, e.g., binding a service required by a subcomponent to a service provided by another subcomponent), *delegation* (delegating a provided interface to an interface provided by a subcomponent) or *subsumption* (using a service obtained as a requirement of the component to satisfy requirements of a subcomponent).

Figure 1 gives an example of CDL declaration of an interface, a component frame and a compound architecture.

```
interface Login {
   CentralPlayerServices login(in string who);
};
frame Client {
   provides:
      Client iClient;
   requires:
      Login iLogin;
      CentralPlayerServices iCPS;
};
architecture CUNI GameGen implements GameGenerator {
   inst GameGeneratorDBServices aGGDBS;
   inst ConfigurationFileParser aConfig;
   inst GameGeneratorFunctionality func;
   bind func:iConfig to aConfig:iConfig;
   bind func:iGGDB to aGGDBS:iGGDB;
   subsume aGGDBS:iDatabase to iDatabase;
};
```

Fig. 1. A sample declaration in the Component Definition Language (CDL)

2.3 Development Process in SOFA

Development of a software component can be divided into two parts, development of the component's specification and later implementing this specification. These roles can be performed by different parties; furthermore, the parties can reuse results produced by other parties in both the specification and the implementation. To demonstrate this, let us give a real-life example:

Netscape provides an interface for plug-ins to extend the functionality of the browser, this interface corresponds to a component frame. The list of service the browser provides to the plug-in (or, more precisely, which the plug-in can require from the browser), can include service which are maintained by other parties, e.g., JDBC$^{\mathrm{TM}}$, which is maintained by Sun Microsystems.

This component frame can be implemented by multiple vendors, each either directly providing an implementation, or assembling a compound component from other components, possibly developed by other parties.

At the *design time* [11] of a compound component, the first step is to design its architecture (which is the gray-box view of the component and specifies only subcomponents only by their frames). The second step is to create an *assembly descriptor*, which specifies both the architecture and the assembly descriptor for each subcomponent, thus recursively binding all nested component frames to a concrete architecture. The next step is to divide the application into *distribution units*, which results into a *deployment form.* Filling this form with exact locations for execution yields a *deployment descriptor.*

2.4 SOFAnet and SOFAnode – Deployment & Execution Model

A single deployment environment in SOFA is called a *SOFAnode*; SOFAnodes are interconnected to form the *SOFAnet.* A SOFAnode consists of the following parts: the *template repository (TR)*, the *Run-part*, the *Made-part*, the *In-part* and the *Out-part.* Every SOFAnode must have a TR, the other parts are optional. Their presence is to be determined by the role the SOFAnode in the flow of the software components, the roles we have identified include a component producer, a pure end-user, a smart user (eventually developing custom components), a retailer and combinations of these. A more detailed elaboration can be found in [1].

The template repository contains component implementations together with their descriptions. (Note that the keyword *template* denotes, that the component implementations stored are actually templates used to create component *instances.*) The Made part is used to create new components and insert them into the TR, the In and Out parts serve to transfer components from one SOFAnode to another and the Run part provides environment for running the component instances; at least the Run part is assumed to be distributed across multiple hosts.

When an application (a top-level component) is launched, component instances are created in the respective units of the Run-part *(deployment docks)* according to the application's *deployment descriptor.* Components can be either pre-installed at the respective deployment docks, or the deployment docks can dynamically download the component implementations from the TR and on-the-fly generate connectors (stubs & skeletons) for interconnecting the components via the respective middleware.

Figure 2 shows a sample scheme of several SOFAnodes connected to form a SOFAnet.

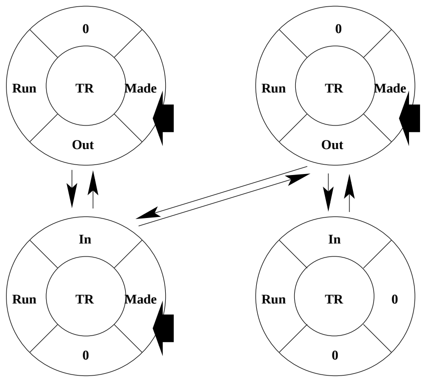

Fig. 2. Scheme of sample SOFAnodes interconnected to form a SOFAnet

2.5 Evolving Specifications Problem – Lack of a Repository

As it was demonstrated in the previous section, parts of specifications are likely to be developed separately, either by different development groups within a single organization, or by different organizations. It is expected that specifications will be replicated to all sites using them, however, at the place of their origin, they are expected to evolve. And eventually, already during the development of the new versions, other groups within the organization may require to access them. Also, let us mention that SOFA contains a hierarchy of description languages – e.g., deployment descriptors reference frames and architectures declared in the CDL language.

This gives us a need to have a facility, which would allow to distinguish among different versions of a specification, allow the coexistence of the respective versions at the same node, and allow easy integration of new versions of either local or third party specifications. Support must be provided for references among parts of specification declared in different specifications languages (at different levels in the hierarchy of languages), these references must be fixed to a particular version, while being able to easily develop new specifications making use of new versions of the referenced specifications. Also, redistribution of new versions to sites interested must be handled.

These needs can be solved with a repository holding the specifications. In the following sections, we describe architecture of a repository we developed; we reason that such a repository satisfies the needs stated above (i.e., support for 1. distributed creation and evolution of specifications; 2. hierarchy of languages; and 3. easy transition to new versions).

3 The Repository Concept

In this section, we introduce the concept of the repository, independent on the underlying component model. To demonstrate the feasibility of the concept, we show an application of the concept integrated into the SOFA component model.

3.1 The Role of a Repository

The task of the repository is to provide storage for individual versions of specifications and to guarantee consistency of the specifications both with respect to (possibly) concurrent development within a single organization and with respect to development done in other organizations (though each specification is to be evolved only by one organization).

A repository will assist the development of the component specifications and will also provide access to the specifications for further steps of component development; this may cover code generating, creating distribution packages or executing the components. Each organization either developing specifications or producing component implementations should be running an instance of the repository, these instances would together form a federation.

Operations the repository must support are *retrieving* existing specifications and *adding* either new specifications or new versions of already existing specifications. We believe that support for multiple versions of a single specification must be provided. When a specification is modified, it is not sufficient to create a new specification with a completely new identification, a framework must exist for establishing a *successor* relation among the new and the old versions of the specification. This task should be addressed by the version model.

We also believe, that it is necessary to allow for concurrent development of the specifications by different groups within a single organization, to achieve this, support must be included both in the repository itself and in the underlying version model. (In our solution, we use the concept of *branching*, used among others also in the CVS [10].)

3.2 The Type Information Repository (TIR) in SOFA

In SOFA, two instances of a repository exist. Inside the Made-part, *Type Information Repository (TIR)* is primarily focused on development of specifications. Its tasks include creating new versions of specifications, creating branches and, under certain circumstances, deletion of particular versions of specifications.

A different instance of the repository is present in TR, providing a differing functionality. As the task of TR is to store component templates and provide them to other parts of the SOFAnode (*Run* and Out), the main task of the TR instance of the repository is to maintain the type information of the components installed on the SOFAnode; this includes cooperating with TIR as well as other parts of the SOFAnode according to the following scenarios:

1. release of a specification from TIR to the SOFAnode (the TR) – running, testing and debugging the components
2. release of a specification to the SOFAnet – after this step, a specification *should not* be deleted
3. transferring a specification from one SOFAnode to another (either as a part of a distribution package or standalone)
4. requesting a new version of a specification, either querying or subscribing for notification on updates
5. providing the contents of one repository visible to the other – (TR to TIR – creating alternative implementations; TIR to TR – instant debugging by running the components in the Run-part directly from the Made part)

Figure 3 demonstrates integration of the concept of the repository into the SOFA component model.

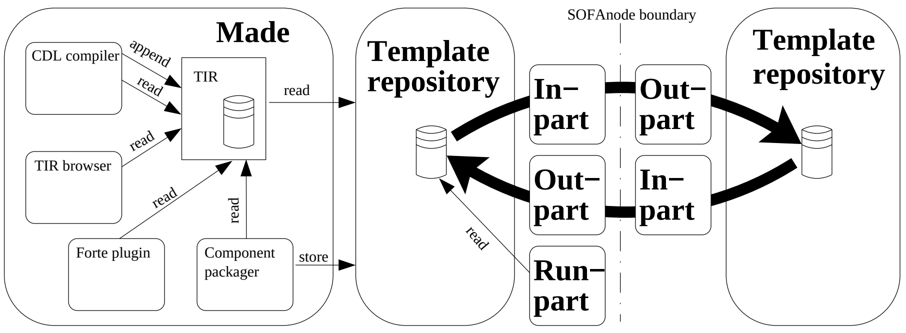

Fig. 3. Interaction of TIR with other elements of the Made part and with TR

3.3 Requirements for a Version Model

In a repository a *version model* has to be employed, i.e., that specifications have to be identified by their name and by a version identifier (VID) identifying the particular revision of the specifications. A version model has to specify the rules for creating and handling the VIDs.

We already stated some of the requirements put on the underlying version model in sect. 3.1, namely establishing a successor relation and supporting concurrent development (branches). Further, we believe version model should keep a record of the evolution history; as a very convenient approach we consider recording the history at least partially inside the version identifier, e.g., by including the version identifier of the preceding version.

An open issue is, whether a version model *must* provide globally unique VIDs. It may be sufficient to provide identifiers unique within a single node only; globally unique identifiers can be achieved by assigning to each node a unique prefix within the global namespace.

The following operations have to be supported by a version model:

compare – test the successor relation; either *unrelated*, *identical*, *successor* (direct or indirect) or *weak successor* (across branchings)

has-successor – test, whether a successor to a particular version exists (this operation is reliable only on the node where the specification is local)

create-initial – creates the initial version

create-next – creates a new version, as a successor of the original version. This operation can be invoked only on *local* specifications and at most once.

create-branch – creates a new version in a new *branch*. The new version becomes the initial specification of the new branch and is related by a *weak* successor relation to the original version.

is-local – test, whether the specification is local to the node. A VID should provide sufficient information to resolve this operation.

3.4 Version Model in TIR and SOFA

When designing a version model, the two basic questions are: 1. How to assure uniqueness of VIDs? and 2. What to construct a VID from?

In a centralized environment (a single node issuing identifiers), a solution can be easily found, relying, e.g., on sequence numbers; some systems also use timestamps and/or (pseudo-)random numbers. In a distributed case, global uniqueness can be achieved either by splitting the global specification name-space, limiting each node to only to the sub-space it has been assigned (disjoint with name-spaces of other nodes), or by reliably constructing unique VIDs. Global uniqueness can be achieved by including an identification of the node in VID, either administratively assigned or it derived, e.g., from a hardware component of the system. Such identification can be combined with a timestamp and/or a random-number.

To provide predictability and readability of VIDs, we selected to use an administratively assigned identification of the originating node, joined with a mechanism based on sequence numbers. Though we assume the global name-space to be administratively divided (in a Java-like reverse-DNS scheme), our version model does not rely on the separation of the name-space and provides globally unique both VIDs and branch identifiers.

In the previous section, we agreed that a successor relation among versions should be maintained. The (meta)information constituting this relation can be kept either separately in the repository (as meta-objects), or inside the VID itself. We chose the latter, storing this information in VIDs in a distributed fashion.

Each VID contains an identification of its branch and a sequence number; from that, VID of either the preceding or the succeeding version can be derived. The benefits of the readability and the evolution history overweight the burden of additional data put on the version identifier. Format of a VID is:

```
version_id : SOFANODE_ID ':'  branch_id ':' SEQUENCE_NUM
branch_id :  /* empty */ |
       version_id ':' SEQUENCE_NUM
```

This approach contains a potential disadvantage of releasing information about internal development to the outside world. Also, note that the length of a VID can grow linearly with respect to nested branchings, however we do not expect an increased number of nested branchings to occur; in practice, the level of branching rarely exceeds 1. The notion of branches used here draws on the concepts used in CVS [10], practical experiences show, that this potential increase of length does not impose a problem.

3.5 Integration of Versioning Scheme and CDL

To allow for convenient managing of different versions of specifications, extensions have been made to the CDL language. These extensions assume a certain level of cooperation from the compiler, the compiler may either use settings in the environment or interact with the user.

The aim of these extensions is to provide direct reusability of source code of CDL specifications even in the case when a specification being referenced evolves and the new version is about to be used.

A specification (a type, interface, frame, . . .) can be referenced either by its full identification (name and VID), or by only its name, omitting the version identifier. In such case, the proper version will be selected according to the environment settings (including the tags), the *profile*, eventually searching for the most current version along the proper branch. The algorithm will be described in sect. 3.6, together with the definition of the tags and the profiles.

A special issue to discuss is the granularity of versioning. In CDL, each declaration, which can be externally referenced, is subject to versioning. This includes all declarations of architectures, frames, interfaces and types, located either at the top level or directly inside a `module` declaration. Declarations nested inside other declarations (e.g., `struct`s declared inside other `struct`s and/or interfaces) inherit the version identifier of the embedding declaration, and also their life-cycle is bound with the life-cycle of the embedding declaration.

3.6 Version Selection – Profiles and Tags

When a new version of a CDL specification is being compiled, two issues must be handled: 1. to look up the appropriate version of specifications being referenced 2. when creating a new version of a specification, to select the version to which the new version is a successor; and eventually to decide whether a new branch should be created. To allow for these issues to be handled properly, two concepts were introduced, *tags* and *profiles*.

Tags are a concept inspired by CVS; tags are textual identifiers used as a substitute for a concrete VID. We recognize *version tags*, identifying a particular revision, and *branch tags*, identify a whole branch; when a branch tag is used, the most recent version from the branch is used. (Note, that for non-local branches, latest updates on the branch may be missed.)

Profiles were introduced to handle multiple lines of development, a profile serves as a snapshot of a set of the most recently used VIDs along that development line. For each CDL name involved in the profile, the profile contains VID of the version most recently used within the profile. When a new version is created with a profile being active, the new VID is stored in the profile; the version previously stored in the profile is treated as the predecessor of this new version. When a specification is referenced only by its name, the VID is retrieved from the profile using the name.

The algorithm for selecting the appropriate version is:

1. When a version is specified by a version identifier or a tag, that version is retrieved (tag is first tried as a version tag; then as a branch tag)
2. When only the entity name is specified, a profile is active, and the name of the entity exists in the profile, the version identifier stored in the profile is used.
3. otherwise, the *head revision is used* – the most recent version along the main branch (in non-local case with the same limitations for retrieving the most recent version).

When a new version is to be created, the following happens:

1. if this is the first version of the object to be created (no previous version exists), the *initial version* is created.
2. otherwise, if a profile is active and the entity exists in the profile, then
   (a) if create-next can be invoked on the version stored in the profile (is local and has no successors), create-next is used
   (b) otherwise, if the version is local, a new branch is created from the version stored in the profile
   (c) otherwise, an error is reported – creating a branch from a non-local version would violate the administrative division of the name-space
3. otherwise, an error is reported – it was not possible to detect the version to which this version is a successor
4. if the version is successfully created and a profile is active, the version is added to the profile.

3.7 Mapping of Versioned CDL to Java

With a version model employed, mapping of CDL to implementation environments (only Java, currently) becomes a question. Without a version model, CDL names might be mapped to implementation names by a straightforward mapping (derived from *IDL to Java* mapping [8]). Making the implementation name depend on the VID would be extremely inconvenient for developers; implementation source files would have to be altered whenever the specification is modified, for these reasons, we decided not to include the VID in the CDL mapping.

This introduces a danger of name-clashes among different versions of a single specification (mapped to the same name in the target environment). We present a solution for Java here, we believe, that solutions for other (future) platforms (C++) can be derived in a similar fashion.

At the application-composition level, this issue can be worked around, e.g., by instantiating individual components in separated name-spaces, each bound with an instance of a classloader.

At the level of architecture of a compound component, the problem can still be overcome the same way as illustrated above (though adapting connectors crossing the boundaries of the incompatible name-spaces may be necessary).

For the case of a single component frame, name-clashes must be avoided. Within a component frame, only non-clashing specification elements are allowed (with respect to the transitive closure along references among the elements). A consequence of this restriction is, that frames cannot provide and/or require different versions of a specification element. Though this prohibits creating components being "backwards compatible" (providing both the old and the new version of an interface), the backward-compatibility issue can still be successfully handled, either by a wrapper component, or by adapting connectors.

The solution for the future C++ environment may be based on separate include files and link modules (by using either `static` symbols or eventually renaming symbols). The scheme used will depend on the exact architecture of the C++ environment.

4 Practical Experience, Evaluation and Open Issues

TIR, which is the application of the concept of a repository to the SOFA environment, is realized in a prototype implementation (in the Java/RMI environment). This implementation encapsulates a set of interfaces related to the repository itself and a set of interfaces specific to the CDL language; extensions for other languages can be added in a straightforward way (e.g., for the *Update Description Languages – UDL*). A CDL compiler interacting with the repository is available, either as a command-line tool, or as a part of a module for the Forte environment; this module also includes a browser of the content of TIR.

Modifications to the content of TIR are done within transactions managed by the client side (the CDL compiler). This maintains the content consistent (e.g., with respect to compile errors in later parts of a source file), and also allows a user to review the impact of the changes being made and eventually to abort the whole transaction. Concurrent access is handled by locking sub-space of TIR accessed within a transaction for the duration of the transaction.

A task for the future is to provide a facility for enriching component distribution packages with an external representation of the type information related to the component; for this task, XML will be employed. Special care will be given to representing references to concrete versions of other elements of the description.

An open issue remains, whether the administrative separation of the namespaces should be strictly enforced. An alternative to the approach presented here is to permit creating local branches for further development of remote specifications [6]. Though such approach is feasible, it suffers from certain disadvantages, especially that the resulting graph of versions of a specification with respect to the successor relation is a forest instead of a tree. Though a specification is to be changed only by its maintainer and the whole concept seem as misleading, this way, a solution would be found for the problem of *transfer of control* of a specification among vendors. Deletion of specification is also an open issue, as reuse of VIDs might lead to collisions; currently we agree, that once a specification is released from the originating node, it may not be removed.

5 Related work

CORBA *Interface Repository (IR)* [7] provides runtime access to type information, however no standardized mean for modifying the content of the repository exists. Though a version identifier can be associated with a specification, repositories are not required to support storing multiple versions of a type. Also, no version model is defined for the version identifiers and therefore there is no relation among different versions.

Microsoft COM/DCOM uses the *Windows registry* and the concept of a fixed size *Global Unique Identifier (GUID)*, employing both a hardware-based identifier and pseudo-random values. This approach implies certain disadvantages, namely: 1. under certain circumstances, the method might not be reliable. 2. the space of potential identifiers is limited and might be eventually exhausted 3. the identifier is opaque. No link to the place of origin of a GUID is managed (the hardware identification does not suffice and may be even subject to change), neither is managed a relation among different versions. In the DCOM environment, this relation is usually managed by inheritance, but such approach burdens the type information itself with an inheritance chain across all preceding versions.

The Gnome project has a similar registry called *GConf*, which is a tree-structure holding key-value pairs; an interesting feature of GConf is a notification service. However, the notification service cannot be used in TIR, as we are not requiring connectivity among nodes of the federation.

CVS is a RCS-based versioning system providing versioning system, its version model (and the notion of branches) has been used in this work. However, CVS is file-oriented and does not handle the distributed case.

A versioning scheme for SOFA has already been proposed in [13], however the scheme is focused on capturing the type of changes among succeeding versions, and not take into account the distributed and concurrent nature of evolution and neither does consider fragility of references among specifications.

6 Summary

It is generally agreed that multiple versions of a specification of a component must be able to coexist within a single node. However, common solutions provide little more than a simple unique version identifier; the issue of establishing a successor relation among the versions is usually not addressed, and neither is the issue of providing a language mapping invariant with the version identifier.

We address these issues by the concept of a repository, together with a version model providing evolution history. A federation of either loosely connected or disconnected repositories redistributes the specifications among the nodes, either attached to a component or standalone. Support is provided for coexistence of multiple versions, preserving a successor relation among the versions, consistency of references to a particular version as well as for easy transition to a new version of a specification for both the specification and the implementation sources.

Having finished the design and implementation of such a repository for the SOFA component model, our efforts now aim at inter-repository communication and transfer of component specifications.

References

1. F. Plášil, D. Bálek, R. Janeček: *SOFA/DCUP:Architecture for Component Trading and Dynamic Updating*, Proceedings of ICCDS'98, May 4-6, 1998, Annapolis, Maryland, USA, IEEE CS Press 1998
2. F. Plášil, D. Bálek, R. Janeček: *DCUP: Dynamic Component Updating in Java/CORBA Environment*, Tech. Report No. 97/10, Dept. of SW Engineering, Charles University, Prague
3. V. Mencl: *Component Definition Language*, Master Thesis, Charles University, April 1998, Advisor: Nguyen Duy Hoa
4. Z. Petrová: *Update description language*, Master Thesis, Charles University, August 1999, Advisor: Vladimír Mencl
5. P. Hnětynka: *Managing type information in an evolving environment* Master Thesis, Charles University, August 2000, Advisor: Vladimír Mencl
6. V. Mencl, P. Hnětynka: *Managing type information in an evolving environment*, Week of Doctoral Students WDS 2000, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic
7. Object Management Group: *CORBA 2.4.2 specification*, formal/01-02-33, `ftp://ftp.omg.org/pub/docs/formal/01-02-33.pdf`
8. Object Management Group: *IDL_Java Language Mapping*, formal/99-07-53, `ftp://ftp.omg.org/pub/docs/formal/99-07-53.pdf`
9. Object Management Group: *CORBA Component Model Specification*, orbos/99-07-01, `ftp://ftp.omg.org/pub/docs/orbos/99-07-01.pdf`
10. CVS: The Concurrent Versions System, `http://www.loria.fr/~molli/cvs-index.html`
11. D. Bálek, F. Plášil: *Software Connectors and their Role in Component Deployment*, accepted to DAIS2001, Krakow, Poland, September 17 - 19, 2001
12. F. E. Redmond III: *DCOM: Microsoft Distributed Component Object Model,* IDG Books Worldwide, Inc.,Foster City, USA, 1997

13. P. Brada: *Component Change and Version Identification in SOFA*, Proceedings of SOFSEM'99, Nov 27 - Dec 4, 1999, Milovy, Czech Republic, Springer LNCS 1725
14. R. Conradi, B. Westfechtel: Configuring Versioned Software Products. Ian Sommerville (Ed.): Software Configuration Management. ICSE'96 SCM-6 Workshop, Berlin, Germany, March 1996, LNCS, Springer-Verlag 1996
15. Larsson, M., Crnkovic, I: *New Challenges for Configuration Management*, Proceedings of SCM-9, Toulouse, France, September 1999. LNCS 1675 Springer-Verlag
16. Object Management Group: *ORB Interface Type Versioning Management* Request for Proposals. Document 96-01-06, OMG 1996
17. Feiler, Peter H.: *Configuration Management Models in Commercial Environments*, CMU/SEI-91-TR-7. Carnegie Mellon University, Pittsburgh, PA, 1991