Dependence Analysis of Component Based Software through Assumptions

Ratneshwer[1], Anil Tripathi[2]

[1] **Department of Computer Science(MMV), Banaras Hindu University**
Varanasi, 221005, India

[2] **Department of Computer Engineering, Institute of Technology,**
Banaras Hindu University
Varanasi, 221005, India

Abstract

This study presents a quantitative approach for dependency analysis of Component Based Software (CBS) systems. Various types of dependency, in a C BS, have been observed through 'assumptions' and based on these observations some derived dependency relationships are proposed. The proposed dependency relationships are validated theoretically and an example illustration has been shown to demonstrate the proposal. The result of the study suggests that these dependency relationships may prove helpful in understanding CBS systems.

Keywords: *Complexity measures*, *Software Component, Component Based Software, Dependency Analysis*

1. Introduction

In this paper, an approach has been given to analyze the dependence problem in software components through a set of 'assumptions' (that a software component may have with respect to other software components). Software developers during their day to day work are constantly making assumptions about the interpretation of requirements, design decisions, the operational domain, the environment and the characteristics of input data [1].These software assumptions can formally be analyzed and documented and can be utilized in dependency management. The gist is that by analyzing the set of assumptions among software components in a quantitative manner, dependence relationships among software components can be estimated. Component dependence analysis is a useful technique that has many applications in software engineering activities including software understanding, testing, debugging, maintenance, and evolution [2]. The dependence problem is intensified because [3] CBS can encompass both components developed in-house and those made available by a third party (e.g., COTS), normally deployed as a "black-box" and often with deficient documentation.

A software component can be defined as an independent executable unit that performs certain functionality when get plugged into an application software system. One of the earliest definitions of software component is given by Greedy Booch [4]: "A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction". Later, Clement Szyperski presented his well known definition of a software component at the 1996 European Conference on Object Oriented Programming [5]: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party."

Many research approaches tackled dependence problem in CBS from different aspects. Most of them are based on graph based approach i.e. to draw a graph among software components based on their dependency relationships and analyze dependencies based on graph properties [6, 7, 8]. These approaches give idea of direct dependency relationships but fail to describe the types and complexity of the dependency relationships at software component level. Li [2] has nicely categorized the dependencies among software components in eight types and represented the dependency relationships as dependency graph and dependency matrix form. He has considered the dependency relationships due to edge complexity but do not cover effect on dependency due to node complexity (software components in case of CBS) which is also an important factor. He also suggested the eight types of dependency matrixes which make analysis tough. A weighted dependency graph approach has also been proposed with additional parameters 'Dependency Strength' and 'Dependency Criticality' [9] but these computations also do not cover component's internal complexity. One approach is formalization of dependency information in a C BS i.e. to describe dependency relationships in some formal language [10, 11]. This approach covers mathematical aspect and do not include programming aspect of CBS. One approach is to represent the component dependency relationships in form of regular expression POMSET [3] and made a d ependency graph CBDM based on these information. These above approaches mainly cover the dependency edge relationships among software components but do not cover the effect on dependency due to implicit and explicit properties of software components. These limitations will become evident in big CBS systems. Merely stating that a component is dependent on a nother component is not sufficient. The type of that dependency, possible effects of that dependency failure and critical factors of that dependency also need to be explored. Dependency need to be represented in some quantifiable manner as its possible effects can be visualized effectively.

In this study, an attempt has been made to correlate assumptions with dependency analysis in a CBS. It is our conjecture that if a software component has more number of assumptions regarding its functionality and behavior, its dependency on other components will be more complex in nature. The likely benefit of this approach may be useful in earlier identification and removal of design and implementation level weaknesses and system can be made more maintainable. We have taken four types of dependency, (in a CBS) i.e. data dependency, control dependency, interface dependency and real time dependency. In the present work only software assumptions are considered.

The paper is organized as follows. In section 2, we briefly mention some existing works related to the dependency analysis in a CBS system. In section 3 a correlation between dependency and 'assumptions' has been demonstrated. In section 4 and following subsections, the proposal of '*describing dependencies among software components through assumptions*' has been explained and some dependency relationships have been derived. In section 5, we evaluate the derived relationships by a mathematical framework proposed by Briand et al. In section 6, an example illustration of the dependency relationships have been shown. In section 7, we discuss some suppositions of dependency of CBS systems. Finally we conclude in section 8.

2. Related Work

In Literature, dependence problem have been studied widely in the context of CBS systems. Substantial work has been reported regarding dependency analysis of CBS systems. Some significant works related to the topic are as follows. Li [2] has described eight types of dependency in Component-based software (CBS) and given a matrix model to analyze the dependencies in a CBS. Vieira and Richardson [3] discussed an approach for describing dependencies of an individual component by using a declarative XML description. Kon and Campbell [12] have given a method to analyze dependencies by prerequisite specifications of software components. Bondrev et. al. [13] observed the influence of input-parameter dependency on the CBS system behavior and performance. Guo [14] has addressed the interconnection dependency problem among software components in a CBS by using category theory. The software industry later discovered various techniques that aim at identification of undocumented functional and behavioral mismatches under the name of assumptions, policy, operational profiles, check lists etc. Analysis through 'assumption' is one of the approaches in this direction. Some works observable in the software engineering literature related to 'assumption' based analysis are reported here. The idea of assumptions management came out of an Independent Research and Development project sponsored by the Software Engineering Institute (SEI) in 2002-2003 in the area of sustainment [15, 16].Lewis and Mahatham [17] developed a prototype that demonstrates the application of assumptions management, including the recording and extraction of assumptions from Java source code into a repository, and the Web-based management of these assumptions. Tirumala et al [18] have considered mismatched assumptions between software components are a prime source of failures in CBS systems. In their work, they introduced a framework to explicitly expose assumptions in software components, and automatically verify these assumptions during system integration. Steingruebl and Peterson [19] argued that Undocumented assumptions are often the cause of serious software system failure. Thus, to reduce such failures, developers must become better at discovering and documenting their assumptions. Steingruebl and Peterson have mentioned the common categories of assumptions in software, discuss methods for recognizing when developers are making them, and recommend techniques for documenting them, which offers value in and of it-self. In the present work, 'assumptions' have been used to analyze dependence among software components. The above contributions demonstrate that although various approaches to analyzing the dependencies are available in the literature but an appropriate *'Quantified Dependency Estimation Model'* especially for a Component-based software system has not yet been found. This serious concern raised by practitioners and researchers turn easily into variety of research issues still to be tackled and properly addressed. This paper extends the above contributions further by suggesting an approach to estimate the dependencies in a quantifiable manner.

3. Dependency versus Assumptions

Every time a d ecision is made- about how to design an interface, how to implement an algorithm, if and how to encapsulate an external dependency- assumptions are made concerning how the software will be used, how it will evolve, and what environment it will operate in [17]. A good simplifying assumption simplifies the design problem significantly without changing the essential character of the program which needs to be implemented [20]. Developers don’t always recognize that they’re even making assumptions, so we must focus on devising techniques that focus on areas where assumptions can occur and assisting developers so that they can methodically examine them. Undocumented assumptions are often the cause of serious software system failure. Thus, to reduce such failures, developers must become better at discovering and documenting their assumptions [19]. These assumptions can be recorded and reviewed in order to get information regarding incompatibilities due to assumption mismatches. Software components depend on each other by service providing/ receiving relationships. If a component ‘X’ is providing some services to component ‘Y’, then ‘X’ will have some assumptions about ‘Y’ and ‘Y’ may also have some assumptions about ‘X’ in terms of structure, behaviour and functionality. If there is some service providing/receiving activity between ‘X’ and ‘Y’, then ‘Y’ has to fulfil all the assumptions of ‘X’. Here, we correlate these software assumptions with dependency measure. If a component has more number of assumptions regarding its use, its degree of dependency (on other components) will be more. If the total number of assumptions for a co mponent can be computed, this information can be used to categorize software components based on their dependency measure.

A CBS can be represented by a set of n components such as:
$C = (C_1, C_2, C_3$-------------------- $C_N)$
We define two subsets of C, X and Y. The kth edge from a vertex c(i) of X to a v ertex c(j) of Y represents a relationship, R: X$\rightarrow$Y, such that the ith component of X ***is providing services to*** the jth component of Y i.e. x_i R y_j meaning thereby x_i of X is providing some services to y_j of Y. We can say that X is a s et of service providing components and Y is a set of service receiving components. A component may provide services to more than one component. In this case, there will be edges between c(i) of X to C(j1), c(i) to c(j2), ... of Y. The dependency information among software components may be obtained from component's meta-data.

The assumption exposed by a software component will form assumption set. The assumption set of a component ‘P’ for a component ‘Q’ consists of a set of assumptions exposed by ‘P’ that needs to be fulfilled by ‘Q’. Let A_i is the set of assumptions for a component C_i. We consider here four types of dependency relationships: data dependency, control dependency, interface dependency and real time dependency.

The set A_{Di} is a s et of data transfer related assumptions where each element of this set is an assumption made by the component regarding its data transfer.
A_{Di} = {ad | ad is a data transfer related assumption}

Similarly, A_{Ci}, A_{Ii} and A_{RTDi} will be the set of assumptions regarding control transfer, interface and real time systems. A_i is the union of all four types of assumptions.

$$Ai = (A_{Di})\ \ U\ (A_{Ci})\ U\ (A_{Ii})\ U\ (A_{RTi})$$

$n(A_i)$ is the total number of assumptions for a component.

Two software components, providing same functionality, can be compared based on their $n(A_i)$ values and the component having lesser $n(A_i)$ value may be chosen. A component may not have to deal with all types of assumptions. A component may have to deal with only specific assumptions that may occur during that type of dependency. Assumptions may be of two types. The one that can be directly measured and quantified and the other one that can not be directly measured but also play an important role in that type of dependency.

4 An Approach towards ‘Dependency Estimation’ Through Assumptions

In this section, we identify assumptions underlying among software components, as we perceive them. Component’s have opposing communication styles, data representation, protocols, synchronization paradigms or processing expectation [21]. In spite of listing all possible assumptions, we concentrate on source of assumptions. We try to find out some factors for different types of dependency that requires special consideration (more assumptions) in terms of its use and behavior. By counting these factors (along with their possible assumptions) one can get idea about the dependence complexity. The required input data can be obtained from the design description of the CBS and interface description of software components.

In the following subsections, different types of dependence analysis have been described.

4.1 Data Dependence

Data assumptions capture what is expected of input or output data. Another use of a data assumption is to capture

the way of data internal processing and about the nature of data [17]. Data assumptions capture what data the program expects to input and output, including that data's format and type and who is checking it for correction [18].
The possible data transfer related assumptions are:

- The size and types of I/O parameters (of methods) of a component,
- The size of meta data,
- The reference variables in a component,
- The shared variables in a component,
- The sequence of execution of data,
- The hardware interfaces and their capacities, and

A data dependence exists when one component provides a value subsequently used by another component either directly or indirectly. During data transfer, assumptions of a component may conflict with the assumptions of another component that leads to failure of service and poor performance of the component. We made an attempt to quantify some of the factors regarding data transfer assumptions that may affect the performance of the component. Some assumptions that could not be quantified also need to be taken care of.
We propose a r elationship Weight of Data Dependence (W_{DDi}) that will give an estimate regarding data related assumptions.
The following factors may influence $W_{DD.}$

(A1) Number of Input Parameters
Number of Input Parameters, of a component, will be the sum of all its methods' parameters. The higher number of such assumptions decreases understandability and modifiability.
If there are 'n' parameters in a method 'M1' and there are 'm' methods in a component then

$W_{DDi} \ \alpha \ \sum_{i=1}^{m} \sum_{j=1}^{n} Nij$

Where, Nij is the jth parameter of the ith method.
Weight of Data Dependence will be proportional to the Total Size of input parameters (in a component).

(B1) Number of Reference Variable
If a component has many reference (pointer type) variables then data that passes through the component might be misunderstood because the other component may not have any idea regarding the data structure. Such variable requires more number of assumptions. Weight of Data Dependence will be proportional to the number of reference variables (in a component).

W_{DDi} α Number of Reference Variables

(C1) Number of shared variables
If components are using some shared data then modification of such data may affect those components. Shared data may become a single point of failure. One has to integrate explicit management of shared data. As the number of shared variables, in a component, increases, the number of data assumptions will also increase.

W_{DDi} α Number of Shared Variables

(IV) Number of Conditions
If a component has many pre-conditions and post-conditions concerning the use of any data then it is tough to understand and modify such data because one has to check conditions every time when the component is going to get plugged into. As the Number of conditions with a component's data increases, the number of data assumptions will also increase.

W_{DDi} α Number of Conditions

The above said factors in terms of their effects on W_{DDi} can be summarized as follows:

W_{DDi} = Number of Input Parameters + Number of Reference Variables + N umber of Shared variables + Number of Conditions

A software organization that engages in development of a CBS using software components may consider the values of these factors for their normalization to work out W_{DDi} in a quantifiable form. The W_{DDi} is data dependence contributed by ith component. The total data dependence weight contributed by all the components in a software system can be expressed as the following:

$DD_{CBS} = \sum_{i=1}^{c} WDDi$

W_{DDi} is the weight contributed by the ith out of 'c' components in a CBS.

4.2 Control Dependence

Control assumptions, for example, capture the expected control flow, including function call ordering and initialization requirements [18]. A software designer or architect can evaluate control assumptions to make sure they are consistent with the application flow [17]. A component C1 is control dependent on component C2 if C2 invokes C1. A control dependence [22] from Component X to component Y means that C2 must be verified if C1 changes. In a CBS, Software components may raise a control for variety of reasons.

- In response to a change in the component's data
- The completion of a long running process in a CBS
- An interruption in service of a component
- Components that represent user interface elements usually raise controls in response to user actions like a button click or menu selection
- When a time of a process expires
- When a counter exceeds its value

- When software or hardware failure occurs
- To notify about an event

In a CBS, during control transfer the two important things are: to receive the control and to handle the control. During a control transfer, assumption mismatches lead to failure of control transfer between the components.

Some possible control related assumptions are:

- The control transfer mechanism,
- The life time of the control,
- Order of the execution of the control
- Effect of the control,
- The number of exceptions with control, and
- The number of conditions with control.

A component organizes its activities with causing of events by it and responses that it furnishes in response to events caused by other components. An interface of a component defines events to send control messages to other components. A component may (or may not) receive responses of an event caused by other components. So, a component has a set of native events (that it causes) and a set of external responses (that it gets from other components). Control dependence, of a component, mainly depends on the native events, external responses and their interactions. Native events may be of many types. For example, a component may send control to another component (1-1 mechanism), a component may send control to a group of components (1-m mechanism), many components may send control to a component (m-1 mechanism) or many components may send control to many other components (m-m mechanism), and hence, the control dependence is related to the types of these native events. We define, Weight of Control Dependence (W_{CDi}) that will indicate an estimation of control assumptions. These factors, as discussed below, are being considered to express the Weight of Control Dependence of a component.

(A2) Events Fan out

For a native event, in a component, one has to define the event class definition, event parameters, and event name and corresponding event exceptions (if any). As the Number of Native Events, in a component, increases the control assumptions of the component will increase. For a native event, we define "Event Fan-out" (Number of component(s) receiving control messages by the event). For example, In case of "1-1 event mechanism" the value of *Event Fan-out (EFO) will be '1'* and in case of "1-m or m-m event mechanism" the value of *Event Fan-out* will be 'm'.

If there are 'n' native events in a component then,

Total Event Fan-out = $\sum_{i=1}^{n} EFOi$

Where, EFO_i is the Event Fan-out of i^{th} event.

Weight of Control Dependence (for a component) will be proportional to the Total Event Fan Out of the component.

W_{CDi} α Total Event Fan-out

(B2) Responses Fan in

A component may get responses caused by other components. As the Number of External Responses increases, the control assumptions will increase. We define "Responses Fan-in (RFI)" (number of responses an event receives from other components). If there are 'n' native events in a component then,

Total Response Fan In = $\sum_{j=1}^{n} RFIj$

Where, RFIj is Responses Fan-in of the j^{th} event.

Weight of Control Dependence (for a component) will be proportional to the Total Response Fan-in of the component.

W_{CDi} α Total Response Fan-in

(C2) Control Communication Weight

A component can send a control message in two ways; either synchronously or asynchronously. Asynchronous method calls [23] use multi-threading so one must be aware of potential problems concerning thread concurrency, state corruption, re-entrance etc. One can count the number of shared variables, the number of states and the number of re-entry points in a thread. As these values increase, the control assumptions will increase.

If there are 't' threads in a component then

Control Communication Weight= $\sum_{i=1}^{t} \mathrm{A(i) + B(i) + C(i)}$

Where A, B and C are number of shared variables, number of states and number of re-entry points respectively (in a thread).

W_{CDi} α Control Communication Weight

(D2) Number of RPCs

Control dependence counts on the range of native events i. e. whether the control will be sent to local component(s) or remote component(s). If a component sends control to a remote component then some *remote procedure calls* will be needed that would increase the control assumptions contributed by this component. One can measure the number of RPCs (Remote procedure call) in a component that can be counted from the internal code of a component. Weight of Control Dependence, of a component, will be proportional to Number of RPCs.

W_{CDi} α Number of RPCs

(E2) Number of Exceptions

Another problem with a control transfer is that of exceptions. When an exception is raised, execution stops and a corresponding error handler are searched among the handlers. Any un-handled exception [23] rose by the subscriber will be propagated to the publisher. Some subscribers may encounter an exception in their handling of the control, not handle it, and cause the publisher to crash. More number of exceptions with an event would attract higher control assumptions.

W_{CDi} α Number of Exceptions

Weight of Control Dependence (for a component) may be expressed as follows:

W_{CDi} = Events Fan Out + Responses Fan In + Control Communication Weight + Number of RPC + Number of Exceptions

A software organization that engages in development of a CBS using software components may consider the values of these factors for their normalization to work out W_{CDi} in a quantifiable form. The W_{CDi} is the weight of control dependence contributed by a component. Suppose in a CBS there are 'c' components then control dependence in a CBS

$CD_{CBS} = \sum_{i=0}^{c} WCDi$

W_{CDi} is the weight contributed by the i^{th} out of 'c' components in a CBS.

4.3 Interface Dependence

Li [2] has described that the interface - event dependence is the main dependence form in CBSs. In practice [24], many failures in a C BS arise because of interface violations among components- where one party breaks the contract. Any interface violation results in disturbances in these interdependencies and consequently breaking of contracts among them. As many interface dependencies, and the interdependence complexities, would be there that many possibilities of their violations and breakage of contracts would be possible.

If a component [25] has multiple access points, each of which represents a different service offered by the component, then the component is expected to have multiple interfaces. If one substitutes a co mponent with another component (having more than one interface), then one has to substitute all its interfaces and one has to take care that replaced interfaces are providing identical services as earlier interfaces.

The possible interface related assumptions are as follows.

- Interface signature matching
- Semantic properties matching
- Hidden interfaces (may be in some cases) in a component
- Multiple versions of an interface
- Multiple interfaces of a component
- Visibility of interfaces
- Wrapper code (if needed)
- Business case of components
- Publishing the properties of an interface

One can consider the total interface assumptions in a CBS in terms of interface dependencies contributed by the individual components. We made an attempt to quantify some of the assumptions regarding interface(s) of a component and some assumptions that could not be quantified also need to be taken care of. These factors, as discussed below, are being considered to estimate the Weight of Interface Dependence (W_{ID}) of a component in a CBS.

(A3) Number of Interfaces in a Component

More Number of Interfaces, per component, would attract more interface dependence because failure of any interface functionality may affect the functionality of the component and more effort would be needed to understand and modify the component. As the Number of Interfaces per Component increases, the interface assumptions will increase.

W_{IDi} α Number of Interfaces in a Component

(B3) Number of Hidden Interfaces

Another worrisome problem facing a CBS is the issue of "hidden interfaces". "Hidden interfaces" are typically channels through which application or component software is able to induce an operating system to execute undesirable tasks or to launch undesirable processes. [26]. A component can be used by a s oftware system, a hardware, another component or network. There are interfaces defined for all these. In spite of that, there are some possibilities of 'hidden interfaces', through which a component can be accessed. 'Hidden interfaces' may be helpful to make components integrable but it m ight be possible that a u ser can access the component through 'hidden interfaces' and can modify the component's attributes and consequently its state. As the Number of Hidden Interfaces, in a component, increases the possibility of failure of component's functionality will increase and it will contribute the complexity that makes understanding and modification difficult.

W_{IDi} α Number of Hidden Interfaces

(C3) Number of Ambiguous Statements in an Interface Specification

Poorly documented interfaces may create ambiguity in understanding them. Ambiguities get created when a statement, in an interface specification, has more than one interpretation. Ambiguity may be derived as follows.

Un-ambiguity = A / B

Where A is the Total number of statements and B is the Total number of possible interpretations of all statements.

Ambiguity = 1- Un-ambiguity

More number of ambiguous statements would make the understandability of a component poor.

W_{IDi} α Number of ambiguous statements in an interface specification

Weight of Interface dependence of a co mponent can be expressed as follows.

W_{IDi}= Number of Interfaces + N umber of Hidden Interfaces + Number of Ambiguous Statements

A software organization that engages in development of a CBS using software components may consider the values of these factors for their normalization to work out W_{IDi} in a quantifiable form. The W_{IDi} is Interface dependence contributed by a component. Suppose in a CBS there are 'c' components, and then Interface dependence in a CBS

$ID_{CBS}=\sum_{i=0}^{c} WIDi$

W_{IDi} is the weight contributed by the ith out of 'c' components in a CBS.

4.4 Real Time Dependence

Real time systems [27] are computer systems in which the correctness of a s ystem depends not only on the logical correctness of the computations performed but also on time factors. Real time system requirements impose some extra constraints for Component-based development like execution time, memory consumption etc. Worst case execution time [28] can be estimated using information about the code that was generated by the compiler.

The possible real time related assumptions are:

- Timing analysis at component level,
- Timing analysis at CBS level,
- Worst case execution time of a component,
- Memory consumption by a component,
- Dependence relations of a c omponent in a CBS,
- Hardware platform of the CBS,
- Bounded communication time between remote components,
- Priority of components,
- Deadline of components,
- Concurrency and synchronization issues in a CBS,
- Composite components in a CBS, and
- Resource uses by a component.

The total real time constraints in a CBS have been considered as accumulative sum of constraints contributed by individual components. We made an attempt to quantify some of the assumptions regarding real time services and some assumptions that could not be quantified also need to be taken care of. Weight of Real time Dependence (W_{RTD}) contributed by a component can be estimated as follows.

(A4) Number of Real Time Constraints

The interface of a co mponent needs some additional constraints (synchronization calls, scheduling, communication calls, timing and memory constraints etc) to fulfil real time requirements. These constraints help components to get composed and function efficiently in a real time CBS. But the other side these constraints may increase the assumptions required to understand and modify the components.

W_{RTDi} α Number of Real Time Constraints

(B4) Number of Non Periodic Events

In CBS, there are two types of events. One is Periodic events, for which the execution time and other properties can be estimated earlier. The other one is non-periodic events that generate due to responses of events. Non-periodic events cannot be estimated earlier. Components [26] may have different time characteristics in different platforms. They can only be predicted earlier. These non-periodic events, in a real time CBS, may affect the execution time guarantee and memory consumption property. If a component has many non-periodic events then its performance would be unpredictable in real time systems and hence understandability and modifiability of such a component would be hard.

W_{RTDi} α Number of Non Periodic Events

(C4) Number of Resources a Component Uses

Resources that can be used [28] by a real time application are usually scarce. Available processor time and memory are limited due to hardware costs. Thus a co mponent can easily influence others simply by consuming too many resources. Resources from different operating systems also make the problem worse. Many resources consumed by a component would attract poor understandability and modifiability.

W_{RTDi} α Number of Resources consumed

Weight of Real Time Dependence (of a component) can be estimated by sum up all the above outcomes.

W_{RTDi} = Number of Real Time Constraints + Number of Non-periodic events+ Number of Resources a Component use

A software organization that engages in development of a CBS using software components may consider the values of these factors for their normalization to work out W_{RTDi} in a q uantifiable form. The W_{RTDi} is Real Time Dependence contributed by a component. Suppose in a CBS there are 'c' components, then Real Time Dependence in a CBS

$RTD_{CBS} = \sum_{i=1}^{c} WRTDi$

W_{RTDi} is the weight contributed by the ith out of 'c' components in a CBS.

The proposed dependency relationships are summarized as follows:

S. No.	Name of Reelationship	Derived Relationship
1	Weight of Data Dependency	W_{DDi} = Number of Input Parameters + Number of Reference Variables + Number of Shared variables + Number of Conditions
2	Weight of Control Dependency	W_{CDi} = Events Fan Out + Responses Fan In + Control Communication Weight + Number of RPC + Number of Exceptions
3	Weight of Interface Dependency	W_{IDi}= Number of Interfaces + Number of Hidden Interfaces + Number of Ambiguous Statements
4	Weight of Real Time Dependency	W_{RTDi} = Number of Real Time Constraints + Number of Non-periodic events+ Number of Resources a Component use

Table 1: Proposed Dependency Relationships

5 Validation of Derived Dependency Relationships by Briand et al Framework

Briand et al [29] have proposed an axiomatic framework for evaluating complexity measures. Their properties have been widely applied to software engineering practices and have been thoroughly discussed in literature [30, 31, 32]. The five criteria proposed in the framework evaluate software metric properties using a formal theoretical basis. The properties are intended to validate complexity measures as a system property. Complexity and dependency both are system properties and based on inter-module relationships. Since there is a strong similarity between complexity measures and dependency measures, so we choose this framework to validate the proposed dependency relationships (given in Table 1).

We have taken some assumptions here for applying the Briand's framework for a CBS. We use following notations: here a CBS system 'S' will be represented as a pair <C,R>, where C represents the set of components and R is set of dependency relation on E ($R \leq C x C$). The Briand's axioms are applied to derived dependency relationships in the following paragraphs.

P1: **Non-negativity**

The dependency of a system S = < C, R > is non-negative.

- If data transfer takes place between two components, the number of input parameters can not be zero. Rest of the values may be zero but will not be negative. So, W_{DD} will always be positive value i.e. non-negative value.
- If control transfer takes place between two components, the events fan out/responses fan in value can not be zero. Rest of the values may be zero but will not be negative. So, W_{CD} will always be positive value i.e. non-negative value.
- In a similar analogy, one can say that W_{ID} and W_{RTD} will also be non-negative value.

P2: **Null Value**

The dependency of a system S = < C, R > is null if R is empty.

- If R is empty means there are no data transfer take place between any two components and they will be treated as independent (in terms of data dependency), so W_{DD} will be null.
- In a similar analogy, one can say that W_{CD}, W_{ID} and W_{RTD} will be null if R is empty.

P3: **Symmetry**

The dependency of a system S = < C, R > does not depend on the convention chosen to represent the relationships between its components. S = < C, R > and S^{-1} = < C, R^{-1} > => d ependency (S) =dependency (S^{-1}). Dependency should not be sensitive to representation conventions with respect to the direction of arcs representing system relationships.

- Data dependency between two components, say C1 and C2, is sensitive to the direction because a data receiving component is more dependence prone as compare to data providing component. So, this property does not hold for data dependency.
- In a similar analogy, one can say that this property does not hold for W_{CD}, W_{ID} and W_{RTD} also.

P4: Module Monotonicity

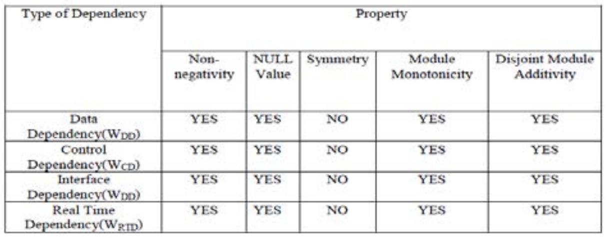
The dependency of a system S= <C, R> is no less than the sum of the dependencies of any two components with no relationship in common.
S = <C, R> and m1= <C_1, R_1> and m2=<C_2, R_2> and (m1 U m2) ≤ C & $R_1 \cap R_2$=ǿ =>
Dependency (S) ≥ dependency (C1) + dependency (C2)

- If two components say C1 and C2 do not participate in data transfer activities i.e. they are not related to each other in terms of data dependency, then W_{DD} will be zero. But some indirect dependencies (relationships) may exist between two components. So, data dependency of the component based system will always greater than the sum of the dependence of any two of its components with no relationship in common.
- In a similar analogy, one can say that this property holds for W_{CD}, W_{ID} and W_{RTD} also.

P5: Disjoint Module Additivity
The dependency of a system S = <C, R> composed of two disjoint modules m1, m2 is equal to the sum of the complexities of the two modules.
S =<C , R> & S=C_1 U C_2 and $C_1 \cap C_2$=ǿ => dependency (S) = dependency (C_1) + dependency (C_2)

- If two components are put together in the same CBS, but they are not providing/ receiving any services to/from each other then they will be treated as two disjoint components in the CBS system and no additional dependency are generated from the internals of one component to the internals of the other. This will be true for all types of dependency. So, W_{DD}, W_{CD}, W_{ID} and W_{RTD} will hold this property.

We summarize our findings:

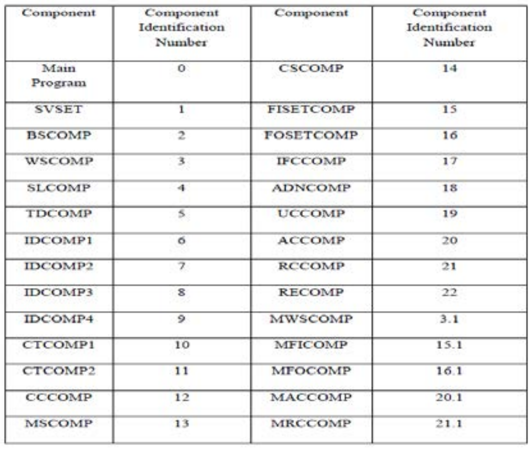


Type of Dependency	Property				
	Non-negativity	NULL Value	Symmetry	Module Monotonicity	Disjoint Module Additivity
Data Dependency(W_{DD})	YES	YES	NO	YES	YES
Control Dependency(W_{CD})	YES	YES	NO	YES	YES
Interface Dependency(W_{DD})	YES	YES	NO	YES	YES
Real Time Dependency(W_{RTD})	YES	YES	NO	YES	YES

We summarize the results in the table, which shows that all the proposed dependency relationships satisfy property 1, 2, 4 and 5 but they fail to satisfy symmetry property.

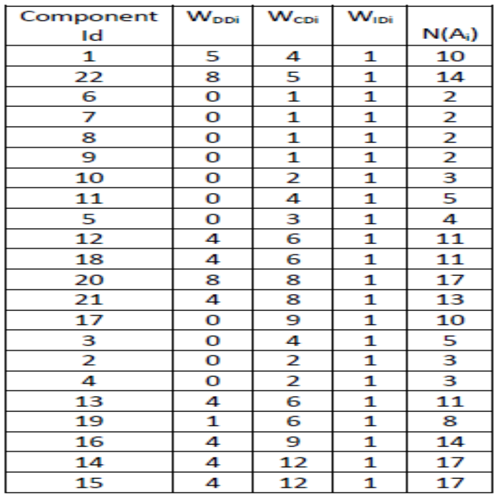
6. Example Illustration of Dependency Relationships

The objective of this example illustration is to obtain quantitative characteristics of these dependency relationships and understand the ways in which these dependencies can be managed/ minimized. We develop some components and a CBS in which the components are providing/receiving the services. The one main problem that we have encountered during the work is lack of some good experimental data from real time environment that may help to verify the above suggested metrics. Thus, we made a co mponent-based software environment 'CIG Information Extraction Tool (CIGIET)' for the proposed experiment [description of the tool is given in the annexure 1]. The tool developed for the purpose takes CIG attributes as an input and give various information, complexity measures, as output. This 'CIGIET' does not covers real time aspects so we did not included real time dependency in the example illustration. Here we have included W_{DD} , W_{CD} and W_{ID} only.
We assign a C omponent Id to each component for ease and main program has been assigned an Id '0'. 3.1, 15.1 etc are updated version of corresponding components.



Component	Component Identification Number	Component	Component Identification Number
Main Program	0	CSCOMP	14
SVSET	1	FISETCOMP	15
BSCOMP	2	FOSETCOMP	16
WSCOMP	3	IFCCOMP	17
SLCOMP	4	ADNCOMP	18
TDCOMP	5	UCCOMP	19
IDCOMP1	6	ACCOMP	20
IDCOMP2	7	RCCOMP	21
IDCOMP3	8	RECOMP	22
IDCOMP4	9	MWSCOMP	3.1
CTCOMP1	10	MFICOMP	15.1
CTCOMP2	11	MFOCOMP	16.1
CCCOMP	12	MACCOMP	20.1
MSCOMP	13	MRCCOMP	21.1

Table 4: Component List with their IDs



Component Id	W_{DDi}	W_{CDi}	W_{IDi}	$N(A_i)$
1	5	4	1	10
22	8	5	1	14
6	0	1	1	2
7	0	1	1	2
8	0	1	1	2
9	0	1	1	2
10	0	2	1	3
11	0	4	1	5
5	0	3	1	4
12	4	6	1	11
18	4	6	1	11
20	8	8	1	17
21	4	8	1	13
17	0	9	1	10
3	0	4	1	5
2	0	2	1	3
4	0	2	1	3
13	4	6	1	11
19	1	6	1	8
16	4	9	1	14
14	4	12	1	17
15	4	12	1	17

Table 5: Outcomes of W_{DD}, W_{CD} , W_{ID} and n(Ai)

One can consider the basic guiding principles for designing a CBS based on understandings regarding the derived dependency relationships that make the software system and the overall complexity of the structure of the given CBS. One may choose a design that has less interdependency edges among components.

7. Discussion

Assumptions may vary in different software environments. Number of assumptions may be an important measure to prioritize different types of dependencies. A survey shown in [18] indicates that algorithmic defects in software occur less frequently than the defects that are related to integration issue. In real time systems, integration defects are caused by assumption mismatches between software components and environmental assumptions which may be invalid. Several catastrophic failures in large scale real time systems can be attributed to the inadequacy of existing interfaces and the inability to track implicit assumptions of components [18]. When a control assumption mismatch occurs, software components have integration conflicts that prohibit them from communicating properly in the system. Hence, control assumptions should given priority.

The dependency would be considered "good" if it is there for extending its services to other components. Similarly the dependency would be considered bad if it appears because of the fact that a component requires help of some other components to construct services provided by it. It is assumed that by reducing the dependencies of a component-based system one can make it more maintainable. Reduced complexity will result in ease in understanding and modification. It is possible to have multiple design blueprints of a CBS with varying presence of dependencies. One or more of these (blue-prints) designs may have minimum values of dependence compared to others, and hence, smaller requirement of the effort required for understanding and modifications for the purpose of maintenance. Here, it is observable that the dependencies can be reduced retaining its full functionality. It is therefore concluded that the designs of a CBS must strive to propose a design and refine and revise it for reducing the complexity of the software and its full functionality to be able to ensure better maintainability.

8. Conclusion

This work explores the concepts related to the various dependencies in the context of CBSE. It proposes inclusion of assumptions in various dependencies in a quantifiable manner. Some meaningful conclusions have been drawn conceptually. It further shares possibilities of quantification of these dependencies in terms of factors that have been identified herein. We understand that the suggested method of quantification of dependencies can be helpful in working out suitable metrics in this context. The suggested quantifiable dependencies can purposefully indicate the maintenance effort required. This initial proposition of such a model may be purposefully employed by the professionals and the corresponding useful feedback may be analyzed. It calls for further extensive research oriented studies, by all concerned, for perfection of details of the model.

References

[1] D . Parnas, "Information Distribution Aspects of Design Methodology", in proceedings *of 1971 IFIP Congress*. New York, NY.

[2] B. Li, "Managing Dependences in Component Based Systems Based on Matrix Mode," in *Net. Object days (NODE) conference*, 2003, Erfurt, Germany.

[3] M. Vieira and D. J. Richardson, "Analyzing Dependencies in Large Component Based S ystems," in *Proceedings of the 17th IEEE International Conference on 'Automated Software Engineering*, 2002, Edinburgh, UK.

[4] G. B ooch, Software Components with Ada: Structures, Tools, and Subsystems, 3rd Edition. Reading, MA: Addition-Wesley, 1993.

[5] C. Szyperski, Component Software- Beyond Object Oriented Programming, Reading, MA: Addition-Weslay, 1999.

[6] S. Alhazbi and A. Jantan, "Dependencies Management in Dynamically Updateable Component-Based System", *Journal of Computer Science*, 3(7):499-505, 2007.

[7] M. Larsson, "Applying Configuration Management Techniques to Component Based System". MRTC Re-port, *IT Licentiate thesis*, Uppsala University, Sweden, 2007.

[8] Ratneshwer and A K Tripathi, Interdependence Analysis in Component Based Software, Journal of Information Science and Technology", volume 6, issue 2, 2009.

[9] S. Alda, M. Won, and A. B. Cremers, “M anaging Dependencies in Component-Based Distributed Applications”, In: Proceeding of the 2nd F IDJI International Workshop on scientific Engineering of Distributed Java Applications. Luxembourg-City, 2002.

[10] M. Belguidoum and F. Dagnat, “Dependency Management in Software Component Deployment”, *Journal of Electronic Notes in Theoretical Computer Science* (ENTCS), Volume 182, June, 2007.

[11] L. Yu, A. Mishra, S. Ramaswamy, “Component co-evolution and component dependency: speculations and verifications”. IET Software 4(4): 252-267, 2010.

[12] F. Kon. and R.H. Campbell, “Dependence Management in Component-Based Distributed Systems,” *IEEE Concurrency*, Volume 8, Number 1, pp. 26-36, January 2000 .

[13] E., Bondarev, P. D. With, M. Chaudron and J. Muskens, “Modelling of Input-Parameter Dependency for Performance Predictions of Component-based embedded systems,”, in Proceedings of 31 EUROMICRO conference on Software Engineering and Advance Applications, Porto, Purtgal, 2005, Aug 30-sept 3.

[14] J. Guo, “Using Category Theory to Model Software Component Dependencies.”, in proceedings of *9th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems(ECBS 2002)*, Lund University, Lund, SWEDEN, April 8-11, 2002.

[15] F. Bachmann. et. al. 2003. SEI Independent Research and Development Projects (CMU/SEI-2003-TR-019 ADA418398). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. <http://www.sei.cmu.edu/publications/documents/03.reports/03tr019.html>.

[16] Seacord, R. (2003). Assumption Management.” news@sei interactive 6, 1,First Quarter 2003.<http://interactive.sei.cmu.edu/news@sei./columns/the_cots_spot/2003/1q03/cots-spot 1q03.htm>.

[17] Lewis, G. A.,Mahatham,T and Wrage, L. (2004). Assumptions Management in Software Development, Technical Note CMU/SEI-2004-TN-021, Carnegie Mellon University, 2004.

[18] A. Tirumala, et al. “Prevention of failures due to assumptions made by software components in real-time systems”, SIGBED Review - Special issue: The second workshop on high performance, fault adaptive, large scale embedded real-time systems (FALSE-II), Volume 2 Issue 3, 2005, pp. 36-39.

[19] Steingruebl, A and Peterson, G. (2009). Software Assumptions Lead to Preventable Errors, *Journal of IEEE Security and Privacy* , Volume 7 I ssue 4, July 2009.

[20] C . Rich, C. and R. C. Waters, The Disciplined Use of Simplifying Assumptions. ACM SIGSOFT Software Engineering Notes- Special Issue on R apid Prototyping, Volume 7, Issue 5, December 1982.

[21] H apner et al., “Patterns of Conflict among Software Components.”, Volume 79, 2006, pp. 537-551.

[22] R. Leitch & E. Stroulia, Accessing the Maintainability Benefits of Design Restructuring Using Dependency Analysis. In Proceedings of Ninth International Software Metrics Symposium, Sydney, Australia, 2003, 3-5 Sept 2003.

[23] J. Lowy, Programming .Net Components. O'Reilly Media, Inc. Sebastopol, CA. pp. 75. Chapter in an edited book, 2005.

[24] Edwards, S. H., Shakir, G., Sitaraman, M., Weide, B. W. and Hollingsworth, J. (1998). A Framework for Detecting Interface Violations in Component Based Software. In Proceedings of the 5th International Conference on S oftware Reuse, Victoria, British Coulmbia, Canada.

[25] Heineinnan, G. T. and Councill, W, T. (2001). Definition of Software Component and its Elements”. Component Based Software Engineering- Putting the Pieces Together. In G. T. Heineinnan and W. T. Councill (Ed.), Addison Weslay, Bostan, MA, pp. 9-10.

[26] Voas, J. (2001). Predicting System Trustworthiness. In Building Reliable Component Based Systems. I. Crnkovic, and M. Larsson, (eds), A rtech House, Inc. Norwod, MA, USA.

[27] Norstorm, C. and I sovic, D. (2001). Components in Real Time Systems. In *Building Reliable Component Based Systems*, I. Crnkovic and M. Larsson (Ed)., Artech House, Inc. Norwod, MA, USA, 2001.

[28] Rastofer, U and Bellosa, F. (2001). An Approach to Component Based Software Engineering for Distributed Embedded Real Time Systems. IEE Software. 148(3), pp. 99-103.

[29] L. C. Briand, and S. Morasca, Property based Software Engineering Measurement. *IEEE Transactions of Software Engineering*, 22(1), pp. 68-86.

[30] L. C. Briand, J. W. Daly, and J. Wüst, “A Unified Framework for Cohesion Measurement in Object-OrientedSystems”, *Empirical Software. Engineering*. 3, 1 (Jul. 1998), 65-117.

[31] B. Xu, Z. Chen, Z., and J. Zhao, “Measuring cohesion of packages in Ada95”, In Proceedings of the 2003 Annual ACM Sigada international Conference on A da: the Engineering of Correct and Reliable Software For Real-Time & Distributed Systems Using Ada and Related Technologies (San Diego, CA, USA, December 07 - 11, 2003).

[32] F. Dandashi, “A method for assessing the reusability of object-oriented code using a v alidated set of automated measurements”, In Proceedings of the 2002 ACM Symposium on Applied Computing (Madrid, Spain, March 11 - 14, 2002). SAC '02. ACM, New York, NY, 997-1003.

Ratneshwer is working as an Assistant Professor in Department of Computer Science (MMV), Banaras Hindu University, Varanasi (India). He has done his Ph.D. in Component Based Software Engineering at Department of Computer Engineering, Institute of Technology, Banaras Hindu University, Varanasi (India). He is currently working on software process, interdependence and composability aspect of component based software engineering. He has eight research papers in international journals and 1 1 research papers in international/national conference proceedings in his credit.

A K Tripathi is serving as Professor at Computer Engineering Department, Institute of Technology, Banaras Hindu University, India. He has 23 years of teaching and research experience. His research areas are parallel/ distributing computing and S oftware Engineering. He has supervised more that 10 doctoral thesis. He has two book chapters, and several research papers in international journals and conference proceedings.