Proceedings of the World Congress on Engineering 2011 Vol II
WCE 2011, July 6 - 8, 2011, London, U.K.

Towards Component-Based System Integration Testing Framework

Sajjad Mahmood

Abstract—**The success of a Component Based System (CBS) relies on integrating individual components. CBS integration testing encounters challenges similar to traditional testing; and aims to prioritize test cases with a potential benefit of improving the rate of fault detection. In this paper, we present a CBS integration testing framework to identify integration test criteria and prioritize test cases. We also present an application of CBS integration testing framework to the Information Management System (IMS).**

Index Terms—**Component Based Systems, Software Components, Integration Testing**

I. Introduction

COMPONENT -based software development is integration centric [15] with a focus on assembling components to build a software system. A CBS can be developed using components written in different programming languages (e.g. Java, .Net technologies); and they interact with different types of external entities. Similar to web applications [2], [3], the heterogeneous nature of components and deployment architectures introduce complexities in the integration process that must be analyzed and validated during a testing process. A component goes through a traditional software testing process at the developer's site [11]. However, due to the integration-centric nature of CBS and heterogeneity of involved components, integration testing [21] plays an important role in detecting faults during a CBS development process.

Software testing can be an expensive process to execute in full, due to the large number of possible test cases derived from a system specification. The large numbers of input fields, input choices and the ability to enter values in any order combine to create a state space explosion problem [2]. To reduce the cost of testing, software testers aim to prioritize test cases based on some measure, with the potential benefit of improving the rate of fault detection [6]. CBS integration testing encounters challenges similar to those found in tradition testing; for example, how to minimize risk associated with selecting testing criteria or prioritize test cases to meet budget and schedule constraints.

Munson et al. [18] have indicated that there is a correlation between the number of faults found in a software component and its complexity. Similarly, Fenton et al. [8] and Emam et al. [7] show that quantitative factors, such as complexity and coupling, have a major impact on the fault proneness of a software application.

Recently, Goseva-Popstojanova et al. [13] have used cyclomatic complexity [16] to estimate the probability of failure of individual components for an architectural-level risk assessment which can be used in the early phases of the software life cycle. We believe that the CBS complexity measures can enable a system analyst to identify fault-prone components; and can be used as a measure to prioritize test suites and improve the rate of fault detection.

Sajjad Mahmood is with the Information and Computer Science Department, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia (e-mail: smahmood@kfupm.edu.sa).

In this paper, we present a process to identify CBS integration test criteria and prioritize test suites based on the complexity metrics. The framework uses the structural control flow coverage criteria as adequacy criteria for CBS integration testing. Next, we use complexity measures [14] to select adequate test criteria for interactions in a CBS. The complexity measures enable a system analyst to identify fault-prone components and associated interactions; and are used as a measure to improve the rate of fault detection. Finally, suitable test cases are identified for a chosen test criteria. To demonstrate the usefulness of this method, it has been applied to the Information Management System (IMS) application.

The rest of the paper is organized as follows. Section II reviews related literature. In Section III, we present the framework. Section IV describes the application of the method, while Section V presents results of the case study. We conclude the paper and discuss future work in Section VI.

II. Related Work

CBS testing techniques mainly focus on validating individual component using black box testing techniques [12], [17], [20]. For example, Build In Tests (BIT) models consists of built-in testing-enabled components which implement mandatory interfaces. Testers access the built-in testing capabilities of BIT components through the corresponding interfaces and which contain the test cases. The Self-testing COTS components [4] strategy proposes to augment a component with functionality of analysis and testing tools thus enabling it to be capable of conducting some or all activities of the component user's testing processes.

Gao et al. [4] have proposed a component test model to analyze API-based component validation and testing. The test model uses the concepts of the component function access graph to represent components access patterns. Further, a set of API-based test criteria is also proposed to evaluate the models. They have also proposed a component regression test approach [10] to identify component changes and their impact on CBS. They have also developed a tool called COMPTest which supports automatic identification analysis of API-based component changes and black box test selection.

A test model using an interaction graph is presented in [21] that depicts a generic infrastructure of a component based system and suggests key test elements. This model puts emphasis on the interaction among components in component base systems and is targeted at revealing inter-component and interpretability faults.

III. CBS Integration Testing Framework

Component integration is a complex and risk-prone process because it is rarely the case that two components are perfectly matched. We believe that successful CBS integration testing needs to address two key issues: (1) specify CBS interactions and (2) identify test criteria that give the optimal coverage. Figure 1 shows the CBS integration testing framework.

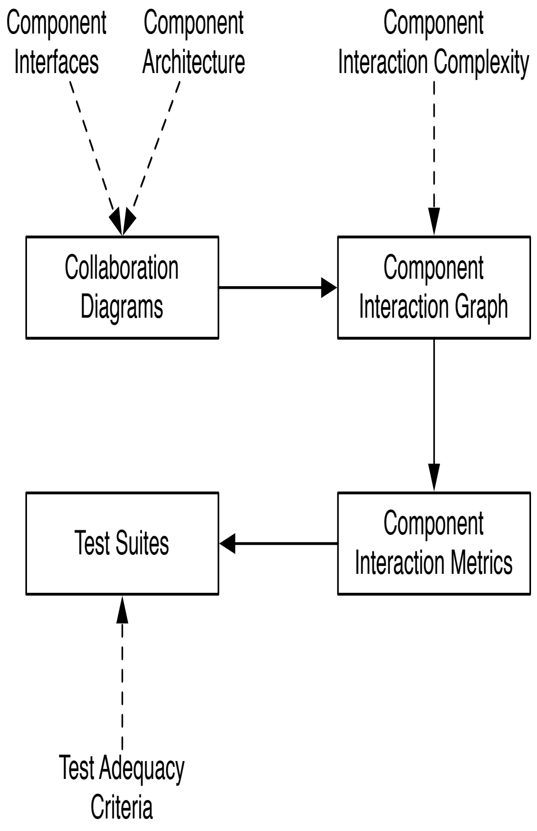

Fig. 1. Integration Testing Framwork

A. Test Adequacy Criteria

A test adequacy criterion is a predicate [9] which is used to determine when software has been adequately tested for a given testing criterion. It acts as means of organizing the testing activity and is a measure of progress toward a testing goal [19]. In this paper, we use the structural control flow coverage criteria [1] for integration testing of a CBS. Control flow coverage criteria define Test Requirements (TR) in terms of properties of test paths in a graph G. A typical test requirements is met by visiting a particular node, edge or path. We discuss these criteria as follows:

- *Category A - Node Coverage:* TR contains each reachable node in G.

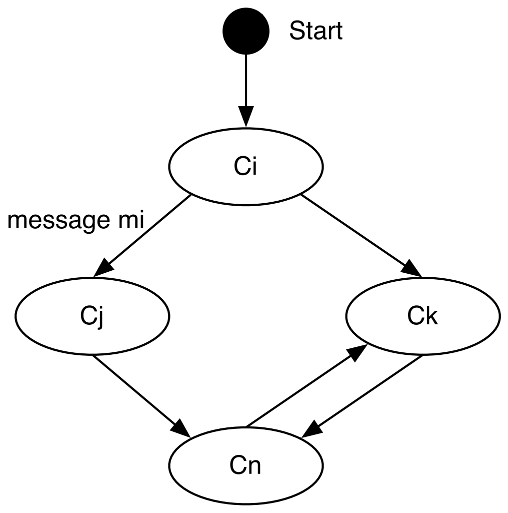

Fig. 2. A Component Interaction Graph

- *Category B - Edge Coverage:* TR contains each reachable path of length up to 1, inclusive, in G.
- *Category C - Prime Path Coverage:* TR contains each prime path in G.

B. Component Interaction Model

Interaction between components is characterized by a component's interface or through using other component's event. The interaction occurs when a component provides an interface and other component uses it. In UML CBSS [5], collaboration diagrams specify the interactions between components. Each collaboration diagram shows one or more interactions, where each interaction shows one possible execution flow.

1) Component Interaction Graph: For each collaboration diagram, we construct a directed graph called Component Interaction Graph (CIG), to represent the interactions between components, as shown in Figure 2. The CIG is defined as follows:

Definition 1: A CIT is defined as a tuple [N, E, Start], where [N,E] is a directed graph, and Start is the root node. N is a set of nodes in a graph, where N = [ni], i = 1 . . . . . . |N|; and E is a set of edges in the graph, where E = [ei], i = 1 . . . . . . |E|.
Definition 2: Node 'n': n $\in$ N represents a component C.
Definition 3: Directed Edge 'e': e $\in$ E represents the control flow transfer from a component Ci to another component Cj. Each edge is annotated by interaction complexity (CIij).
Definition 4: Path 'p': A path p is a sequence [N1,N2, ... Nm] of nodes, where each pair of adjacent nodes, (n_i, n_{i+1}), 1 < i < M, is in the set E of edges.

The steps to convert collaboration diagrams into a CIG are defined as follows:

1) The CIG construction process starts with selecting a collaboration diagram.

2) Every component Ci in a collaboration diagram will correspond to a node Ni ∈ N of CIG where name of Ni is same as that of the component Ci.

3) For every message mi and mj between two components in a collaboration diagram, a direct edge will be created from Ni ∈ N to Nj ∈ N in CIG.

4) A weight w is assigned to every directed edge Ei between N_i and N_j in a CIG, as

$$w = \sum_{ij=1}^{max} IF_{ij} \times \sum_{k=1}^{n} CM_k \tag{1}$$

where IF_{ij} is the total interaction frequency and CM_k is the content complexity of the data types involved in the information exchange between two nodes n_i and n_j.

5) Select the next collaboration diagram to be merged into the CIG and repeat step 2 - 3 for all new nodes and edges.

6) Identify the common edges E_{common} between the collaboration diagram and the CIG.

7) Update weight w assigned to the common edge Ei between N_i and N_j in a CIG, as

$$w = w_{old} + \sum_{ij=1}^{max} IF_{ij} \times \sum_{k=1}^{n} CM_k \tag{2}$$

where w_{old} is the previous weight of the edge.

C. Test Criteria Selection

The test criteria selection phase starts with investigating a component's interactions in a CBS. We propose to use complexity metrics to estimate the fault proneness of interactions among components in a CBS. Complexity measure is chosen as a quantitative factor because it has a proven impact on fault proneness [13], [7]. We perform a complexity measure for a component specification with a focus on identifying competent attributes that affect its complexity at both component and intra-component levels. We use interface and interaction complexity measures [14] to rank component interactions and subsequently select suitable test criteria. For an example showing the approach to measure the complexity of a component, please refer to our previous work [14].

TABLE I
COMPONENT INTERACTION METRICS

	C1	C2	C3	C4
C1	X	<ITc>	<ITc>	-
C2	<ITc>	X	-	<ITc>
C3	<ITc>	-	X	<ITc>
C4	-	<ITc>	<ITc>	X

1) *Component Interaction Metrics:*

We introduce the notation of a Component Interaction Metrics (CIM) to represent the interface and interaction complexities between components in a CBS. Table I shows a CIM where each interaction is expressed as a relation between pairs of components. Each element CIM(i) is represented as <ITc> where ITc represents the sum of the interface complexity measures of the involved components. A '-' at an element CIM (i) indicates that the corresponding components do not directly interact with each other. Further, each element CIM (i,j) where i = j is denoted by 'X'. We propose the following set of rules to select a suitable testing adequacy category based on interface complexities of components.

Rule 1: Select test adequacy criteria A for each element (i,j) in the CIM

$$IF(ITc[i,j] < (Average(ITc) - StandardDeviation) \tag{3}$$

Rule 2: Select test adequacy criteria B for element (i,j) in the CIM

$$IF((Average(ITc) + StandardDeviation) > ITc[i,j]) \tag{4}$$

Rule 3: Select test adequacy criteria C for element (i,j) in the CIM

$$IF(ITc[i,j] > (Average(ITc) + StandardDeviation) \tag{5}$$

Finally, test cases are generated based on the selected test adequacy criteria using the notion of appropriate path coverage of the CIG.

IV. An Application

In this section, we describe an application of the CBS integration testing process to the Nomad IMS[1] application. Nomad IMS is an open source system developed using Eclipse[2] development framework and uses both specialized and third-party components (plug-ins). Furthermore, Nomad IMS has well documented list of faults identified during different versions of the application. Nomad IMS allows the tracking of personal data, notes, diary, money and contact details. It also provides support for scheduling and time tracking. Figure 3 shows Nomad IMS components and the interactions between them.

TABLE II
FAULTS IDENTIFIED IN NOMAD PIM VERSION 7

Faults	Description
1	Double clicking on the time interval opens entity.
2	Can not open activity with return in current activities view.
3	Spend time view not updated.
4	Week overview not updated correctly when linked to calendar.
5	Week formatter returns wrong year.
6	Calendar not updated when using navigation buttons.
7	Week overview view not updated correctly.

[1]http://nomadpim.sourceforge.net/
[2]http://www.eclipse.org/

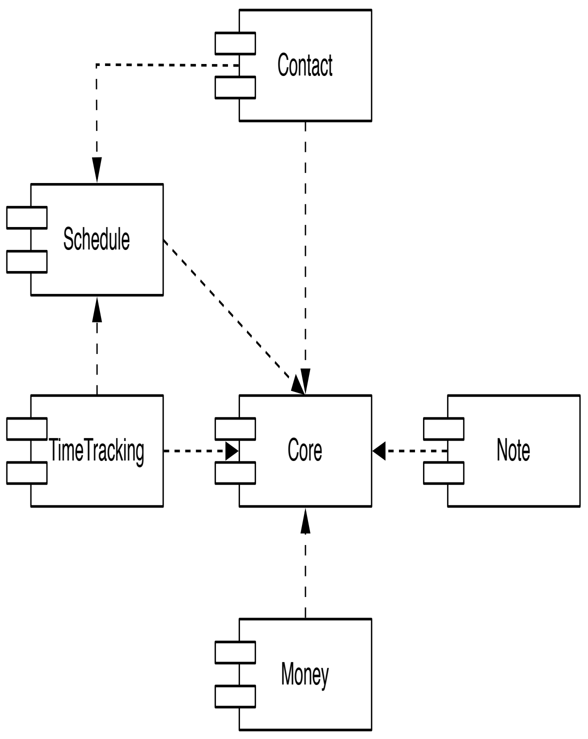

Fig. 3. Nomad IMS Component Specification Architecture

To analyze the proposed testing process, we use Nomad IMS version 7. There are 15 faults reported by the users of Nomad IMS which can be classified into two groups: unit level faults and integration level faults. Out of these 15 faults, 8 are unit level faults and the remaining 7 faults involve interaction between Nomad IMS components. In this analysis, we only consider the integration level faults and present a brief description of these faults in Table II.

Figure 3 shows that there are seven interactions between IMS components. The 'Core' component directly interacts with all the remaining five components. The 'Schedule' component also interacts with 'Contact' and 'TimeTracking' components. We start the adequate test criteria selection for each interaction by developing the CIM, as shown in Table III. Furthermore, based on the rules defined in section III-A, we classify each interaction as either 'Category A', 'Category B' or 'Category C'.

TABLE III
IMS Component Interaction Metrics

Interactions	Complexity Measures	Testing Adequacy Criteria
Core-Schedule	<252>	C
Core-Contact	<189>	B
Core-TimeTracking	<245>	B
Core-Note	<217>	B
Core-Money	<224>	B
Schedule-Contact	<161>	A
Schedule-TimeTracking	<217>	B
Average Complexity Measure	<215>	

Lastly, based on rules defined in section III-C, we identify the suitable test suites for each component interaction, as shown in Table IV. Furthermore, each test case was executed against IMS version 7 and outcomes were examined against the known faults.

TABLE IV
IMS Test Cases

Interactions	No. of Test Cases
Core - Schedule	31
Core - Contact	21
Core - Time Tracking	24
Core - Note	17
Core - Money	25
Schedule - Contact	7
Schedule - Time-Tracking	22

V. Result Analysis

Table V shows the faults found in IMS. The results indicate that CBS complexity measures can be used as a guideline to prioritize CBS testing. However, the testing process failed in detecting the two faults, namely, fault no. 2 and fault no. 4.

It is important to note that the IMS application only had seven integration faults and there is a need for studies on more complicated programs.

TABLE V
Faults Found in IMS

Fault No.	Status
1	Found.
2	No test suite generated that could cover it.
3	Found.
4	No test suite generated that could cover it.
5	Found.
6	Found.
7	Found.

VI. Conclusions and Future Works

In this paper, we present an initial outline of CBS integration testing framework which shows the potential of using software complexity measures in adequate test criteria and test case selection. The primary strength of the integration testing framework is the notion that complexity measures can be used to identify testing adequacy criteria. The framework uses software complexity measures to help prioritize CBS integration testing with the potential benefit of improving rate of fault detection. Furthermore, complexity metrics help address the challenge of selecting suitable test suites that will increase the probability of detecting faults. For future work, we plan to empirically validate the benefits of the framework on medium to large applications.

Acknowledgement

The author would like to acknowledge the support provided by the Deanship of Scientific Research at King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia under Research Grant JF090010.

References

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2009.
[2] Anneliese A. Andrews, Jeff Offutt, and Roger T. Alexander. Testing web applications by modeling with fsms. *Software and Systems Modeling*, 4(1):326 – 345, 2004.

[3] Anneliese A. Andrews, Jeff Offutt, Curtis Dyreson, Christopher J. Mallery, Kshamta Jerath, and Roger T. Alexander. Scalability issues with using fsmweb to test web applicaitons. *Information and Software Technology*, 52(1):52 – 66, 2010.

[4] S. Beydeda and V. Gruhn. Merging components and testing tools: the self-testing cots components (stecc) strategy. In *Proceedings of 29th Euromicro Conference*, pages 107–114, 2003.

[5] John Cheesman and John Daniels. *UML Components A Simple Process for Specifying Component Based Software*. Addison-Wesley, 2001.

[6] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Test case prioritization:a family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159 – 182, 2002.

[7] Khaled El Emam, Walcelio Melo, and Javam C. Machado. The prediction of faulty classes using object oriented design metrics. *Journal of Systems and Software*, 56(1):63 – 75, 2001.

[8] N.E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8):797–814, 2000.

[9] Phyllis G. Frankl and Elaine J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483 – 1498, 1988.

[10] J Gao, D Gopinathan, Quan Mai, and Jingsha He. A systematic regression testing method and tool for software components. In *30th Annual International Computer Software and Application Conference (COMPSAC '06)*, pages 455 – 466, Chicago, USA, 2006.

[11] Jerry Zeyu Gao, H.-S. Jacob Tsao, and Ye Wu. *Testing and Quality Assurance for Component Based Software*. Artech House, 2003.

[12] Jerry Goa, K. Gupta, and S. Gupta. On building testable software components. In *Proceeding of the First International Conference on COTS Based Software Systems*, pages 108–121. Springer-Verlag, 2002.

[13] Katerina Goseva-Popstojanova, Ahmed Hassan, Ajith Guedem, Walid Abdelmoez, Diaa Eldin M. Nassar, Hany Ammar, and Ali Mili. Architectural-level risk analysis using uml. *IEEE Transactions on Software Engineering*, 29(10):946–960, 2003.

[14] Sajjad Mahmood and Richard Lai. A complexity measure for uml component system specification. *Software-Practice and Experience*, 38(2):117–134, 2008.

[15] Sajjad Mahmood, Richard Lai, Yong Soo Kim, Ji Hong Kim, Seok Cheon Park, and Hae Suk Oh. A survey of component based system quality assurance and assessment. *Information and Software Technology*, 47(10):693 – 707, 2005.

[16] T.J. McCabe. A complexity measure. *IEEE Transaction on Software Engineering*, 2(4):308–320, 1976.

[17] C. Mueller and B. Korel. Automated black-box evaluation of cots components with multiple-interfaces. In *Proceedings of the 2nd International Workshop on Automated Program Analysis, Testing and Verification, ICSE 2001*, 2001.

[18] J. Munson and T. Khoshgoftaar. Software metrics for reliability assessment. *Handbook of Software Reliability Engineering*, pages 493 – 529, 1996.

[19] Matthew J. Rutherford, Antonio Carzaniga, and Alexander L. Wolf. Simulation-based test adequacy criteria for distributed systems. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM Press, 2006.

[20] L. Tahat. Requirement based automated black-box test generation. In *Proceedings of the 25th Annual International Computer Software and Application Conference*, pages 489 – 495. IEEE Computer Society Press, 2001.

[21] Ye Wu, Dai Pan, and Mei-Hwa Chen. Techniques for testing component-based software. In *Proceedings of Seventh IEEE International Conference on Engineering of Complex Computer Systems*, pages 222–232, 2001.