Managing Dependencies in Component-Based Systems Based on Matrix Model

Bixin Li

CWI(the National Research Institute for Mathematics and Computer Science)
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
Email: B.Li@cwi.nl

Abstract. Component-based software development technique and its intensive use in the industry has led to the wide research in various aspects of component-based systems (CBSs). Dependency analysis is an useful technique that has many applications in software engineering activities including software understanding, testing, debugging, maintenance, and evolution. In this paper, we propose a matrix-based approach to analyzing dependencies in CBSs. [1] By further investigating the application of matrix to analyzing dependencies in component-based system, we find that it is a good way to manage dependencies in a CBSs. To make it possible, we first discuss something about a dependency-based representation called the *component dependency graph* (CDG) and then construct the *dependency matrix* (DM) t Based on the CDG and DM, we build a mathematical basis for managing and analyzing dependencies in a CBS. Finally, we discuss some possible applications of our dependency analysis technique in component-based system understanding, testing, maintenance, and evolution.

1 Introduction

Component-based software construction technique has brought a new revolution in software engineering. Many researchers have done much research in this area [1,5]. We know that there are many interesting issues that need be furthermore explored. For example, dependency analysis in CBSs is important but complicated. Similar to object-oriented systems, in which object is the basic building block, in CBSs, component is the basic, but usually black-box, 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 CBSs. In general, as soon as a new component is installed in a system, it has an impact on a part of the system. The new component may refer to certain components, and also be used by other components – this is a kind of explicit direct dependency. In addition, there are also indirect dependencies, derived from the components which are used by the new component, and also implicit dependencies, that are related

[1] M. Larsson has mentioned similar topics, but his discussion is very preliminary [7].

to the system environment. In CBSs, there are at least four types of dependencies: explicit direct dependency, explicit idirect dependency, implicit direct dependency and implicit indirect dependency. [2]

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 in order to supply a complex system functionality. Any modification to a component can cause the change of composite functionality, because the composite functionality is reflected in different components. In addition, the replacement of a new version component will also cause the change of dependency between components [8].

Traditional dependency analysis is based on control and data flow analysis [3, 4]. When analyzing dependencies in CBSs, our aim is to analyze and solve these problems, which were presented by Stafford and Wolf [6], where they solved these problems using a method called *chaining*. In this paper, we bring up these issues again, and propose a solution based on matrix analysis.

- **Question 1**: Which other components are also required when one component is to be reused in another system?
- **Question 2**: What other components might be affected when one component is changed?
- **Question 3**: What is the minimal set of test cases that must be rerun when one component is changed?
- **Question 4**: What is the minimal set of components of the system that must be inspected when a failure of the system occurs?

The rest of the paper is organized as follows. Section 2 identifies all types of primary dependencies in a CBS. Section 3 build the component dependency matrix model, which consists of some component dependency graphs and the dependency matrice. Section 4 shows some applications of our dependency analysis technique. Section 5 discusses related work. Concluding remarks are given in Section 6.

2 Dependencies in Component-Based System

In a running CBSs, components interact, communicate, cooperate and coordinate each other, which will produce all types of dependencies in a CBSs. For example: (1) *Data dependency*, which is produced by data integration between different COTS components. In general, data dependency represents that the data defined in one component, but used in another one. (2) *Control dependency*, which is produced by control integration in CBSs, it is not explicit dependency. Control integration is realized by broadcasting, remote procedure calls

[2] Here we have some ideas similar to the idea in this draft, the difference is that we will construct dependency matrix, they construct component coupling graph. http://www.cis.ohio-state.edu/ weide/sce/miscellaneous/CCDs.pdf

or by general passing. (3) *Interface dependency* is produced by user-interface integration. Usually, the interface-event dependency is the main dependency form in CBSs. When one component need another component to do something, it first send a message to trigger an event through its interface, then the event activates another component to response the message through one interface of another component. (4) *Time dependency* represents that the behavior of one component precedes or follows the behavior of another one in CBSs. (5)*State dependency* represents that the behavior of a basic component will not happen unless the system, or some part of the system, is in a specified state. (6)*Cause and effect dependency* represents that the behavior of one component implies the behavior of another component. (7)*Input/Output dependency* represents that a component requires/provides information from/to another component. (8)*Context dependency* represents that a component runs must be under special context environment. [3]

3 The Construction of Component Dependency Matrix Model

To understand a CBS, we must first understand each component and the relationships between components in the system. In this paper, we use the *component dependency graph* (CDG) and *dependency matrix* to model various types of dependencies between components in the system so as to analyze and manage these dependencies in a good way.

3.1 Component Direct Dependency Graph and Adjacency Matrix

The *Component Direct Dependency Graph* (CDDG) of a CBS, denoted by $G_{CDDG} = (N_c, E_c)$, is a directed graph, where N_c is a set of vertices each represents a component in the system, and E_c is the set of dependency edges each represents a direct dependency between components. In order to construct the CDDG of a CBS, we must identify all components involved in the system, and find all direct (explicit and implicit) dependencies between these components.

Source Code is Available. When the source code of a component is available, we can locate explicit direct dependencies by looking for component relationship key words, which usually appear in the code immediately after the component name; We can locate implicit direct dependencies by looking for parameter passing [4].

– Identifying component:

[3] We can find some of these concepts in [6]. In our method, we call these dependencies *feature dependency*.

[4] Refer to http://www.cis.ohio-state.edu/ weide/sce/miscellaneous/CCDs.pdf

1. In the code (for example, written in C++ or Java), find the name of the concrete component, i.e., the name of modular, class or interface that is the main component,.
2. Find all the other components that this component directly depends on by examining all **#include** statements. Recursively draw the complete CDDG for every component that appears in a **#include** statement, even for every component that appears in **#include** statement inside a formal comment. The components defined in these files are all abstract components. You can examine these files in any order.

– Analyzing Dependency relationships between components
1. Find each *explicit direct dependency* between a component and the component it depends on. For each explicit direct dependency, we draw an edge between the two components.
2. Find each *implicit direct dependency* between a component and the component it depends on. For each implicit direct dependency, we draw an edge between the two components.

Source Code is not Available When a new component is installed, it must register all information (metadata), including all types of dependencies, context or running environment etc. We can construct the CDDG based on this information.

Adjacency Matrix We can use adjacency matrix $AM_{n\times n}$ to represent direct dependencies of component-based systems [5]. In the matrix, each component is represented by a column and a row. If a component c_i is dependent on another component c_j, then $AM[i,j] = 1$. More formally, the values of all elements in $AM_{n\times n} = (d_{ij})_{n\times n}$ are defined as follows:

$$d_{ij} = \begin{cases} 1 \text{ if} c_i \rightarrow c_j \\ 0 \text{ otherwise} \end{cases}$$

If position $AM[i,j]$ in the matrix is 1 then there is an edge from i to j. After the calculation each position denotes that there is a path from a vertex to another vertex.

3.2 The Derivation of Indirect Dependency Relationships

According to the description in above section, the direct dependencies can be represented in a adjacency matrix. Because of the transitivity of dependencies, we can further to gather all indirect dependencies in the system by calculating the transitive closure of dependencies. The alogrithm of calculating the transitive closure is Warshall's algorithm.

[5] Similar meanings of adjacency matrix mentioned in the reference [7]

3.3 Component Dependency Graph

In fact, the nodes in component dependency graph (CDG) are the same as that in the CDDG. Therefore, to construct the CDG, we need only add all indirect dependency relationships to the CDDG, which include explicit direct dependencies and implicit direct dependencies, which we derive using Warshall's transitive closure algorithms based on the adjacency matrix direct dependency relationships (explicit and implicit).

3.4 Component Dependency Matrix

Let S_{com} be a CBS consisting of n components: $C_1, C_2, \cdots, C_n$. We can define dependency matrix as follows. Without loss of generality, we only include eight types of dynamic dependencies between components:

$$\text{DM}= \begin{bmatrix} d_{11} & d_{12} & \cdots & d_{1n} \\ d_{21} & d_{22} & \cdots & d_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ d_{n1} & d_{n2} & \cdots & d_{nn} \end{bmatrix}$$

Where,

$$d_{ij} = cd_{ij} + dd_{ij} + td_{ij} + sd_{ij} + rd_{ij} + io_{ij} + ed_{ij} + id_{ij} \quad (1)$$

In above equation, "+" is a bool operator: 1+1=1,1+0=1,0+1=1, and 0+0=0. The elememts of d_{ij} is defined as follows: let Γ^h denote the dependency relationship between two components, $h = 1$ denotes the direct dependency relationship between two components, $h > 1$ denotes the indirect dependency relationships between the two components, then, predicate $C_i \Gamma C_j$ is true iff component C_i directly depends on C_j. Let Γ_{cd} denote control dependency relationship between two components, then, $C_i \Gamma_{cd} C_j$ is true iff component C_i directly control depends on C_j. Similarly, Γ_{dd}, Γ_{sd}, Γ_{td}, Γ_{rd}, Γ_{io}, $\cdots$, denote data dependency, state-based dependency, temporal dependency, causal dependency, input/output dependency, $\cdots$, respectively. So, we have

$$DM = CDM + DDM + TDM + RDM + SDM + RDM + IOM + IDM \quad (2)$$

where CDM denotes *control dependency matrix*, DDM denotes *data dependency matrix*, TDM denotes *temporal dependency matrix*, RDM denotes *causal dependency matrix*, SDM denotes *state dependency matrix*, IDM denotes *interface dependency matrix*, IOM denotes *input/output dependency matrix*, and EDM denotes *context or environment dependency matrix*. We call all these matrice *feature dependency matrix*. Now, we define the values of elements of all feature dependency matrices.

$$cd_{ij} = \begin{cases} 1 \text{ if } C_i \Gamma^h_{cd} C_j \\ 0 \text{ if } \neg C_i \Gamma^h_{cd} C_j \end{cases} \quad dd_{ij} = \begin{cases} 1 \text{ if } C_i \Gamma^h_{dd} C_j \\ 0 \text{ if } \neg C_i \Gamma^h_{dd} C_j \end{cases} \quad td_{ij} = \begin{cases} 1 \text{ if } C_i \Gamma^h_{td} C_j \\ 0 \text{ if } \neg C_i \Gamma^h_{td} C_j \end{cases}$$

$$rd_{ij} = \begin{cases} 1 \text{ if } C_i \Gamma^h_{rd} C_j \\ 0 \text{ if } \neg C_i \Gamma^h_{rd} C_j \end{cases} \quad sd_{ij} = \begin{cases} 1 \text{ if } C_i \Gamma^h_{sd} C_j \\ 0 \text{ if } \neg C_i \Gamma^h_{sd} C_j \end{cases} \quad io_{ij} = \begin{cases} 1 \text{ if } C_i \Gamma^h_{io} C_j \\ 0 \text{ if } \neg C_i \Gamma^h_{io} C_j \end{cases}$$

$$id_{ij} = \begin{cases} 1 \text{ if } C_i \Gamma^h_{id} C_j \\ 0 \text{ if } \neg C_i \Gamma^h_{id} C_j \end{cases} \quad ed_{ij} = \begin{cases} 1 \text{ if } C_i \Gamma^h_{ed} C_j \\ 0 \text{ if } \neg C_i \Gamma^h_{ed} C_j \end{cases}$$

3.5 Analyzing Dependencies Based on Dependency Matrix

Based on different forms of dependency matrix, we can understand all types of dependency relationships in CBSs, for example:

- Upper-triangular dependency matrix–If dependency matrix is upper-triangular dependency matrix, all dependency relationships between components are uni-directioon. Component dependency graph is in fact a directed tree.
- Diagonal dependency matrix–If dependency matrix is diagonal dependency matrix, then there is no any dependency relationship between components. Component dependency graph is in fact a graph composing of only some isolated points.
- Sparse dependency matrix–If dependency matrix is sparse dependency matrix, then the CBS is a loosely coupled system.
- Data dependency matrix–If $DM = DDM$, then there are only data dependency relationship between components. Data dependency is one of the best dependeency forms, because it can reduce inter-component coupling.
- Control dependency matrix–If $DM = CDM$, then there are only control dependency relationship between components.

4 Application Analysis

In this section, we discuss some applications of our dependency analysis, we first show how to answer the four questions mentioned in section 1, and then discuss some other applictions such as how to manage changes, how to order components etc.

4.1 Possible Solutions to The Four Questions

Question 1: Which other components are also required when one component is to be reused in another system ? Let $S = (Com, D)$ be a CBS, where $Com = \{C, C_1, C_2, \cdots, C_{n-1}\}$, $D = \{dd, cd, td,\ sd, rd, id, ed, io\}$. For any component C in a CBS $S = (Com,\ D)$ all other components in the system can be classified into two groups: *Related-Group* and *Irrelated-Group*. *Irrelated-Group* denotes the set of all components which have not any dependency relationship with component C; *Related-Group* denotes the set of all components which have dependency relationship with component C. *Related-Group* can be furthermore divided into two groups: *Depend-Group* and *Depended-by-Group*. *Depend-Group* denotes the set of all components which depend on component C; *Depended-by-Group* denotes the set of all components which are depended by component C.

In old environment, components belong to *Depended-by-Group* may affect the behavior and structure of component C. When one component moves from the old environment to a new environment, on the one hand, it must meet all dependency relationships in new CBS, on the other hand, in order to perform the functionality related to new system, it must be considered all components in set *Depended-by-Group*.

Question 2: What other components might be affected when one component is changed ? When a component C changes, some of the components which depend on C will be affected, i.e., *Depend-Group*. The steps for finding *Depend-Group* using dependency matrix is as follows:

1. Create dependency matrix for the CBS;
2. Find the corresponding column of dependency matrix to component C;
3. Find all non-zero elements;
4. Determine all components that correspond to these non-zero elements;
5. The set of all these components is *Depend-Group* of component C, marked as *Depend-Group(C)*.

Question 3: What is the minimal set of test cases that must be rerun when one component is changed ? Based on the CDG of a CBS, we can define some dependency-based test coverage criteria for the system . For example,

- *all-data-dependency-paths*: Test cases selected must coverage all data dependency paths on the CDG.
- *all-control-dependency-paths*: Test cases selected must coverage all control dependency paths on the CDG.
- *all-temporal-dependency-paths*: Test cases selected must coverage all temporal dependency paths on the CDG.
- *all-state-dependency-paths*: Test cases selected must coverage all state dependency paths on the CDG.
- *all-casual-dependency-paths*: Test cases selected must coverage all casual dependency paths on the CDG.
- *all-context-dependency-paths*: Test cases selected must coverage all context dependency paths on the CDG.
- *all-input/output-dependency-paths*: Test cases selected must coverage all input/output dependency paths on the CDG.
- *all-interface-dependency-paths*: Test cases selected must coverage all interface dependency paths on the CDG.

When a component C changes, all dependency paths between C and any components in *Depend-Group(C)* or *Depended-By-Group(C)* need be retested. Let *MinTestSet(C)* be the minimal set of required test cases, *TestCase1(C)* be the set of all test cases used to test the old dependencies between component C and any components in *Depend-Group(C)* or *Depended-By-Group(C)*, and *TestCase2(C)* be the set of all test cases used to test new dependencies between

any other components which depend on C or depended by C. The minimal set of test cases that must be rerun can be computed using the following equation:

$$MinTestSet(C) = Testcase1(C) \cup Testcase2(C) \tag{3}$$

Question 4: What is the minimal set of components of the system that must be inspected when a failure of the system occurs ? When a failure of a CBS occurs, we first determine the component (position) where the failure occurs, and then determine all the components on which the component depends. We assume that the failure occurred in component C, all the components which C depends on form the set *Depended-By-Group(C)*. The steps for finding *Depended-By-Group(C)* using dependency matrix as follows:

1. Create dependency matrix for the CBS;
2. Find the corresponding row of dependency matrix to component C
3. Find all non-zero elements;
4. Determine all components that correspond to these non-zero elements;
5. The set of all these components is *Depended-By-Group(C)*.

4.2 Other Applications

In this section, we discuss some other applications using our dependency matrix and feature dependency matrix. First we introduce a new operation "[]" transforming "boolean value" to number value, which will be used in this section.

$$[d_{ij}] = \begin{cases} 1 \text{ iff } d_{ij} = 1 \\ 0 \text{ iff } d_{ij} = 0. \end{cases}$$

$$[(d_{ij})_{n\times m}] = ([d_{ij}])_{n\times m}$$

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.

When changing a component C_i (such as removing or modifying), the values of some of $d_{i1}, d_{i2}, \cdots, d_{in}$ in corresponding dependency matrix will likely be changed. So if a value is changed, its correspond component will be affected by the change of component C_i. So we can judge which components will be affected if one component has been changed based on the row of dependency matrix; we can judge which components affect one component based on the column of dependency matrix, i.e.,

$$Affected - by - C_k = (d_{k1}, d_{k2}, \cdots, d_{kn})_{1\times n} \tag{4}$$

Which represents all components with d_{ij} ($d_{ij} \neq 0$ and $j = 1, 2, \cdots, n$) form the set that will be affected by component C_i when C_i changes.

$$Affect - C_k = (d_{1k}, d_{2k}, \cdots, d_{nk})^T_{n \times 1} \tag{5}$$

Which represents all components with d_{ij} ($d_{ij} \neq 0$ and $i = 1, 2, \cdots, n$) form the set that will affect component C_i when their corresponding components change.

Ordering Components. In analyzing a CBS, we need to know, for example, which components are more important than others? and which components are isolated? i.e., which components are principal? To answer these questions, we can use *dependency coefficient* (DC). The dependency coefficient of a component C_k can be defined as follows:

$$DC(C_k) = \sum_{j=1}^{n} [d_{kj}] + \sum_{j=1}^{n} [d_{jk}] - 2[d_{kk}] = \sum_{j=1, j \neq k}^{n} ([d_{kj}] + [d_{jk}]) - 2[d_{kk}] \tag{6}$$

dependency coefficient represents the degree of one component depends on the rest part (or components) of the system. The bigger the value of $DC(C_k)$ is, the stronger the dependencies between C_k and the rest part of the system. If $DC(C_k) = 0$, then C_k is an isolated component in the system considering all dependencies mentioned in this paper.

Constructing Feature Dependency Chains Based on Feature Dependency Matrix Feature dependency chains trace special dependency relationships in a CBS, and aid one to understand the system in detail, and are useful to structural testing. For example, data dependency chain traces the data dependencies, control dependency chain traces control dependencies in the system, etc., which are useful to test a CBS. We can construct feature dependency chains: *shortest feature dependency chain* based on feature dependency matrix using Floyd's or Dijkstra's shortest path algorithm, shortest feature dependency chain can be used to further analyze CBSs.

The Comparison Between Two Versions A CBS may be updated to meet new requirements. There are many ways to update a system, for example, to redevelop a new system to replace the old one (this case is irrelated to our topic), to introduce new components to the system, or to only enhance the functionalities of some components in the system. Here we consider how to compare the differences between two versions of a system with the following two cases:

- *Case 1*: Updating the old components, don not augment the number of components in the CBS;
- *Case 2*: Introducing new components to old CBS. Assume that we introduce m new components to old system.

Let $S = (Com, D)$ be a CBS, $S' = (Com', D')$ is the modified version of S, and DM' be the dependency matrix of S'.

Case 1: The ranks of matrix DM and matrix DM' are the same. Let $DM(S, S') = [DM - DM'] = ([d_{ij}] - [d'_{ij}])_{n \times n}$. We can get some interesting results:

- $[d_{ii}] - [d'_{ii}] = 0$, $i = 1, 2, \cdots, n$ i.e., main diagonal elements are all equal to zero;
- If $[d_{ij}] - [d'_{ij}] = 0$, $i \neq j$, the dependency relationships between component C_i and component C_j are not changed;
- If $[d_{ij}] - [d'_{ij}] \neq 0$, $i \neq j$, the dependency relationships between component C_i and component C_j are changed. Furthermore, $[d_{ij}] - [d'_{ij}] > 0$ means some dependency relationship between component C_i and component C_j disappear in new version system; $[d_{ij}] - [d'_{ij}] < 0$ means some new dependency relationship are added between component C_i and component C_j in new version system.

So, we can determine the difference of dependencies of two versions of a system by analyzing the dependency matrix.

Case 2: The ranks of matrix DM and matrix DM' are different, the rank of matrix DM is n, the rank of matrix DM' is $n + m$, in order to apply above method in case 1, we can enlarge matrix DM with m rows and m columns element zero. i.e.:

$$DM = \begin{bmatrix} dd_{11} & cd_{12} & \cdots & dd_{1n} & 0 \cdots 0 \\ dd_{12} & dd_{22} & \cdots & dd_{1n} & 0 \cdots 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ dd_{n1} & dd_{n2} & \cdots & dd_{nn} & 0 \cdots 0 \\ 0 & 0 & \cdots & 0 & 0 \cdots 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 0 & 0 \cdots 0 \end{bmatrix}_{n+m \times n+m}$$

5 Related Works

Stafford and Wolf [6] developed an architecture-level dependency analysis technique, called *chaining*. Chains represent dependency relationships in an architectural specification. The individual chain links within a chain associate elements of an architecture that are directly related, while a chain of dependencies includes associations among architectural elements that indirectly related. Chaining is the process of applying dependency algorithms to architectural descriptions in order to create these sets of related components and/or elements. There are three types of chains: affected-by, affects, and related. An affected-by relationship indicates that the element of interest could be affected by the elements linked together in the chain. An affects chain contains the set of elements on which the element of interest could potentially have an effect. Related chains are the union of affected-by and affects chains. Different chains can be used to solve different questions mentioned in introduction section.

Vieira and Richardson [9,10,11] used component-based dependency model (CBDM) to manage dependencies in CBSs. The CBDM is a graph (a set of graphs) that represents the "special associations" among the system's components based on their services. Those associations are distict as relationships that could lead to dependency among components service. They define these "special associations" based on pomsets description of each component.

Zhao [12] has begun work in the area of dependency analysis of formally described software architecture. According to his method, the software architecture should first be formalized in Architecture Description Language (ADL), such as Wright, ACME, RAIPE etc., then analyze some dependency relationships based on the formal specification of the software architecture. In contrast to traditional dependency analysis, This *architectural dependency analysis* technique is designed to operate on an architectural description of a software system rather than the source code of a conventional program.

M. Larsson [7] has mentioned representing dependencies using adjacency matrix, but they did not explain how to analyze and manage dependencies in CBSs using matrix analysis.

Our approach is different from the approaches mentioned above in three aspects:

- Using the CCDG and adjacency matrix to analyze direct dependency relationship between components, and derive indirect dependency relationship by transitive closure algorithm over adjacency matrix.
- Using component dependency graph to model total CBS, and using dependency matrix to manage various dependency relationships and furthermore to manage changes in a CBS. Therefore, based on dependency matrix and matrix operation, we can answer some interesting questions in understanding and maintaining of CBSs.
- dependency matrix is the composition of different feature dependency matrix. Using feature dependency matrix, we can trace and manage various dependency relationships in a CBS.

6 Concluding Remarks

In this paper, we proposed a matrix-based approach to analyzing and managing dependencies in CBSs. To make it possible, we first identified some types of dependencies in a CBS, and then presented a dependency-based representation called the *component dependency graph* (CDG) and the *dependency matrix* (DM) to explicitly represent these dependencies in a CBS. Based on the CDG and DM, we built a mathematical basis for managing and analyzing dependencies in a CBS. Finally, we discussed some important applications of our dependency analysis technique in component-based system understanding, testing, maintenance, and evolution.

As current and future work, we are working on a calculus based on the dependence matrix model, and extending our dependency analysis approach to analyze distributed CBSs and real-time CBSs, as well as other CBSs. In addition,

we would like to develop a dependency analysis tool for CBSs and also to perform some industrial case studies for our approach.

Acknowledgement

The authors thank Dr. Jianjun Zhao and Prof. Conradi Reidar for their useful comments and suggests on the original version.

References

1. A. Brown, K. Wallnau. The Current State of Component-Based Software Engineering. *IEEE Software*, September (1998)37-47.
2. J. Cheesman, J. Daniels. UML Component: A Simple Process for Specifying Component-Based Software. *Addison Wesley* ISBN 0-201-70851-5, (2001)45-100.
3. J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependency Graph and Its Use in Optimization. *ACM Transaction on Programming Language and System*, Vol.9, No.3, (1987)319-349.
4. S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependency Graphs. *ACM Transaction on Programming Languages and Systems*, 22(1), January (1990)26-60.
5. G. Pour. Component-Based Software Development Approach: New Opportunities and Challenges. *Proceedings of Technology and Object-Oriented Language*, TOOLS'26.,(1998)375-383.
6. J. A. Stafford and A. L. Wolf. Architecture-Level Dependency Analysis for Software Systems. *International Journal of Software Engineering and Knowledge Engineering*, Vol. 11, No. 4, August (2001)431-453.
7. M. Larsson, Applying Configuration Management Techniques to Component-Based System, MRTC Report, IT Licentiate thesis, 2000-07, Uppsala University.
8. M. Vieira, et al. Describing Dependnecy at Component Access Point *Proc. of Workshop on Component-Based Software Engineering (at ICSE 2001)*, Toronto, Canada, May 2001.
9. M. Vieira, D. J. Richardson: Classifying and Dealing with dependencies in Large Component-Based Systems, accepted for publication at *the Fifteenth International Conference on Software & Systems Engineering & their Applications (ICSSEA 2002)*, Paris, December 2002
10. M. Vieira and D. J. Richardson. Analyzing Dependencies in Large Component-Based Systems. *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, September 23-27, 2002, Edinburgh, UK.
11. Vieira, M., Richardson, D. J. : The Role of Dependencies in Component-Based Systems Testing and Evolution, In: Proceedings of the IWPSE-02, 24th International Conference on Software Engineering (ICSEN02, Orlando, USA), May,(2002)62-68.
12. J. Zhao. Using Dependency Analysis to Support Software Architecture Understanding. in Li, M.(Ed.), *New Technologies on Computer Software*, International Academic Publishers, September (1997)135-142.