Version Management and Composition of Software Components in Different Phases of Software Development Life Cycle

Parminder Kaur, Hardeep Singh
Department of Computer Science and Engineering,
Guru Nanak Dev University, Amritsar-143005, India.
Email: parminderkaur@yahoo.com, hardeep_gndu@rediffmail.com

Abstract

The key factor of component-based software development is the composition of pre-fabricated components. Although research efforts have focused on this issue, yet the optimal performance of component-based systems has not been achieved. If the concept of version management is introduced along with composition technology then it will help in locating the component mismatch in the earlier stages of software development life cycle. This paper analyses the significance of version management and composition of software components in different phases like analysis phase, design phase and deployment phase of software development life cycle. A comparative analysis of different available component models like COM, .NET, Enterprise JavaBeans (EJB), CORBA, SOFA, UML 2.0 and Web Services with respect to these two issues is also discussed.

Keywords: - Component, Component-Based Systems, Version Management, Software Composition, Component Model

Introduction

Software Configuration Management (SCM) deals with development, assembly, configuration, evolution and maintenance of software systems [1]. It has a significant role in system development process i.e. from analysis phase to deployment phase. It is well established in the late phases of software development i.e. during programming and integration but less commonly used in the early phases, i.e. analysis and design.

There are two main reasons for not using SCM in early phases. First, there are generally not many versions of analysis and design documents. Making backup copies of these documents lead to a significant increase in the complexity and the number of versions in early phases. Secondly, SCM systems [2] are not well adapted to the needs and conditions of managing documents in the early phases of the development process. SCM systems like RCS, CVS, Subversion and SCCS work only on files that consist of lines of text. These systems are not aware of the logical structure of the document contained in a file. These systems mainly deal with modification of the document, e.g. the insertion/deletion of a statement in a program and fail to work with document management in the early development phases. Three major conditions, which are not fulfilled by SCM systems while managing the documents, are: -

- Documents are stored in the form of UML diagrams and diagrams are further stored in files, either in printable formats or in XML formats. In the case of a class diagram, a few lines of text in the file represent each class. A small change in the diagram can lead to a complete change of the file contents and a large number of significant textual differences. This type of change is not fully incorporate by SCM systems.
- Analysis, design and implementation are considered as parallel activities, which mean that even simple modifications can affect several files, or parts of files, belonging to different development phases. Conventional SCM systems are not able to correctly represent such complex changes.
- Fine-grained data modeling is not supported by SCM systems.

Changes in documents occur due to several reasons, e.g. extending the functionality or restructuring of components, which leads to the large number of versions. A developer wants to store some consistent and intermediate versions of a document while working on a specific phase of a project. Conventional version management systems do not support temporary versions or relations between versions and tasks or phases of projects. Rather, these systems can only create successive or parallel versions, which can also be accessed by other developers.

Version Management in Different Phases of Software Development

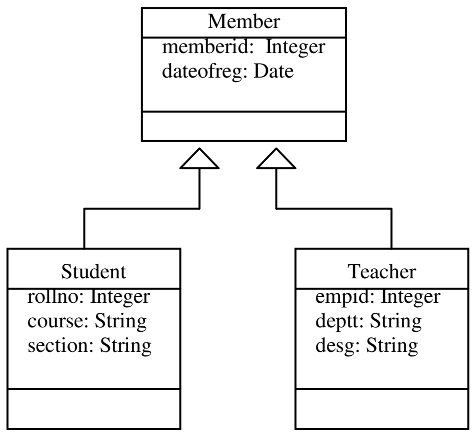
Analysis Phase

During the analysis phase, developer creates a model, which consists of the conceptual classes. This model represents the whole task. Each class carries its own function. Two versions of same example are shown in Figure 1. Version 2 is obtained after making expansion in version 1 of the example under consideration.

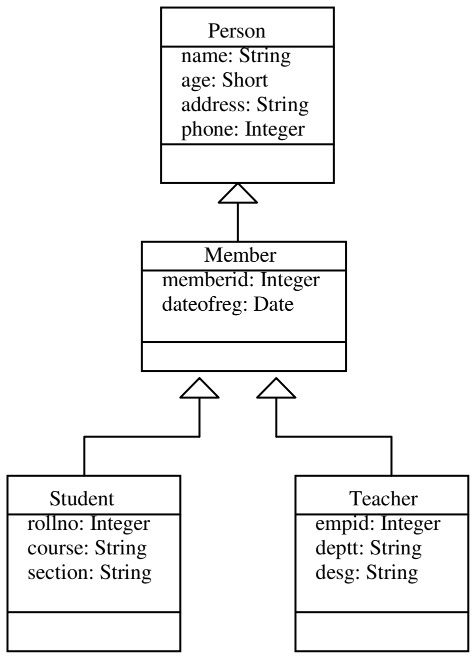
Design Phase

Change occurs in the design phase due to the extension of model with further classes. The reason of change can be addition of new functions or correction of errors. Due to changes, the software architecture, i.e. the class structure is usually modified again and again.

This concept can be illustrated with the help of an example class *Student* as shown in figure 2(a). This class acts as a container for S*tudent* along with *search( ), add( ), delete( )* operations. Suppose a class *Teacher,* which offers same functionality, later, extends the system. Therefore the developers extend the model with a common super class *Member* figure 2(b), which shows that the common methods are shifted from the class *Student* into the class *Member,* which leads to the shifting of a block of text between two files or within one file. Conventional SCM systems

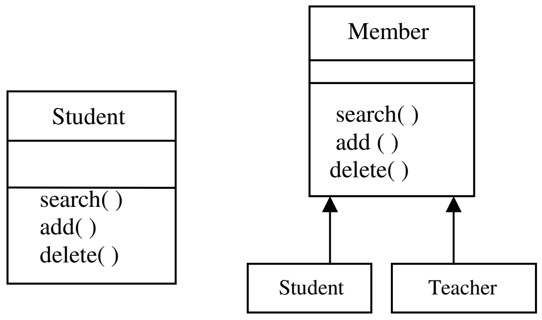


a) Version 1



b) Version 2

Figure 1: Class diagram of example model.



(a) Before Extension (b) After Extension

Figure 2: Example Class diagrams.

are not able to identify this shift correctly. These systems consider this process as the deletion of one block of text at the first location and the insertion of a new block of text at the second location [3].

This situation can be avoided if fine-grained data modelling is combined along with version management. By doing so, it becomes possible to detect the shift of a method and similar modifications. Figure 3 shows a meta-model of a fine-grained model for UML class diagrams. The component relationships between objects are taken as bi-directional so that the whole structure can be traversed from root to leaves and vice versa.

The various component relationships between the object types are:

- A document contains classes, a class contains methods and attributes and methods depend upon parameters.
- Classes can express inheritance, aggregation or association relationships between them.

The Meta model for figure 2 using figure 3 can be shown as in figure 4.

There are two requirements, which are to be fulfilled by software configuration management tools while using a version model for a fine-grained data model i.e.

- SCM tools modify the syntax tree of the documents in such a way that all modifications of the documents are represented as operations upon the syntax tree.
- SCM tools shift the objects across the document instead of deleting and recreating.

The second requirement results in new objects with new identities. The whole syntax tree needs to be searched in order to find new positions. Almost all SCM tools use *tool transactions* (TTAs) [4] offered by the repository to operate on the data, therefore all modifications, which are made in context of a TTA lead to an automatic versioning of the modified objects and relationships with the restriction that each object and each relationship can be versioned only once in a TTA. Each object and each relationship is versioned independently, thus it creates its own version tree. The automatic creation of versions increases the probability that the version needed by a user at a later access actually exists. Along with this, it leads to large number of object versions. So it becomes necessary to store the consistent versions together. This is possible if all versions of objects and their relationships are combined under configuration structure as shown in figure 5.

Configuration structure consists of configurations like *working* configuration, *base* configuration and *frozen* configuration. Base configuration refers to that configuration from which accessible versions can be determined. The working configuration refers to the current configuration or successor of base configuration. In figure 5, configuration 1.1 refers, as base configuration whereas configurations 1.3, 2.2, 3.1 are referred as working configurations. Configurations 1.2 and 2.1 are considered as frozen configurations. The objects or components with latest versions could be one of the following: -

- Currently modified version in the working configuration
- Frozen version from the base configuration
- One of its predecessors

All configurations having a successor cannot be changed any more and are stored as persistent objects in the repository as shown in figure 6, so that user can make simple access. The configuration objects can also be extended by information with

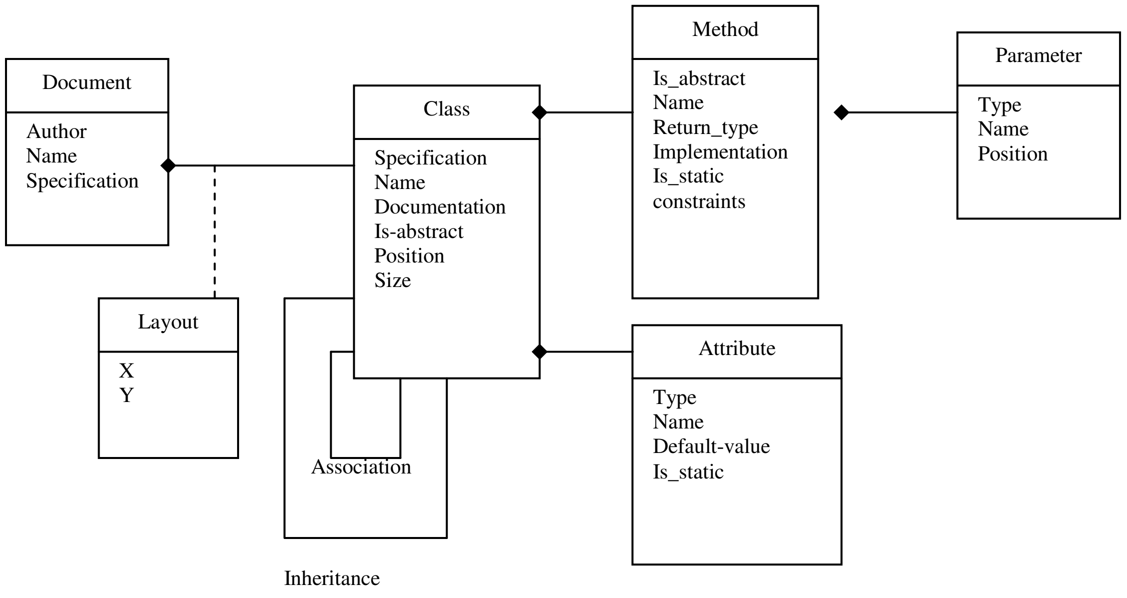


Figure 3: Meta model of a fine-grained data model [3].

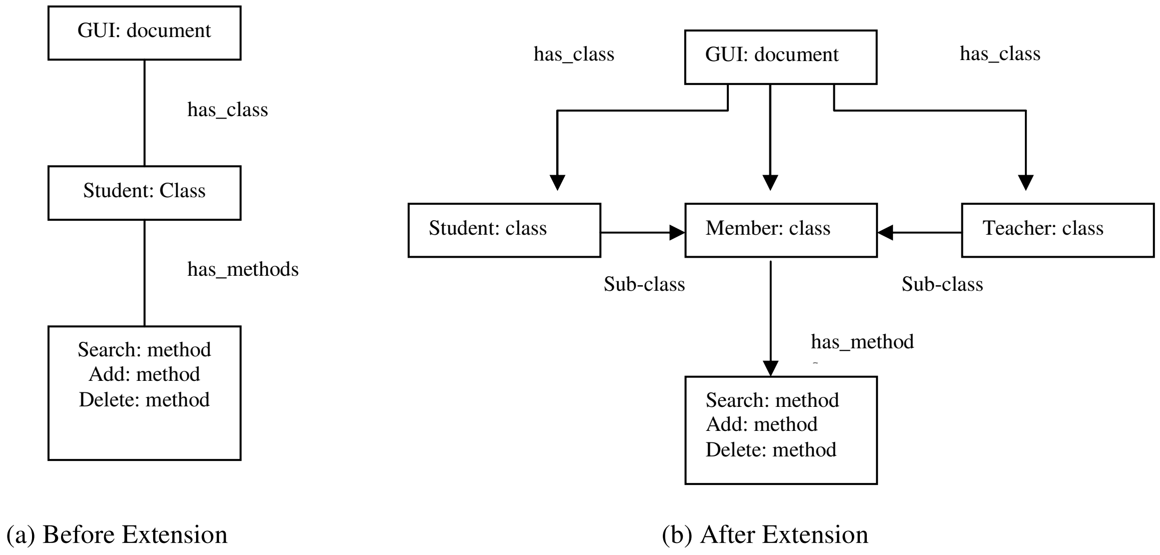


Figure 4: Example of a fine-grained object structure.

respect to executed changes, for example a modification comment.

All modifications done to complete the process or a phase of a process can be represented through *Design transactions* (DTAs) [4] figure 7. The major difference between DTAs and tool transactions (TTAs) is that a DTA refers to a logical frame within which the TTAs are executed. The DTA is not authorized to change the documents directly. All changes can be done inside the TTAs, which are executed inside an operating system process. Termination of a TTA results in the termination of operating system process but DTA is not bound to an operating system process and has a longer run time. Therefore tool developer has the freedom to use this versioning concept in a wide variety of software development processes e.g. the waterfall model, the spiral model, or the Unified Process.

Deployment Phase

Configuration hierarchy allow user to add more configurations instead of modifying the whole structure. It also makes it possible to roll back the upgrade if the new configuration breaks some existing functionality. Actually when a new configuration of a component is added, a new version or variant takes place. The developer has an idea about the change, going to take place in the behaviour of that component. Behaviour that was not meant to be changed but has changed is exactly the breaking of existing functionality. To overcome this problem, Cook [5] presented a framework, Hercules, figure 8, for highly reliable upgrading of the components. Reliability is provided by keeping existing versions of the component running and removing the old component when it is determined that the new component fully satisfies its role.

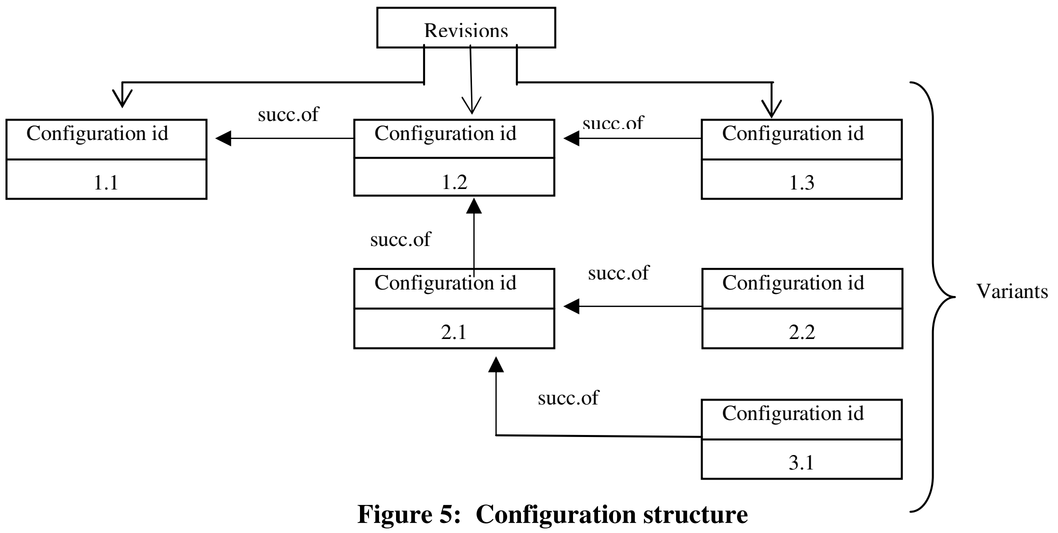


Figure 5: Configuration structure

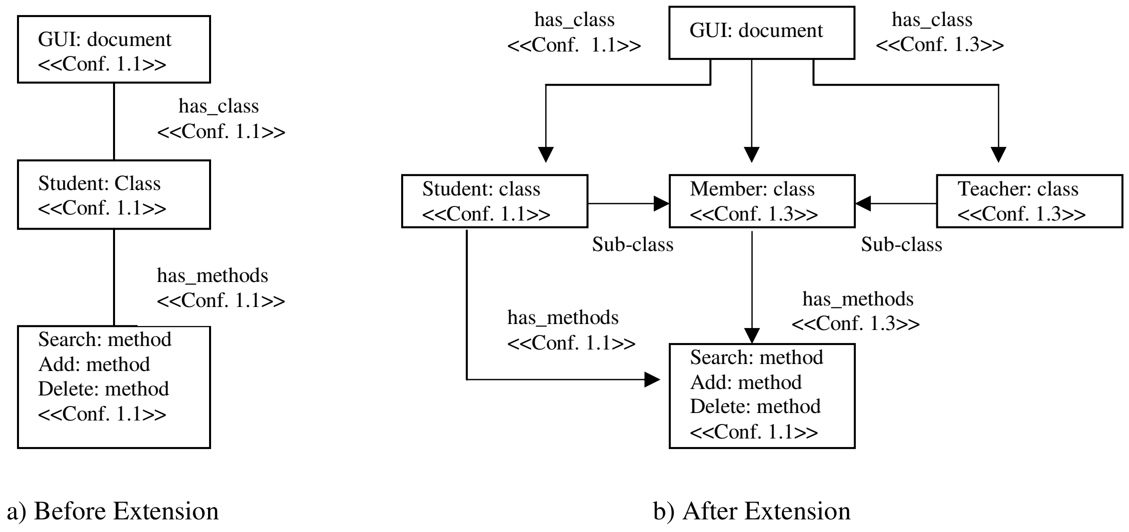


Figure 6: Object structure in repository along with configuration details.

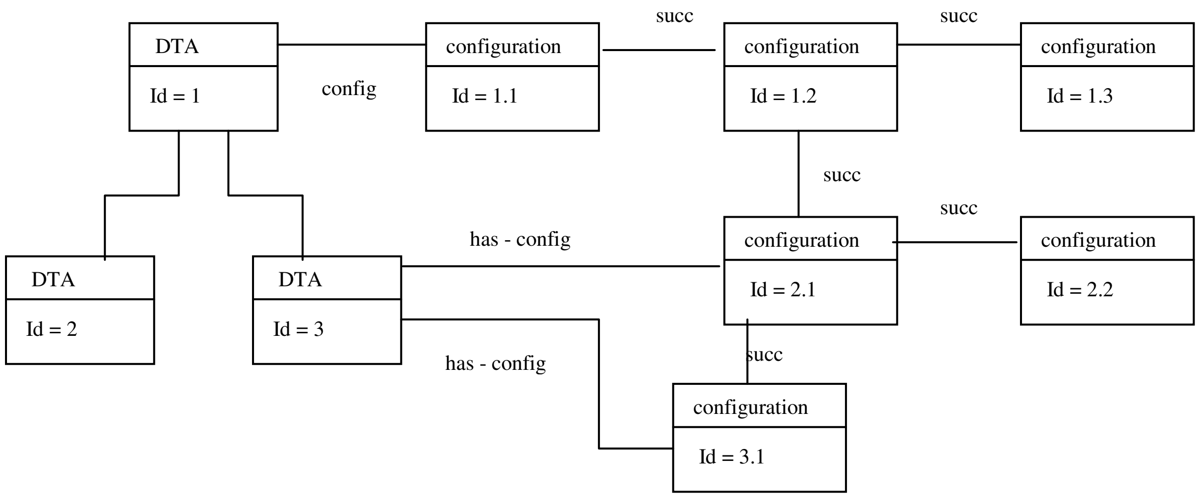


Figure 7: **Configuration structure through Design Transactions.**

The Arbiter invokes each of the component versions when the system requests it, and sends the selected result back to the system. The Arbiter also contains component management facilities for dynamically adding and removing component versions. Figure 9 shows the overall process of component upgrading when a new version of the component is created and tested, and given a specific domain. After installation of a new version into the running system, statistics could be gathered on all running versions.

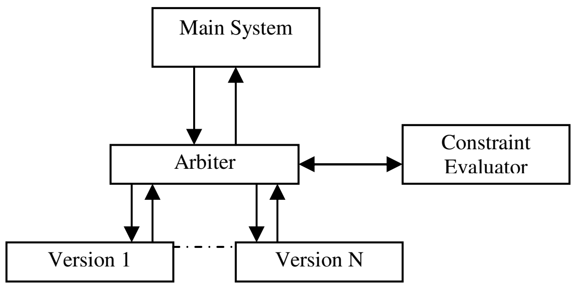


Figure 8: The HERCULES framework.

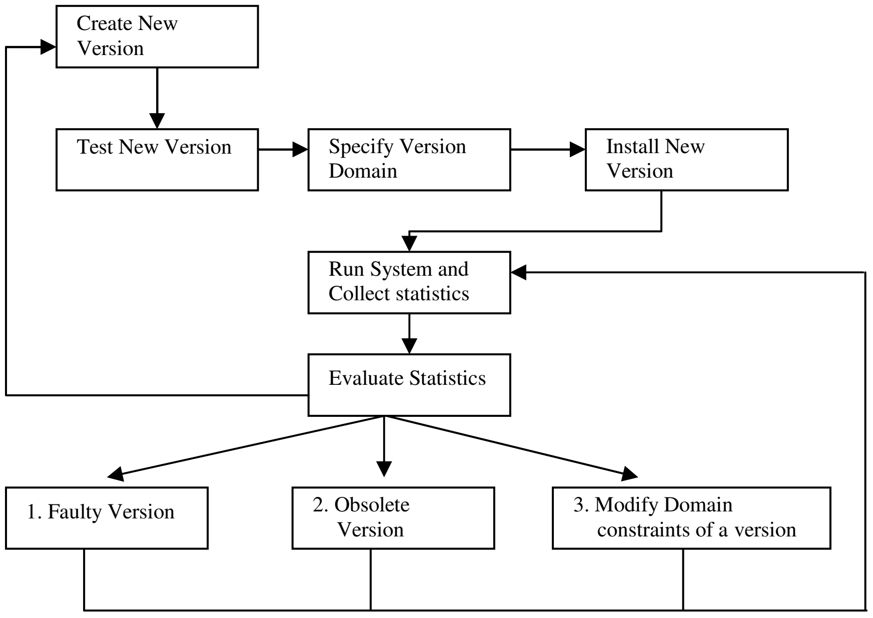


Figure 9: The Component Upgrading Process.

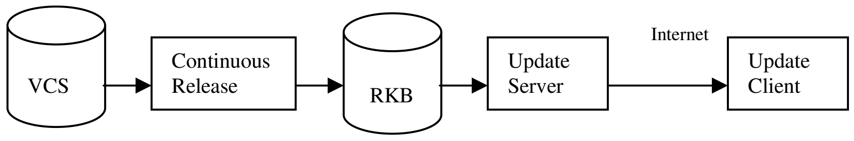


Figure 10: **Continuous Release Architecture [7].**

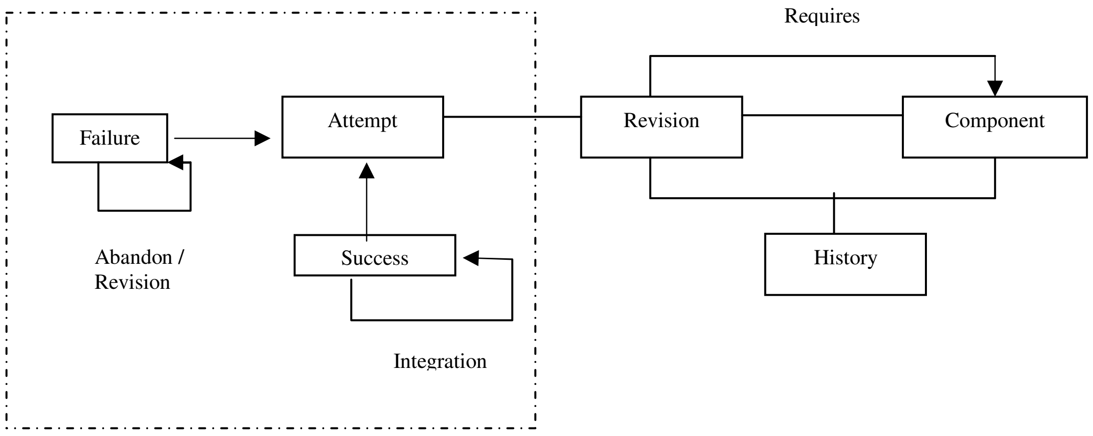


Figure 11: Continuous Integration Component Model.

When a component with new version is installed, the change occurs in its dependencies. The declaration of dependencies along with its new version number is specified in file, known as RPM specification file [6]. If a new release of a component is introduced, the declaration of its version number is also updated along with the declaration of its dependencies. Since such dependencies always refer to released components, makes component-based releasing a recursive process. The changes between the versions of components are stored with the help of version control systems. Users are free to decide whether they accept an upgrade or not within the limits of consistency. Upgrades can be loaded at any time without additional operating cost from development. Figure 10 shows that how version control system (VCS) stores all changes during continuous release. Release Knowledge Base (RKB) keeps the track of those revisions, which have passed integration process. The integrated component revisions should pass through quality assurance stages before passing to users. Update server can query RKB in order to compute updates from releases. These updates can be delivered to the user through Internet.

Continuous release architecture consist all updated versions of components, which are accessible to the user according to their composition. Before releasing the new version of the component, it should pass through several revisions for successful integration. Figure 11 elaborates the architecture of component model with continuous attempts of successful integration with different revisions of a component.

In Continuous Integration Model, figure 11, *Component* is considered as a set of components [7]. As changes occur, new components may evolve or existing components may be retired or modified. In order to incorporate the changes, component revisions are performed which can be represented as:

$$Revision \subseteq Component \times N$$

Where N refers to number of revisions performed on the component. Each component passes through more than one revision and differences between the revisions can be stored by version control system. In the present work, Subversion, a version control system is used to store the differences between revisions. Subversion stores full text of the most recent revision of a file as well as related historical revisions. This facility of Subversion allows retrieving any version of a file in a quick manner. All revisions are stored in repository in the form of history, which can be defined with the help of following relation:

$$History \subseteq Time \times (Component \times Revision)$$

In Subversion, all the revisions of same component at different times can be viewed with the help of Log table. The Log table maintains the information regarding who made the change, at what time and why. When change evolves, dependencies in components also change. Since Subversion implements a virtual versioned file system, it helps in keeping the track of changes to whole directory trees over time. Subversion supports the versioning of files as well as directories.

One portion of the figure 11, enclosed in dotted square, marked as Continuous Release System shows the continuous successful integration of a component. Successful attempt of integrating a component depends upon various revisions of a from component time to time. The relation Attempt records all the attempts of a component for successful integration. If the component does not match with the current environment, it may either abandon or consider for further revision. The relations Success, failure and Integration can be shown as:

$$Success \subseteq Attempt \qquad Failure \subseteq Attempt$$

$$Integration \subseteq Success \times Success$$

In many cases, different users or organisations use different releases like alpha, beta, etc. of the same component. These releases indicate the direct component revisions through an organisation, starting from development and ending with actual users. Revisions on component are performed to make it adaptable according to the environment. Figure 12 shows the adaptation framework, suggested by [8], for two components A and B, which provide an appropriate functionality, having distinguishable type of specifications. One specification represents the target state of application i.e. project specification and other represent as-is component specification. As per project specifications, Analyser will read the component specifications and compare them with demanded functionality. It will then provide this information i.e. regarding required functionality, to the Linkage Management. Then Linkage Management will perform a search in the Component Pool with respect to the required component. After search, there can be two possibilities-firstly, the required component is available for adaptation and secondly, there is no component for adaptation i.e. there is need to develop the component from scratch. If the component with desired functionality is available in the component pool, Linkage Management will integrate it into the application. After adaptation, Analyser will again verify the functionality of the application. If the required component is not available then Component Pool will be extended with new software components.

The major concern here is to define the semantic and to analyse the compatibility of black-box components. The behaviour of these components can be analysed with the help of Architecture Description Languages like Unified Modelling Language (UML) [9], Object Constraint Language (OCL) [10] and Meta Information Definition Language (MIDL) [11]. The software development process becomes more transparent when the right component is retrieved from the Component Pool. This helps in reducing the development time as well as inconsistencies.

Available Software Component Models

In component-based software development, the design of components is carried out separately from their deployment process, so that third parties independently compose them according to the environment. In current component models like JavaBeans [12], COM/COM+/.NET [13][14], CORBA [15], Open Service Gateway Initiative (OSGI) [16], Web Services [17], Koala [18], SOFA [19], Architecture Description Languages [20] and UML 2.0 [21], components are defined in the form of objects, classes and architectural units.

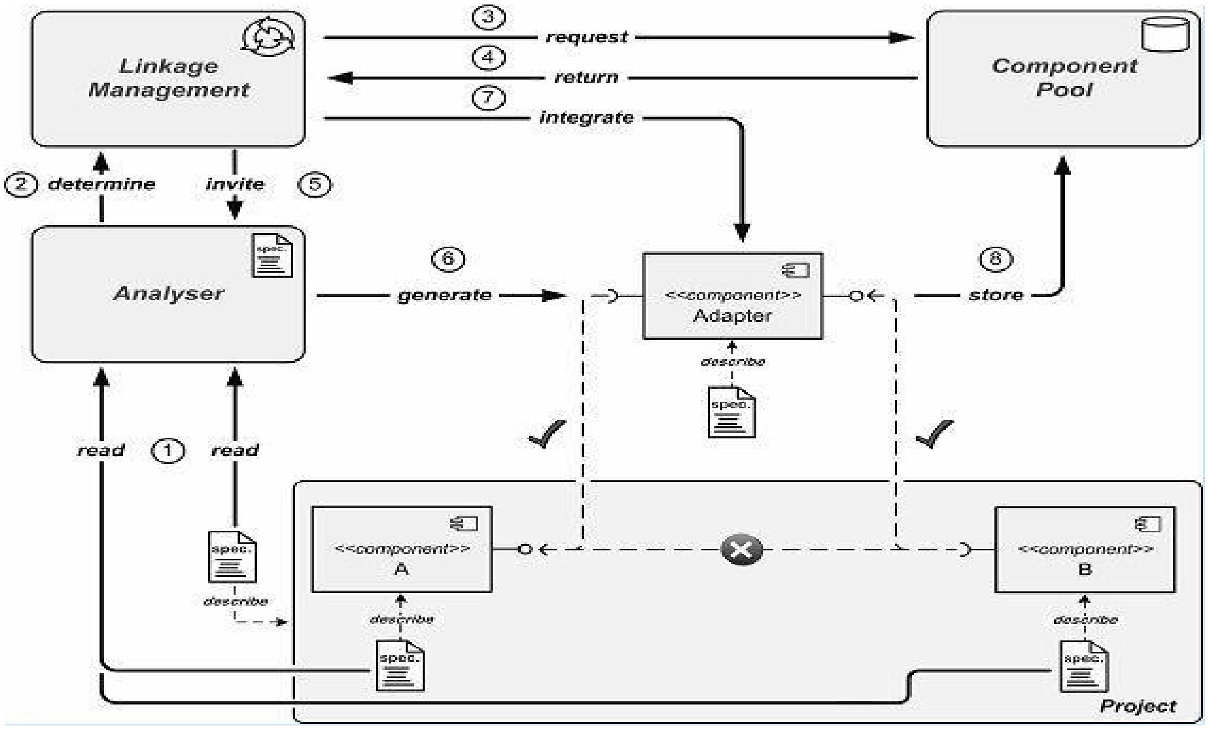


Figure 12: Adaptation Framework [8].

A software component model consists of [22- 24]:

- Syntax of components, i.e. how they are defined, constructed, and represented.
- Semantics of components, i.e. what components are meant to be.
- Composition of components, i.e. how they are composed or assembled.

The syntax of software components: The language used for constructing software components determines the syntactic rules for them. Therefore this language should be defined in a component model. In current component models, programming languages are used to construct components. For example: COM objects can be created with Object-oriented languages, such as C++. Similarly, in both java and EJB, a component is defined as a java class.

The semantics of software components: There is a commonly accepted abstract view of what a component is, i.e. a software unit that contains:

(i) Name
(ii) Code for performing services
(iii) An interface for accessing these services. To provide its services, a component may require some services.

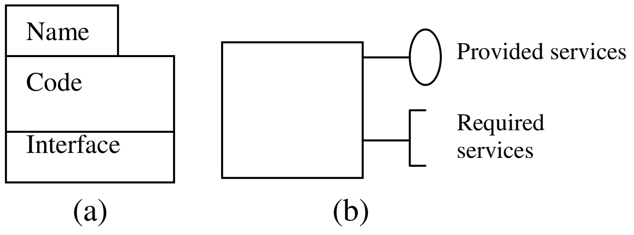


Figure 13: (a) Component Semantic (b) A software component.

The provided services are the operations performed by the component. The required services are the services needed by the component to produce the provided services. The interface of a component specifies dependencies between provided as well as required services [25]. Components are defined according to the component model. For example, in OO programming, components are referred as *objects* and methods of these objects are referred as *provided services*. Since these components do not specify their required services, therefore they are deployed into a container, through which they become accessible by other components. Components in Java are referred as a *java class*, deployed in a container known as BeanBox [26-28]. In CORBA Component Model (CCM) and UML 2.0, components are referred as architectural units.

The Composition of Software Components: The composition of software components can be done with the help of composition language [29]. The composition language should have suitable semantics and syntax that are compatible with those of components in the component model. Component models like COM, .NET use Microsoft Interface Description Language (IDL), CCM uses Object Management Group (OMG) IDL, UML2.0 uses the Unified Markup Language (UML) notation and web services use Business Process Execution Language (BPEL) to define components.

Comparative Analysis of Component Composition and Versioning Support in Component Models

In order to establish a framework for comparison, it is necessary to compare component models at the same level of services like: Distribution, Inheritance, Interface, Inheritance, Implementation, Dynamic Invocations, Platforms, Security, Design, Versioning Mechanism etc.

Table 1 divides the available software component models in four categories by taking into account three phases i.e. *design* phase, *deployment* phase and *run-time* phase [23-24],[30]. Component models of Category 1 like UML 2.0, has no repository in the design phase, therefore, components are all constructed from scratch. Composition is also possible in this phase. In the deployment phase, no new composition is possible. The composition of the component instances in the runtime phase is the same as that of the components in the design phase. These models can be defined as *Design without Repository.*

In Category 2, new components can be deposited in a repository but cannot be retrieved from it, in the design phase. Composition is possible, that is, composites can be formed, but composites cannot be retrieved from the repository as they do not have identities of their own. In the deployment phase, no new composition is possible. So composition of the component instances in the runtime phase is the same as that of the components in the design phase. These models can be defined as *Design with Deposit-only Repository.*

In Category 3, as far as design phase is concerned, new components can be deposited in a repository but cannot be retrieved from it. Composition is not possible. So no composites can be deposited in the repository. In the deployment phase, components can be retrieved from the repository and their binaries can be formed and composed. JavaBeans is only the component model of this category. This category can be defined as *Deployment with Repository.*

In the design phase of component models of Category 4, new components can be deposited and can be retrieved from the repository. Composition is possible and composites can be deposited in the repository. In the deployment phase, no new composition is possible. Therefore, composition of the component instances in the runtime phase is the same as that of the components in the design phase. The component models of this category can be defined as *Design with Repository.*

The comparison of versioning support provided by above discussed component models is also shown in table 1. From the table, it can be concluded that component models like CORBA, COM cannot handle more than one versions of a single component at the same time, whereas .NET, SOFA and UML 2.0 have full versioning support i.e. more than one versions of same component can be handled at one time. JAVA does not provide any automated tool to track the version information. Web Services support execution of multiple versions at a same time but again do not introduce a versioning policy and automated tools to detect syntactical or semantical changes or mechanisms to discover incompatibilities [24][31].

Conclusion

Through this paper, an effort has been made to define the significance of version management in different phases of software development life cycle. If version information is handled at earlier stages, it helps in producing good quality components. Composition of software components in various phases is discussed with respect to available component-based systems. An adaptation framework is discussed which helps in detecting syntactical or semantical incompatibilities of third-party components. Version support provided by these component models is also listed in the table, which shows the fact that though these component models are working as good middleware's still there is a need of mechanism which automatically detects the compatibilities between various components.

Component Model	**Target Environment**	**Component Specification / Syntax**	**Component Semantics**	**Component Composition**	**Handling Multiple Versions of a Single component**	**Versioning Support**
COBA	C++, Unix, Windows	OMG IDL	Objects	Design with Deposit-Only Repository	No	No
COM	Windows	Microsoft IDL e.g. C, C++, Ada	Objects	Design with Deposit-Only Repository	No	No
.NET	Windows	MSIL, CLS e.g. C#, VB, C++	Objects	Design with Deposit-Only Repository	Yes	Full
Web Services	Multi Platform	WSDL	Objects	Design with Deposit-Only Repository	Yes	Limited
EJB	Java	Java Prog. Language	Classes	Deployment with Repository	Via method *IsCompatibleWith*	Limited
UML 2.0	Java	Visual UML	Architectural Units	Design without Repository	Yes	Full
SOFA	Java	SOFA CDL	Architectural Units	Design with Repository	Yes	Full

Table 1: Comparison of Component Composition and Versioning Support in different Component Models [24], [31].

.References

[1] Peter H. Feiler (1991): Configuration Management Models in Commercial Software Development Environments, *SEI Technical Report CMU/SEI−91−TR−7*, March 1991.

[2] R. Conradi and B. Westfechtel (1998): Version models for software configuration management. *ACM Computing Surveys*, June 1998, 30(2), pp. 232–282.

[3] D. Ohst, U. Kelter (2002): A Fine –grained Version and Configuration Model in Analysis and Design, *Proc. of the IEEE International Conference on Software Maintenance (ICSM2002), 3-6 October 2002, Montréal, Canada*, pp.521-527.

[4] U. Kelter, M. Monecke, and D. Platz. (1999): Constructing distributed SDEs using an active repository. *Proc. 1st Int. Symposium on Constructing Software Engineering Tools (COSET '99); May1999, Los Angeles, CA*, pp.149– 158.

[5] Cook J.E., Dage J.A. (1999): Highly Reliable Upgrading of Components, Proc. *21st International Conference on Software Engineeing, Los Angeles, May 1999*. pp. 203-212.

[6] E. C. Bailey. (2000): Maximum RPM. Taking the Red Hat Package Manager to the Limit. Red Hat, Inc., 2000. Online: http://www.rpm.org/max-rpm.

[7] Van Der Storm, T. (2005): Continuous release and upgrade of component-based software, *Proc. of the 12th international Workshop on Software Configuration Management (Lisbon, Portugal, September 05 - 06, 2005). SCM '05. ACM, New York,* pp. 43-57.

[8] Rolf Andreas Rasenack (2008): Adaptation of Black-Box Software Components, *Ann. Univ. Tibiscus, Comp. Sci. Series 6, 2008*, pp 153-168.

[9] Cheesman, J., and J.Daniels (2000): UML Components- A Simple Process for Specifying Component-Based Software, *Reading, MA: Addison- Wesley*, 2000.

[10] Warmer, J., and A. Kleppe (2000): The Object Constraint Language, *Reading, MA: Addison – Wesley*, 2000.

[11] Karsten Wolke (2007): Higher Availability of Services in Heterogeneous Distributed Systems, *PhD Thesis, Leicester (GB),* October 2007.

[12] Sun Microsystems, JAVA, http://java.sun.com

[13] Microsoft, ".NET Development (MSDN)", http://msdn.microsoft.com/en-us/library/aa139615.aspx.

[14] Microsoft, "The Component Object Model: A Technical Overview (MSDN)", http://msdn.microsoft.com/en-us/library/ms809980.aspx

[15] OMG, CORBA, http://www.omg.org/corba

[16] OSGI. "OSGI Service Gateway Specification," Release 1.0, http://www.osgi.org.

[17] G. Alonso, F. Casati, H. Kuno, and V. Machiraju (2004): Web Services: Concepts, Architectures and applications, *Springer-Verlag*, 2004.

[18] R. van Ommering (2002): The Koala Component Model : Building Reliable Component-Based Software systems, *I. Crnkovic and M. Larsson, eds., Artech House, 2002*, pp. 223-236.

[19] F. Plasil, D. Balek, and R. Janecek (1998): SOFA/DCUP: Architecture for Component Trading and Dynamic Updating, *Proc. Fourth Int'l Conf. Configurable Distributed Systems (ICCDS '98) 1998,* pp. 43-52.

[20] P. Clements (1996): A Survey of Architecture Description Languages, *Proc. Eighth Int'l Workshop Software Specification and Design (IWSSD '96), 1996,* pp. 16-25.

[21] Unified Modeling Language Superstructure, V2.1.2, (OMG UML), http://www.omg.org/docs/formal/07-11-01.pdf

[22] Hardeep Singh, Parminder Kaur (2005): Use Of Component Models In Component - Based Development Systems", *8th Punjab Science Congress, Patiala, India, Feburary 2005.*

[23] Kung-Kiu Lau, Zheng Wang (2006): A Survey of Software Component Models", *2nd edition, Preprint Series, Manchester*, 2006.

[24] Kung-Kiu Lau, Zheng Wang (2007): Software Component Models, *IEEE Transactions on Software Engineeing, Vol 33, No. 10, Oct. 2007,* pp. 709 – 724.

[25] D'Souza, D., and A. C. Wills (1998): Objects, Components and Frameworks: *The Catalysis Approach, Reading, MA: Addison-Wesley,* 1998.

[26] Ivica Crnkovic, Magnus Larsson (2002): Building Reliable Component-Based Software Systems", *Artech House, Boston, London,* 2002.

[27] JavaBeans Architecture: BDK Download. Sun Microsystems, (2003): http://java.sun.com/products/javabeans/software/bdk_download.html.

[28] Java 2 Platform, Enterprise Edition Sun Microsystems, (2007): http://java.sun.com/j2ee/.

[29] M. Lumpe, F. Achermann, and O. Nierstrasz (2000): A Formal Language for Composition, *Foundations of Component Based Systems, G. Leavens and M. Sitaraman, eds., Cambridge Univ. Press, 2000,* pp 69-90.

[30] B. Christiansson, L. Jakobsson, and I. Crnkovic (2002): CBD Process, *Building Reliable Component-Based Software Systems, I. Crnkovic and M. Larsson, eds., , Artech House, 2002,* pp. 89-113.

[31] Parminder Kaur, Hardeep Singh (2008): Automated Versioning Mechanism in Component-Based Systems, *CSI Communications, Vol. No. 32, Issue 6, Sept. 2008,* pp 21-28.