Journal of Computer Science 3 (7): 499-505, 2007
ISSN 1549-3636

Dependencies Management in Dynamically Updateable Component-Based Systems

[1]Saleh Alhazbi and [2]Aman Jantan
Computer Science Department, Qatar University, Doha, Qatar
School Of Computer Science, Universiti Sains Malaysia, 11800, Penang, Malaysia

Abstract: This paper discusses dependencies analysis significance when updating component-based system dynamically. It presents a service-based matrix model and nested graph as approaches to capture components' dependencies; it discusses using dependencies analysis for safe dynamic updating in component-based software systems; we advocate using service-based dependencies rather than component-based which refelect accurate effect during dynamic reconfiguration.

Key Words: Component-based software, dependencies analysis, dynamic updating.

INTRODUCTION

Component-based software systems are those built by assembling pre-exiting components, which provides high flexibility and reusability. The major work with component-based development (CBD) is component integrating rather than writing code and developing everything from scratch. In conventional software development, the concept of complexity is related to the difficulty to analyze source code, modify, and maintain its modules. However, this concept is different in CB systems because the maintenance and reconfiguration only involves replacing, adding, and deleting components rather than source code changes. Therefore, in CB systems, the complexity resides in the dependencies among components, which is captured by the system architecture [1]. In this paper, we discuss managing components' dependencies in our framework (Dynamic Protocol-based Component-based Software–DPICS) [2], which supports building software systems by wiring software components. In DPICS, the functionality of the system is accomplished through protocol-based interaction between components routed by soft bus. DPICS aims to support updating the system during runtime. Traditionally, software modifications require shutting down the system, update the system, and restarting it. This approach is not suitable for critical systems that require 24/ 7/365 availability, such as banking or telecommunications systems, or systems that are critical-mission systems such as air-traffic controllers. Therefore, such systems require dynamic updating which means modifying the system at run-time without service interruption. In component-based software systems, dynamic updating includes adding, removing, and replacing a component on the fly. Updating the system dynamically requires exploring the effects of this modification on the rest of system's components in order not to lead the system to inconsistent state.

Dependency between components can be defined as the reliance of a component on other(s) to support a specific functionality; therefore, we consider dependency as binary relationship between two components: antecedent, and dependent [3] . Antecedent is the free component that has an effect on the dependent one if it is removed or modified, on the other hand, dependent component is the one that related to its antecedents where changes in them might lead dependent to malfunction or fail (see Fig. .1).

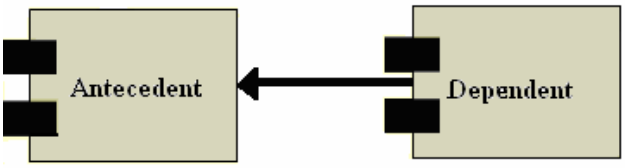


Fig .1: Dependency Relationship

Formally, Larsson and Crnkovic [4] define a relation → called "depend on", where
$C_i \rightarrow C_j$ means that component C_i is the dependent and it requires correct operation of C_j (the antecedent) in order to function correctly. For a component-based system that has a set of components ***S,*** the set of all dependencies is defined as

Corresponding Author: Saleh Alhazbi, Computer Science Department, Qatar University, Doha, Qatar

$D=\{(Ci, Cj): Ci, Cj \in S \wedge Ci \rightarrow Cj\}$

According to this, the current configuration is set of all component and their dependencies

$Con=(S,D)$

Requirements for Dependency Analysis: Dependences analysis is fundamental task for understanding, maintaining, and updating software systems [5,6]. Traditionally, dependence analysis was based on investigating the source program to find dependencies such control and data flow relationships among program variables and functions in order to optimize compilation process [7]. In component-based system, dependency management is essential part of system configuration [6,8]. Moreover, updating system at runtime lacks the test phase when developing software which makes such updating more risky, thus analyzing dependencies between components is necessary in order to safely keep the system running continuously and not crash the system. In this section, we discuss the significance of analyzing the dependencies when dynamically updating the system.

When adding a new Component: Before the new component can be added to the system, it is needed to understand its relationships with other components and its roles as dependent and antecedent. As dependent component, components that would provide services to this new one should be recognized and checked if they are already among systems' components or needed to be loaded. As antecedent, the added component will offer new services to others components; this might require creating new dependencies or might require adding or replacing other components that could be dependents on this one. More specifically, when adding a new component, dependency analysis should answer the following questions

Q1) If there are components in the system need also to be updated in order to benefit of the services provided by this new one (antecedent role), what is the order of updating those components safely ?

Q2) What are the new dependencies (direct and indirect) if this new component will depend on pre-existed ones (dependent role)?

Formally, we can define the configuration of the system after adding a new component safely as

$Con'=(S',D')$

The difference between original configuration *Con* and the new one *Con'* is the new components and new dependencies which can be defined foramll as following

The new component $C_{new} \in S_d$ and $S_d = S \cap \overline{S'}$

The new dependency is the set

$D_{new} = \{(C_{new}, C): C_{new} \rightarrow C\} \cup \{(C, C_{new}): C \rightarrow C_{new}\}$

When deleting an existing components: Before deleting a component from the system, dependencies management is necessary to understand the effect of removing that component. Removing a component might not only have effect on its direct dependents but might affect others transitively, which requires tracing these dependencies from a component to other. Such management of dependency is important for system safety as removing a required component might lead the system to crash which is not accepted with continuously running systems. When removing a component from the system, dependency analysis should answer the following questions:

Q3) What are the components in the system that will get affected by removing this component directly or transitively?

Q4) What is the order of updating the dependents on removed one ?

Formally, the deleted component

$\overline{C_{removed}} \in S_d$ and $S_d = S \cap S'$

$D_{removed}=\{(C, C_{removed}): C \rightarrow C_{removed}\}$

When replacing a component: Dependency analysis is required when replacing a component with a new version in order to evaluate the effect of this modification and take the proper action. The action depends on the relation between old component and new version. Regarding the effect on its dependents, replacing a component with a new version can be categorized into two types:

1. Implementation updating: In this case, the new version has the same interface as old one. Therefore, it has no effect on its dependents as it still provides the same services with same interfaces.
2. Interface updating: in this type, the new version has different interfaces comparing to old one's.

This includes adding, deleting, or/and modifying an interface(s). Adding new service while continuing provide old ones would not affect other old component. But in order to benefit from the extra services provided by the new version, either other components required to be updated or another new component(s) might be added to use them. Modifying and missing services in the new version will affect components depend on those services, thus dependencies analysis should answer the following questions:

Q5) What are the components in the system that will get affected by replacing this component directly or transitively?

Q6) If this replacement requires updating other components, what is the order of those updates?

Formally, modifying a component can be viewed as series of deleting and adding new component. so generally $C_{modified} \in S$ as the set of components doesn't changed

$D_{modified} = D_{removed} \cup D_{new}$

Dependency Representation: Managing and analyzing dependency efficiently requires a good modeling to represent the dependencies among the components. This representation should offer answers for the questions above when updating the system. Commonly, direct graph and adjacency matrix is used to represent the dependencies between components [9, 8,10].

The Component Direct Dependency Graph*(CDDG) =(S,D)* is a direct graph where *S* is a finite nonempty set vertices represent system's components, and D is set of edges between two vertices such that $(a,b) \in D$ means $a \rightarrow b$, and $D \subseteq (S \times S)$

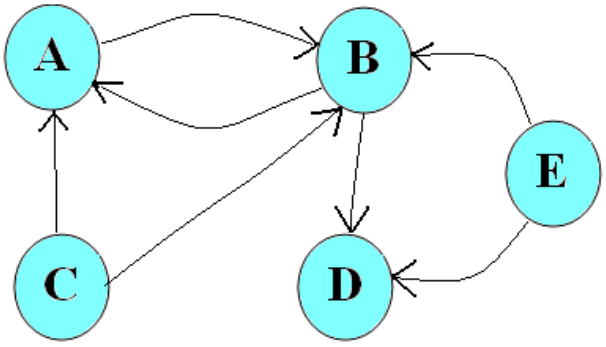


Fig. 2: Component Direct Dependency Graph

Fig. 2 describes the direct dependency where

D={(A,B), (B,A), (B, D), (C, D), (C, B), (E, B), (E,D)}

To represent components' dependencies using adjacency matrix, a matrix $M_{n \times n}$ is used, where each component is represented by a column and a row. If Component C_i depends on C_j then $MD_{i,j}=1$, and in general.

According to this the previous dependency described in Fig. 3 can be represented using adjacent matrix as depicted in Fig. 3

	A	B	C	D	E
A	0	1	0	0	0
B	1	0	0	1	0
C	1	1	0	0	0
D	0	0	0	0	0
E	0	1	0	1	0

Fig. 3: Adjacent Matrix representation for direct component dependencies

Obviously, *CDDG* and adjacent matrix above only describe direct dependency between components. On the other hand, updating a component can affect others transitively, for example in Fig. 3.2, *A* depends on *B*, and *B* on its turn depends on *D*, thus updating *D* might affect *B* and consequently might affect *A*. In order to derive indirect decencies, a transitive closure is calculated to produce component dependency graph *(CDG)*, Fig. 3. which has the same components, it includes direct and indirect dependencies.

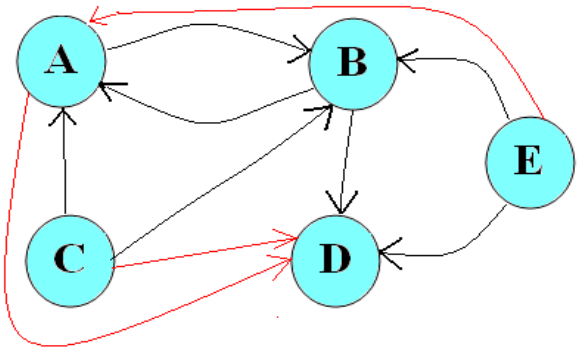


Fig. 4: Component Dependency Graph

In Fig. 4, when calculating transitive closure, self dependency is excluded as the component is the module

of updating and our concern here is the inter-components dependencies.

Correspondingly, indirect dependency can be represented in a matrix by calculating the transitivity using Warshall's algorithm showed in Fig. 5. The algorithm uses the matrix represents direct dependencies $MD_{n\,x\,n}$ to produce the matrix $MA_{n\,x\,n}$

```
For 1⇐i⇐n do
for 1⇐r⇐n do
if MD[r,i]=1 then
    for 1⇐k⇐n do
        if k<>r then
            MA[r,k] := MD[r,k] or MD[i,k]
```

Fig.5: Warshall's algorithm to calculate the transitive closure

Fig. 6 shows the matirxi *MA* which represents direct and indirect component dependencies.

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	0	1	0
C	1	1	0	1	0
D	0	0	0	0	0
E	1	1	0	1	0

Fig. 6: Adjacent Matrix direct and indirect component dependencies

Service Level of Dependencies: Normally, when a component depends on another it relies on some but not all of its services [11]. According to this, during dynamic updating, modifying an antecedent component not necessary to result in inconsistencies with its dependents. For example, in Fig. 7, C1 depends on C3 where its service S11 requires S31 in order to accomplish its functionality. C2 depends also on C3 where its service S21 requires S32 from C3.

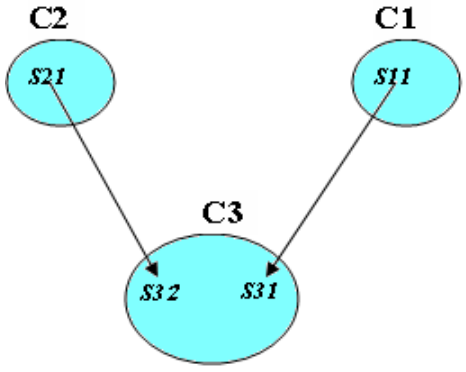


Fig. 7: Service Level Dependencies

Considering only component level dependency, If C_3 got updated; both C_2 and C_3 are considered to be affected, which might not be completely true. Assume that service S_{31} in the new version of C_3 has no changes comparing to that in old version, and S_{32} has changed, then only component C_2 will be affected with this replacement. Therefore, component level of dependency is not enough to trace effects of component updating. On the other hand, service level of dependency will help understand more detail about the consequence of component modification.

Moreover, service dependency can be used to discover all true direct and indirect components dependencies. For example, in Fig. 3.8 service S11 in component C_1 depends on service S21 in Component C_2, and C2 depends on C3 where C2 has a service, S22, which depends on service S31 in C3. Taking into account only component level of dependency, C_1 would depend on C3 indirectly, but with more details through service dependency, C_1 does not depend on C3.

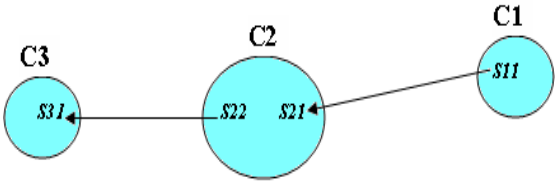


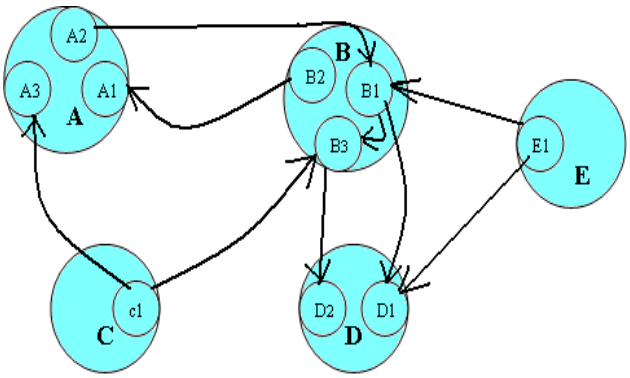
Fig. 8: Service Level Indirect Dependency

But what if service S21 depends on S22 (intra-component dependencies) in Fig. 8 ? Likewise what if service S31 depends on S32 in Fig. 7. As a result of that, with service dependencies, intra-component dependencies (dependencies between component's services) play a rule when calculating components dependencies.

Service Level Dependencies Representation: Using graph and adjacent matrix are sufficient to model dependencies in component-based system as component level, but that is not enough to trace component dependencies accurately. Hence, instead of using simple graph to represent component dependencies, nested graph is used to model dependencies at service level, which gives more details of components relationships.

The Service Level Dependency Graph *(SLDG)=(C,S,A)* is a nested graph where C is a finite nonempty set vertices represent system's components, S is a finite

nonempty set of inner vertices represent component's services, and A is set of edges between two vertices(inner vertices) such that $(S_i, S_j) \in D$ means $S_i \rightarrow S_j$, where $S_i, S_j \in (Ci \cup Cj)$ and $D \subseteq (S \; X \; S)$.



F

Fig. 9: Service Level Dependency Graph

Fig. 9 is another description of the example presented in Fig. .2. To compute the transitive closure, an adjacency matrix is required to capture such graph. Similarly, with component dependencies, two dimensional matrix $S_{m \; x \; m}$ is used to represent services dependencies, where m is the number of all services in all components. Likewise matrix in Fig. .3, $S_{x,y}=1$ if service X depends on service Y. Fig. 3.10 depicts a matrix that represent services dependencies described in Fig. 9,

	A1	A2	A3	B1	B2	B3	C1	D1	D2	E1
A1	0	0	0	0	0	0	0	0	0	0
A2	0	0	0	1	0	0	0	0	0	0
A3	0	0	0	0	0	0	0	0	0	0
B1	0	0	0	0	0	1	0	1	0	0
B2	1	0	0	0	0	0	0	0	0	0
B3	0	0	0	0	0	0	0	0	0	0
C1	0	0	1	0	0	1	0	0	0	0
D1	0	0	0	0	0	0	0	0	0	0
D2	0	0	0	0	0	0	0	0	0	0
E1	0	0	0	1	0	0	0	1	0	0

Fig. 10: Adjacent Matrix representation for direct service-based dependencies

The transitive closure also can be calculated using Warshall's algorithm described in Fig. 5. Fig. 11 depicts the matrix resulted of computing transitive closure, which represents the direct and indirect service dependencies.

	A1	A2	A3	B1	B2	B3	C1	D1	D2	E1
A1	0	0	0	1	0	1	0	1	1	0
A2	0	0	0	1	0	0	0	0	0	0
A3	0	0	0	0	0	0	0	0	0	0
B1	0	0	0	0	0	1	0	1	1	0
B2	1	0	0	0	0	0	0	0	0	0
B3	0	0	0	0	0	0	0	0	0	0
C1	0	0	1	0	0	1	0	0	1	0
D1	0	0	0	0	0	0	0	0	0	0
D2	0	0	0	0	0	0	0	0	0	0
E1	0	0	0	1	0	1	0	1	1	0

Fig.11: Adjacent Matrix representation for direct and indirect service-based dependencies

Now from the matrix in Fig. 11, we can map the service back to its components so we can have clear picture about real direct dependencies between components, for example, from Fig. 3.11, we can find that services belongs to component E has neither direct nor indirect dependencies with services in component A, so updating A will have no effect on E, which is against Fig. 6 indication.

Applying Dependencies Analysis during Dynamic Updating: When a component is updated dynamically, its dependencies with other components in the system should be checked in order to keep the system running without fail. Adjacent matrix representation of service-based dependencies is a good computational approach to answer the questions above raised when adding, removing, or modifying a component.

When adding a new component: Adding a new component to the system has no effect on existing components' dependencies but this requires replacing some of old components in order to use the new one. To answer question 1, regarding the order of components updating , first the new component should be added first then starting update the components that will benefit of this new one (its dependents) [12]. Replacing those components requires analysis dependencies related to component replacement which discussed in 4.3. The adjacent matrix will be modified in order to reflect the changes in dependencies structure, new rows and columns are added to represent the direct dependencies added between the new component's services and other components. Also, using Warshall's algorithm, the matrix will be changed when computing new indirect dependencies added with the new component (question 2).

When removing an existing component: Removing a component while system is running might lead the system to inconsistent state and result in crashing the system. In order to find all affect components by removing component *C*, we search for non-zero elements in columns corresponding to its services in Fig. 11. The no-zero elements indicate direct or indirect service dependencies, therefore, the components of those services are dependants on *C*. Consequently, those components will be affected when removing *C* (Question 3).

In order to keep the system running safely when removing a component, the dependents of that components need to be updated before removing the component. The goal of updating its dependents is either to delete those services were depending on services of deleted one or to modify them so they mask the changes. For example in Fig. 9, if component D would be removed from the system, then according to matrix in Fig. 11, either component B replaced and modified services B1,B3 so they can mask this update and not depend on D1,D2 anymore, or remove those service from B which in its turn requires updating its dependents before that. Note in Fig. 9, we have circular dependency. In this case, both components should be updated together[12] (question 4).

When replacing an existing component: The effect of replacing a component dynamically depends on the relation between new version of the component and old one. If the new component still provides the same services as the old version, then no dependents will be affected by this updating.

If there are services removed in the new version, then from adjacent matrix in Fig. 11, in the columns represent those service, non-zero elements indicates dependent services which means their components will be affected (question 5).

If the component has extra service comparing to old version, then this new component should be replaced first, then updating its direct depends in order to use its new services, and this also might require again updating the dependents in the second level, this tracing can be found from direct matrix in Fig. 10 (question 6).

If the component has some services missing, this case is like the one when removing a component, either to update its dependents in some level to mask such modification or to delete those depended services from all components (updating) starting from outer level (components that have no dependents) toward the components.(question 6).

Many research tackled dependencies analysis in component-based systems from different aspects. In our work[1], dependencies analysis was used to measure the complexity of system's architecture which indicates the effort needed to maintain the system. Li in [7], used adjacent matrix model to capture components' dependencies and applied to system maintenance, testing, and evolution. Li used matrix-based model only for component level of dependencies which-as we discussed above- not describe the dependencies accurately. In [12], the authors focused on type safety when updating a class dynamically and investigated different cases when updating two depending components, their focus was mostly on the direct dependencies.

CONCLUSION

Updating component-based system dynamically requires analyzing dependencies between components in order to inspect affected components and take the proper action so the system continues running consistently. Service-based matrix representation is an appropriate model to capture components' dependencies; computationally, this matrix can be used to analysis dependencies when a component is added, removed, or replaced and according to that, other components might require adaptation in a specific order.

REFERENCE

1. Alhazbi S., 2004. Measuring the complexity of component-based system architecture, in Proc. of the 1st IEEE Intl. Conference on Information and Communication Technologies: From Theory to Applications (ICTTA-04), Damascus, Syria, IEEE Computer Society.
2. Alhazbi S. ,Jantan A. ,2007. A Framework for Dynamic Updating in Component-based Software Systems, Accepted for Conference on Information Technology Research & Application (CITRA) ,Selangor- Malaysia.
3. Hasselmeyer Peer, 2001. Managing Dynamic Service Dependencies,12th International Workshop on Distributed Systems: Operations & Management (DSOM), Nancy, France.
4. Larsson M., and Crnkovic I., 2001. Configuration management for component-based systems. In Proceedins of the Tenth International Workshop on Software Configuration Management, Toronto, Canada.

5. Zhao J,. 1997. Using Dependence Analysis to Support Software Architecture Understanding, New Technologies on Computer Software, pages 135–142.
6. Stafford J.A. and Wolf A.L., 1998. Architecture-Level Dependence Analysis in Support of Software Maintenance. In Proceedings of the Third International Software Architecture Workshop, pages 129–132.
7. Horwitz S., Reps T.,Binkley D., 1990. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems 12, 1, pp: 26-60.
8. Li B., 2003. Managing Dependences in Component-Based Systems Based on Matrix Mode, Net. Objectdays(NODE) conference, Erfurt, Germany.
9. Cui Y. and Nahrstedt K.,2001. Qos-aware dependency management for component-based systems. In International Symposium on High Performance Distributed Computing 2001. San Francisco, CA, August 2001.
10. Larsson M.,2000. Applying Configuration Management Techniques to Component-Based System, MRTC Report, IT Licentiate thesis, Uppsala University
11. Tansalarak N. and Claypool K., 2003. CGC: An Architecture to support Better and Faster Component Evolution In Second International Workshop on Unanticipated Software Evolution, Warsaw, Poland.
12. Murarka Y., Bellur U., Joshi R., 2006. Safety Analysis for Dynamic Update of Object Oriented Programs APSEC-2006, 13th Asia Pacific Software Engineering Conference, Bangalore.