Dependency Management in Component Based Systems

Ramandeep Kaur[*1] Aman Arora [2]

CSE & Punjab Technical University *CSE&PTU*
Sai Institute of engineering &Technology, Amritsar *Sai Institute of engineering &Technology, Amritsar*
raman_nandha006@yahoo.co.in *aman_study@yahoo.com*

Abstract— **Dependency in component-based software system basically deals with controlling the configuration of the components. Software developed with commercial off the shelf (COTS) has many advantages as compared to traditional software development. But still while developing component-based software, some factors need to be considered. Because the components are developed separately from the system, there is a need to manage dependencies among components. As the time passes, certain changes are required in the components as well as in the system. When software is updated, certain compatibility issues need to be addressed. This paper tries to illustrate dependency management concepts in component-based systems. A tool named as Dependency Walker is used to calculate the component dependencies with respect to different versions of same software.**

Keywords— **Component Dependency Management, Managing Components, Component Interface, Managing Change Dependencies, Dependency Walker**

I. INTRODUCTION

Component-based software (CBS) is built through the ways of composition and integration, this development style is quite favorable for time and budget constrained [1]. Component-based software engineering improves productivity, quality and reusability and reduces maintenance overheads and time to market [2]. Dependency analysis is a useful technique that has many applications in software engineering activities including software understanding, testing, debugging, maintenance, and evolution. Similar to object-oriented systems, in which object is the basic building block, in component-based software systems, component is the building block. So it is very important to analyze component's context and its running environment in order to efficiently manage all kinds of dependencies in component-based software systems. Larsson [6] has defined dependency management as follows: Dependency Management (CM) is a discipline, which controls the consistency between the parts of the entire system, and can increase the reliability of component-based products. Configuration management (CM) is used to manage the development of complex systems.CM covers version, change, build, release and workspace management [5][12][13][14]. In order to apply managing dependencies between different versions of a component is an important issue in component configuration management (CCM). A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only [10]. When developing component-based software, it is really difficult to keep track of components. As this is to be done in early stages of software development so firstly components need to be identified. Then during the up gradation and assembly phase, dependency management is an important function. Dependency management is usually done with dependency graphs etc. So in component dependency management, firstly components are identified, then a dependency model is selected. Afterwards change management is done and at the last stage dependencies among various components are managed.

II. RELATED WORK

In estimation of maintenance cost of software i.e. higher is the impact of change in a part of the system; more is the cost of implementing that change. A dependency matrix based approach is proposed for recording the interdependencies in software components [3]. A matrix based approach is used to understand and manage the different forms of dependencies between components with the help of component dependency life cycle. Component dependency metric is used to represent the dependencies between components and the use of component interaction density metric shows relationship between component dependencies and their architecture [4]. Larsson [5] has given discussed some of the problems faced in Component-based systems and their possible solutions. Configuration Management (CM) is used for managing the complex components. In configuration management, version, change, build, release and workplace management are covered for this purpose. [7] has applied the Dependency Structure Matrix (DSM) to check design violation in architectural designs. In this approach every I, Jth cell of matrix contains the strength of dependencies in component pair (i, j). Vieira and Richardson [8] [9] used component-based dependency model (CB DM) to manage dependencies in Component-based software systems (CBSs). The CBDM is a graph that represents the "special associations" among the system's components based on their service. Stafford et.al, [15] developed an architectural level dependency analysis technique called chaining. Chains represent dependency relationships in an architectural specification. The individual chain associates elements of architecture that are directly related.

III. MANAGING COMPONENTS

Components typically consist of shared libraries, where the component functions are implemented. The programs using components do not refer to the libraries directly but to the component interfaces. The libraries are implementations of the interfaces. There are needed to keep track of changes on both logical and physical levels as well as their relations. Both libraries and interfaces must be identified. Component

Configuration Management must work on both levels. Versioning of interfaces is a more difficult task, because the interface is an abstraction without information about the physical representation. For this reason, separate the problem of managing components onto two levels: Managing libraries and managing interfaces [6]. Managing libraries prevented the executable from being updated when a new version of the library was released and Managing interfaces establish connection between a component and its user. If an interfaces is changed, the user needs to know that it has been changed and how to use the new version. Kruchten defines an interface as a collection of operations that are used to specify a service of component [11]. An interface serves to name a collection of operations and specify their signatures and protocols. An interact focuses upon the behavior, not the structure, of a given service.

IV. MANAGING CHANGE DEPENDENCIES

Managing Change, One of the major challenges in CBSs is how to manage changes, because the primary objective of a component is that it must be easily replaceable, that means two aspects: (1) Replaced by completely different implementation of the same functions, and (2) Replaced by an upgraded version of current implementation. When a system's various components evolve and its requirement changes, this objective places the emphasis on the architecture of the system, on being able to manage the total system.

Directed Graph:

Let V be a finite nonempty set, and let E⊆V V. The pair (V, E) is then called a directed graph, in which V is the set of vertices, or nodes and E is a set of directed edges or arcs represented by ordered pairs. Such a directed graph is denoted G = (V, E). The notation a b denotes (a, b) as edge.

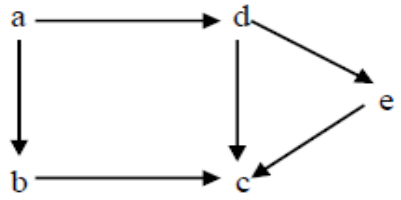

Fig 1: A graph G with 5 nodes a, b, c, d, e

Node	Adjacency List
A	B,D
B	C
C	Empty
D	C,E
E	C

Table 1: Adjacency Lists of G

This figure 1 shows an example of a graph G = (V, E), in which V= {a, b, c, d, e} and E = {(a, b), (a, d), (b, c), (d, c), (d, e), (e, c)}. Placing an arrow on the edge indicates the direction of the edge. Which is its list of adjacent nodes, also called its successors or neighbors. Paths are introduced to be able to define dependencies between components. An example of a path from a to d in Figure 1 is <a, b, c, d> since each pair (a, b) and (a, d) is a part of the set of edges E. Knowing that there is a path from a to d indicates that a is dependent on d, since a is affected if d changes. There are many algorithms to find all the paths between two nodes. Warshall's algorithm is mostly used for this purpose. When the dependencies have been calculated, it is possible to create a system structure, as defined in [16], with different levels of components. On the lowest level of components are components without dependencies to other component. This system structure is used as a model to calculate quality properties such as complexity and localization factors. The complexity is proportional to the number of dependencies between the components. The localization factor denotes the number of levels between components. A configuration is a set of components and their dependencies to other components. The configuration is a baseline since it represents a version of a system at a particular time.

V. DEPENDENCY WALKER: A TOOL

The tool "Dependency Walker" (www.dependencywalker.com) helps us to find dependencies by simply parsing the components. It is used for the evaluation of the presented dependency model. It parses through the system; finds all shared libraries and generates the dependency graph. Scanning all shared libraries and executables in a system creates a basic dependency graph. As the new version of the component is installed, the task of component dependency management is to handle all the conflicts in that situation. Because in such a case, the new component may have some additional dependent files. So these are the issues to be handled by version management. The information required by version management is mostly made available by this dependency walker tool. The information provided by the dependency walker for this purpose is:

- **General information regarding the file**
- **Module version numbers**
  - Image Version, OS Version, Subsystem Version, Linker Version
- **Types of Dependencies**
  - Implicit Dependency, Delay-load Dependency, Forward Dependency, ExplicitDependency, System Hook Dependency
- **Application Profiling**
- **Dependency Tree View**
  - Module List View, Parent import Function List View, Export Function List View, Log View.

For the purpose of version management, it is useful to gather all the information required during the comparison of two components or while looking at the dependent tiles of the components. Otherwise it won't be possible to gather all this information required. As the goal of using this is dependency management, so after collecting the information regarding the

dependencies of a component, it is required to compare the two versions of same component (figure 2).

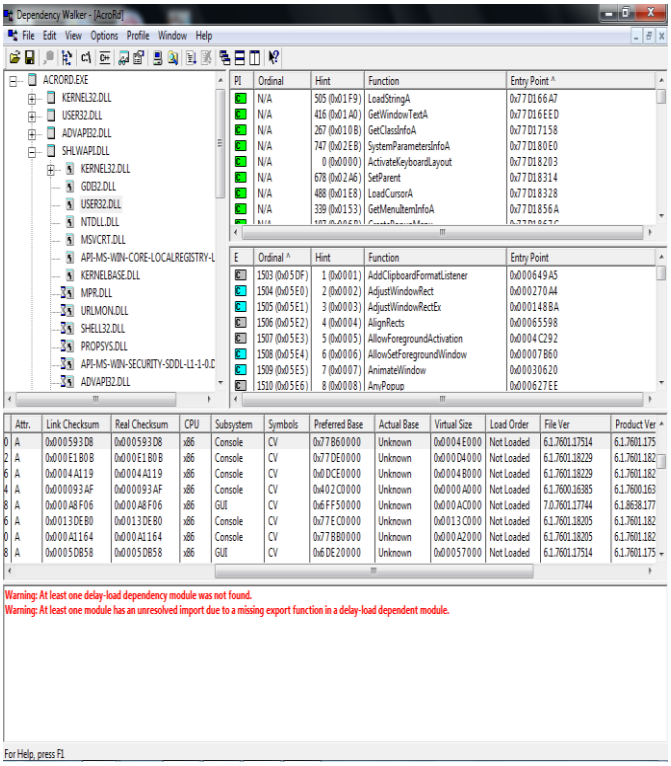

Fig 2: Shows the various views of the dependency walker.

VI. EXPERIMENTAL SETUP

Dependency Walker (DW) is basically used for version management. So in order to do that, more than versions need to be studied. In this section, two versions of same exe file are taken. They are then studied in dependency Walker. For instance, two versions of the software Acrobat Adobe Reader are taken. One is version 9.0 and the other is 11.0. Both of these exe files are opened in DW. Now for the purpose of version management, all the necessary information which is required and available is taken. Following are the changes which are studied in them:

1. Number of .dll files under each of the exe file: As can be seen from the snapshots given below (figure 3a, 3b), there is difference in the number of dll files in the each of the exe file. In version 9.0, there are 6 main dll files which in turn contain many dll files. And in version 11.0, there are just 20 main dll files which also contain in turn contain many dll files. So some difference in the dependencies can be made out here also.

2. Number of missing files or modules: The number of dll files which are studied, it can be easily seen that if some files name contain an icon which can be of any sort but is red in color. Then it shows that that particular file is missing or giving some warning.

3. Different version values for almost all fields: As it is already discussed that many different types of versions are created. So, they can also be used to compare the two exes. Whether file version is only new or the others also like OS version, Product version etc. it can be analyzed that whether all the dll files are updated or only selected one. Then after this the change studied for further analysis.

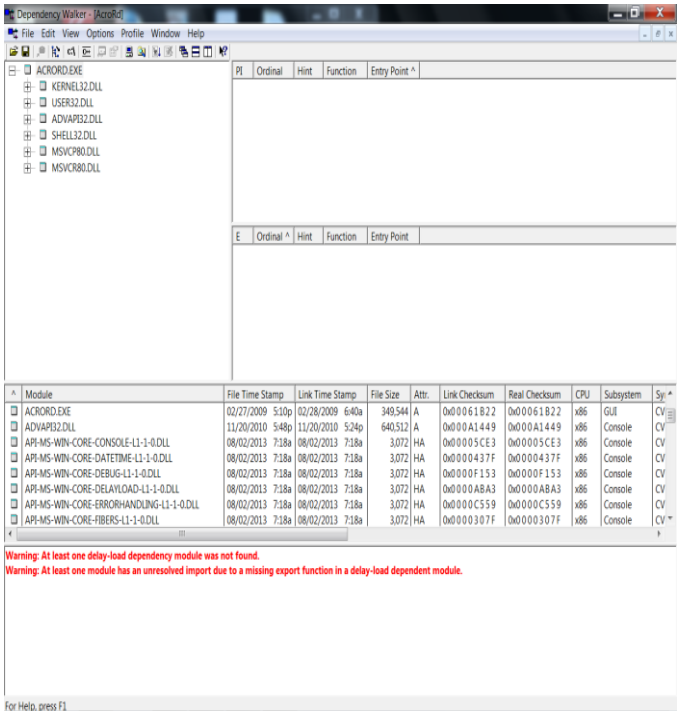

Fig 3a: Showing DLL Files of Adobe Acrobat Reader version 9.0

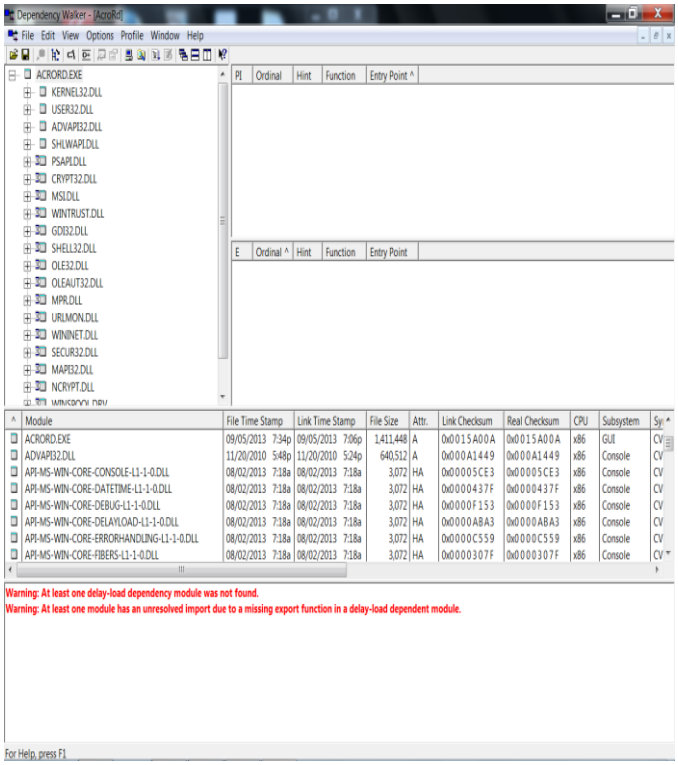

Fig 3b: showing DLL files of exe of Adobe Acrobat Reader version 11.0

Acrobat Adobe Reader version with details	Acrobat Adobe Reader 9.0	Acrobat Adobe Reader 11.0
No. of Components	6	20
Kernel DLL Files	25	25
User DLL Files	8	8
Advapi DLL Files	20	20
Shell DLL Files	48	82

Table 2: Showing dll files with version details

This case study shows that number of components in two versions of the same software are different i.e. Adobe Acrobat Reader with version 9.0 has 06 components means 06 dll files (first level dependencies) has 06 components means 06 dll files (first level dependencies) whereas version 11.0 has 20 components means 20 dll files (first level dependencies). Version 9.0 has greater dependency at next level where as version 11.0 consists more functionalities at first level so less complex at higher level. This shows that the various software product operated in same environment, have different number of dependencies. The reduced number of dependencies may indicate toward the simple architecture of the component integration. Components can be easily removed from their original positions and plugged at new locations. This shows that there exists a relationship between dependencies and functionalities provided by the respective software. First level dependencies can be measured manually but for calculation of high level dependencies, an automated tool is required as system level calculations cannot be performed manually.

VII. CONCLUSIONS AND FUTURE WORK

The Components provide system functionalities by interacting, cooperating and coordinating. Interaction, cooperation and coordination will produce dependencies among them. Usually, a group of components depend on each other to supply complex system functionality. When the system evolves new components are added or deleted. As a result, new dependencies occur. A tool named as dependencies walker by Microsoft is used to calculate the first level dependencies between the components. The case study shows that when new versions releases, number of DLL files also increases, which result the increase in functionality of new version. The future work includes the implementation of component design metrics in order to measure the size of the component. This can calculate the maintainability index factor and measure the strength of weight dependencies.

REFERENCES

1. Rajender Singh Chiller, "Measuring complexity of component-based system using weighted assignment technique" IPCSIT vol. 55(2012).

2. Kirti Tyagi, Arun Sharma, "Reliability of Component Based Systems – A Critical Survey" Issue 2,Volume 11, February (2012).

3. Kuljit Kaur, Hardeep Singh and Debasish Jana, "A Framework to Analyze impact of change in component based software engineering", Proceedings for 2nd International Workshop on Tool Support and requirements Management in Distributed Bangalore, India August 17th (2008).

4. Parminder Kaur, Hardeep Singh, "A Metric-Based Analysis of component Dependencies", Journal of the CSI, Vol.38, Issue No., Oct., Dec., (2008).

5. Larsson M., "Applying Configuration Management Techniques To component Based Systems", IT Licentiate Thesis, 2000-07, Department of Information Technology, Uppsala U niversity, (2007).

6. Larsson M. and Crnkovic I., "New Challenges for Configuration Management", MRTC Report, IT Licentiate thesis, 2000-07, Uppsala University.

7. Sanghal N.Jordon E., Sinha V and Jackson D., "Using Dependency Models to manage Complex Software Architecture", OOPSLA 05, San Deigo, California, USA (2005).

8. M.Viera, D.J.Richerdson, "Classifying and Dealing with dependencies in large Component Based Systems", 15th International Conference on software & systems Engineering & their applications, Paris, December (2002).

9. M. Viera, D.J. Richerdson, "The role of Dependencies in Component Based Systems testing and Evolution", Proceedings of the IWPSE-02, 24th International Conference on Software Engineering (ICSE 02, Orlando, USA), May (2002).

10. C.Szyperski, D.Gruntz, and S.Murer, "Component Software: Beyond Object-Oriented Programming", 2nd Edition, New
York: Addison-Wesley/ACM Press, (2002).

11. Kruchten Philipe, "Modeling Component Systems with the unified modeling language", http://cis.cs.tu_berlin.de/Lehre/WS/0001/Sonstiges/Konteng_sem/papers/Modeling_Componentsystems_with_UML.Pdf",(2001).

12. Somerville I., "Software Engineering", Addison Wesley (2001).

13. Asklund, U., Configuration Management for Distributed Development - Practice and Needs, Dissertation 10, Department of Computer Science Lund University, (1999).

14. Continuus Software Corporation, Continuus, http://www.continuus.com.

15. Stafford, J.Alexander, W., "Architecture level dependence analysis in support of software maintenance", proceedings of 3rd International Software Architecture Workshop, Orlando Florida, USA, ACM press, November (1998).

16. Crinkovic, I., "Large Scale Software System Management, Ph.D. Thesis, Department of Electrical Engineering, University of Zagreb, (1991).

