A Proposal for New Software Testing Technique for Component Based Software System

Bayu Hendradjaya

School of Electrical Engineering and Informatics, Institut Teknologi Bandung, INDONESIA

Abstract: A Component-Based Software (CBS) system consists of integrated components that work together to perform specific tasks. Different components are selected and integrated to form a new software system. The components may have been developed by other third party, thus it is expected that the development time and effort can be reduced significantly. However just like any traditional development, the testing activities requires a specific evaluation to assess the software. Most of the components do not come with the source code, but only some information of the components. Thus, a specific testing technique is required. In this paper, we propose a new testing technique for Component-Based Software system. Older techniques had been developed based on traditional metrics, while other CBS system had different strategy from what we propose in this research. The technique help determining test adequacy criteria based on a set of complexity and criticality metrics of a CBS system. Based on this test adequacy criteria, our experimental studies have shown that it has assisted in reducing the number of test suite and test cases. For software testers, this technique would significantly reduce the testing time and effort in CBS development.

Keyword: Component Based Software (CBS), Component Based Metrics, Component Integration Metrics, Software Testing, Component Based Software Engineering

1. Introduction

Component-Based Software (CBS) system is proposed to allow the easy, fast and reduced cost of development of a software project[1]–[8]. Component-based technology allows the creation of components with certain functionalities. However there are also some negative impacts such as complexity issues [9], [10], [19], [11]–[18], increased criticality [16], [20] and increased interaction among components [19], [21]–[25]. Some of these issues have been evaluated since the development of object-oriented methods/techniques[26], [27], but still existed in the component based system. These impacts lead to other problems such as compatibility interaction behavior [28], software anomalies [29], integration difficulties [20], [30], testing issues, [20], [30], or cost issues [31]–[33]. The effect of the negative issues of complexity, criticality and increased interaction can increase the effort to perform the testing activities.

In this paper, we propose the new technique to test CBS system by employing the use a set of CS metrics that have been defined in our previous research[34]. A testing technique for CBS system could be effectively performed by using specific CBS characteristics. The right set of metrics can help reveal these CBS system characteristics.

Each integration of components requires significant extra testing[35], thus testing activities are a must to ensure each expected behavior and performance is well taken care of. A component needs to be tested sufficiently before it is ready to be used by the application clients. On the other hand, some traditional testing techniques for a single component were still practical to be used [36]. Many traditional unit testing techniques were still applicable [37].
Once a component has been integrated, it is subject to integration testing. Integration of a component means that the component operates in a new environment. The focus is not only to test the interfaces that glue the components but also to analyze the interactions between components [21], [23], [36], [38]–[40].

Component users usually are not provided with the source code of the component. Therefore, the test is conducted based on the information at hand. However, a component provider should

Received: December 18th, 2017. Accepted: March 23rd, 2018
DOI: 10.15676/ijeei.2018.10.1.5

present a metadata and a clear description of the behavior of the component [36], [41]. We can utilize this information to assist in testing activities.

In this paper, we suggest a software testing technique for component based system. Previous research on component testing had various proposals which attacks the faults between components and their interoperability, but others still suggest traditional faults. Many has proposed techniques to uncover erorrs, some of these are combination of older techniques [42] [43] [36], [40] [44], however many more have proposed specific technique that use the CBS information [43] [42] [45] [46] [47] [48] [49] [50]–[55]. In this research, a testing technique is proposed which utilize component information from CBS metrics to reduce the number of test suite and test cases.

The organization of the paper are as follows. In section 2, we provide the literature survey of the current testing techniques. Then we described our proposal of the new testing technique. In section 4, we describe our experimental study and followed by the evaluation of the proposal. The conclusion is presented in the last chapter with the future work.

2. Literature Survey

Faults in integrated component testing can be classified as Inter-component faults, Interoperability faults and traditional faults[42]. Inter-component faults appears when a component is combined with other components, while interoperability faults may be detected from components that are built from different infrastructures (different operating system or libraries) or misinterpretation of specifications or different programming languages. The traditional faults occur within individual components. We can use conventional testing techniques to uncover this kind of faults.

Testing for CBS has several specific characteristics as the following:

- The testing should focus on the interface that combine the components to the new system, thus requiring investigation on interaction among components[56]
- Black box testing may need to be applied if the source code is not included by component developers[42]. The CBS tester may need to determine the test coverage and criteria, based on the information at hand. A test adequacy criteria for a component-based system is proposed to assist in the determination of test coverage[43].
- A component may have passed a test in one environment, however another test still needs to be performed in a new environment [39]. Some reports show that testers cannot rely on previous testing [36], [40].

Several proposals have been presented to uncover the faults, however there are common issues faced by component users as follows:

- Limited or no access to the component’s source code. Many proposals have imposed suitable information (metadata) on the component [7], [33], [57], [58].
- Even if a component provider offers the source of the component, the component user may use a different development language [59]. Hence, a testing tool at the user side may have a problem recognizing the original language of the component.
- A component provider does not know in advance what the real requirements will be. Therefore, the provider needs to test the component as a context-independent unit of software[35], [59].
- The component user should be aware that a component developer may have not enough time to undertake adequate testing[40].

To solve the issues above, several proposals on component-based testing suggested criteria to adequately test component integration. Some suggested to reduce the number of test cases from a test suite, subsequently risking the capability to detect faults [44]. However, some research claims that their reducing strategy either did not compromise fault detection [60] or only compromised it to a small degree [61]. Others proposed a framework or a test model to help the testing process and also to enable automated testing. Automated testing should be a benefit in performing regression testing.

Some of the proposals on software testing for components were as follows:

- Proposals on test adequacy criteria
    - A paper by Rosenblum [43] proposed two formal definitions of criteria that were used to analyze whether a test suite has an adequate amount of testing for a component-based system.
    - Ye and Dai [42] proposed the use of a component interaction graph to generate a family of test adequacy criteria.
    - Gao et al. [45] presented a dynamic approach to adequately test model and test coverage criteria for component validation using a *component function access graph*.
    - Jin and Offut [46] proposed a coverage criteria based on interconnections between two components, as a fault in one component may affect the coupled component. Experimental studies by Jalote et al. [62] showed that every component has test buddies, which are coupled components that may affect the connected component.
- Proposals on testing frameworks and test models
    - Cho and McGregor[47] proposed a testing framework that addresses component interoperation in message protocols on a formal specification.
    - Belli and Budnik [48] provided a framework to help automate the test case and test script generation.
    - Edwards [49] proposed a framework for black box testing of CBS. The framework offers an automatic generation of a test driver, test data and test oracles.
    - Wu et al. [63] introduced a component-based test model that used UML diagrams to model the component's behavior.
    - Some frameworks for component testing used information from the component's metadata [64], [65] or are based on black box testing techniques[48].
    - Liang and Xu proposed an test-driven component integration with the support of UML 2.0 testing and monitoring profile (U2TMP). Their proposal generated test cases in integration-level automatically using this profile [66].
    - Elsafi et. Al. suggested an improved learning algorithm to infer a model of integrated components[67] .
- Proposals on automated component-based testing
    - Some frameworks provided by Belly and Budnik [48]. and Edwards [49], [68] helped automate test case and test script generation by using their frameworks.
    - Gallagher and Offutt [69] proposed the use of a finite state machine model to describe component interaction and provide a test method to automate test sequence generation.
    - Some approaches [50]–[55] embedded a component with a suitable executable test case, also known as built-in testing or BIT [70].
    - Kang and Park [71] have proposed an automatic generation algorithm of expected results for a component based software system.
    - Braione et al. [72] have proposed a code-based test generations on industrial software compenents.
    - Saglietti and Pinte have proposed an automated generation of test cases for unit and integration [73] using evolutionary algorithm such as Genetic Algorithm.
- Regression Testing
    - Orso et al. [41], [74] provided techniques to address the problem of regression test selection for component-based applications using the component's metadata
    - Jiang et al. [75] proposed a process to do black-box regression testing. Their study showed that their process can reduce the number of regression tests.
- Other proposals
    - Silva et al. [76] provided an experimental study to examine the use of a case tool to support component testing. Two workflows were presented as a guideline for the component developer/user.
    - Grundy et al. [77] proposed 'Dynamic validation agents' for testing software component deployment.

- Imran et al. [78] have proposed strategies to develop software testing.
- Alegroth et al. [79] have proposed component-based testing by using Visual GUI testing.
- Aktouf [80] have proposed a testing-location based for component based mobile application
- Weber [81] have developed a tool for supporting fuzz testing of component based system.

While most of the proposal suggested the use of conventional technique, in this paper, we propose a new technique based on the metrics that is specifically design for the CBS system.

The proposed testing technique uses a set of CBS metrics. Many other research for software metrics had been proposed [82]–[89] however for this technique we use CBS metrics that is define in [34]. Their set of metrics requires the CBS system be presented in a graph connectivity. The nodes represent the components and the link represent the connectivity between components. Interactions occur through interfaces and events that are arriving in. For the purpose of this testing technique, we only use several type of the metrics. The original definition of the metrics provide a set of static and dynamic metrics.

Table 1. The Static Metrics [34]

Name	**Formulae**	**Description**
Component Interaction Density Metric	$CID = \#I/\#I_{MAX}$	#I is the number of actual interactions and $\#I_{MAX}$ is the number of maximum available interactions.
Component Incoming Interaction Density	CIID=#Iin/#ImaxIN	$\#I_{IN}$ is the number of incoming interactions used and $\#Imax_{IN}$ is the number of incoming interactions available.
Component Outgoing Interaction Density	$COID = \#I_{OUT}/\#Imax_{OUT}$	$\#I_{OUT}$ is the number of outgoing interactions used and $\#Imax_{OUT}$ is the number of outgoing interactions available.
Component Average Interaction Density	$CAID= \Sigma_n CID_n/\#components$	$\Sigma_n CID_n$ is the sum of interactions density of n component and #components is the number of the existing component
Size Criticality Metric	$CRIT_{SIZE}= \|\{c\|size(c) > SizeThreshold\}$	#size_component is the number of component,which exceeds a given critical value.
Link Criticality Metric	$CRIT_{LINK} = \|\{c\|link(c) > LinkThreshold\}\|$	Link(c) is the number of connected components from an individual component
Inheritance Criticality Metric	$CRIT_{INH} = \|\{c\|inheritance(c) > InhThreshold\}\|$	#root_component is the number of root components which has inheritance.
Bridge Criticality Metric	$CRIT_{BRIDGE} = \|\{c\| bridge(c)\}\|$	#bridgecomponent is the number of bridge components.
#Criticality Metrics	$CRIT_{ALL}= CRIT_{LINK}+CRIT_{BRIDGE} +CRIT_{INHERITANCE} +CRIT_{SIZE}$	The sum of all critical components

3. Proposal of a New testing Technique for CBS System

The technique starts by analysing software functional specifications that can be obtained from a UML diagram or a specification/design document. Then, test specifications are produced from the definition in software functional specifications.

Software functional specifications can be obtained from software requirement documents, software design documents, source code, or executable programs. A definition of interfaces of included components is usually introduced in the design document or in the source code. A component's interface can be categorized into import interfaces and supply interfaces [36]. Import interfaces are ports where the component discovers services from other components, and supply interfaces are ports where other components discover services provided by the component. UML diagrams may also be an important source for generating test cases. The UML diagrams usually are found in software specifications or design documents.

Test specifications use specific test adequacy criteria that are based on complexity/criticality metrics. A test adequacy criterion is a predicate that is used to establish sufficient testing on a software application[90]. Tests are sufficient if all elements defined in the criteria are covered. The test adequacy criteria can also be used to measure the progress of testing activities.

Information from the metrics is used to generate test cases and scenarios. Furthermore, the generated test cases/scenarios are executed, and the test results are analyzed by exercising information from the metrics. The overall approach is illustrated in Figure 1.

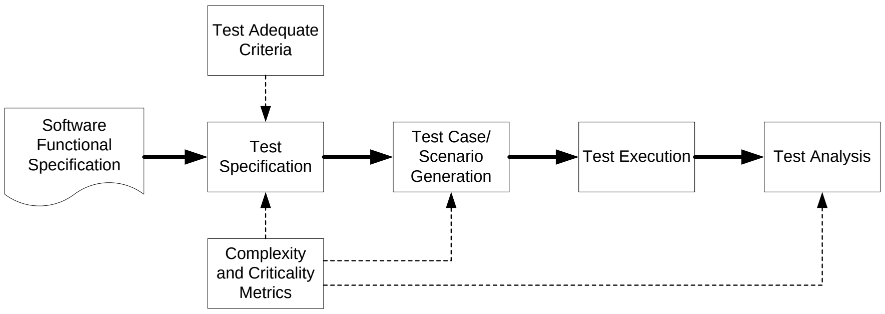
Test Adequate Criteria
Software Functional Specification
Test Specification
Test Case/ Scenario Generation
Test Execution
Test Analysis
Complexity and Criticality Metrics

Figure 1. Component Test using Component Integration metrics

To help create test cases, we can use a Category Partition Method[91]. The method has been discussed in a number of research studies [44], [60], [92]–[96] and the idea has also been used for supporting component-based testing [65], [97]. A test case contains a set of inputs and/or expected conditions to be tested on the program under test.

The Category Partition Method (CPM) is used as a strategy to reduce the number of test cases. CPM helps refine the functional specification into the categories and its environment conditions that may have an effect on the execution behavior of a function. Environment condition is a required property for a certain functional unit. Furthermore, several significant values called choices are selected from each category. A suite of test cases is obtained by putting together all possible combinations of choices for all categories. Some constraints are produced to prevent redundant, not meaningful or contradictory choices.

We use five test adequacy criteria, which may help the software tester to create test cases from a component-based system. The criteria are as follows:

Criterion A: Test all functional units, category choices, parameters and environmental conditions of a component.

Criterion B: Test all functional units, all category choices and all parameters of a component.

Criterion C: Test all functional units and all category choices of a component.

Criterion D: Test one category choice, one parameter and one environmental condition of a component.

Criterion E: Test all functional units and one category choice of a component.

The idea for these criteria was adopted from the four test adequacy criteria of Mahmood's research[25] .However the sequences and items to be tested are different because of the nature of the proposed metrics. Criterion A is the strongest criterion, whilst criterion E is the weakest.

To select the appropriate criterion, we use the component integration metrics. Previous research on complexity metrics has shown that complexity has an association with fault proneness[98]. Our proposed criticality metrics are expected to give an indication of the critical components in a CBS. Critical components are subjected to complete testing[56], [99], since they are most likely to be fault prone.

Each critical component deserves special attention in test case generation due to their specific characteristics. Size critical components have a tendency to have more faults because the bigger the size is, the more mistakes that may be introduced by the programmer [100]. Inheritance critical components also have a tendency to introduce error[31], [101]. Link critical components have more interactions with other components, therefore more testing is needed[38], [69]. Bridge critical components may cause the failure of the entire system if the faults are not found [102].

We suggest the following steps to determine a suitable test criterion by using the component integration metrics:

- Calculate CID for each component (CID_X) and CAID for the integration of components. A CID value indicates how a specific component interacts with other components, and a CAID value gives the overall picture of how components interact with each other;
- For each component, generate the value of nSize(x), nLink(x), nInheritance(x), and isBridge(x));
  - nSize(x) is the size of the component (by definition, the size can be the number functions, statements, methods or LOC[1])
  - nLink(x) is the number of links connected to the component
  - nInheritance(x) is the inheritance depth of a component
  - isBridge(x) is a true or false status of whether a component is a bridge or not.
  - The default values of nSize, nLink, and nInheritance are zero and isBridge is false.
- By comparing the value of nSize, nLink, nInheritance to a predefined threshold value, we can determine the criticality status of the component. IsBridge value could be determined subjectively by the software designer.
- For each component, the following rules are used:
  - Select Test adequacy criterion A if a component has more than one criticality **or** a component has one criticality and CID value > CAID value **or** a component is a bridge criticality.
  - Select Test adequacy criterion B, if a component is only size critical
  - Select Test adequacy criterion C, if a component is only inheritance critical **or** CID value > CAID value.
  - Select Test adequacy criterion D, if a component is only link critical
  - Select Test adequacy criterion E, if a component has no criticality and its CID value < CAID value.

The steps are illustrated in figure 2.

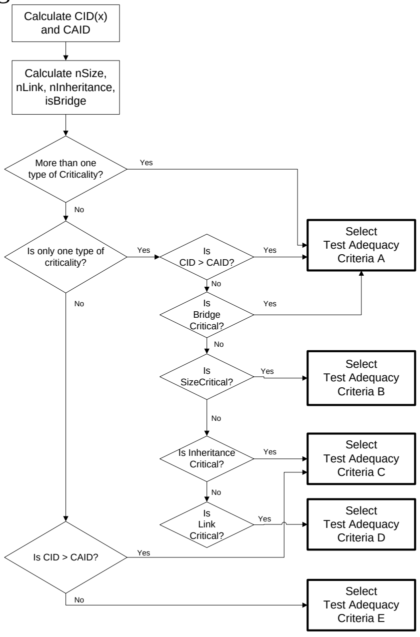
Calculate CID(x) and CAID
Calculate nSize, nLink, nInheritance, isBridge
More than one type of Criticality?
Yes
No
Is only one type of criticality?
Yes
Is CID > CAID?
Yes
No
Is Bridge Critical?
Yes
No
Is SizeCritical?
Yes
No
Is Inheritance Critical?
Yes
No
Is Link Critical?
Yes
Is CID > CAID?
Yes
No
Select Test Adequacy Criteria A
Select Test Adequacy Criteria B
Select Test Adequacy Criteria C
Select Test Adequacy Criteria D
Select Test Adequacy Criteria E

Figure 2. Determination of Test Adequacy Criteria

[1] *LOC can be used if source code is available or if the developer supplies this information.*

If a component has only one specific criticality, the testing is focused on this particular criticality's characteristic. Bridge criticality requires the strongest criteria as a bridge connects other components, and its role is important for the system. Size criticality involves the size and thus the testing effort focuses on testing all the functionalities and all categories and all of their parameters. Testing for inheritance criticality needs to focus on all functionalities and all their category choices. Testing for link criticality needs to concentrate on the calling of interfaces from/to other components and thus in their implementation requires examination of each category choice, its parameters and also the specific environment. A higher component interaction value, compared to the average interaction value, requires the testing of all functional units and its category choices.

4. Experimental Study

To validate the proposed technique we have designed the an experimental study. The experimental study consisted of the following steps:

1. Find a component-based application that allows us to see the source code, and has UML diagram and a set of testing suite.
2. Generate the metrics from UML design and component implementation of the application
3. Generate the list of test suites and test cases with and without proposed testing technique.
4. Analyze the result by comparing the original number of test suite/test cases to the new number of test suite/test cases using proposed technique.
5. Evaluate the result by examining the reduction of the number of test suite/test cases. This is the criteria that should demonstrate the efficiency of new technique.

By following the design above, we have used an application called Nomad PIM[2] (Personal Information Manager) to validate the use of component integration metrics on the testing process. Nomad PIM was a personal information management system that allows the recording of personal data such as schedule, contact, notes, personal finance, time tracking, and fitness measurement.

Nomad PIM was selected because it consisted of several components with a quite rich set of functionalities. Moreover, Nomad PIM was released as an open source software application, and thus we could perform a detailed examination of the system. In addition, it had a UML diagram and a set of testing suites. However, there was only one UML diagram (component diagram) and the test suites are available only for Schedule, Contact, TimeTracking, Note, Money and Core components. The component diagram is presented in Figure 3. A component in Nomad PIM consists of several classes and interfaces.

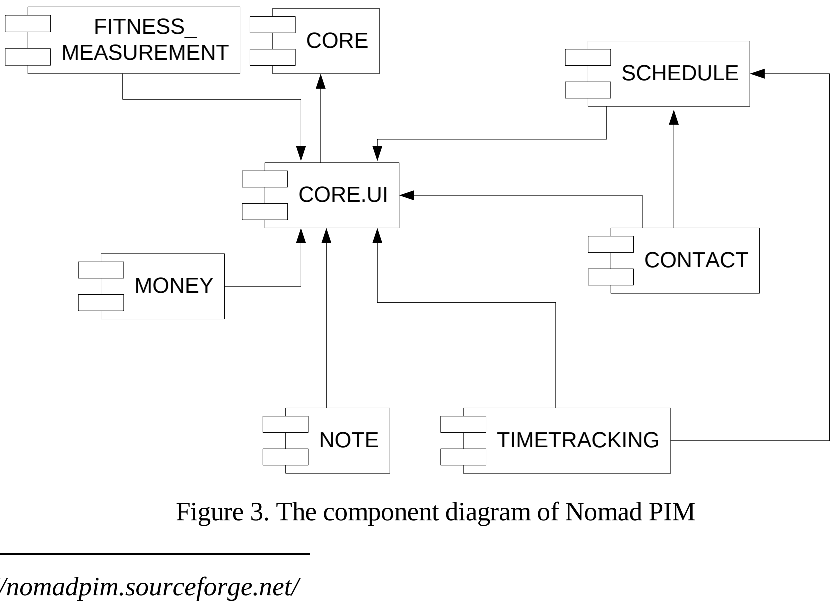

Figure 3. The component diagram of Nomad PIM

[2] *http://nomadpim.sourceforge.net/*

For this experimental study, we have performed the following activities:
1. Generate the metrics from UML design;
2. Generate the metrics from component implementation of Nomad PIM;
3. Generate the list of test suites and test cases of Nomad PIM;
4. Analyze the results by comparing the metrics' values to the number of test suites and test cases.

A. *The Metrics Generation*

The number of metrics that can be generated from their UML design is quite limited, as there is not much information provided in their design document. Table 1 shows the generation of the number of links from the UML design (Figure 3). The table clearly shows that the CORE.UI component is the closest candidate to become a link critical component. By our metric's definition, Link Criticality counts the number of components in which their links exceed a threshold number. The threshold number can be found by finding extreme values or the outliers using a boxplot calculation. Statistical calculation of the number of links found that CORE.UI is the only outlier, and thus it is concluded that the $CRIT_{LINK}$=1.

Table 2. The Metrics produced for the componentes

Component Name	#Link	CID	LOC	#Statements	#Classes and Interfaces	#Nmethod	Inheritance Level Maximum
CONTACT	2	1.00	322	125	8	23	3
CORE	1	0.94	1046 0	3791	210	785	2
CORE.UI	7	0.94	5574	2395	123	403	3
MONEY	1	1.00	2829	1166	56	219	2
NOTE	1	0.98	563	219	19	34	3
SCHEDULE	3	1.00	2577	1125	52	192	3
TIMETRACKING	2	0.98	2837	1260	56	193	4
FITNESS_ MEASUREMENT	1	0.99	427	192	8	21	4
		CAID=0.98					

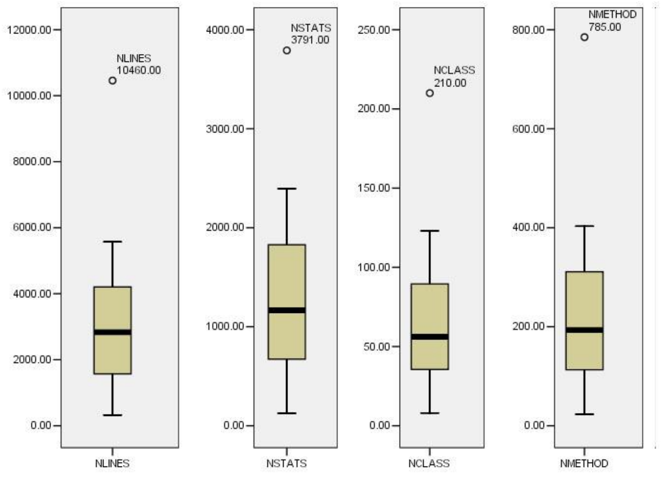

Figure 4. The Boxplot of Size Criticality Calculations of Nomad PIM

From this UML design, it is also obvious that the CORE.UI component is a candidate for a bridge component and thus we conclude that this design has $CRIT_{BRIDGE}$=1. However we could not generate other metrics (CID, CAID, Size Criticality, Inheritance Criticality or Bridge Criticality) because there was not enough information on this design. Nomad PIM is released as opensource, thus we can examine the source code, we can then calculate the components' interaction within

Nomad PIM using our Perl script. Source Monitor software tool is used to calculate the size in LOC, the number of statements, the number of classes/interfaces and the number of methods. The result is summarized in Table 2.

Boxplot diagrams of LOC, the number of statements, the number of classes and interfaces and the number of methods are shown in figure 4. The figure clearly reveals that the CORE component is the only outlier component compared to other components. Thus, we can conclude that $CRIT_{SIZE} = 1$.

To calculate link criticality, we have executed our Perl program to count the linkages between components in actual program code. We have found many links from and to a component. The top 10 large numbers of links are shown in Table 3.

Table 3. The Numbers of Links in Components

id	Component 1	Component 2	#link
53	CORE-UI	CORE	3864
155	TIMETRACKING	CORE	1663
79	CORE	CORE-UI	1658
115	MONEY	CORE	1307
141	SCHEDULE	CORE	1288
86	CORE	SCHEDULE	954
84	CORE	MONEY	870
59	CORE-UI	SCHEDULE	813
60	CORE-UI	TIMETRACKING	807
87	CORE	TIMETRACKING	737

Table 4. The List of Links from Critical Classes

Component	Java Class	#link
CORE	EntityContainer	427
	NewThreadExecutor	376
	NullRunnable	375
	SaveJob	376
	SimpleLazyMapWithDefault	315
	Space	325
CORE.UI	AbstractConvertingAction	379
	AbstractEntityActionDelegate	379
	AbstractEntityContainerView	380
	AbstractTextComponentAda	328
	CheckboxComponentAdapter	328
	ComboBoxComponentAdap	328
	ComponentAdapter	328
	DeleteAction	375
	DisplayAsyncExecutor	376
	EditorService	377
	InternalListContentProvider	416
	NomadPIMApplication	375
	OneDayBackActionDelegate	375

Component	Java Class	#link
CORE.UI	OneDayForwardActionDele.	375
	ShowViewActionDelegate	378
	TableContentProvider	384
	TableViewerFilterAction	375
	TodayActionDelegate	375
FITNESS_ME	AbstractFitnessMeasurement	386
MONEY	AccountListView	381
	CalculationContentProvider	385
	NullAccount	340
	TransactionsOutlineTableC	385
NOTE	DayView	384
	LiteratureNoteListView	376
	OpenDayInViewOperation	376
SCHEDULE	CalendarDateServiceListener	376
	PastEventsView	391
	ScheduleView	406
	WeekOverviewTreeContentP	389
	WeekOverviewView	379
TIMETRACK	CurrentActivitiesView	386

The statistical calculation could not find the outlier for this component, thus we can conclude that $CRIT_{LINK}= 0$. However, after further investigation into these components, we can extract the information from each component, and found that several classes have a higher link degree than other classes. By using the same calculation, we can find 38 classes where their links are the outliers from the rest of the links' dataset. By examination of the classes, we can conclude that $CRIT_{LINK}$=38 or there are 38 classes that have link criticality.

We can find that two components (Schedule and TimeTracking) have the level of inheritance 4. Our empirical investigation suggests that the component that has a level of inheritance 4 is critical. Thus, it is concluded that $CRIT_{INH}$ = 2. By further inspection of each component, we found classes that have high inheritance levels. **Error! Reference source not found.**shows the components and their classes that have inheritance levels 2, 3 and 4.

Table 5. Maximum Inheritance Level of Each Component

Component	Inheritance Level Maximum
CONTACT	3
CORE	2
CORE.UI	3
FITNESS_MEASUREMENT	2
MONEY	3
NOTE	3
SCHEDULE	4
TIMETRACKING	4

To find bridge criticality, we undertook manual analysis of the components and their classes, and concluded that CORE is the bridge component that is responsible to link other components. A fail in the CORE component most likely will interrupt the whole application. Thus, we determine that $CRIT_{BRIDGE}$=1. And finally, we can summarize that the criticalities of Nomad PIM are as follows:

- $CRIT_{SIZE}$=1
- $CRIT_{LINK}$=0
- $CRIT_{INH}$=2
- $CRIT_{BRIDGE}$=1
- $CRIT_{ALL}$ = 1 + 0 + 2 + 1 = 4

B. Test suite and test cases generation

Table 6. Test suites and Test Cases of a NOTE Component

Java Class Name	**Test Suite**	**Test Cases**
DiaryViewTest Class	1. DateShouldBeNormalized	Test 1/2/2000
	2. TestBug1115269	TestCurrentDate
	3. TestGetSameDaysAreEqual	Test First Day
		Test Second Day First Time
		Test Second Day Second Time
		Test Equal Second Day First time to Second day second time
		Test Inequality second day first time to FirstDay
		Test inequality of second day second time to FirstDay
TypeExtensionPluginTest Class in Note Folder	4. Test RegisteredSpacetypes1	TestDiary.TypeName
TypeExtensionPluginTest Class in Diary Folder	5. Test RegisteredSpacetypes2	TestDiary.TypeName
TypeExtensionPluginTest Class in WorkArea Folder	6. TestRegisteredSpacetypes3	TestDiary.TypeName

Nomad PIM comes with its own test suites. Each test suite has its own test cases. A component can have one or many test suites, and a test suite can have many test cases. Table 6 shows a sample of test suites and test cases from a NOTE component. The summary of the number of test suites and their test cases is listed in **Error! Not a valid bookmark self-reference.**.

Table 7. Test Suites and Test Cases of Nomad PIM

Component	TestSuite	TestCases
CONTACT	10	28
CORE	227	521
CORE.UI	45	84
FITNESS_MEASUREMENT	6	21
MONEY	81	184
NOTE	6	11
SCHEDULE	10	27
TIMETRACKING	14	31

C. *Analysis*

By examining the value of CID, we found that most of the components interact intensively to other components. It is a common phenomenon that in-house components are developed conjointly when the application is developed. Nomad PIM does not use external components, and thus all components are developed in-house. Therefore, their classes and functions are developed specifically from the requirements of Nomad PIM, and thus the CID values are high (above 90%). The size criticality metric shows one component is critical in size (LOC, classes, methods and statements). The table and boxplot diagram clearly demonstrate that the CORE component is a critical component. Correlation analysis between the size of LOC, Statements, Classes and Methods shows a significant relationship (significant at 0.01) to test suite and test cases (Table 8).

Table 8. Pearson Correlation of LOC, the Number of Statements, Classes, Methods, Test Suites and Test Cases

		NLOC	**NSTAT**	**NCLASSES**	**NMETH**	**TSUITE**	**TCASES**
NLOC	PC	1	.994(**)	.998(**)	.999(**)	.906(**)	.893(**)
	Sig.		.000	.000	.000	.002	.003
NSTAT	PC	.994(**)	1	.995(**)	.992(**)	.864(**)	.847(**)
	Sig.	.000		.000	.000	.006	.008
NCLASSES	PC	.998(**)	.995(**)	1	.996(**)	.891(**)	.875(**)
	Sig.	.000	.000		.000	.003	.004
NMETHODS	PC	.999(**)	.992(**)	.996(**)	1	.916(**)	.903(**)
	Sig.	.000	.000	.000		.001	.002
TSUITE	PC	.906(**)	.864(**)	.891(**)	.916(**)	1	.999(**)
	Sig.	.002	.006	.003	.001		.000
TCASES	PC	.893(**)	.847(**)	.875(**)	.903(**)	.999(**)	1
	Sig.	.003	.008	.004	.002	.000	

PC: Pearson Correlation

** Correlation is significant at the 0.01 level (2-tailed), a Listwise N=8

An outlier analysis of test suites and test cases shows that the CORE component is the outlier component. Thus, we can conclude that the $CRIT_{SIZE}$ metric can help find the critical component for helping the testing process.

We also generate the correlation of the number of Links to test Suites and test cases. Table 9 shows that the Links have a significant relationship to test Suites at less than 0.05 level, however the Links have only 0.051 significance level to test Cases. Based on our proposed examination of link criticality, there was no component at criticality. The number of Links on the CORE and

CORE.UI component is quite high, but not enough to be an outlier if compared with other numbers of component links. However, a tester still should pay extra attention to the existence of this high number of links, as it suggests a high connection with other components, and therefore requires more test suites and test cases.

Inheritance criticality analysis shows that the *TimeEvaluationView* class in the TIMETRACKING component and the *WeekOverView* class in the SCHEDULE component have the highest component inheritance levels in Nomad PIM. This means that high inheritance levels are present in the corresponding classes. Thus, the testing effort should focus on this particular class and their parents. Figure 3 shows the detail of their parents' classes.

Table 9. Pearson Correlation of Links to Test Suites and Test Cases.

		Test Suites	Test Cases
LINK	PC	.733(*)	.705
	Sig.	.039	.051

PC: Pearson Correlation
** Correlation is significant at the 0.05 level (2-tailed),*
a Listwise N=8

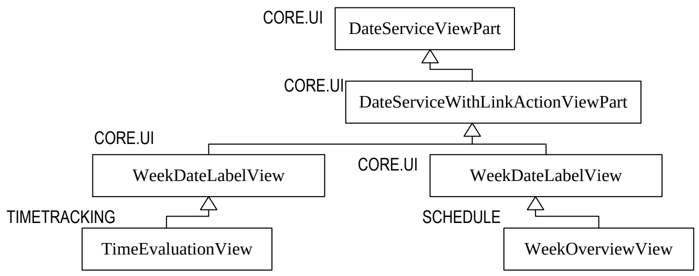

Figure 5. The Hierarchical View of Critical Inheritance Components

The implementation of Nomad PIM has demonstrated that it has a bridge criticality value of one and so does its design. However the actual bridge component is different. The UML class diagram showed CORE.UI is a bridge component, but the implementation revealed that CORE is the bridge component. We believe that this is a normal situation as the implementation sometimes varies from the design. However, we also think that the actual implementation is not much different from the design. In regards to the testing process, locating a bridge component also uncovers the need to put extra effort into building test suites and test cases.

Table 10. Reduction of Test Suites and Test Cases

	Before		After		Reduced percentage of TS	Reduced percentage of TC
Component	TS	TC	TS	TC		
CONTACT	10	28	9	12	10%	57%
CORE	227	521	227	521	0%	0%
CORE.UI	45	84	44	62	2%	26%
FITNESS_MEASUREMENT	6	21	4	4	33%	81%
MONEY	81	184	61	68	25%	63%
NOTE	6	11	4	5	33%	55%
SCHEDULE	11	27	11	18	0%	33%
TIMETRACKING	14	31	10	11	29%	65%
				Average	17%	47%

Note: TS: Test Suites, TC: Test Cases

By employing this new process, we can reduce the number of test suites and test cases from the original list. As shown in
table 10, the test suite has been reduced by an average of 17% and test cases by 47%. The CORE component is not reduced because this component is the most critical component, and thus requires all possible generations of test suites and test cases. The numbers of test cases are reduced substantially compared to test suites. The most reduction comes from components that are not critical.

5. Result and Analysis

This experimental study has shown that the metrics have implications on the generation of the test suites and test case:

- High interaction density implies high testing requirements. Therefore by choosing appropriate test adequacy criteria for choosing test cases can help reduce the effort.
- The size criticality metric shows a close relationship to the size to the number of test suites and test cases. These results confirm the research on the relationship between software size and software test case generation[36], [59], [74].
- In our case study, the link criticality of components could not find any component that is critical in their link to other components. However, information on their links helps in deducing the test suites and test cases.
- Further inspection of components that have high inheritance levels shows that the corresponding classes have shown the need to generate test suites and test cases that examine the child functionality that is linked to their parents.
- A component that has a bridge criticality is shown to have the largest number of test suites and test cases. Therefore, it is confirmed that finding a bridge component is important to a complete generation of test suites and test cases.

While conducting this experimental study, we also noticed the following findings:

- Discovering a bridge component was not difficult[3]. It was assisted by first generating some metrics such as CID and Link criticality metrics. Relatively small number of components involved in component integration may also assist in this discovery.
- When confirming the finding of bridge component, we had a better knowledge of the design and implementation of this application. In the testing process, this knowledge is important to generate suitable test suites and test cases.
- Generation of the metrics values requires extracting raw information from the design and source code. This information is kept in a repository and it helps us in examining the metrics result.

Our approach uses the integration component metrics to help reduce the number of test suites and test cases. The reduction is performed by evaluating the metrics' value of the complexity or the criticality of integrated components. A high complexity and/or high critical component should go through a more thorough testing process.

In regards to finding the faults, the metrics help finding the inter-component faults and also traditional or other faults. The detail is summarized in table 11.

Table 11. Fault type and related metrics.

Fault Type	The related metrics
Inter-component faults	Interaction Complexity Metrics, Link Criticality metrics, Inheritance criticality Metrics and Bridge criticality metrics.
Traditional/other faults	Size Criticality Metrics

[3] *Finding a bridge component may not be easy if we have less information about the system or if the system contains large numbers of components. However by understanding the nature of a system and the generation of other metrics can help finding the bridge component.*

Most of the work on reducing test suite and test case generation was performed manually. To generate the metrics, we have used some tools. In the future an integrated tool for CBS system should help the process.

The metrics are generated using a specific software application, which only examines the source code. The availability of the source code has helped us in building the test suites and test cases. The case tool can also keep the repository and therefore, a tester gets helped in defining the particular information. For example, an inspection of inheritance criticality requires finding the parent of each component/class.

On the other hand, the reduction of test suites and test cases risks the capability of detecting faults in the application, therefore a tester should always review the reduced test suites and test cases.

6. Conclusions

In this paper, we have proposed a new technique to test CBS system. We have conducted an experimental study to validate this technique. The technique use component integration metrics as part of the testing process. We have demonstrated that the new technique has helped in reducing the number of test suites and test cases by using information from the metrics. We also have learned that the generation of the metrics helps in understanding the software specification and furthermore it can be of assistance in the generation of test suites and test cases. There are five criteria that should be chosen based on the value of the metrics for each component. However, at this stage, the validation of the new technique had been performed using manual inspection, but it had been applied carefully. A CASE tool could be developed to help extract the metrics and produce the test suites and test cases based on the new technique. This testing technique can be used not only for the development of a component which may consists of several other components but also for the application development which uses components as part of their development strategy.

For the future work, we can have more application with more components that can help validate this metrics or could possible enhanced the technique. A domain specific or an industrial use case application could be used to validate the work. It may also be interesting to see how the proposed technique can be applied to different platforms such as web-based application or mobile-based application. We also suggest that a specific test procedure or technique or method can be possibly used to undertake testing activitities using the different CBS metrics with modified testing adequacy criteria.

7. References

[1]. C. Szyperski, D. Gruntz, and S. Murer, *Component software : beyond object-oriented programming*, 2nd ed. New YorkLondon ; Boston: ACM Press ;Addison-Wesley, 2002.

[2]. Object Management Group, “Corba Component Model Specification version 4.0 OMG formal document 06-04-01.” Object Management Group, 2006.

[3]. K.-K. Lau and Z. Wang, “Software Component Models,” *IEEE Trans. Softw. Eng.*, vol. 33, no. 10, pp. 709–724, 2007.

[4]. J. Sametinger, *Software engineering with reusable components*. Berlin ; New York: Springer, 1997.

[5]. B. Boehm and C. Abts, “COTS integration: Plug and pray?,” *IEEE Comput.*, vol. 32, no. 1, 1999.

[6]. A. W. Brown, *Large-Scale Component-Based Development*. Upper Saddle River, New Jersey: Prentice Hall PTR, 2000.

[7]. P. C. Clements, “From Subroutines to Subsystems: Component-Based Software Development,” in *Component Based Software Engineering*, A. W. Brown, Ed. Los Alamitos, California: Carnegie Mellon University - Software Engineering Institute - IEEE Computer Society Press, 1996.

[8]. C. Szyperski, D. Gruntz, and S. Murer, *Component software : beyond object-oriented programming*, 2nd ed. London ; Boston, MA: Addison-Wesley, 2003.

[9]. L. Brownsword, D. Carney, and P. Oberndorf, “The Opportunities and Complexities of Applying Commercial-off-the-Shelf Components,” *CrossTalk J. Def. Softw. Eng.*, vol. 11, no. 4, pp. 4–6, 1998.

[10]. D. J. Carney, E. J. Morris, and P. R. H. Place, “Identifying commercial off-the-shelf (COTS) product risks: the COTS usage risk evaluation,” Carnegie Mellon Software Engineering Institute (SEI), 2003.

[11]. V. B. Misic and D. N. Tesic, “Estimation of effort and complexity: An object-oriented case study,” *J. Syst. Softw.*, vol. 41, pp. 133–143, 1998.

[12]. D. Tran-Cao, A. Abran, and G. Levesque, “Functional Complexity Measurement,” *International Workshop on Software Measurement (IWSM’01)*. Montreal, Quebec, Canada, 2001.

[13]. D. P. Darcy and C. F. Kemerer, “Software complexity: Toward A Unified Theory of Coupling and Cohesion,” *ICSc Workshop Spring 2002*. 2002.

[14]. J. Shao and Y. Wang, “A New Measure of Software Complexity Based on Cognitive Weights,” *Canadian Conference on Electrical and Computer Engineering, 2003. IEEE CCECE 2003.* . 2003.

[15]. E. Lee, W. Shin, B. Lee, and C. Wu, “Extracting Components from Object-Oriented System: A Transformational Approach ,” *IEICE-Transactions Info Syst.*, vol. E88–D, no. 6, pp. 1178–1190, 2004.

[16]. V.Lakshmi Narasimhan and Bayu Hendradjaya, “Component Integration Metrics,” in *Proc. of the 2004 International Conference on Software Engineering Research and Practice (SERP’04)*, 2004

[17]. D. P. Darcy, C. F. Kemerer, S. A. Slaughter, and J. E. Tomayko, “The structural complexity of software an experimental test,” *IEEE Trans. Softw. Eng.*, vol. 31, no. 11, pp. 982–995, 2005

[18]. S. Mahmood and R. Lai, “Measuring the Complexity of a UML Component Specification ,” *Fifth International Conference on Quality Software (QSIC’05)*. IEEE Computer Society, 2005

[19]. S. Mahmood and R. Lai, “A Complexity Measure for UML Component-based System Specification,” *Softw. Pract. Exp.*, 2006.

[20]. S. Sedigh-Ali, A. Ghafoor, and R. A. Paul, “Metrics Guided Quality Management for Component-Based Software System,” in *Proceedings of 25th Annual International Computer Software and Application Conference (COMPSAC)*, 2001, pp. 303–308.

[21]. N. Pryce and S. Crane, “Component interaction in distributed systems,” in *Configurable Distributed Systems, 1998. Proceedings. Fourth International Conference on*, 1998, pp. 71–78.

[22]. N. R. Mehta, N. Medvidovic, and S. Phadke, “Towards a Taxonomy of Software Connectors,” *22nd International Conference on Software Engineering (ICSE ’00)*. 2000.

[23]. A. W. Williams and R. L. Probert, “A Measure for Component Interaction Test Coverage,” *ACS/IEEE International Conference on Computer Systems and Applications (AICCSA’01)*. 2001.

[24]. S. Becker, S. Overhage, and R. H. Reussner, *Classifying Software Component Interoperability Errors to Support Component Adaption*. Springer, 2004

[25]. S. Mahmood, R. Lai, and Y. S. Kim, “Survey of component-based software development,” *IET Softw.*, vol. 1, no. 2, pp. 57–66, 2007

[26]. S. R. Chidamber and C. F. Kemerer, “A Metrics Suite for Object-oriented Design,” *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, p. 476–493., 1994.

[27]. S. R. Chidamber and D. P. Darcy, “Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis,” *IEEE Trans. Softw. Eng.*, vol. 24, no. 8, pp. 629–639, 1998

[28]. D. Garlan, R. Allen, and J. Ockerbloom, “ Architectural mismatch: Why reuse is so hard,” *IEEE Softw.*, vol. 12, no. 6, 1995

[29]. C. Szyperski, *Component Software : Beyond Object-Oriented Programming*. New York: Addison-Wesley, 1998.

[30]. A. W. Brown and K. C. Wallnau, "Engineering of Component-Based System," *Second IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'96)*. p. 414, 1996.

[31]. C. Abts, B. W. Boehm, and E. B. Clark, "COCOTS: a COTS software integration cost model - model overview and preliminary data findings," USC Center for Software Engineering, 2000.

[32]. B. Boehm, *Software cost estimation with Cocomo II*. Prentice Hall, 2000.

[33]. L. Brownsword, T. Obendorf, and C. A. Sledge, "Developing New Processes for COTS-Based Systems," *IEEE Softw.*, pp. 48–55, 2000.

[34]. V. Lakshmi Narasimhan and Bayu Hendradjaya, "Some Theoretical Considerations for a Suite of Metrics for the Integration of Software Components," *Inf. Sci.*, vol. 177, no. 3, pp. 844–864, 2007.

[35]. E. Weyuker, "Testing component-based software: A cautionary tale," *IEEE Softw.*, vol. 15, no. 5, pp. 54–59, 1998.

[36]. J. Z. Gao, H.-S. . Tsao, and Y. Wu, *Testing and Quality Assurance for Component-Based Software*. Boston, London: Artech House, 2003.

[37]. J. D. McGregor and D. A. Sykes, *A Practical Guide To Testing Object-Oriented Software*. Addison-Wesley, 2001.

[38]. T. Parsons, A. Mos, M. Trofin, T. Gschwind, and J. Murphy, "Extracting Interactions in Component-Based Systems," *Softw. Eng. IEEE Trans.*, vol. 34, no. 6, pp. 783–799, 2008.

[39]. R. Abernethy, R. Morin, and J. Chahín, *COM/DCOM Unleashed*. Indianapolis, Ind.: Sams Publishing, 1999.

[40]. E. J. Weyuker, "The trouble with testing components," in *Component-based software engineering: putting the pieces together*, G. T. Heineman and W. T. Councill, Eds. Addison-Wesley Longman Publishing Co., Inc., 2001, pp. 499–512.

[41]. A. Orso, H. Do, G. Rothermel, M. J. Harrold, and D. S. Rosenblum, "Using component metadata to regression test component-based software," *Softw. Testing, Verif. Reliab.*, vol. 17, no. 2, pp. 61–94, 2007.

[42]. W. Ye, P. Dai, and C. Mei-Hwa, "Techniques for testing component-based software," in *Engineering of Complex Computer Systems, 2001. Proceedings. Seventh IEEE International Conference on*, 2001, pp. 222–232.

[43]. D. S. Rosenblum, "Adequate testing of component-based software," University of California at Irvine, Technical Report TR97-34, , 1997.

[44]. G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Softw. Testing, Verif. Reliab.*, vol. 12, no. 4, pp. 219–249, 2002.

[45]. J. Gao, R. Espinoza, and H. Jingsha, "Testing coverage analysis for software component validation," in *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, 2005, vol. 1, p. 463–470 Vol. 2.

[46]. Z. Jin and A. J. Offutt, "Coupling-based criteria for integration testing," *Softw. Testing, Verif. Reliab.*, vol. 8, no. 3, pp. 133–154, 1998.

[47]. I.-H. Cho and J. D. McGregor, "A formal approach to specifying and testing the interoperation between components," *Proceedings of the 38th annual on Southeast regional conference*. ACM, Clemson, South Carolina, pp. 161–170, 2000.

[48]. F. Belli and C. J. Budnik, "Towards self-testing of component-based software," in *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, 2005, vol. 2, p. 205–210 Vol. 1.

[49]. S. H. Edwards, "A framework for practical, automated black-box testing of component-based software," *Softw. Testing, Verif. Reliab.*, vol. 11, no. 2, pp. 97–111, 2001.

[50]. F. Barbier and N. Belloir, "Component behavior prediction and monitoring through built-in test. ," in *The 10th IEEE International Conference on Engineering of Computer-Based Systems (ECBS 2003)*, 2003, pp. 17–22.

[51]. S. Beydeda and V. Gruhn, "Merging components and testing tools: The self-testing COTS components (STECC) strategy," in *The 29th EUROMICRO conference (EUROMICRO 2003),* 2003, pp. 107–115.

[52]. H.-G. Gross and N. Mayer, "Built-In Contract Testing in Component Integration Testing," *Electron. Notes Theor. Comput. Sci.,* vol. 82, no. 6, pp. 22–32, 2003.

[53]. E. Martins, C. M. Toyata, and R. L. Yanagawa, "Constructing self-testable software components," in *IEEE International Conference on Dependable Systems and Networks (DSN 2001),* 2001, pp. 151–160.

[54]. M. Momotko and L. Zalewska, "Component+ built-in testing: a technology for testing software components," *Found. Comput. Decis. Sci.*, pp. 133–148, 2004.

[55]. Y. Wang, G. King, and H. Wickburg, "A method for built-in tests in component-based software maintenance," in *The 3rd European Conference on Software Maintenance and Reengineering (CSMR 1999),* 1999, pp. 186–189.

[56]. V. Lakshmi Narasimhan and B. Hendradjaya, "Some theoretical considerations for a suite of metrics for the integration of software components," *Inf. Sci. (Ny).*, vol. 177, no. 3, 2007.

[57]. C. S. Gall *et al.*, "Semantic software metrics computed from natural language design specifications," *Software, IET,* vol. 2, no. 1, pp. 17–26, 2008.

[58]. D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call Graph Construction in Object Oriented Languages.," in *Object Oriented Programming Systems, Languages and Applications,* 1997, pp. 108–124.

[59]. M. J. Harrold, D. Liang, and S. Sinha, "An Approach To Analyzing and Testing Component-Based Systems," *Workshop on Testing Distributed Component-Based Systems (ICSE'99).* 1999.

[60]. D. Jeffrey and N. Gupta, "Test suite reduction with selective redundancy," in *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05.*, 2005, pp. 549–558.

[61]. W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," *Softw. Pract. Exp.*, vol. 28, no. 4, pp. 347–369, 1998.

[62]. P. Jalote, *CMM in practice : processes for executing software projects at Infosys*. Reading, Mass ; Wokingham: Addison-Wesley, 2000.

[63]. Y. Wu, M.-H. Chen, and J. Offutt, "UML-Based Integration Testing for Component-Based Software," 2003, pp. 251–260.

[64]. F. Jabeen and M. Jaffar-ur-Rehman, "A framework for object oriented component testing," in *Emerging Technologies, 2005. Proceedings of the IEEE Symposium on*, 2005, pp. 451–460.

[65]. Y.-S. Ma, S.-U. Oh, D.-H. Bae, and Y.-R. Kwon, "Framework for third party testing of component software.," in *Eighth Asia-Pacific Software Engineering Conference,* 2001, pp. 431–434.

[66]. D. Liang and K. Xu, "Test-driven component integration with UML 2.0 testing and monitoring profile," in *Proceedings - International Conference on Quality Software*, 2007, pp. 32–39.

[67]. A. Elsafi, D. N. A. Jawawi, and A. Abdelmaboud, "Inferring approximated models for integration testing of component-based software," in *2014 8th Malaysian Software Engineering Conference, MySEC 2014,* 2014, pp. 67–71.

[68]. F. Berzal, I. Blanco, J.-C. Cubero, and N. Marin, "Component-based Data Mining Frameworks," *Commun. ACM,* vol. 45, no. 12, 2002.

[69]. L. Gallagher and J. Offutt, "Automatically Testing Interacting Software Components," *Workshop on Automation of Software Test (AST 2006).* Shanghai, China., pp. 57–63, 2006.

[70]. R. V Binder, *Testing Object-oriented systems: Models, Patterns, and Tools*. Boston: Addison-Wesley, 2000.

[71]. J. S. Kang and H. S. Park, "Automatic generation algorithm of expected results for testing of component-based software system," in *Information and Software Technology,* 2015, vol. 57, no. 1, pp. 1–20.

[72]. P. Braione, G. Denaro, A. Mattavelli, M. Vivanti, and A. Muhammad, “Software testing with code-based test generators: Data and lessons learned from a case study with an industrial software component,” *Softw. Qual. J.*, vol. 22, no. 2, pp. 311–333, 2014.

[73]. F. Saglietti and F. Pinte, “Automated unit and integration testing for component-based software systems,” in *Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systems - S&D4RCES ’10*, 2010, p. 1.

[74]. A. Orso, M. J. Harold, D. Rosenblum, G. Rothermel, M. Lou Soffa, and H. Do, “Using Component Metacontent to Support the Regression Testing of Component-Based Software,” *Proceedings of the IEEE International Conference on Software Maintenance (ICSM’01)*. IEEE Computer Society, p. 716, 2001.

[75]. Z. Jiang, L. Williams, B. Robinson, and K. Smiley, “Regression Test Selection for Black-box Dynamic Link Library Components,” in *Incorporating COTS Software into Software Systems: Tools and Techniques, 2007. IWICSS ’07. Second International Workshop on*, 2007, p. 9.

[76]. F. R. C. Silva, E. S. Almeida, and S. R. L. Meira, “An approach for component testing and its empirical validation,” *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, Honolulu, Hawaii, pp. 574–581, 2009.

[77]. J. Grundy, G. Ding, and J. Hosking, “Deployed software component testing using dynamic validation agents,” *J. Syst. Softw.*, vol. 74, no. 1, pp. 5–14, 2005.

[78]. M. Imran, A. Raza, and K. Asghar, “Component Based Software Testing Strategies to Develop Good Software Product,” in *1st International Conference on Emerging & Engineering Technologies (ICEET-2014)*, 2015, vol. 1.

[79]. E. Alegroth, Z. Gao, R. Oliveira, and A. Memon, “Conceptualization and evaluation of component-based testing unified with visual GUI testing: An empirical study,” in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015 - Proceedings*, 2015.

[80]. O. E. K. Aktouf, T. Zhang, J. Gao, and T. Uehara, “Testing location-based function services for mobile applications,” in *Proceedings - 9th IEEE International Symposium on Service-Oriented System Engineering, IEEE SOSE 2015*, 2015, vol. 30, pp. 308–314.

[81]. J.-F. Weber, “Tool Support for Fuzz Testing of Component-Based System Adaptation Policies,” in *International Workshop on Formal Aspects of Component Software*, 2016, pp. 231–237.

[82]. A. Sharma, P. S. Grover, and R. Kumar, “Dependency analysis for component-based software systems,” *SIGSOFT Softw. Eng. Notes*, vol. 34, no. 4, pp. 1–6, 2009.

[83]. A. Sharma, R. Kumar, and P. S. Grover, “A Critical Survey of Reusability Aspects for Component-Based Systems,” in *PROCEEDINGS OF WORLD ACADEMY OF SCIENCE, ENGINEERING AND TECHNOLOGY*, 2007, vol. 21.

[84]. H. Washizaki, H. Yamamoto, and Y. Fukazawa, “Metrics Suite for Measuring Reusability of Software Components,” in *9th International Software Metrics Symposium*, 2003, pp. 211–223.

[85]. S. Sedigh-Ali, A. Ghafoor, and R. A. Paul, “Metrics-Based Framework for Decision Making in COTS-Based Software Systems,” in *7th IEEE International Symposium on High Assurance Systems Engineering (HASE’02)*, 2002.

[86]. S. Mahmood and R. Lai, “A complexity measure for UML component-based system specification,” *Softw. - Pract. Exp.*, vol. 38, no. 2, pp. 117–134, 2008.

[87]. E. S. Cho, M. S. Kim, and S. D. Kim, “Component Metrics to Measure Component Quality,” in *The 8th Asia-Pacific Software Engineering Conference (APSEC)*, 2001, pp. 419–426.

[88]. O. P. Rotaru and M. Dobre, “Reusability metrics for software components,” in *The 3rd ACS/IEEE International Conference onComputer Systems and Applications, 2005.*, 2005, pp. 85–92.

[89]. L. Kharb and R. Singh, “Complexity Metrics for Component-oriented Software Systems,” *SIGSOFT Softw. Eng. Notes*, vol. 33, no. 2, p. 4:1-4:3, 2008.

[90]. J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *Proceedings of the international conference on Reliable software*. ACM, Los Angeles, California, pp. 493–510, 1975.

[91]. T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating fuctional tests," *Commun. ACM,* vol. 31, no. 6, pp. 676–686, 1988.

[92]. A. Andrews, R. France, S. Ghosh, and G. Craig, "Test adequacy criteria for UML design models," *Softw. Testing, Verif. Reliab.*, vol. 13, no. 2, pp. 95–127, 2003.

[93]. A. Bertolino and S. Gnesi, "PLUTO: A Test Methodology for Product Families," *5th International Workshop of Software Product-Family Engineering* - . Springer, Siena, Italy., pp. 181–197, 2004.

[94]. T. Y. Chen, P. Pak-Lok, and T. H. Tse, "A choice relation framework for supporting category-partition test case generation," *Softw. Eng. IEEE Trans.*, vol. 29, no. 7, pp. 577–593, 2003.

[95]. M. Grindal, B. Lindstr, J. Offutt, and S. F. Andler, "An evaluation of combination strategies for test case selection," *Empir. Softw. Engg.*, vol. 11, no. 4, pp. 583–611, 2006.

[96]. M. Grindal, B. Lindström, J. Offutt, and S. F. Andler, "An evaluation of combination strategies for test case selection," *Empir. Softw. Eng.*, vol. 11, no. 4, pp. 583–611, 2006.

[97]. J. Hartmann, C. Imoberdorf, and M. Meisinger, "UML-based integration testing," in *ACM Sigsoft Software Engineering Notes*, 2000, vol. 25, no. 5, pp. 60–70.

[98]. M. Alshayeb and W. Li, "An Empirical Validation of Object-Oriented Metrics in Two Different Iterative Software Process," *IEEE Trans. Softw. Eng.*, vol. 29, no. 11, 2003.

[99]. P. V Bhansali, "A systematic approach to identifying a safe subset for safety-critical software," *SIGSOFT Softw. Eng. Notes*, vol. 28, no. 4, p. 1, 2003.

[100]. N. Ohlsson, A. Eriksson, and M. Helander, "Early Risk-Management by Identification of Fault-prone Modules," *Empir. Softw. Eng.*, vol. 2, no. 2, pp. 166–173, 1997.

[101]. W. M. P. van der Aalsta, K. M. van Heeb, and R. A. van der Toornb, "Component-based software architectures: a framework based on inheritance of behavior," *Sci. Comput. Program.*, vol. 42, no. 2–3, pp. 129–171, 2002.

[102]. V. L. Narasimhan and B. Hendradjaya, "Some Theoretical Considerations for a Suite of Metrics for the Integration of Software Components," *J. Inf. Sci.*, vol. 177, no. 3, pp. 844–864, 2007.

Bayu Hendradjaya received B.Eng and M.Eng degrees in Informatics Engineering from Institut Teknologi Bandung. He holds a PhD in software engineering from La Trobe University. His research involved software requirements, software process improvement, software V & V, software development methodology and e-Government system. Dr. Bayu Hendradjaya is at School of Electrical Engineering and Informatics, Institut Teknologi Bandung, Jl. Ganesha 10 Bandung, Indonesia 40132 or at bayu@stei.itb.ac.id.