Software Components' Coupling Detection for Software Reusability

Zakarya A. Alzamil
Software Engineering Department
King Saud University
Riyadh, Saudi Arabia

***Abstract*—Most of the software systems design and modeling techniques concentrates on capturing the functional aspects that comprise a system's architecture. Non-functional aspects are rarely considered on most of the software system modeling and design techniques. One of the most important aspects of software component is reusability. Software reusability may be understood by identifying components' dependence, which can be measured by measuring the coupling between system's components. In this paper an approach to detect the coupling between software system's components is introduced for the purpose of identifying software components' reusability that may help in refining the system design. The proposed approach uses a dynamic notion of sequence diagram to understand the dynamic behavior of a software system. The notion of data and control dependence is used to detect the dependences among software components. The components' dependences are identified in which one component contributes to the output computation of the other component. The results of the experiments show that the proposed algorithm can help the software engineers to understand the dependences among the software components and optimize the software system model by eliminating the unnecessary dependences among software components to enhance their cohesiveness. Such detection provides a better understanding of the software system model in terms of its components' dependences and their influence on reusability, in which their elimination may enhance software reusability.**

Keywords—Software component coupling; software component dependence; software component reusability; components interdependence; components dependence testing

I. Introduction

Software components interact with each other by maintaining a duct. Such interactions incorporated via software connectors. Software connectors play different roles in providing interaction among set of components, in which a protocol specification defines its properties such as the types of interfaces it is able to mediate, assurances about interaction properties, rules about interaction ordering, and interaction commitments (e.g., performance). Software connectors facilitate the interaction as communication, coordination, conversion, or facilitation [1]. Components interactions may be incorporated using different types of software connectors such as procedure call, data access, event, stream, linkage, distributor, arbitrator, and adaptor. In addition, software component may be interacted using composite connectors i.e., multi-type connector.

One of the software design basic principles when designing software component's is maximizing the component's cohesiveness and minimizing component coupling. Software coupling was first introduced by Stevens at el. [2]. Coupling is a measure of the interdependence degree between software components [3]. The Components interaction may take different forms in terms of the degree of their interdependence. Coupling can occur in various ways, however, the concentration are on dependencies between components that arise from associations and collaborations. Software components may exhibit different levels of interdependence; Myers [4] has identified the levels of coupling as follows; content coupling, common coupling, external coupling, control coupling, stamp coupling, and data coupling. Ideally, the best case for components' reusability is to have no coupling among software components, however, such case may not be achievable in most of the cases, therefore, such coupling's levels can be used as an interdependence measure between software components. These levels of coupling can be ordered based on their effects on components' understandability, maintainability, and reusability from the worst to the best as follows; content coupling, common coupling, external coupling, control coupling, stamp coupling, and data coupling [5], in which the content coupling is the worst because it represents the high and tight components' coupling; and the data coupling is the best because it represents the low and loose components' coupling. Data coupling occurs when a simple data i.e., a simple argument, is passed between the interconnecting components. Stamp coupling occurs when a data portion of data structure is passed between components, control coupling occurs when a control such as flag is passed between components, external coupling occurs when the two components are tied to an environment or medium that is external to the system such as communicating via I/O device or file, common coupling occurs when the interacting components reference a global data. Content coupling occurs when one component uses or changes the data or control information maintained within the boundary of another component [6]. Additional types of coupling have been introduced; such as tramp coupling [5], scalar data coupling, scalar control coupling, non-local coupling, and global coupling [7].

Low coupling is desirable because less interaction between components reduces the possibility of the affects that may be caused by a failure or change in one component to other connecting components [5]. In addition, low coupling enhances components independence which leads to software understandability as well as software reusability. Software dependence is one of the core factors for the software reusability measurement, in which component's dependency should be measured to understand software component's reusability. Such dependency can be measured by measuring the coupling between system's components. The coupling measures the strength of the relationship between two modules. In the case of object-oriented designs, modules are classes. Since the introduction of this measure, a large number of coupling measures have been proposed [8], which correspond to different types of relationships between classes.

In this paper, an approach to detect the coupling between software system components is introduced for the purpose of identifying software components' interdependence which may contribute to understanding the software components' reusability that may help in refining the system design. The proposed approach uses a dynamic notion of sequence diagram to understand the dynamic behavior of a software system. The notion of data and control dependence is used to detect the dependences among software components. The dependences among two software components are identified such that such dependence of one component influences or contributes to the output computation of the other component. This paper is organized as follow; the following section presents the related work, section III provides some basic definition of the terms used throughout this paper. The proposed algorithm is described in section IV, and section V provides the experimental study of this work. The conclusions and the future works are presented in section VI.

II. The Related Work

Most of the software testing research has concentrated on the implementation phase of the software development life cycle. Software testing at early stages of software system development has been recognized in past few years and many software modeling testing researches has been conducted in the literatures. Software components' coupling has been investigated in the literatures as a metric for different purposes such as complexity [9], modularity [10], maintainability [11, 12], dependencies [8, 13], reusability [14, 15, 16], dependability [17],

Software coupling has been used for structured design to identify the modules' dependence. An early attempt to use module coupling based on the measurement of information flow between system components has been proposed in [9] for evaluating the structure of large-scale systems. Among the proposed metrics, in addition to module coupling, are procedure complexity and module complexity.

A framework of coupling measurement in object-oriented systems has been presented in [8] based on a standard terminology, formalism, and a review of the existing frameworks and measures for coupling measurement in object-oriented systems. A unified framework based on this review was developed in which the existing measures were classified. The proposed framework provides a mechanism for comparing measures and their potential use, integrating existing measures as well as defining new ones, and selecting from existing measures for a specific goal of measurement. It has been reported that most of the coupling measurement in object-oriented systems focuses on components' static dependencies and much less has investigated components' dynamic dependencies.

In [15] static measures of indirect coupling have been proposed to assess the reusability of Java components retrieved from the internet by a search engine. The class coupling has been traditionally described as when a class accesses one or more of another class's variables or invokes at least one of its methods. However, such description ignores inheritance based coupling but a variant includes it. The proposed measures intended to overcome the limitation of the existing static measures to handle indirect coupling such as inheritance. An empirical comparison of the proposed measures has been presented to test such metric.

A new design pattern coupling role and component concepts have been proposed in [13] to solve the challenge of building the appropriate coupling of separated code elements of components, and reducing the build-level dependencies. Roles are related to the functional aspects of a target software program (composition and collaboration of functional units) and components correspond to the physical distribution of code elements with limited build-level dependencies. The proposed coupling is enabled to instantiate a software program using a generic main program to retrieves and composes functionalities at run-time according to a description file.

An information theory has been used to propose coupling measures for modular systems [10]. An abstraction of software system, such as graph, has been used to represent system in which inter-module coupling and intra-module coupling have been proposed to assess or predict the quality of software system. Inter-module coupling measures system level coupling based on relationships between modules, and intra-module coupling is similar, but measures a different subgraph, i.e., measures coupling at subsystem level.

An indirect coupling metric that identify the exact relationship between indirect coupling and maintainability has been presented in [12]. A chain that is expressed in terms of graph vocabulary has been used as a central attribute to detect the indirect coupling. The proposed metrics focus on the reflection of "strength" as it is a fundamental component of coupling which is viewed as the relationship between a given pair of classes as well as on the aggregation of coupling relationships with respect to a single class with the intent of seeing how much influence a given class has over the system.

A dynamic coupling metric has been proposed in [14] to measure the direct coupling of object-oriented software at the object level based on the structural relationships, method call types, and the number of method calls between classes. The proposed metric is designed for embedded systems that are based on component-based or object-oriented systems to produce efficient and reusable component.

A module coupling has been used to propose a spatial impact metric to capture the extent of error propagation in a software system by identifying the location of dependability components called detectors and correctors at early stage of software system development [17]. The proposed metric is based on the hypothesis that modules with high coupling values are most likely potential locations for detectors.

A survey of the components dependencies has been done in [16] in which a classification of such dependencies is introduced based on composition, distribution and platform dependencies to promote component reusability. The authors assessed the contemporary component models for networked embedded systems using Loosely-coupled Component Infrastructure (LooCI), which is a platform-independent component model designed for networked embedded systems. LooCI eliminates composition dependencies at compile time with explicit definitions of interfaces and receptacles. The authors have found that most of the component models in such systems eliminate composition dependencies but not distribution and platform dependencies.

A static analysis tool for measuring the coupling between Java classes has been presented in [11] for the purpose of maintainability, based on source code analysis aiming to identify the types of couplings that are not available until after the implementation is completed. It uses interdependencies between objects to define coupling types, in which it defines four types of coupling; parameter coupling, external/file coupling, inheritance coupling, and global coupling.

An empirical study is presented in [18] that analyzed the coupling among number of open source software projects to identify two types of coupling; logical coupling and structural coupling. This study aims at to determine the interplay between the two types of coupling, the coupling strength between classes, and the level of stability between the coupled classes as stable or unstable. In addition, this study aims to understand the impact of the two types of coupling on each other. Statistical tests have been used to compute the correlation between the strengths of logical and structural dependencies. Although the achieved results cannot be generalized, statistical analysis has shown that interplay occurs between structural and logical dependencies in most of the analyzed software projects.

A component ranking method based on non-dominated sorting for the purpose of software components reuse is presented in [19] in which a specification of the relative importance of non-functional properties is used for a partial ordering. In addition, components' coupling has been used as a measure for the external and internal dependencies between classes; however, such measure is restricted to the entry class of candidates determined by test-driven search evaluation. An explorative study has been applied on a set of components obtained from the Maven Central repository.

A study of coupling measures between software components has been presented in [20] to determine the most significant coupling measure among a set of measures. The authors have categorized the coupling measures in two categories; ratio oriented and ratio less. The analysis of the coupling measures has been conducted by defining two types of class interactions; Operation-Operation interaction which is defined as the interaction between two operations of two or more different objects or classes, and Class-Class interaction which is defined as the interaction between two classes if any one of the above two interaction occurs. A case study has been performed on three industrial software systems.

A measure of the level of coupling for components within a software system has been used in [21] to predict the maintenance efforts for the purpose of evaluating the relationship between system design decisions and the costs of maintenance. The aim of this paper was reduce the cost of redesign a software system by predicting the released value of such redesign, or what has been called architectural debt. The authors have measured system coupling for two software systems; one has a hierarchical design, the other has a core-periphery design, and have shown that, the tightly-coupled components cost more to maintain than loosely-coupled components.

In [22] software metrics have been used to classify the software components into cyclic and non-cyclic for the purpose of understanding the relationship between the dependencies of cyclic components and defect profiles of cyclically dependent components. A static analysis has been used to identify the components' coupling, in which some measures have been used such as coupling between classes and response for class. An empirical study of six object-oriented programs along with some statistical tests has been conducted to investigate the components' cyclic dependencies and their impact on detecting defective components. The study has shown that components with cyclic dependencies are the more defective than non-cyclic components which is similar to the results of related studies.

A multiple dependency metric based on network analysis has been proposed in [23] to investigate the relationship between structural features of classes and their functions within a network system. The metric measures the degree of reusability of a component, as well as its direct and indirect coupling. The measurement of coupling (direct and indirect) may indicate the construction cost of new class. The authors have conducted an empirical study on several open source codes, which has shown that, the used metric is useful in analyzing the complexity, stability, and maintainability of classes. In addition, it has shown that, classes with multiple dependencies have more complicated functions that are less cohesive than other less complicated classes.

The coupling between object classes (CBO), as an object-oriented design metric, has been introduced in [24], as a count of the number of other classes to which a class is coupled with, in which methods of one class use methods or instance variables of another.

A dynamic coupling measure is presented in [25] for change proneness of classes in object-oriented software. The data is collected and analyzed through a dynamic analysis of the code at runtime or from the dynamic design models to collect such measures to identify the objects interaction. Such dynamic measures capture more properties that static measures.

Although the aforementioned related work proposed several techniques for testing and detecting different types of software components' dependences and couplings, the main purpose of detecting software components' dependence is to identify the components' coupling that reduces components' reusability. Most of the presented related works identify software components' coupling without investigating whether such coupling is contributing to the components' computation i.e., component's output. Such is based on the premise that each dependency among components within a software system should influence at least one component's output; otherwise, such dependency is unnecessary and may be eliminated without influencing the software semantic.

This paper introduces an approach that detects the software components' coupling based on dependence that influences or contributes to the components' output computation which may help in understanding components' reusability for the purpose of refining the system design.

III. Basic Definitions

Software components collaborations can be dynamically modeled by a sequence diagram model, such dynamic model of the system is represented by a sequence of messages passed between the components showing the message-sends involved in specific collaborations in order to carry out the system functionality. In UML, sequence diagrams are employed to model the runtime of the software system.

The sequence diagram describes the dynamic behavior of system and can be viewed as a set of sequences of events, referred to as traces, where each event represents an occurrence of a message passed between components. For a given sequence diagram *S*, a trace of the sequence diagram is defined and referred to as T_s. For a finite set *R* of roles and a finite set *M* of messages, a message label is defined as a function *g* that maps each message in *M* to a triple (*l*, *s*, *r*) where *l* denotes a label of the occurred message, *s* and *r* denote roles in *R*, called sender and receiver, respectively [26]. Assume that there are an infinite set *O* and a finite set *L* for participating objects and labels of messages, respectively, an event $e_p \in T_s$ as a triple $g(e_p) = (l_{e_p}, s_{e_p}, r_{e_p})$ is defined, where the *p* is the event number within the trace T_s, $l_{e_p} \in L$ for the label, $s_{e_p} \in O$ for the sender, and $r_{e_p} \in O$ for the receiver. Because a code statement within an event may be executed several times during a trace, an execution position for each executed code statement within an event is defined as Y^i in which *Y* is the code statement number and *i* is the position of an executed code statement within the message's code statements. Y^i is referred as executed code statement or execution position interchangeably.

The event *e* can be represented as a directed graph (*V*, *E*), where *V* is a set of nodes, and *E* is a set of arcs. The nodes represent the objects associated with an event (sender and receiver). The arcs represent the dependence among the participating objects within a given event. Such dependence can be identified as data or control dependence. Every graph has an entry node V_0 and an exit node V_x. The program dependence graph has been proposed in [27] for the purpose of program optimization. Program dependence graph is a control flow graph with nodes corresponding to statements and control predicates, and arcs corresponding to data and control dependencies. It has been widely used for program analysis for different purposes such program testing and program optimization.

The data dependence can be defined in terms of defining or using passed data among participating objects via message passing within an event in a trace T_s. The data passed among objects within an event e_p in a trace T_s via label l_{e_p} are stored in a memory address referred to as variable. A passed variable might be simple data type, data structures, or complex objects. Also, a variable may contain data that is used as a control flag. A use of a variable occurs when such variable is referenced, and a definition of a variable occurs when a value is assigned to it. A variable *v* that is passed via a label l_{e_p} is said to be used at r_{e_p} if such variable is referenced. A variable *v* passed via a label l_{e_p} is said to be defined at r_{e_p} if a value has been assigned to that variable. A variable might be defined (assigned another value) several times and may be at different objects (receivers) within an event in a trace T_s. A label is considered as used if at least one of its passed variables has been referenced, and is considered as defined if at least one of its passed variables has been defined. The last definition *LD*(*v*) of a variable *v* at execution position *m* within a receiver object of an event e_p in a trace T_s is defined as the closest execution position Y^i within the sender or receiver object of the event e_p that contains a definition of *v* such that $i < m$. Another type of dependence comes in the form of returned value of the receiver to the sender such that the sender is dependent on the receiver. Therefore, returned value of passed variable *v* via a label l_{e_p} is defined if a value is returned from the receiver r_{e_p} to the sender s_{e_p}. Such returned value may come in the form of passed value, reference value, shared data structures, or common variable.

The data dependence captures a situation where one object (sender) assigns a value to a variable and the other object (receiver) uses that value. In terms of the directed graph, the sender object assigns a value to a variable before the entry node V_0 of a directed graph, and the receiver object uses that value before the exit node V_x. The control dependence captures the situation when the execution of a statement within an object (receiver) depends on the evaluation of a test statement (i.e., a predicate) of a conditional statement within another object (sender). Originally, the control dependence has been defined in [27]. The proposed definition is modified to fit the control dependence among software components. Formally, let *Q* and *Z* be two code statements within participating sender and receivers objects of an event, respectively, and (*Q*, *X*) be a branch of *Q*. Code statement *Z* postdominates code statement *Q* *iff* *Z* is on every path from *Q* to the exit node V_x of the event. Code statement *Z* postdominates branch (*Q*, *X*) *iff* *Z* is on every path from *Q* to the event's exit node V_x through branch (*Q*, *X*). *Z* is control dependent on *Q* *iff* *Z* postdominates one of the branches of *Q* and *Z* does not postdominate *Q*. As stated earlier, coupling is a measure of the interdependence degree between software components [3], in which coupling is defined between two components if they exhibit data or control dependences.

Due to various reasons, components may exhibit dependence (data or control) that is not contributing to their output. The output of a component may be a regular output statement or return statement. The dependence that is not contributing to the components' output computation is unnecessary or useless dependence, and may occur due to fault in the system model or poor design. Therefore, the notion of influence between components is identified, such that component c_1 influences component c_2 *iff* the data or control dependence contributes to the output computation of component c_2. As described earlier, coupling is a measure of the interdependence degree and the strength of the relationship between software components. Hence, the coupling that is based on the influence among the components is recognized for the purpose of software reusability. However, the other components' dependences that are not contributing to the component's output are considered as unnecessary coupling that should be minimized to improve the software reusability. The proposed approach aims to provide a new type of information that is based on the premise that the components' dependence is computed based on the influence of a component on another component's output.

IV. Components Coupling Detection Algorithm

The proposed algorithm is based on the notion of data and control dependence defined in the earlier section to identify the component coupling. A trace analysis has been used to determine the defined-used chain; in which different sets are defined to store some collected data for later analysis. Therefore, the sequence diagrams of the system model under analysis should be instrumented and recorded in a trace T_s. As described earlier, the sequence diagram trace T_s consists of a series of events that records the message label l_{e_p} along with its passed parameters, the sender s_{e_p} along with its defined variables, and the receiver r_{e_p} along with its used variables. In the trace T_s, the used and defined variables are stored at each executed code statement. The trace T_s of a given system is instrumented based on test cases that may be generated based on random inputs, certain inputs, or based on system operational profile [28], which is a description of how the system is used. It is usually developed during the system engineering or requirements definition phase.

As described earlier, the aim of the proposed approach is to identify the coupling among systems' components for the purpose of understanding components' reusability which may contribute to system design refinement. For that purpose, the proposed algorithm analyzes the system model under analysis in a backward fashion that requires the sequence diagram trace T_s to be recorded and the data are collected and stored in defined sets, in which the analysis starts from the end of the sequence diagram trace T_s and goes backward. Such approach, i.e., backward analysis, is appropriate for analyzing the dynamic model of the system under test in which the model is executed first, and the data is collected and stored in the trace, so that the model is analyzed dynamically based on its actual execution. However, such approach can be used for static analysis of a model, in which the model is analyzed statically, i.e., without its execution, to collect and store data in the trace based on all possible execution traces.

The proposed algorithm, as presented in the subsequent paragraph, requires the trace of the sequence diagram T_s as an input. The algorithm starts by defining two sets for the data and control dependences and identifying the used variables at s_{e_p} and r_{e_p}. Then it sets all executed output and return code statements in the participating objects within each event (r_{e_p} and s_{e_p}). Then the algorithm iterates from the end of the trace T_s and goes in backward fashion to identify the data and control dependence for every event in the trace T_s.

ALGORITHM
INPUT: T_s
OUTPUT: Coupled components

```
1   DEFINE DataDependent() and ControlDependent() as two sets
    of data and control dependences
2   DEFINE USED($O, Y^i$) as the set of used variables at $Y^i$ of a
    given object ∈ $O$.
3   SET all output/return executed code statements within $s_{e_p}$ and $r_{e_p}$
    as marked for each event.
// start from the end of the $T_s$ in backward fashion
4   WHILE (not the beginning of $T_s$)
5     FOR (every $e_p$) DO
6       WHILE (not $V_0$ of $e_p$)
7         FOR (every marked executed code statement $Y^i$∈ $r_{e_p}$)
          DO
8           FOR (every variable v ∈ USED( $r_{e_p}$, $Y^i$)) DO
// returns the executed code statement $X^i$ as the last definition of $v$
9             $X^i$= Find_LD($v$);
10            SET $X^i$as marked executed code statement;
11            IF ($X^i$∈ $s_{e_p}$) THEN
12              ADD ($s_{e_p}$, $r_{e_p}$) to DataDependent();
              END_IF;
            END_FOR;
// returns $Z^i$ that controls $Y^i$ or zero if no control dependence
13          $Z^i$= Find_CD($n$);
14          IF ($Z^i$> zero) THEN
15            SET $Z^i$as marked executed code statement;
16            IF ($Z^i$∈ $s_{e_p}$) THEN
17              ADD ($s_{e_p}$, $r_{e_p}$) to ControlDependent();
              END_IF;
            END_IF;
          END_FOR;
        END_WHILE;
      END_FOR;
    END_WHILE;
18  DISPLAY DataDependent() & ControlDependent() as the
    coupled components.
```

As described earlier, the algorithm starts by defining several sets to be used during the analysis and marking all executed output code statements in the trace T_s within each executed component. The algorithm analysis starts from the end of the trace T_s and goes backward and iterates for every event while not reaching its entry V_0. Then for every used variable at every marked executed code statement, the algorithm finds its last definition, and marks it. If the marked executed code statement is within the sender object, it means that the data dependent among the sender and receiver object contributes to the output computation of the receiver object. As a result, the sender and receiver objects are added to the *DataDependent* set as data dependent objects. Next the algorithm checks whether the marked executed code statement is control dependent on any other code statement, if so, such executed code statement is marked. In addition if this marked executed code statement is within the sender object, the sender and receiver objects are added to the *ControlDependent* set as control dependent objects. The algorithm iterates until it reaches the beginning of the trace T_s, and at the end, it displays the coupled components as the dependent components based on the influence on the output computation. In addition, for the purpose of investigating useless dependences, the algorithm can provide all data and control dependences among components regardless of their contributions to the computations of the components' output.

Although, there are many dependencies that might occur among software components, the proposed algorithm minimizes such information by only identifying the dependencies between components that contribute to the computations of the components' output. The software engineers can use such information to identify the coupled components for the purpose of software reusability. In addition, software engineers can consider the rest of dependences among software components as an unnecessary coupling that might occur as a result of poor or inefficient software components' design. The software engineers might use such information to investigate such dependencies for potential design problems. The contribution of the proposed approach is that, it computes the components' dependences based on the premise of components' output computation, such that a component influences another component if it contributes to its output computation.

V. Experimental Study

To illustrate the applicability of the proposed algorithm a small experimental study has been conducted. This study consists of three samples of sequence diagrams for selected operations within software system. In addition, testing is performed on randomly selected operations within four software systems that were modeled by different groups of students as system modeling project of real world systems. According to the proposed algorithm, the success criterion for detecting coupled components is the detection any form of data or control dependences. To demonstrate how the proposed algorithm is applied, consider the events in Figures 1, 3, and 5. The components model presented in the figures are simple examples of events that show sequence diagrams of a portion of lending library system, portion of home surveillance system, and student class enrolment. The presented samples of sequence diagrams and traces have been simplified for the purpose of demonstrating the algorithm. In addition, based on the specification of the given object, a note has been added at the activation bar of the receiver object containing a pseudo code that describes the execution of the sent message.

Figure 1 shows the sequence diagram of a lending library system event, in which a message *LendCopy*(*title*) is being sent to the *Main* object to search for a given *title*, and then *GetNumAvailable*() message is sent to the *aCopy* object to return how many copies are available for a specific *title* that is returned as *num* variable. Figure 2 shows the trace T_s of the lending library system event. The trace starts with the event's entry V_0 which shows the function $g(e_1)$ as the event triple, i.e., message label, sender, and receiver. When inspecting this trace by the proposed algorithm, the code statement at the execution position 4^4 is marked as an output code statement, then the analysis starts from the event's exit V_x in backward fashion. The marked execution position 4^4 is identified and its last definition at position 3^3 is marked. Note that this code statement does not belong to the sender object, so no data dependent is detected. The algorithm iterates until it reaches the event's entry V_0. According to this quick analysis, no data or control dependent is detected among these two objects (Main and aCopy), in which a conclusion can be drawn that no coupling is detected between Main and aCopy, in which either of these two components can be reused within another system or subsystem without the need of attaching the other component.

Another example is presented in Figure 3 which shows a simplified sequence diagram of an event of within a home surveillance system, in which a message *Activate_Deactivate*(*sensor*) is being sent to the *Control_Panel* object to search for an object *Sensor* to activate it or deactivate it. Figure 4 shows the trace T_s of the event of the home surveillance system. As can be seen the message *Activate_Deactivate*() is being sent to the object *Sensor* to examine its state, in which the *Sensor* is activated if its state is inactive and deactivated if its state is active. The code statement at the execution position 5^5 is marked by the algorithm as an output code statement, which returns the *Sensor* state to the control panel. The algorithm detects no used variables at this marked code statement, and as a result no data or control dependences are detected between the *Control_Panel* and *Sensor* objects in which no coupling is identified among those two objects, and they can be reused separately.

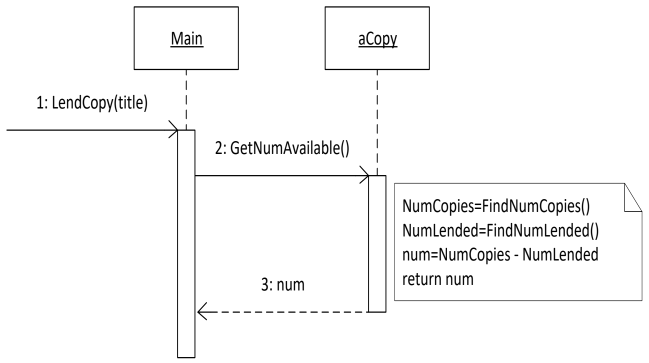
Main
aCopy
1: LendCopy(title)
2: GetNumAvailable()
NumCopies=FindNumCopies()
NumLended=FindNumLended()
num=NumCopies - NumLended
return num
3: num

Fig. 1. Sample Sequence Diagram of a Lending Library System.

```
V_0 g(e_1)=(LendCopy(title), Main, aCopy)
1       Send "LendCopy(title) to Main"
        1^1   FindTitle(title)
2   Send "GetNumAvailable() to aCopy"
        1^1   NumCopies=FindNumCopies()
        2^2   NumLended=FindNumLended()
        3^3   num= NumCopies - NumLended
        4^4   return num
3   Send "num to Main"
4   V_x
```

Fig. 2. The Trace T_s of the Sequence Diagram of a Lending Library System in Fig 1.

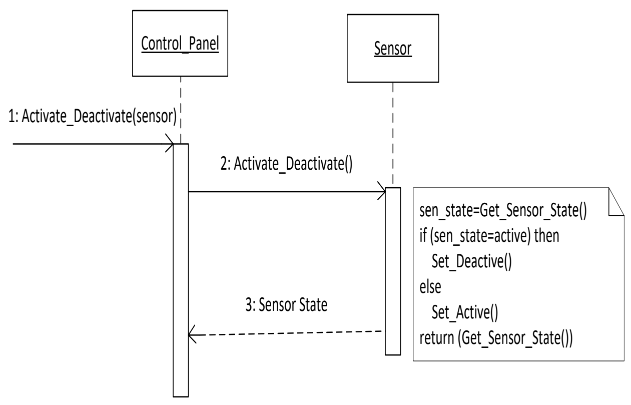
Control_Panel
Sensor
1: Activate_Deactivate(sensor)
2: Activate_Deactivate()
3: Sensor State
sen_state=Get_Sensor_State()
if (sen_state=active) then
Set_Deactive()
else
Set_Active()
return (Get_Sensor_State())

Fig. 3. Sample Sequence Diagram of Home Surveillance System.

```
V0  g(e1)=(Activate_Deactivate(sensor), Control_Panel, Sensor)
1  Send “Activate_Deactivate(sensor) to Control_Panel”
2  Send “Activate_Deactivate() to Sensor”
   1¹     sen_state=Get_Sensor_State()
   2²     if (sen_state = active) then
   3³         Set_Deactive()
   4⁴     else Set_Active()
   5⁵     return Get_Sensor_State()
3  Send “Sensor State to Control_Panel”
4  Vx
```

Fig. 4. The Trace T_s of the Sequence Diagram of Home Surveillance System in Fig 3.

The last example is the sequence diagram of student class enrolment that is shown in Figure 5. As shown the message *Register*(*std, CourseID*) is sent to the *aStudent* object to search for the course, then the message *Enroll*(*std, IsPrerequisite*) is sent to the *aCourse* object to search for the session number and assign the student to it if the prerequisite course is satisfied, next the session number is returned to the *aStudent* object. Figure 6 shows the trace T_s of the event of the student class enrolment system. When analyzing this trace by the proposed algorithm in backward fashion, with the assumption that the *IsPrerequisite* flag is true, the code statement at the execution position 5^5 is marked, then the last definition of the *SessionNumber* is identified at execution position 3^3 and marked. The algorithm iterates searching for marked code statement and looking for the last definition of all used variables at each marked code statement and marks them. As a result, the code statement at the execution positions 2^2 is marked in which a control dependent is identified between the two objects *aStudent* and *aCourse* because the passed flag *IsPrerequisite* is used at the predicate of the execution position 2^2. Also, data dependent is identified as well between these two objects because the passed variable *std* is used at the marked execution position 3^3. The detected data and control dependent contribute to the output computation of this event; therefore, a coupling can be identified between the two objects *aStudent* and *aCourse*. Therefore, the software engineer should consider the coupling among these two components when reusing either one.

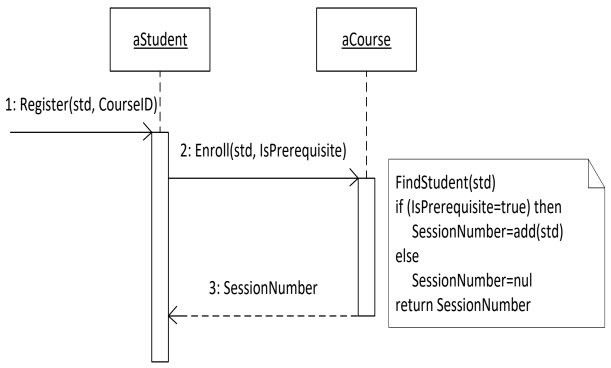

Fig. 5. Sample Sequence Diagram of Student Class Enrolment.

```
V0  g(e1)=(Register(std, CourseID), aStudent, aCourse)
1  Send “Register(std, CourseID) to aStudent”
   1¹     FindCourse(CourseID)
   2²     CheckPrerequisite(CourseID)
2  Send “Enroll(std, IsPrerequisite) to aCourse”
   1¹     FindStudent(std)
   2²     if (IsPrerequisite = true) then
   3³         SessionNumber = add(std)
   4⁴     else SessionNumber = nul
   5⁵     return SessionNumber
3  Send “SessionNumber to Student”
4  Vx
```

Fig. 6. The Trace T_s of the Sequence Diagram of Student Class Enrolment in Fig 5.

The remaining of this section describes some real world software systems which have been used to test the proposed algorithm. Table 1 summarizes the four software systems that have been modeled by different groups of students as a system modeling project of real world systems to demonstrate their skills in software design and architecture course. These four systems have been selected as semi-commercial software systems to test the applicability of the proposed algorithm for the real world systems.

TABLE I. EVENTS OF THE SELECTED SYSTEMS WITH COMPONENTS' DEPENDENCES

System Name	**Components (s_{e_p} & r_{e_p})**	**Message Name**	**Coupling**
NCAAA System	Website & Database	validate(ID)	No
University RFID	Sensor & RFID_Reader	read(signal)	Yes
Studying Abroad Advising System	Registration_System & Authentication_System	Validate_User_Information(info)	No
Truck Car Traffic Tracking System	RTD & DB_Driver	Retrieve_Driver_Data(ID)	Yes

The first system model is the NCAAA Accreditation System which is an automation system aims to automate all the processes that are required for any university applying for accreditation at the Saudi National Commission for Academic Accreditation and Assessment (NCAAA). The purpose of this system is to develop a system model that can be implemented to automate the accreditation process via an online system. The selected message *validate*(*ID*) is a validation event that occurs as a result of sending the message *validate*(*ID*) from the website object to the database object. This message validates the login of a client against the registered clients in the database. After analyzing this event by the proposed algorithm based on the specifications of the participating objects, no data or control dependence was identified among the object website that influences the output computation of the database object. Such result indicates the independence of the two components from each other, and the software engineer may reuse any of the two components without the need to attach the other one.

The second system model is the University RFID that is a part of an access control system that allows the university to control and monitor the access to its buildings and properties. The modeled system manages the access to the buildings, classrooms, labs, and offices within the university via the use of hardware solutions such as RFID cards that are linked to a software solution to manage and monitor the access. The selected message *read*(*signal*) is a reading event that occurs as a result of sending a signal to the sensor component which causes a *read*(*signal*) message to be sent to the *RFID_Reader* object to read the signal. After analyzing this event using the proposed algorithm based on the specifications of the participating objects, it has been identified that, the computation of the returned value of the *RFID_Reader* object is dependent on the passed data of the sensor object. Therefore, such dependence influences the output computation of the *RFID_Reader* object, and as a result, there is a coupling among the two objects sensor and *RFID_Reader* that must be considered when reusing either of these two components within another system.

The third system model is the Studying Abroad Advising System which aims to automate the supervision program at the ministry of higher education in Saudi Arabia. The ministry has established a scholarship program for Saudi students to study their BSc, MSc, and PhD in various specialties. This program is offered for the students to study at local private universities as well as at abroad universities. The ministry would like to manage this program such as facilitating the admission to the program, local/abroad universities subscriptions, monitoring students' performance, and facilitating the communications with the students/guardians. The selected message *Validate_User_Information(info)* is sent from the *Registration_System* object to the *Authentication_System* object. This event has been analyzed by the algorithm based on the specifications of the participating objects and found no data or control dependence of the object *Registration_System* that influences the computation of the output of the *Authentication_System* object, such detection identifies the independence of the two components, in which the two components may be reused separately within other system or subsystem.

The fourth system model is the Truck Car Traffic Tracking System that aims to help the Riyadh Traffic Department (RTD) to manage the trucks movement within the city highways and local roads due to the congestions that caused by these truck cars during the day and rush hours and to monitor the vehicles movements on the roads and highways by linking the RTD system with GPS system. The purpose of the modeled system is to guide the truck vehicles to follow alternative roads during the rush hours or in the case of congestions in the highway. Also, the system aims to provide different services such as monitoring the vehicles movements and issuing violation tickets. A data retrieving event has been selected that occurred as a result of sending the *Retrieve_Driver_Data*(*ID*) message from the RTD object to the *DB_Driver* object. After analyzing this event by the proposed algorithm, based on the specifications of the participating objects, control dependence is discovered by the RTD object that influences the computation of the output of the *DB_Driver* object, which causes a coupling among these two objects that should be considered when reusing either one.

Although the experiments have been investigated on several events within some selected software systems' models, the results of the experiments have shown encouraging results for the applicability of the proposed algorithm in detecting software components' coupling. The proposed algorithm can help the software engineers to better understand the software system model and the relationships among its components. This may help the software engineers to comprehend the software components' dependences in order to optimize the software system model by eliminating the unnecessary dependences among software components. Such minimization definitely, will increase the components' cohesiveness, and as a result, the software components' reusability should be improved. In addition, the algorithm provides the software engineer with information about the influence among components that identifies components' coupling in which a component should be attached with another reused component within other system or subsystem.

VI. Conclusion

In this paper an approach for detecting coupling among software components has been proposed. The proposed approach analyzes the software system model under test in terms of a sequence diagram trace which represents the dynamic behavior of the system under analysis. The notion of data and control dependence is used in the proposed approach to identify the dependence among system components. The notion of influence between components has been introduced in which a data or control dependence of a component is considered if it contributes to the output computation of the other component. The applicability of the proposed algorithm in identifying the software components' coupling has been presented in the paper through an experimental study.

The experimental study has shown encouraging results in detecting the coupling between the software components. The software engineer may use the results to have a better understanding of the software system model under analysis in terms of the dependences among its components and how they may influence their reusability. Software engineers may use the provided information by the algorithm to eliminate the unnecessary components' couplings for the purpose of enhancing software reusability.

The proposed algorithm has been applied manually by inspection and walkthrough of the tested model, in which the trace is inspected manually. The future plan is to implement the proposed algorithm in which the system model under test can be automatically instrumented and the trace is recorded dynamically, in which the model analysis can be performed at the runtime. In addition, an integration of the proposed algorithm within one of the open source environment may be implemented in which the system model can be constructed and examined. Furthermore, an investigation may be conducted to extend the algorithm to identify the different types of coupling among components such as content coupling, common coupling, external coupling, control coupling, stamp coupling, and data coupling.

REFERENCES

[1] R. Taylor, N. Medvidovic, and E. Dashofy, Software Architecture Foundations, Theory, and Practice, 2010, Wiley.

[2] W.P. Stevens, G.J. Myers, and L.L. Constantine, "Structured design". IBM Systems Journal, 1974, 13(2), pp. 115-139.

[3] L.L. Constantine and E. Yourdon, Structured Design, 1979, Prentice-Hall, New Jersey.

[4] G. Myers, Reliable Software through Composite Design, Mason and Lipscomb, New York, 1974.

[5] P.-J. Meilir, Practical Guide to Structured Systems Design, Prentice Hall, 2nd Edition, 1988.

[6] R. Pressman, Software Engineering: a Practitioner's Approach, 1992, McGraw-Hill.

[7] A.J. Offutt, M.J. Harrold, and P. Kolte, "A software metric system for module coupling", The Journal of Systems and Software, March 1993, 20(3), pp. 295-308.

[8] L. Briand, J. Daly, and J. Wust, "A unified framework for coupling measurement in object-oriented systems", IEEE Transactions on Software Engineering, 1999, Vol. 25, No. 1, pp. 91-121.

[9] S. Henry and D. Kafura, "Software structure metrics based on information flow", IEEE Transactions on Software Engineering, 1981, Vol. SE-7, No. 5, pp. 510-518.

[10] E.B. Allen and T.M., Khoshgoftaar "Measuring coupling and cohesion: an information-theory approach", Proceedings of the 6th International Software Metrics Symposium, 1999, pp. 119 – 127.

[11] J. Offutt, A. Abdurazik, and S. Schach, "Quantitatively measuring object-oriented couplings", Software Quality Journal, 2008, Vol. 16, No. 4, pp. 489 – 512.

[12] H.Y. Yang and E. Tempero, "Measuring the strength of indirect coupling", IEEE Proceedings of the Australian Software Engineering Conference (ASWEC'07), 2007.

[13] J.-B. Fasquel and J. Moreau, "A design pattern coupling role and component concepts: Application to medical software", The Journal of Systems and Software, 2011, No. 84, pp. 847–863.

[14] M. Choi and J. Lee, "A dynamic coupling for reusable and efficient software system", IEEE 5th International Conference on Software Engineering Research, Management and Applications, 2007, pp. 720-726.

[15] G. Gui and P.D Scott., "Ranking reusability of software components using coupling metrics", The Journal of Systems and Software, 2007, No. 80, pp. 1450–1459.

[16] W. Horré, D. Hughes, K.L. Man, S. Guan, B. Qian, T. Yu, H. Zhang, Z. Shen, M. Schellekens, and S. Hollands, "Eliminating implicit dependencies in component models", Proceedings of the IEEE 2nd International Conference on Networked Embedded Systems for Enterprise Applications (NESEA) , 2011, pp. 1-6.

[17] A. Jhumka and M., Leeke "The early identification of detector locations in dependable software", Proceedings of the 22nd IEEE International Symposium on Software Reliability Engineering, 2011, pp. 40-49.

[18] N. Ajienka and A. Capiluppi, "Understanding the interplay between the logical and structural coupling of software classes", The Journal of Systems and Software, No. 134, 2017, pp. 120–137.

[19] M. Kessel and C. Atkinson, "Ranking software components for reuse based on non-functional properties", Inf Syst Front, No.18, 2016, pp. 825–853.

[20] K. Hasan and M. Hasan, "Principal Component Analysis of Coupling Measures for Developing High Quality Object Oriented Software", International Conference on Computer and Communication Engineering (ICCCE 2010), 2010.

[21] A. MacCormack and D. Sturtevant, "Technical debt and system architecture: The impact of coupling on defect-related activity", The Journal of Systems and Software, No. 120, 2016, pp. 170–182.

[22] T. Oyetoyan, D. Cruzes, and R. Conradi, "A study of cyclic dependencies on defect profile of software components", The Journal of Systems and Software, No. 86, 2013, pp. 3162– 3182.

[23] M. Yutao, H. Keqing, L. Bing, and Z. Xiaoyan, "How multiple-dependency structure of classes affects their functions a statistical perspective". The 2nd International Conference on Software Technology and Engineering, 2010, pp. V2-60-V62-66.

[24] S. Chidamber and C. Kemerer, "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, Vol. 20, No. 6, 1994, pp. 476-493.

[25] E. Arisholm, L. Briand, and A. Føyen, "Dynamic Coupling Measurement for Object-Oriented Software", IEEE Transactions on Software Engineering, Vol. 30, No. 8, 2004, pp. 491-506.

[26] S.M. Cho, H.H. Kim, S.D. Cha, and D.H. Bae, "A semantics of sequence diagrams", Information Processing Letters, 2002, No. 84, pp. 125–130.

[27] J. Ferrante, K. Ottenstein, and J. Warren, "The program dependence graph and its use in optimization", ACM Transactions on Programming Languages and Systems, 1987, Vol. 9, No. 3, pp. 319-349.

[28] J. Musa, "Operational profiles in software reliability engineering", IEEE Software, March 1993, Vol. 10, No. 2, pp. 14-32.