Component Evolution and Versioning State of the Art

Alexander Stuckenholz
FernUniversität in Hagen
Lehrgebiet für Datenverarbeitungstechnik
Universitätsstrasse 27, 58084 Hagen, Germany
Alexander.Stuckenholz@FernUni-Hagen.de

October 26, 2004

Abstract

Emerging component-based software development architectures promise better re-use of software components, greater flexibility, scalability and higher quality of services. But like any other piece of software too, software components are hardly perfect, when being created. Problems and bugs have to be fixed and new features need to be added.

This paper analyzes the problem of component evolution and the incompatibilities which result during component upgrades. We present the state of the art in component versioning and compare the different methods in component models, frameworks and programming languages. Special attention is put on the automation of processes and tool support in this area. The concluding section sketches a possible solution of these problems we are currently working on.

1 Introduction

During the last years component based software development changed from a pure scientific research field to a widely used technique [1]. A number of component models for different layers (desktop, server) have been established [2–5]. After consolidation of standards and specifications in this area, it is now time to look at the problems that arise by the practical use of software components in everyday projects.

Just like any other piece of software too, software components are hardly perfect, when they are created [6]. Problems and bugs have to be fixed and new features need to be added [7]. Software development generally may be defined as the process of creating and propagating changes [8]. By this process, new versions of software components come into existence.

To take advantage of these updates, one has to deploy the new components into the systems that use them. Doing this, the main question is: Does the system run with the new component, or is the system negatively affected by the upgrade? [9]

As we will show in the following, a series of different approaches exist in this area (see section 4 on page 4). They vary in the component model that is used, the kind of meta information about the component, which is used for compatibility checks, the kind of information about the running system as well as the lifecycle of the component in which the mechanisms are applied. The most frequently used technology in this area is versioning. Versioning mechanisms are typically used to distinguish evolving software artifacts over time. As we will show in section 4 on page 4, these mechanisms play an important role in component based software development too.

This paper presents an overview of current versioning mechanisms and substitutability checks in the area of component based software development. In a comparison, these systems will be evaluated against their usability in the case of component upgrades. Here we come to the conclusions that the available solutions are not sufficient to reduce version conflicts in component systems. To overcome this weakness, we are proposing an improved upgrade mechanism that can applied to all major component models. This paper is organized as follows: Section 2 on the following page will focus on the ultimate problem of evolving components, its reasons and related works.

Section 3 on the next page will render more precisely the authors comprehension of the term software component. We illustrate the linkage mechanisms between components, which influence their substitutability, and the problems that arise during component upgrades. Further on an overview about versioning mechanisms, generally used terminology in this area and the semantics of version numbers is given.

Section 4 on page 4 introduces a component based software system, which is used as a running example in the next sections to clarify the different versioning mechanisms. We compare a series of component models, programming languages and other systems which promise solutions to the upgrade problem by using particular versioning mechanisms. The author focuses on four main questions:

1. Does the system have the ability to distinguish different versions of the same component at all?
2. How are syntactical and semantic changes detected and how are these changes reflected in the version information?
3. How is the compatibility and incompatibility between different component versions detected and how is that reflected in the version information?

4. Will the provided mechanisms warn the component user of incompatible upgrades and does the system provide solutions for such situations?

In section 5 on page 8 the results will be categorized and evaluated with special attention to the questions and the main problem. Special importance is attracted to the automation level of the used practises to minimize the burdens for component developers and system administrators.

In section 6 on page 9 we will propose an advanced component upgrade system called *intelligent component swapping* that seizes and enhances a number of mechanisms in this area to minimize the negative effects during component upgrades and to initiate counteractive measures as automatic as possible.

2 The Problem

Software components undergo dynamic evolution during which a client component experiences the effects of modifications made to a service component even though these occurred after the client was built [10].

Dynamic evolution means that all characteristics of a component that are observable from external clients may change over time. These characteristics are part of different contract levels. Beugnard et. al have identified four classes of contracts in the software component world (see [11]): basic or syntactic (1), behavioral (2), synchronization (3) and quantity (4). The higher the level of contracts in which changes have been performed, the lower is the chance to detect them. So far there are a couple of approaches to detect syntactical and only a few approaches to detect behavioral changes (see section 4 on page 4), but we are not aware of any approaches that considers the level of synchronization or quantity.

Whichever level of contract is affected by component changes, such changes are necessary in many situations. In many cases bugfixes need to be enforced, which entail changes in the component's implementation. In some cases the requirements, like the required precision of results or the used types, may change over time, which causes changes at the interface level of the component next to its implementation. The simple addition of functionality is in most cases unproblematic.

In the case of a component upgrade, problems may arise in the interaction of the changed component and other components in the system. In these situations of incompatibilities it may come to malbehavior of system parts or the crash of the whole system.

This problem did not carry so much authority, because during the last years the majority of components were created by those developers that also had the responsibility for the whole system. Hence incompatibilities could be removed on the spot. But the problem will emerge in the future, because a growing number of components will be created by third persons (Off-The-Shelf Components), who do not have control over the client systems and do not have any information about the way their components are used.

There are currently no automatic tools or methods for versioning components in such a way that incompatibilities could be predicted and corrected before a component is deployed to a system. As long as this situation lasts, the advantages of component based software development such as flexibility, scalability and higher quality of services are abrogated. The problem of versioning and check for substitutability is therefore one of the most burdensome in this area.

2.1 Related Work

System Evolution is the stepwise development of systems or models due to environment changes [12]. These environment changes are, e.g., the need for new functionality, changes in use cases, new conditions (laws), implementation of new technologies or major changes in the processed data. At the Fraunhofer Institute for Software- and System Technology (ISST) the project *Continuous* is focusing on evolving systems [12–14], which use component based software development (CBS) as a base technology.

In general, Configuration Management (CM) is the discipline for organizing and controlling evolving systems [15]. According to this definition, controlling component evolution is a subdiscipline of CM. However, the term Software Configuration Management (SCM) includes the creation phase of software. Procedures like construction and team work are in the center of interest [16]. Component Evolution also includes managing the deployment of components and system management.

Class Evolution is defined as the process of evolving class hierarchies in object oriented programming. Since almost all current component models are based on object oriented development, class evolution plays an important role in this area. [17] distinguishes the categories Class Tailoring, Class Surgery, Class Versioning and Class Reorganization as solutions for evolving classes. For component evolution especially versioning plays an important role.

3 Components, Models and Versioning

3.1 Definitions

A widely accepted definition of the term software component is from Szyperski [18]. He defines software components as a coarse grained blackbox software element with contractually specified syntax and semantics on both the provided and required side of the interface. He claims that a software component can be deployed independently and is subject to composition by third parties.

The rules for the creation, composition and communication of individual components are defined by component models and put into operation by component frameworks.

A component encapsulates specific knowledge which is accessible only by its provided interfaces (i_p). Furthermore a component needs access to some general framework services and to other components or services to work. These necessities are expressed by the components required interfaces (i_r). The specification (s) of the component, especially the semantics of the component's interfaces, their constraints, data formats and protocols as well as detailed information about timeouts or quality of services may also be specified by OCL, in terms of predicate calculus or graphical description techniques like state charts or Petri nets. The

higher the degree of formalization of such descriptions is, the better is the utilization by automatic triggers, checks or verification mechanisms [12].

In current component models there is no obligation to enrich components with that kind of information. As a matter of fact, this has different reasons. Component based software development is an inherently complex technology, which would become even more difficult, if component developers would be forced to embrace formal specifications. These are tedious and difficult to write [9]. Additionally formal specifications are really needed in least projects. Currently most components do simple things like displaying controls on user interfaces. Using extensive formal descriptions for those components would break a fly on a wheel.

In general, metadata that specifies different aspects of software components is needed in case of dynamic linkage, where the only information about component usage is the component itself [10].

The above mentioned distinction between required interfaces, provided interfaces and the specification has advantages with regard to versioning of changeable parts and can also be found in [19] and [20].

3.2 Component Usage and Changes

Components are put together in order to build more complex components or "composable" software systems. The degree of substitutability of a component in a system does not only depend of the facilities of the components themselves, but also on how components are glued together and what kinds of composition mechanisms have been used [19].

The stronger the dependency between a component, other components and the system that makes use of it is, the harder it is to do upgrades with new component versions. This influences the requirements for a component versioning and upgrading system.

Some of the current component models, like JavaBeans [2], are based on object oriented programming languages. Although the idea of component based software development does not deal with object oriented concepts, they often go hand in hand in current component models.

Hence the strongest possible dependency between components in a system is inheritance. Changes of classes in components from which other components or classes derive, may cause the Fragile-Base-Class problem as described in [21] and [18]. Although this kind of usage contradicts the proposed paradigm that the knowledge of components should only be accessible via specified interfaces, this is often done by reasons of flexibility. Especially components for the GUI[1] layer are often customized by derivation [22].

The most common way of component usage is to invoke direct (procedural) or indirect (object) interfaces based on strong typing. Strong typing comes with a couple of problems in context of component evolution. Minor modifications of such interfaces, even switching the order of method parameters, may cause an invalidation of the component to a client. Parts of the system that make use of the changed component and expect older versions of an interface crash when they invoke the new methods.

[1] Graphical User Interface

Westphal [23] described this problem and suggested the replacement of strong-typing by a so called strong-tagging mechanism and a single entry point for components. Strong tagging detaches the parameters of methods from their position in method calls and bind them to their names. Thereby the parameter order is irrelevant at method calls. Westphal uses XML to pass parameters to methods, thereby he also assures independence from platform dependent type representations and enables implicit type casting.

Next to this technique to avoid the invalidation of component interfaces at all there are mechanisms to detect problems of incompatible interfaces before they arise. Invalidation checks of direct component interfaces are not a difficult issue. Several techniques, like fingerprinting, are currently used in component models and programming languages (see section 4 on the following page). However, indirect interfaces like object references that are passed over component boundaries, should also be integrated into this analysis. Since the dependencies between object references build a directed cyclic graph such calculation is getting complex.

3.3 Component Upgrades

When upgrading a component, a user has different possibilities at hand depending on the component model. The most common way to do an upgrade is to remove the old component and to replace it by another, usually newer, version (e.g. JavaBeans [2] or simple DLLs). This procedure is designated especially to systems that do not use a centralized registration and identification service like COM [24]. Applications which make use of such component models are bound to specific components by component IDs or other naming mechanisms. Since a new version of a component is also connected to a new component ID, simple replacement won't work for upgrade purpose.

Another possibility to upgrade a component is to deploy the component to the system and to pursue the new component in parallel to the old one. This has the advantage that different applications may use different versions of one component. In certain aspects, this reduces problems of incompatibility but circumvents simple bugfixes since in this situation applications need to be reconfigured or rebuilt for other component versions.

An inconvenient way of performing component upgrades is to use multiple versions simultaneously in one application. Rakic and Medvidovic [7] use a so-called Multiple Version Connector to test new component versions in parallel with older versions in a single application (see section 4.9 on page 7).

3.4 Versioning

The terms version, version model, revision and change originate from the area of Software Configuration Management (SCM). They can be directly transferred to the area of component evolution. A version of a component is a specific instance on the time axis, which came into existence due to a revision or change. The way how a version is identified by a version identifier and which characteristics are included into computation is defined in a specific version model [25]. This may also contain a metric which allows conclusions about the kind of changes in reference to the version identifier. In general, version identifiers consist of version

numbers, which are n-tuples of natural numbers often denoted $X.Y.Z$. The most commonly used versioning mechanism is the so-called Major-Minor-Build scheme, where major changes induce an increment of the major-version number (e.g. X), whereas minor bugfixes or enhancements lead to an increment of minor version numbers.

In its first occurrence to versioning binary modules to distinguish them during deployment and runtime (see section 4.3 on the following page), this scheme also tried to introduce special semantics to version numbers. Their original sense was to discover incompatibility between different versions at the first glance. Different major versions should be incompatible whereas different minor and build versions reflect compatibility. Over the years this meaning was more and more deluded. In the meantime version numbers are merely used for marketing purpose only.

The manual assignment of version numbers to a specific component is problematic. In general, version numbers do not have the potential to state which characteristics have changed and to which extent. The only information that can be derived from such numbers is that one component is newer than another. The only approach, known to the author, of giving a well-defined meaning (semantics) to such version numbers is the concept of Premysl Brada (see section 4.9 on page 7).

One advantage of version numbers is their human readability instead of hashes, fingerprints or manifold specifications. Indeed version numbers have no great expressiveness but they can be caught by humans at first glance.

4 Component Versioning

4.1 Example

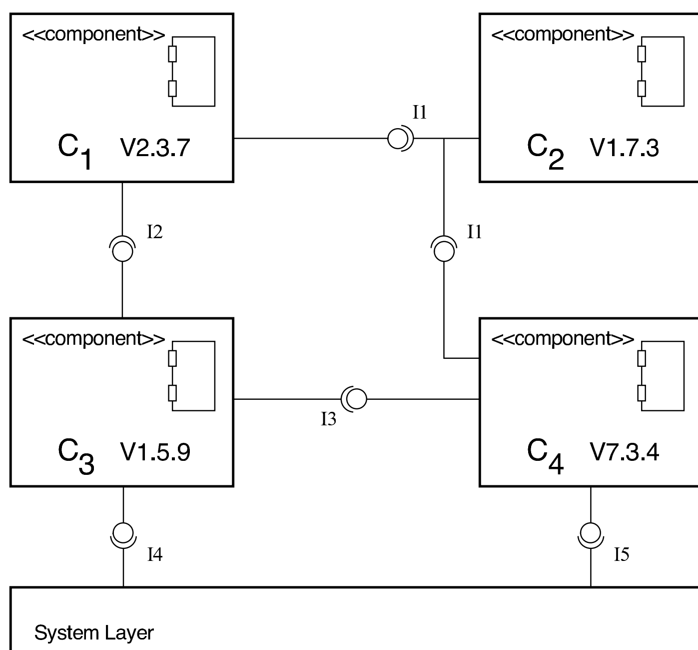

Figure 4.1: Common situation in component based systems

Figure 4.1 illustrates a typical situation in a component based system[2]. A couple of components by different producers use and provide services from each other by means of their interfaces.

The whole system may be an Enterprise Resource Planning System (ERP) and C_1 encapsulates the management of the warehouse. A couple of methods in I_2 deal with finding the price of a specific product. The manufacturer of C_1 discovers that float numbers are perhaps not the best data type for representing dollar values. After all, a price is never $2.33333333333 but uses only two digits after the comma. Additionally that representation causes some rounding errors due to the way that trade prices are calculated by the system. So the manufacturer of C_1 decides to change the data type from a float to an integer, where it is understood that the integer value is a price in cents[3].

Implementing that major revision, version 3.0.0 of C_1 comes to existence, which could be smoothly deployed to the system. But the above mentioned change has a hazardous impact on the rest of the system, as C_3 directly uses services of C_1 and C_4 is indirectly connected to C_1 over C_3.

Concepts and tools in component models and programming languages for version control of evolving components should enable developers to avoid such situations of surprising incompatibilities. The following sections will focus on current mechanisms in this area.

4.2 Fingerprinting

Mechanisms of version control in component and module based software development were introduced first with the intention to reduce the number of recompilations of separate modules.

Bernard Crelier [27] described two different inceptions, the layered model and the object model for the portable Oberon-2 compiler OP2. Both models compute a fingerprint over certain characteristics of module interfaces. The coarse grained layered model keeps a list of changes for a module interface for invalidation checks. The history information consists of the fingerprints of the whole interface. The finer grained object model computes fingerprints of all exported objects of interfaces (parameters, types). This also includes recursive types. The programming language Component Pascal also uses such a fingerprinting mechanism to reduce recompilation [28].

Within this fingerprinting mechanism developers can automatically detect syntactical changes of components, modules and libraries. Because the mechanism was intended only to detect the presence of a change, Crelier did not design a special version policy. Hence it does not reflect the kind or extend of changes and does not give answers in the case of incompatibilities.

In the example introduced in section 4.1 the major change can be detected by the sketched fingerprinting mechanisms, because it also includes an interface change which also modifies the computed fingerprint. However, the developer only knows that *something* has changed. He neither has an idea what was changed nor does he know to which extend it has been modified.

[2] A UML 2 component diagram is used to illustrate the structure of the system.
[3] The example is taken from [26].

4.3 Library Interface Versioning in Unix Systems

Sun Microsystems SunOS introduced dynamic shared libraries to UNIX in the late 1980s [29]. Right from the start Sun established versioning mechanisms to allow the evolution of dynamic shared libraries.

For library wide versioning, the filenames of the libraries were enhanced by version numbers (major.minor) initially. Thus different major version numbers represent incompatibility between libraries as different minor version numbers represent compatibility. This semantics was kept in all further versioning enhancements. The Link Editor (ld) simply recorded the filename of the library within the application binary, the application depended upon [30]. At runtime the Library Linker made sure to dynamically load the library with the same major release and the highest available minor version [31]. This course grained mechanism had the disadvantage that an application built with a given minor release of a library might, but cannot be certain to run on an earlier minor release level of the library [30].

To eliminate that drawback, Sun developed a fine grained versioning mechanism based on the ELF binary format which contains libraries as well as executables and introduced it to UNIX System V [29]. This new mechanism decorates the exported symbols (e.g. methods) of a shared library with version numbers and stores them in the header of the ELF file. By the use of a mapfile or special control-code one can influence the visibility and the versioning of the respective symbols. At runtime the Dynamic Linker seeks for the library by means of the SONAME[4] and analyzes the ELF header for the required symbol or a compatible version. On Solaris systems an ELF library may only contain compatible symbols[5] (cf. [32]) whereas Linux Systems with the GNU linker and glibc from 2.1 may also host incompatible symbol versions in a single library [33]. This proceeding along with the ELF binary format became the quasi standard on UNIX and UNIX like systems (Linux, BSD) [29].

In our example (cf. section 4.1 on the preceding page) the revision of C_1 would create a new library $c1.so.3.0$ which is incompatible to older versions. The versioning information of the exported methods of I_2 are stored in the ELF header. All applications that require the old version of I_2 are not affected by the deployment of the new library.

An evaluation of the UNIX library version model with respect to the four main questions from section 1 on page 1 shoes the following results. On the basis of fine grained symbol versioning the UNIX library versioning model allows the parallel existence of several library and symbol versions. Since the versioning information is attached manually to the libraries the developer may consider syntactical as well as semantical changes. The detection of compatibility and incompatibility between different library versions is limited reliable. Syntactic changes are not distinguishable from semantic changes by means of the version numbers. Due to the fact that the version numbers are manually attached to symbols and libraries during development by mapfiles or pseudo-code, there is a huge risk of creeping errors that are difficult to maintain in huge systems (cf. [31], p. 9).

[4]ELF object files may contain an SONAME – a specific means of naming the library (superceding the library's filename) stored within the library's object file [30].

[5]Symbols that have the same major release.

4.4 CORBA

CORBA components are specified by their interfaces in the Interface Definition Language [6] (IDL). The IDL description is mapped to a specific programming language following precise rules [4]. If a CORBA component changes, its specification in IDL changes too. Especially changes that cause an invalidation of the client cause worry lines of CORBA-developers foreheads. As a matter of fact these are all changes to IDL except the addition of new methods or interfaces to a component.

The CORBA specification [4] does not contain any approaches to handle component evolution at all. On the basis of the current CORBA specification, it is neither possible to enrich components with version information, nor to run more than one version of a single component in a system.

One workaround for this problem has been demonstrated by Hamilton and Radia in the Spring Experimental Distributed System [6]. They created new interface versions by derivation and added version numbers to the interface names. The advantage of using such a strategy is to have multiple versions of an interface in a system. However these versions need to be separated by explicit naming conventions. Hence clients do not automatically profit from latest versions.

Using that mechanism with respect to the major change from section 4.1 on the preceding page means that the developer has to create a completely new component C_2new by deriving it from C_2. All other components that want to use services of C_2new need to be rebuilt. This disagrees with the intrinsic sense of component upgrades.

Unfortunately, CORBA does not really have an answer to the versioning problem.

4.5 Web-Services

Web-Services are not units of deployment, but they also encapsulate specific knowledge accessible via their interfaces. By the use of web-services and some glue-code it is possible to compose new applications, which is very close to the idea of components.

Unfortunately, standards like WSDL [34] (to specify the interfaces) or SOAP [35] (the transport protocol) do not address the evolution of web-services. Nevertheless, web-service developers recognized that web-services may also change over time and therefore adopted versioning workarounds.

A common workaround is shown in [36] and [26]. A web-service belongs to a unique namespace which is specified in its WSDL description. The namespace-string can be used to append a date or version stamp. This follows the general guidelines given by the W3C for XML namespace definitions. Doing so, it is possible to run different versions of one web-service to support applications that require older versions of the service. Due to the strong-typing of web-services, it is, for example, not possible to change a parameter-type of a service without invalidating existing

[6]ISO/IEC 14750:1999

clients. Automatic techniques to announce those kinds of changes are missing.

The usage of XML namespaces to distinguish different web-service versions is just a workaround. Neither Web-Services nor CORBA components (see section 4.4 on the page before) introduce a versioning policy, automated tools to detect syntactical or semantical changes or mechanisms to discover incompatibilities.

4.6 From the DLL-Hell to Windows XP

The concept of Dynamic Link Libraries (DLL) was introduced by Microsoft into their operating systems to bundle functionality, which could be used by several applications simultaneously. The main purpose was to load such libraries dynamically at runtime, thus building the basis for a couple of component concepts (see 4.7), in which DLLs are acting as component containers.

The underlying DLL system of MS-Windows did not specify any mechanisms for introspection or versioning. In this context, single applications could replace libraries during their installation by other versions. This often caused crashes of applications that had been installed previously and relied on older versions of that library. This phenomenon is called DLL-Hell and represents the most quoted versioning problem in the windows world (see [37, 38]).

As part of the development of Windows 98 SE and Windows ME, Microsoft introduced the possibility to control the dynamic linkage of their libraries. Special meta-files could redirect the loading process to local, isolated libraries, which could be deployed to one or more applications only (Isolated Applications, see [39]).

Since Windows XP, libraries could be endowed by manifest-files in a special XML-format, which contains the name, the type, the processor architecture and a version number of the library. The version number is a quadruple of major, minor, build and revision number. Using such a manifest, application developers are able to control the process of dynamic linkage, so that only specified versions of a library can be loaded. The application is bound to a major and minor version of a lib. To enable bug-fixes (Quick Fix Engineering, QFE), the build number and the revision number may vary [40].

By the introduction of the above mentioned version mechanisms Microsoft introduced library versioning with external meta-information to Windows. Techniques like this have been existing in the Unix world for years (see section 4.3 on the page before).

With respect to the example from section 4.1 on page 4 the developer could create a new dynamic link library which would contain component C_1. During creation he attaches a manifest to the library with manually created version information. If he found that his library is compatible to the old version, he could copy the file to the applications directory and redirect the dynamic linkage to the new library by creating a meta file. By alternatively installing the library locally to the application's directory (isolated library) the developer ensures that his update does not negatively affect other applications. Obviously this either needs to be done with every application which wants to benefit from this update or he installs the new library version as a shared library to the system folder where it is globally usable by all applications on the system. This installation method could have the same impact as before.

The advantages of the above sketched versioning mechanisms are ruined if the assumptions of the library developer with respect to the extent of changes, their effect to the system and the possibly resulting incompatibilities between different library versions are wrong. This can have hazardous impact to the whole system and should be replaced by automated versioning tools (cf. [20], p. 4-5).

4.7 From COM to .NET - The Microsoft Way

After Microsoft made some bad experiences with the DLL-Hell (see 4.6) the company decided to forbid changes in existing components in their component models (COM, and related ones like ActiveX and DCOM) completely.

Once an interface of a COM component is published, it gets a unique interface identifier (IID) by which the interfaces can be identified, also beyond the boundaries of one computer (DCOM). Rather than changing an interface, the developer actually creates a new interface and the new interface gets a new IID (see [41]). This practice is useful to ensure that component clients are never disabled by installing a newer version of a component. But this also prevents clients to know the features of newer component versions without rebuilding them.

With the development of the .Net framework, Microsoft introduced a couple of new technologies, sometimes well known from other platforms, to the windows world. Beside a runtime environment that uses bytecode like Java and is endowed with meta data they also designed a new component model with some interesting versioning features.

.Net resources like classes, executables and therefore also components are shipped in so called assemblies together with a manifest. These assemblies are the objects of versioning, where the version number is a quadruple of 16-bit integers, which is specified manually by the developer. The manifest may contain, in addition to the version number, some metadata like the name of the assembly, a description and some info about the manufacturer, which can be received at runtime by reflection.

Next to private assemblies, which can be used by local applications only, one can deploy multiple versions of an assembly to the Global Assembly Cache (GAC) to share it with all applications on the computer system. Shared assemblies must be extended with a Strong Name, which is some kind of UID based on a public-key signature to ensure authenticity and integrity [42].

The manifest of an assembly records all dependencies to external assemblies specified by their name, their version number and the strong names, if existing. A reference to an entity which was a not yet loaded causes the Fusion Utility to search first the GAC and after that the local application directory for the appropriate version of the required assembly which is then passed to the Library Loader [10]. If the Fusion Utility is not able to find the assembly version the application was built with[7] an exception is thrown.

[7] Microsoft calls this a compatible version

By means of the .Net Configuration Tool this default processing can be replaced by a custom version policy. Thereby it is possible to redirect the linkage of external assemblies to other versions or version ranges than the originally demanded. Additionally it is possible to define global custom policies for assemblies in the GAC to control their usage.

On top of the .Net component model and the .Net framework Eisenbach, Jurisic and Sadler modelled an extended component cache to ensure its consistency in relation to required and provided services [10]. They adopted the version model of the .Net framework and made use of the assembly metadata to check for inconsistencies in their assembly cache in the case of a component upgrade.

The versioning system of the .Net framework is currently the most progressive mechanism in the area of component versioning. The .Net framework prevents the effects of the DLL-Hell (see 4.6 on the preceding page) because it permits the simultaneous existence and usage of multiple versions of one component. Nevertheless the techniques can only be as good as the provided metadata, especially the version number, of the components. As developers need to specify them manually, changes which are not reflected correctly by the version numbers will cause unpredictable effects.

4.8 The Java Way

Both JavaBeans [2, 22] as well as Enterprise Java Beans (EJB) [3, 43, 44] are specifications, based on the object oriented programming language Java. Java maps classes to single files and packages to directories in a file system. In difference to the package term in UML, a Java package is a physical organization of classes, resources and a manifest. A Java package is a unit of deployment, which can be seen as a component.

The standard class loader resolves references along the CLASSPATH and returns the first occurrence of the component. This prevents the simulcast existence of two or more different versions of a component, except custom classloaders [10] are used (see below). As Java binaries contain meta information, reflection mechanisms can be used to get information on exported interfaces and types of components at runtime.

Class serialization in Java is used for persistence and for Remote Method Invocation (RMI). In this context the Java specification defines compatible and incompatible changes for type evolution (see [45]). Compatible changes are that kind of class revisions that may be used instead of the older class version, to unserialize data-streams.

In Java, packages are the objects of versioning. The package manifest may optionally be used to enrich a component with course grained version information, which serves for identification [46]. This includes the package title, package version, specification title and the specification version. Separating specification and implementation allows the two to evolve independently. All those attributes are of the type *string*, which gives evidence about the cloudiness of their entropy.

Additionally component developers are able to add a special class to their components deriving from the abstract class *Package*, which contains a set of methods next to the aforementioned attributes. The method *isCompatibleWith* receives a specification version to check the compatibility of the current component to the given version.

Many of the J2EE application servers that are currently in use have several classloaders, organized into hierarchical structures. A typical implementation of a classloader in a hierarchy will ask its parent classloader to find and load the class first. In a chain of several classloaders, the effect is that a classloading request propagates all the way to the top of the hierarchy, and then filters down the chain until found [47]. By using such classloaders it is possible to have multiple versions of components in a system and to implement user defined versioning policies.

Unfortunately, there are no rules or regulations to realize such systems, so that components that have been created for one system, might not run in another. Not only does this contradict the Java paradigm *write once, run anywhere*, but in this context the term *jar-hell* arose (see [48]). Component developers are not forced to enrich components with version information and there is no predefined version policy. Furthermore component users are responsible to evaluate version information by themselves, e.g. to create workarounds for well known bugs [46].

To close this gap, currently different case tools (i.e. Krysalis[8]), come into existence, which support component developers in adding version information to Java packages and evaluating their runtime environment.

The whole versioning concept of Java components is based on some fungous specifications which do not allow conclusions to the kind of changes and their extent. Further on it was not designated that multiple package versions exist in one system. By the use of hand-crafted classloaders this situation can be circumvented, but this is not part of the specification. By manually adding the method *isCompatibleWith* to Java packages, developers can at least provide a vague notion or pruning if one component is compatible to another. As Java does not provide automated tools to do these kinds of calculations, wrong assumptions of the developers may cause system crashes.

4.9 Others

Next to the well known component standards like CORBA or .Net, some smaller component systems, sometimes primary developed for academic use, have integrated mechanisms for component versioning. Furthermore there are systems that intend to enhance existing systems with versioning mechanisms and substitutability checks.

One really interesting approach for detecting incompatibilities in the case of component upgrades is presented by McCamant and Ernst (cf. [9] and [49]). The approach is focused on semantic changes in components and their impact on their operational abstraction[9]. The operational abstraction of the new component which has to be deployed to the system is created automatically by the component developer by means of test tools and is provided together with the blackbox component to the component user. On the other side the operational abstraction of the old component is

[8] http://krysalis.org/version/index.html

[9] McCamant and Ernst have a broad understanding of components which also includes software modules, classes and even single procedures.

derived online during its usage and in the context of the real application. By the comparison of these two abstractions and the thereby extracted pre- and post conditions possible incompatibilities can be detected before the upgrade is initiated.

In the example from section 4.1 on page 4 the developer could create version 2.3.8 of component C_1 which provides the same interface as 2.3.7 but has major changes in its implementation. For example, the developers changed the allowed range of prices in the ERP system, which is not visible through the syntax of the interface. The operational abstraction of the new component contains this information. If the old component was called with parameters outside the new price range, this would be recorded to its usage profile and the comparison of the abstractions would detect the incompatibility.

The advantages of that approach are the high level of automation in deriving the operational abstraction, its adaptability to black-box components and its consideration of the usage profile of the component in the users application. Unfortunately, McCarmant and Ernst do not mention the adaptability of their approach in the case of syntactical changes or the replacement of more than one component by a set of new components which are again compatible among each other.

The most sophisticated methods in the area of component versioning, substitutability analysis and dynamic component upgrades have been elaborated at Charles University in Prague at the Departement of Distributed Systems[10]. By means of the SOFA distributed system, the SOFA component model and the DCUP system to initiate dynamic component updates, the problem of component evolution and component upgrades was studied and solutions have been proposed in a series of publications (see [50–53]). In this area, Premysl Brada designed a scheme for component versioning suitable for automated processing and supporting component distribution and retrieval [20]. His concept is built upon the ELF-meta model, which enables abstract descriptions of components and its characteristics coming from different component models. Once the abstract component description is available in the ELF format, automatic tests are able to analyze the differences between component versions to identify compatibility or incompatibility between them. By means of these specification comparisons which are based on subtyping rules, human readable version numbers that allow conclusions which parts of a component have changed (provided parts, required parts or ties) are created and attached to the components. This is the only approach known to the author, which joins such semantics to component version numbers. The mechanisms are integrated into the SOFA component model, but can be applied to all component models for which ELF representations can be generated.

In the example from section 4.1 on page 4 the developer first needs a parser which transforms the components (the IDL description or source code) to an abstract ELF description. If he creates new versions of the components, the automated comparison tools will find out incompatibilities and create according version numbers. As an example, version 3.2.1 of a component means that the provided parts have changed three times, the required parts twice and ties have been left unchanged (cf. [20], p. 72). By simple comparison of the adequate version number parts, compatibility can be detected before replacing a component by a new version.

[10] http://nenya.ms.mff.cuni.cz/

The approach of Brada solves a series of problems in this area. He established a completely new versioning policy which minimizes new burdens to component developers because it uses automatic tests and parsers to transform existing informations and external specifications like IDL into the required ELF model. By means of his approach the situation in component upgrades could be improved considerably. But the mechanism depends on the availability of an abstract description of components in the ELF format. This is easy to create with component models that already use an IDL as an external description. But it is a non trivial task for models like JavaBeans or .Net components which are specifications upon programming languages and do not have external descriptions (see [20], p. 139-143). We also believe that it is not sufficient to detect incompatibilities before a component update is executed. This is only the first step into the right direction. Section 6 on the next page will sketch an approach for solving remaining problems.

Another component versioning related technique is presented by M. Rakic and N. Medvidovic (cf. [7]) which is based on explicit software connectors. Using such connectors it became possible to create a Multi Versioning Connector (MVC) which encapsulates two or more versions of a component in order to monitor the execution of multiple component versions and perform comparisons of their performance (i.e., execution speed), reliability (i.e., number of failures), and correctness (i.e., ability to produce expected results). Thereby new components can be tested without affecting the rest of the system as for a transitional period the results of the new component are only logged. By the simultaneous use of more than one version of a component, the system increases its reliability [54]. Therefore voting schemes are used to decide which version(s) are correct [55]. But this can only work, if the two different component versions have either the same interface or an intermediator like a component wrapper that translates procedure names and converts data. Hence this technique cannot be used at major changes between component versions.

In the example from section 4.1 on page 4 the developer could create an MVC to monitor the correctness of an upgrade of component C_1 from version 2.3.7 to 2.3.8 by comparing the logged results of both components after a specified period. Unfortunately, there are no tools provided for automatically deriving the information if the new component works as predicted.

The sketched technique of Rakic and Medvidovic is not a versioning mechanism itself but can be seen as an enhancement of other versioning mechanisms. It does not establish a specific version policy and rules to derive and attach version numbers or how to detect incompatibilities. Thus this mechanisms will not being considered in the concluding comparison.

5 Summary

The concepts and technologies around component based software development did not appear over night. In addition to the con-

cepts and processes especially the concrete models for which the developers create their components needed to evolve. The fact that components often pass through an evolution and change over time was recognized only lately.

The previous sections compared a series of component models, frameworks and programming languages that promise solutions to the upgrade problem by using their versioning mechanisms. Thereby we dwelled on the four main questions that have been sketched at the beginning (see section 1 on page 1). The results will are summarized in table 5.1 on the following page.

Most of the current component models, programming languages and frameworks integrate rudimentary versioning support to distinguish different versions of components and libraries. But the expressiveness of version numbers in the Major-Minor-Build scheme, which is used in the majority of cases, is limited. Especially if component developers need to assign version numbers to their components manually and do not have proper instructions that define which changes in what level of contract conduct a new version, those version numbers at most rest for marketing use and do not ensure compatibility between different components.

Among the current systems which are commercially in use both .Net assembly versioning and the Unix library versioning present the most practical approaches, because multiple versions of components or libraries can exist in parallel in one system and can be used in different applications without side effects.

Automatic detection of component changes and substitutability checks based on the versioning history of single components can be found in the scientific approach of McCamant and Ernst and in the SOFA experimental system only. Furthermore the SOFA system and its versioning policy is the only existing approach with dedicated interconnection between changes of different component characteristics and version numbers.

None of the compared systems is able to reduce side effects in a component system in the case of an incompatible component upgrade. The minority of the systems are able to detect such incompatibilities at all. In general, developers and system administrators need tools to automate the detection of incompatibilities and to minimize their effects.

6 Future Work

As discussed in the previous sections, component based software development especially lacks of mechanisms which are able to transfer systems into a version conflict free state in the case of incompatible upgrades of one or more new component versions.

Some systems, especially the SOFA system, its versioning policy and substitutability checks, indeed detect such incompatibilities. But in the author's opinion evolutionary components need to be analyzed with respect to the system structure in which they will be used and in the interaction of all other possible component versions. This puts the original usage of components, their composition, more into the right perspective.

As an example consider a system with a couple of components. In an upgrade two of them need to be replaced by new versions. Each of the new versions are incompatible to the old system. But by replacing them both, a new conflict free system emerges. None of the systems evaluated in previous sections could find such a solution.

This is the reason why we are currently working on a new approach which combines different ideas to find answers in the sketched situations. The approach tends to combine available component versions in one system, so that in case of initially incompatible upgrades of one or more components, a conflict free system arises again. This technique is called *intelligent component swapping*.

For the implementation of such a technique the author currently designs an abstract component description comparable to the ELF metamodel (see section 4.9 on page 7) which will be limited to syntactic characteristics of components (interfaces, exceptions) for different reasons. The main reason is the current lack of methods to specify the usage profile between component dependencies. The current compatibility and substitutability checks (see section 4.9 on page 7) are based on behavior specifications of one single component, at which it is possible to detect behavioral incompatibilities by comparing the specification changes in a version history by subtyping. Unfortunately, we do not know any possibility to perform these checks between completely different components, because it is not possible to discover how one blackbox component uses another.

The next step to an intelligent component swapping system (ics) is to gather techniques to create the so called version reachability graph of all available component versions in a system or a component repository. In this graph all components are recorded as nodes with directed edges to their component versions. Component versions implement one or more interface versions which they provide to other components. On the other hand components may also require specific interface versions from other components. Our comprehension is, that the interfaces of a component are the object of versioning. Special kinds of components (connectors) have the ability to bridge between the different interface versions of components, that could not interact without the existence of those bridges. All of these dependencies between components, interfaces and connectors are expressed in the version reachability graph.

Figure 6.1 on the next page shows an example of a version reachability graph. The component repository contains three components C_1, C_2, C_3 and the connector C_4. For each of the components at least one concrete component version exist (C_1 exists as version 1.1, C_2 as 2.1 and C_3 as 6.2). These components provide and require a number of interfaces. Required interfaces can only be satisfied by provided interface with equal version. In Figure 6.1 on the following page the component adapter C_4 bridges between interface version 2.7 of component C_1 and interface version 1.9 of component C_3. The version reachability graph contains all possible combinations between interfaces of component versions. If there was another component than C_2 that could fulfill the requirements of C_1 from the example, that dependency would be part of the graph also.

If an application only consists of components which are elements of the version reachability graph, the dependencies of this application can be seen as a subgraph of the reachability graph. In the case of a single- or multi-component upgrade our objective function needs to find a compatible subgraph of the reachability

	Oberon, Component Pascal	Unix Library Versioning	CORBA, Web-Services	Windows XP	.NET Components	Java Versioning	McCamant and Ernst	SOFA
versioning support	○	●		●	●	●		●
detection of changes	●						●	●
subsititutability checks							●	●
minimizing incompatibilities								

○ basic support, ● full support

Table 5.1: Comparison of component models, programming languages and frameworks

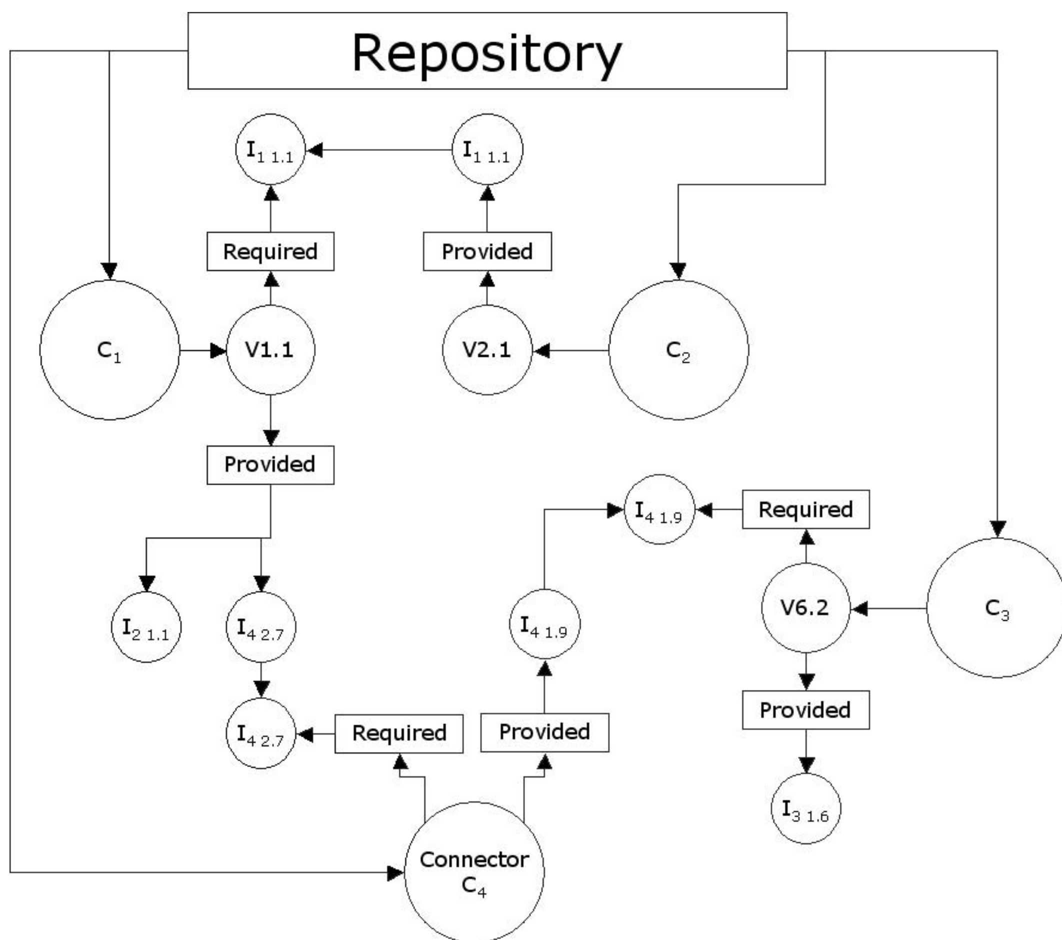

Figure 6.1: Version reachability graph

graph with the highest congruence to the original application in order to reduce version conflicts.

A specific component version from the system can be replaced by another version without further activities if the new version has the same dependencies as the old component version. If the new component requires other component versions than existent in the system, the objective function tries to replace those components by according component versions or add them to the system.

As the reachability graph is a directed cyclic graph (dcg) neither its creation above hundreds of component versions nor the search for conflict free solutions is a trivial problem. Even with only three different components with one single component version each, the version reachability graph from figure 6.1 on the page before becomes very complex. In a number of examinations the author furthermore realized that some components cannot be swapped in systems. These components either have direct hardware dependencies or are too costly, e.g. transformation of huge databases, to be swapped. Hence these components must be excluded from swapping. During the search for a conflict free subgraph we may find multiple solutions or no solutions at all. By the existence of a great number of suitable component versions from the repository, the problem of finding a conflict free subgraph becomes a NP-complete search problem.

The objective function to reduce version conflicts by component swapping is additionally equipped with a number of constraints. As component swapping probably can be enforged during system breaks only, one constraint may be to reduce the upgrade duration and for this purpose the number of required component swaps. Another constraint is to ensure that the system uses the latest possible component versions. This conforms to the definition of well versioned systems seen in [10].

The main aim of the approach is to provide fully automated tools to component developers and system administrators to perform component upgrades transparently. These tools should automatically find conflict free systems, collect missing components and perform component swaps.

The author hopes to reduce the situations of unforseen incompatibilities in component upgrades by means of the above sketched approach and thereby to leverage component based software development sustainable.

References

[1] O. Zwintzscher, *Komponentenbasierte & generative Softwareentwicklung - Generierung komponentenbasierter Software aus erweiterten UML - Modellen.* W3L GmbH, 2003, (in German).

[2] Sun Microsystems, “Javabeans api specification,” Tech. Rep., August 1997, last visited: 04/2004. [Online]. Available: http://java.sun.com/products/javabeans/

[3] B. Shannon, “Java 2 platform enterprise edition specification,” Sun Microsystems, Tech. Rep. v1.4, November 2003, last visited: 04/2004. [Online]. Available: http://java.sun.com/j2ee/index.jsp

[4] Object Management Group, Inc, “Common object request broker architecture: Core specification, Tech. Rep. Version 3.0.2 - Editorial update, December 2002, last visited: 03/2004. [Online]. Available: http://www.omg.org/technology/documents/formal/corba_iiop.htm

[5] ECMA, “Standard ecma-335 - common language infrastructure (cli),” Tech. Rep., Dezember 2002, last visited: 10/2004. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-335.htm

[6] G. Hamilton and S. Radia, “Using interface inheritance to address problems in system software evolution,” *ACM SIGPLAN Notices*, vol. 29, no. 8, pp. 119–128, 1994.

[7] M. Rakic and N. Medvidovic, “Increasing the confidence in off-the-shelf components: a software connector-based approach,” in *Proceedings of the 2001 symposium on Software reusability*, 2001, pp. 11–18.

[8] A. Zeller and J. Krinke, *Open-Source-Programmierwerkzeuge, Versionskontrolle - Konstruktion - Testen - Fehlersuche*, 2nd ed. dpunkt.verlag, 2003, in German.

[9] S. McCamant and M. D. Ernst, “Early identification of incompatibilities in multi-component upgrades,” in *Proceedings of the 10th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Helsinki, Finland, June 14–18, 2003, pp. 287–296.

[10] S. Eisenbach, V. Jurisic, and C. Sadler, “Managing the evolution of .NET programs,” in *6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS 2003)*, November 2003.

[11] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins, “Making components contract aware,” in *IEEE software*, june 1999, pp. 38–45.

[12] A. Borusan, M. Große-Rhode, H. Ehrig, R.-D. Kutsche, S. Mann, J. Padberg, A. Sünbül, and H. Weber, “Kontinuierliches engineering: Grundlegende terminologie und basiskonzepte,” Fraunhofer ISST, Tech. Rep., 2000, (in German).

[13] M. Große-Rhode, R.-D. Kutsche, and F. Bübl, “Concepts for the evolution of component based software systems,” Fraunhofer ISST, Tech. Rep., 2000.

[14] F. Bübl, “Towards the early outlining of a component-based system with concoil,” Technische Universität Berlin, Tech. Rep., 2000.

[15] A. Zeller, “Configuration management with version sets - a unified software versioning model and its applications,” Ph.D. dissertation, Technische Universitaet Braunschweig, April 1997.

[16] M. Bar and K. Fogel, *Open Source Development with CVS*, 3rd ed. Paraglyph Publishing, 2003.

[17] O. Nierstrasz and D. Tsichritzis, *Object-Oriented Software Composition*. Prentice Hall International, 1995.

[18] C. Szyperski, *Component Software: Beyond Object-Oriented Programming - Second Edition*. Addison-Wesley, 2002.

[19] S. Mann, A. Borusan, H. Ehrig, M. Große-Rohde, R. Mackenthun, A. Sünbül, and H. Weber, "Towards a component concept for continuous software engineering," Fraunhofer ISST, Tech. Rep., 2000.

[20] P. Brada, "Specification-based component substitutability and revision identification," Ph.D. dissertation, Charles University in Prague, August 2003.

[21] L. Mikhajlov and E. Sekerinski, "The fragile base class problem and its impact on component systems," in *Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP '97)*, W. Weck, J. Bosch, and C. Szyperski, Eds. Turku Centre for Computer Science, September 1997, pp. 59–67.

[22] R. Englander, *Developing Java Beans*. O'Reilly, 2001.

[23] R. Westphal, "Strong Tagging als Ausweg aus der Interface-Versionshölle," *Objekt Spektrum*, vol. 04/2000, 2000, in Germam.

[24] Microsoft Corporation, "The component object model specification," Tech. Rep., October 1995, last visited: 10/2004. [Online]. Available: http://www.microsoft.com/com/resources/comdocs.asp

[25] R. Conradi and B. Westfechtel, "Version models for software configuration management," *ACM Computing Surveys*, vol. 30, no. 2, pp. 232–282, 1998.

[26] K. Brown and M. Ellis, "Best practice for web service versioning - keep your web services current with wsdl and uddi," IBM, Tech. Rep., January 2004.

[27] R. Crelier, "Separate compilation and module extension," Ph.D. dissertation, Swiss Federal Institute of Technology, 1994.

[28] K. Hug, *Module, Klassen, Verträge : ein Lehrbuch zur komponentenbasierten Softwarekonstruktion mit Component Pascal*. Vieweg, 2001, (in German).

[29] J. R. Levine, *Linkers & Loaders*. Morgan Kaufmann, September 1999, vol. one.

[30] D. J. Brown and K. Runge, "Library interface versioning in solaris and linux," in *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, Georgia, USA, October 2000,, pp. 10–14.

[31] R. A. Gingell, M. Lee, X. T. Dang, and M. S. Weeks, "Shared libraries in sunos," *Proceedings of the USENIX 1987 Summer Conference*, pp. 131–145, 1987.

[32] U. Drepper, "How to write shared libraries," Red Hat, Inc., Research Triangle Park, NC, Tech. Rep., April 2004, last visited: 10/2004. [Online]. Available: http://people.redhat.com/drepper/dsohowto.pdf

[33] ——, "Using ELF in glibc 2.1," Cygnus Solutions, Sunnyvale, CA, Tech. Rep., March 1999, last visited: 10/2004. [Online]. Available: http://people.redhat.com/drepper/elftut1.ps

[34] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web services description language (wsdl) 1.1," W3C," 2001, March 2001.

[35] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen, "Soap version 1.2 w3c recommendation 24 june 2003," W3C, Tech. Rep., June 2003, last visited: 04/2004. [Online]. Available: http://www.w3.org/2000/xp/Group/

[36] R. Irani, "Versioning of web services - solving the problem of maintenance," InSync Information Systems, Inc, Tech. Rep., August 2001, last visited: 10/2004. [Online]. Available: http://www.webservicesarchitect.com/content/articles/irani04.asp

[37] P. Devanbu, "The ultimate reuse nightmare: Honey, i got the wrong dll," in *the 5th Symposium on Sofware Reuseability*, 178–180 1999. [Online]. Available: citeseer.nj.nec.com/devanbu99ultimate.html

[38] S. Pratschner, "Simplifying deployment and solving dll hell with the .NET framework," Microsoft Corporation, Tech. Rep., November 2001, last visited: 04/2004. [Online]. Available: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/dplywithnet.asp

[39] Microsoft Corporation, *Microsoft Platform SDK Documentation*, Microsoft Corporation, April 2004, last visited: 04/2004. [Online]. Available: http://msdn.micorsoft.com

[40] D. Beyer, *C# COM+ Programming*. M & T Books, 2001.

[41] D. Rogerson, *Inside COM - Microsofts Component Object Model*. Redmond, Washington: Microsoft Press, 1997.

[42] J. Löwy, *Programming .Net Components*. O'Reilly, 2003.

[43] R. Monson-Haefel, *Enterprise JavaBeans*. O'Reilly & Associates, 2001.

[44] E. Roman, S. Ambler, and T. Jewell, *Mastering Enterprise JavaBeans*, 2nd ed. New York: Wiley Computer Publishing, 2002.

[45] Sun Microsystems, "Java object serialization specification," Tech. Rep., 2003, last visited: 10/2004. [Online]. Available: http://java.sun.com/j2se/1.4.2/docs/guide/serialization/spec/serialTOC.html

[46] ——, "Java product versioning specification," Tech. Rep., November 1998, last visited: 04/2004. [Online]. Available: http://java.sun.com/j2se/1.4.2/docs/guide/versioning/spec/versioningTOC.html

[47] E. Eide, “Manage your software with the java product versioning specification - an introduction to component versioning with java,” JavaWorld, Tech. Rep., September 2002.

[48] A. R. B. Jack, “Jar hell,” Krysalis Community Project, Tech. Rep., January 2004. [Online]. Available: http://krysalis.org/version/jar-hell.html

[49] S. McCamant and M. D. Ernst, “Early identification of incompatibilities in multi-component upgrades,” in *ECOOP 2004 — Object-Oriented Programming, 18th European Conference*, Olso, Norway, June 16–18, 2004.

[50] P. Brada, “Component revision identification based on idl/adl component specification,” in *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, 2001, pp. 297–298.

[51] ——, “Towards automated component compatibility assessment,” in *Workshop on Component-Oriented Programming (WCOP’2001)*, June 2001, Position Paper. [Online]. Available: http://research.microsoft.com/ cszypers/events/WCOP2001/

[52] P. Hnetynka and F. Plasil, “Distributed versioning model for mof,” in *WISICT 2004*, ser. ACM international conference proceedings. Cancun, Mexico: Computer Science Press, January 2004, pp. 489–494.

[53] S. Visnovsky, “Checking semantic compatibility of sofa/dcup components,” Master’s thesis, Charles University, Faculty of Methematics and Physics, Prague, 1999.

[54] A. Avizienis, “The n-version approach to fault-tolerant software,” *IEEE Transactions on Software Engineering*, vol. 11, no. 12, pp. 1491–1501, 1985.

[55] J. E. Cook and J. A. Dage, “Highly reliable upgrading of components,” in *International Conference on Software Engineering*, 1999, pp. 203–212. [Online]. Available: citeseer.ist.psu.edu/cook99highly.html