2013 10th International Conference on Information Technology: New Generations

Towards Test Focus Selection for Integration Testing using Method Level Software Metrics

Shadi Banitaan, Mamdouh Alenezi, Kendall Nygard, and Kenneth Magel
Department of Computer Science
North Dakota State University
Fargo, ND 58108, USA
shadi.banitaan, mamdouh.alenezi, kendall.nygard, kenneth.magel@ndsu.edu

***Abstract*—The aim of integration testing is to uncover errors in the interactions between system modules. However, it is generally impossible to test all the interactions between modules because of time and cost constraints. Thus, it is important to focus the testing on the connections presumed to be more error-prone. The goal of this research is to guide quality assurance team wherein a software system to focus when they perform integration testing to save time and resources. In this work, we use method level metrics that capture both dependencies and internal complexity of methods. In addition, we build a tool that calculates the metrics automatically. We also propose an approach to select the test focus in integration testing. The main goal is to reduce the number of test cases needed while still detecting at least 80% of integration errors. We conducted an experimental study on several Java applications taken from different domains. Error seeding technique have been used for evaluation. The experimental results showed that our proposed approach is very effective for selecting the test focus in integration testing. It reduces considerably the number of required test cases while at the same time detects at least 80% of integration errors.**

***Index Terms*—Integration Testing, Test Case Reduction, Software Metrics, Test Focus Selection**

I. Introduction

Software testing is the process of executing a program with the intent of finding errors [1]. Software testing is very costly. It requires approximately 50% of software development cost [1]. Many programs contain large number of errors. One reason these errors persist through the software development life cycle is the restriction of testing resources. These resources are restricted by many factors such as time (e.g., the software should be delivered in specific time) and cost (e.g., testing the whole software system requires a large team). Thus, if testing effort can be focused on the parts of a software system where errors are most likely to occur, then the available resources can be used more effectively, and the produced software system will be more reliable at lower cost.

Object-oriented software systems contain large number of modules which make the testing process very difficult and time-consuming. The aim of integration testing is to uncover errors in the interactions between system modules. Correct functioning of object-oriented software depends upon the successful integration of classes. While individual classes may function correctly, several new errors can arise when these classes are integrated together. However, it is usually very difficult to test all the connections between classes. Therefore, it is important to focus the testing on the interactions that are more critical and faulty.

The goal of this research is to reduce both cost and time required for integration testing. We are aiming to provide testers with an accurate assessment of which connections are most likely to contain errors, so they can regulate the testing efforts to target these dependencies. Our assumption is that using a small number of test cases to test the highly ranked error-prone dependencies will detect most of the integration errors. This work presents an approach to select the test focus in integration testing. It uses method-level software metrics, such as coupling, to give a weight for each method-pair dependency and then to predict the number of test cases needed to test each dependency. A dependency is a relationship between two components where changes to one may have an impact that will require changes to the other [2]. For example, we say that component A depends on component B. We also say that A has outbound dependency and B has inbound dependency. A component is a dependent to another component if it has outbound dependency on that component. A component is a dependee to another component if it has inbound dependency from that component.

In general, there are two main types of dependencies according to the dependency extraction method: static and dynamic. Static dependencies are extracted from binary files while dynamic dependencies are extracted during run-time. In our work, we extract static dependencies.

To summarize, we make the following key contributions in this work:

1) Use method-level object-oriented metrics. The metrics are selected according to the following two assumptions: A) the degree of dependencies between the two methods that have an interaction are strongly correlated with the number of integration errors between the methods; B) the internal complexity of the two method that have an interaction are strongly correlated with the number of integration errors in the interaction between them. We use four method level dependency metrics and three method level internal complexity metrics.
2) Build a tool to calculate the metrics automatically. We develop a tool using R language to compute the metrics automatically.

978-0-7695-4967-5/13 $26.00 © 2013 IEEE
DOI 10.1109/ITNG.2013.55

CPS Conference Publishing Services

3) Propose an approach to reduce cost and time of integration testing. We use a combination of object-oriented metrics to give a weight for each method-pair connection. Then, we predict the number of test cases needed to test each connection. The objective is to reduce the number of test cases needed to a large degree while still detecting at least 80% of integration errors.
4) Conduct an experimental study on several Java applications taken from different domains.

The rest of the paper is organized as follows: Section II discusses related work. Section III describes the proposed approach. The experimental evaluation and discussion are presented in Section IV. Section V concludes the paper.

II. Related Work

Class fault-proneness can be defined as the number of faults detected in a class. There is much research on building fault-proneness prediction models using different sets of metrics in object-oriented systems [3], [4], [5], [6]. The metrics are used as independent variables and fault-proneness is the dependent variable. In [3], the authors identified polymorphism measures to predict fault-prone classes. Their results showed that their measures can be used at early phases of the product life cycle as good predictors of its quality. The results also showed that some of the polymorphism measures may help in ranking and selecting software artifacts according to their level of risk given the amount of coupling due to polymorphism. Briand et al. [4] empirically investigated the relationships between object-oriented design measures and fault-proneness at the class level. They defined fault-proneness as the probability of detecting a fault in a class. Their results showed that coupling induced by method invocations, the rate of change in a class due to specialization, and the depth of a class in its inheritance hierarchy are strongly related to the fault-proneness in a class. Their results also showed that using some of the coupling and inheritance measures can be used to build accurate models for class-proneness prediction.

Predicting fault-prone classes can be used in the unit testing process such that the testers can focus the testing on the faulty classes. On the other hand, identifying fault-prone classes can not be used effectively in the integration testing process because we do not know what the fault-prone connections are. Therefore, we need to identify error-prone connections between method-pairs in order to focus the testing on them.

Zimmermann and Nagappan [7] proposed the use of network analysis on dependency graphs such as closeness measure to identify program parts which are more likely to contain errors. They investigated the correlation between dependencies and defects for binaries in Windows Server 2003. Their experimental results show that network analysis measures can detect 60% of binaries that are considered more critical by developers. They mentioned in their paper that most complexity metrics focus on single components and do not take into consideration the interactions between elements. In our work, we take the interactions between methods in our consideration. We use metrics that define dependencies on both method level and method-pair level to identify the interactions that are more likely to contain integration errors.

Borner and Paech [8] presented an approach to select the test focus in the integration testing process. They identified the correlations between dependency properties and the number of errors in both the dependent and the independent files in the previous versions of a software system. They used information about the number of errors in dependent and independent files to identify the dependencies that have a higher probability to contain errors. The main differences between our approach and Borner and Paech approach are the followings:

- Their approach just works for systems that have previous versions while our approach does not need previous versions of the system under test.
- Their approach predicts the error-prone dependencies while our approach not only predicts the error-prone dependencies but also predicts the number of test cases needed to test each dependency. In addition, we rank dependencies by giving a weight for each dependency in which a dependency with a higher weight is assigned more test cases than a dependency with a lower weight.
- They identify the correlations between the dependency properties and the number of errors in the dependent and the independent file. In our work, we do not identify the correlations between our metrics and the number of errors in files because we are working on the method level and information about number of errors at the method level is not available. In our view, identifying the correlations with the number of errors in files is not an accurate measure because errors belong to different classes inside the file and we cannot tell in which class the errors reside.

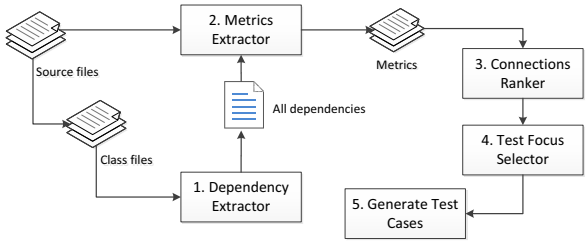

Fig. 1. An overview of the proposed approach.

III. The Proposed Approach

In this section, we discuss the proposed approach. Figure 1 provides an overview of the proposed approach. Our approach is divided into five steps. First of all, the dependency extractor extracts the dependencies from the compiled Java code. The result of this step is an XML file that contains the dependencies at method and class levels. After that, metrics extractor calculates the metrics using both the source code and the dependencies. The output of this step is the metrics at method level and method-pair level. Then, a weight is calculated for each method-pair using a combination of the metrics defined in

the previous step. The ranks of the connections indicate which connections should be focused on during the testing process. Next, the test focus selector selects the error-prone connections as a test focus and predicts the number of test cases needed to test each method-pair connection based on the ranks of the connections produced in the previous step and given the initial minimum number of test cases needed. The last step is to generate test cases manually to test the required application. The following sections explain these steps in detail.

A. System Representation

We first define a representation for a software system.

Definition 1: (System, Classes, Methods). A software system S is an object-oriented system. S has a set of classes $C = \{c_1, c_2, \cdots, c_n\}$. The number of classes in the system is $n = |C|$. A class has a set of methods. For each class $c \in C, M(c) = \{m_1, m_2, \cdots, m_t\}$ are the set of methods in c, where $t = |M(c)|$ is the number of methods in a class c. The set of all methods in the system S is denoted by $M(S)$.

B. Dependency Extractor

The dependency extractor extracts dependencies from Java class files. It detects three levels of dependencies: 1) class to class; 2) feature to class; and 3) feature to feature. The term feature indicates class attributes, constructors, or methods. We use the Dependency Finder tool [9] to extract the dependencies. In our work, we are just considering dependencies between methods. We consider two types of dependencies; method to method dependency and method to field dependency. We use the dependencies to define some of the metrics in the next section.

C. Metrics Extractor

The aim of this step is to identify set of metrics and to extract those metrics automatically from both the dependencies produced in the previous step and the source code. This section describes the software metrics that we use in our work. The metrics are defined for object-oriented systems. We define the metrics on both method level and method-pair level. Those metrics cover the dependencies between methods and the internal complexity of the methods. We consider the dependency metrics because errors are found during integration testing exactly where couplings typically occur [10]. In addition, complex methods are mostly more error prone.

1) Method Level Metrics: We define metrics on individual methods within a class. For method m_i in class c_i, we define the following metrics.

- Inbound Method Dependencies (IMD): Methods in other classes that depend on method m_i.
- Outbound Method Dependencies (OMD): Methods in other classes that method m_i depends on.
- Outbound Field Dependencies (OFD): Fields in other classes that method m_i depends on.
- Local Variables ($LVAR$): The number of local variables used by method m_i.
- Number of Complex Method Parameters ($NOCMP$): The number of complex input parameters in method m_i. The input parameter is complex if it is not a primitive type.
- Maximum Nesting Depth (MND): The maximum depth of nesting in method m_i. This metric represents the maximum nesting level of control constructs (if, for, while, and switch) in the method.

2) Method Pair Metrics: We define the following metrics on the method pair (m_i, m_j) where m_i is a dependent and m_j is a dependee, $m_i \in c_i$, $m_j \in c_j$ where $c_i \neq c_j$.

- Inbound Common Method Dependencies ($ICMm_im_j$): Number of common methods that depend on both m_i and m_j.
- Outbound Common Method Dependencies ($OCMm_im_j$): Number of common methods that both m_i and m_j depends on.

D. Connections Ranker

In this step, we use a combination of metrics defined in the previous step to calculate the weight for each method-pair. For method-pair (m_i,m_j), the weight for the connection between m_i and m_j is calculated as follows:

$$weight(m_i, m_j) = (weight(m_i) + weight(m_j)) \times (ICMm_im_j + OCMm_im_j + 1)$$

Both ICM metric and OCM metric indicate the indirect dependency between the two methods. We give more weight for the connection if there are common dependencies between the two methods as appeared in the previous equation. In addition, the ICM and OCM metrics do not contribute to the weight if the value for both of them is zero.

The weight of method m_i ($weight(m_i)$) is calculated as follows:

$$weight(m_i|c_k, c_l) = \frac{IC_{m_i} \times (IMDm_i + OMDm_i + OFDm_i)^2}{\sum_{y \in M(c_k, c_l)} IC_y \times (IMDy + OMDy + OFDy)^2}$$

where $M(c_k, c_l)$ is the set of methods in both c_k class and c_l class. As we see in the previous equation, the weight for a method depends on: 1) method to method dependencies (IMD, OMD); 2) method to field dependencies (OFD); and 3) the internal complexity of the method (IC). The denominator is used to get a normalized weight. IC_{m_i} is the internal complexity of method m_i and it is measured as follows:

$$IC_{m_i} = MNDm_i + NOCMPm_i + LVARm_i$$

E. Test Focus Selector

The Test focus selector predicts the number of test cases needed to test each connection based on the weights of the connections produced in the previous step and given the initial minimum number of test cases needed. We would like to start with a small initial number of test cases. In our work, we assume the initial number of test cases needed to test a system S to be $\frac{|M(S)|}{2}$. After that, the number of test cases can be

adjusted depending on the error discovery rate. For example, if the initial number of test cases to test a system was 50 and the error discovery rate was 60%, then we generate and run more test cases until we reach 80% of error discovery rate. The method-pair weight is computed using the equation in the previous section. The class-pair weight is computed as the summation of all method-pair weights that belong to the two classes. If the class-pair weight is zero, we specify one test case to test the class-pair connections. Figure 2 explains the testing process used in our approach. We start by creating w test cases where w is the initial number of test cases given by the approach. We then run the test cases against the seeded versions of the applications and we compute the error discovery rate. We stop if we achieve 80% error discovery rate. Otherwise, we create more test cases until the 80% is achieved.

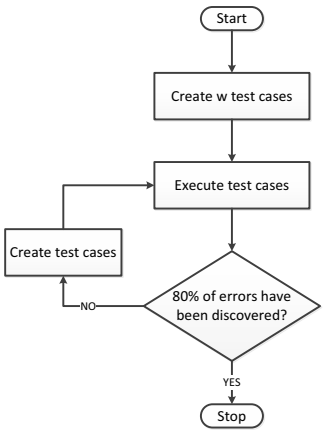

Fig. 2. Our testing approach.

F. Generate Test Cases

In this step, test cases are created manually to test the required application. For evaluating the testing process, error seeding technique [11] is used where a number of seeded integration errors are inserted manually into the source code by a third party. The assumption is that if the test cases found $y\%$ of the seeded errors, those test cases can find $y\%$ of the actual errors.

G. Tool Implementation

We build a **S**oftware **M**etrics for **I**ntegration **T**esting (SMIT) tool to calculate the metrics automatically. The tool is developed using R programming language [12]. Binary versions of this language are available for several platforms including Windows, Unix and MacOS. It is not only free but also an open source. R provides powerful functions to deal with regular expressions. Our tool uses the DependencyFinder tool to extract dependencies from compiled Java code. Our tool calculates the metrics using both the dependencies and the source files. The outputs of our tool are four files in CSV format: method level file, method-pair level file, class-pair level file and a final report that specifies the number of test cases needed to test each method-pair interaction.

IV. Experimental Evaluation

In order to evaluate the proposed approach, we select open-source applications implemented in Java. Table II shows a summary of the selected applications. The selected applications vary in size and domain and can be downloaded freely from the Internet. The Monopoly application presents a Monopoly-like computer game. Monopoly provides many features that appear in the Monopoly board game. The CruiseControl application simulates a car engine and its cruising controller, and PureMVC is a light weight framework for creating applications based on the classic Model, View and Controller concept. There are two versions of the PureMVC framework: Standard and MultiCore. We use the standard Version that provides a methodology for separating the coding interests according to the MVC concept. The Cinema application is a cinema management system that manages tickets booking and movie schedules. The ApacheCli is a library that provides an API for parsing command line options passed to programs.

TABLE II
A SUMMARY OF THE SELECTED APPLICATIONS.

Project	# of classes	# of methods	Source
Monopoly	57	336	http://realsearchgroup.com/rose/
CruiseControl	4	35	http://sir.unl.edu
PureMVC	22	139	http://puremvc.org/
Cinema	10	106	http://alarcos.esi.uclm.es
ApacheCli	20	207	http://commons.apache.org/cli/

Table III shows part of the method-level metrics output for the CruiseControl application. For example, the number of local variables for CarSimulator.run() is 4, the maximum nesting depth is 3, and the number of outbound method dependencies is 2. Table IV shows part of the method-pair level metrics output for the CruiseControl application. The last column in the Figure represents the weight of the method pair. Table V shows the class-level output for the CruiseControl application. Table VI shows part of the final report for the CruiseControl application. For example, the approach suggests to write one test case to test the method-pair (CruiseControl.handleCommand(),CarSimulator.accelerate()). We start with the initial number of test cases needed according to our approach and we stop after the first iteration because the created test cases detect at least 80% of the errors.

We use error seeding technique in order to evaluate our approach. Integration errors are placed in the source code by a third party. The third party injects the following types of integration errors [13]:

- Wrong function error: it occurs when the functionality provided by the dependee module is not the required functionality by the dependent module.
- Extra function error: an extra function error happens when the dependee module contains functionality that is not required by the dependent module.

TABLE I
RESULTS.

Application	# of seeded errors	Error type	Detected	Not Detected	Error detection rate
CruiseControl	10	Wrong function errors	10	0	100.00%
CruiseControl	4	Miscoded Call Errors (Missing instruction)	3	1	75.00%
CruiseControl	3	Miscoded Call Errors (Extra call instruction)	3	0	100.00%
Total	**17**		**16**	**1**	**94.12%**
Monopoly	12	Wrong function errors	11	1	90.91%
Monopoly	7	Miscoded Call Errors (Missing instruction)	7	0	100.00%
Monopoly	2	Miscoded Call Errors (Extra call instruction)	1	1	50.00%
Total	**21**		**19**	**2**	**90.48%**
PureMVC	10	Wrong function errors	8	2	80.00%
PureMVC	8	Miscoded Call Errors (Missing instruction)	7	1	87.50%
PureMVC	1	Miscoded Call Errors (Extra call instruction)	1	0	100.00%
Total	**19**		**16**	**3**	**84.21%**
Cinema	11	Wrong function errors	10	1	90.91%
Cinema	6	Miscoded Call Errors (Missing instruction)	5	1	83.33%
Cinema	3	Miscoded Call Errors (Extra call instruction)	2	1	66.67%
Total	**20**		**17**	**3**	**85.00%**
ApacheCli	10	Wrong function errors	8	2	80.00%
ApacheCli	8	Miscoded Call Errors (Missing instruction)	7	1	87.50%
ApacheCli	4	Miscoded Call Errors (Extra call instruction)	4	0	100.00%
Total	**22**		**19**	**3**	**86.36%**

TABLE III
PART OF THE METHOD-LEVEL OUTPUT FOR THE CRUISECONTROL APPLICATION.

	LVAR	NOCMP	MND	IMD	OMD	OFD
CarSimulator.CarSimulator()	1	0	0	1	1	0
CarSimulator.accelerate()	1	0	2	1	0	0
CarSimulator.brake()	1	0	2	1	0	0
CarSimulator.engineOff()	1	0	0	1	0	0
CarSimulator.engineOn()	1	0	1	1	2	0
CarSimulator.getBrakepedal()	1	0	0	0	0	0
CarSimulator.getDistance()	1	0	0	0	0	0
CarSimulator.getIgnition()	1	0	0	0	0	0
CarSimulator.getSpeed()	1	0	0	2	0	0
CarSimulator.getThrottle()	1	0	0	0	0	0
CarSimulator.run()	4	0	3	0	2	1
CarSimulator.setThrottle(double)	2	0	1	1	0	0
Controller.Controller(CarSimulator)	2	1	0	1	2	0
Controller.accelerator()	1	0	1	1	1	0
Controller.brake()	1	0	1	1	1	0
Controller.engineOff()	1	0	1	1	1	0

TABLE IV
PART OF THE METHOD-PAIR LEVEL OUTPUT FOR THE CRUISECONTROL APPLICATION.

Dependent	Dependee	OCM	ICM	Weight
CruiseControl.CruiseControl()	CarSimulator.CarSimulator()	1	0	0.048
CruiseControl.handleCommand(String)	CarSimulator.accelerate()	0	0	0.806
CruiseControl.handleCommand(String)	CarSimulator.brake()	0	0	0.806
CruiseControl.handleCommand(String)	CarSimulator.engineOff()	0	0	0.802
CruiseControl.handleCommand(String)	CarSimulator.engineOn()	0	0	0.833
SpeedControl.recordSpeed()	CarSimulator.getSpeed()	0	0	0.034
SpeedControl.run()	CarSimulator.getSpeed()	0	0	0.246
SpeedControl.run()	CarSimulator.setThrottle(double)	0	0	0.242
CruiseControl.CruiseControl()	Controller.Controller(CarSimulator)	1	0	0.135
CruiseControl.handleCommand(String)	Controller.accelerator()	0	0	0.822
CruiseControl.handleCommand(String)	Controller.brake()	0	0	0.822
CruiseControl.handleCommand(String)	Controller.engineOff()	0	0	0.822
CruiseControl.handleCommand(String)	Controller.engineOn()	0	0	0.822
CruiseControl.handleCommand(String)	Controller.off()	0	0	0.822
CruiseControl.handleCommand(String)	Controller.on()	0	0	0.841
CruiseControl.handleCommand(String)	Controller.resume()	0	0	0.822
Controller.Controller(CarSimulator)	SpeedControl.SpeedControl(CarSimulator)	1	0	0.338
Controller.engineOn()	SpeedControl.clearSpeed()	0	0	0.043
Controller.accelerator()	SpeedControl.disableControl()	0	0	0.173
Controller.brake()	SpeedControl.disableControl()	0	0	0.173
Controller.engineOff()	SpeedControl.disableControl()	0	0	0.173
Controller.off()	SpeedControl.disableControl()	0	0	0.173
Controller.on()	SpeedControl.enableControl()	0	0	0.216
Controller.resume()	SpeedControl.enableControl()	0	0	0.173
Controller.getState()	SpeedControl.getState()	0	0	0.009
Controller.on()	SpeedControl.recordSpeed()	0	0	0.095

TABLE V
THE CLASS-PAIR LEVEL OUTPUT FOR THE CRUISECONTROL APPLICATION.

Class-pair	Weight	Norm. weight	No. of test cases
CarSimulator-CruiseControl	3.294	0.292	5
CarSimulator-SpeedControl	0.521	0.046	1
Controller-CruiseControl	5.910	0.523	9
Controller-SpeedControl	1.567	0.139	2

- Missing function error: a missing function error occurs when there are some inputs from the dependent module to the dependee module which are outside the domain of the dependee module.
- Missing instruction error: it happens when the invocation statement is missing.
- Extra call instruction error: it occurs when the invocation statement is placed on a path that should not contain such invocation.
- Wrong call instruction placement: it happens when the invocation statement is placed in a wrong position on the right path.
- Interface error: interface errors occur when the defined interface between two modules is violated.

Table I shows the experimental results of the proposed approach. We calculate the error detection rate by dividing the number of detected seeded errors by the total number of seeded errors. For the CruiseControl application, 17 integration errors are inserted and 16 of them are detected by our test cases. For the Monopoly application, 21 integration errors are inserted and 19 of them are detected. For the PureMVC application, 19 integration errors are inserted and 16 of them are detected. For the Cinema application, 20 integration errors

TABLE VI
PART OF THE FINAL REPORT FOR THE CRUISECONTROL APPLICATION.

Dependent	Dependee	No. of test cases
CruiseControl.CruiseControl()	CarSimulator.CarSimulator()	0
CruiseControl.handleCommand(String)	CarSimulator.accelerate()	1
CruiseControl.handleCommand(String)	CarSimulator.brake()	1
CruiseControl.handleCommand(String)	CarSimulator.engineOff()	1
CruiseControl.handleCommand(String)	CarSimulator.engineOn()	1
SpeedControl.recordSpeed()	CarSimulator.getSpeed()	0
SpeedControl.run()	CarSimulator.getSpeed()	1
SpeedControl.run()	CarSimulator.setThrottle(double)	0
CruiseControl.CruiseControl()	Controller.Controller(CarSimulator)	0
CruiseControl.handleCommand(String)	Controller.accelerator()	1
CruiseControl.handleCommand(String)	Controller.brake()	1
CruiseControl.handleCommand(String)	Controller.engineOff()	1
CruiseControl.handleCommand(String)	Controller.engineOn()	1
CruiseControl.handleCommand(String)	Controller.off()	1
CruiseControl.handleCommand(String)	Controller.on()	1
CruiseControl.handleCommand(String)	Controller.resume()	1
Controller.Controller(CarSimulator)	SpeedControl.SpeedControl(CarSimulator)	1
Controller.engineOn()	SpeedControl.clearSpeed()	0
Controller.accelerator()	SpeedControl.disableControl()	0
Controller.brake()	SpeedControl.disableControl()	0
Controller.engineOff()	SpeedControl.disableControl()	0
Controller.off()	SpeedControl.disableControl()	0
Controller.on()	SpeedControl.enableControl()	0
Controller.resume()	SpeedControl.enableControl()	0
Controller.getState()	SpeedControl.getState()	0
Controller.on()	SpeedControl.recordSpeed()	0

are inserted and 17 of them are detected. For the ApacheCli application, 22 integration errors are inserted and 19 of them are detected. Figure 3 shows the error detection rate for the applications under test. Table VII shows the number of test cases created to test the applications. It also shows the percentage of connections that are covered by test cases. For example, in the Monopoly application, we write 30 test cases to test 30 connections out of 449 connections and we only covered 6.68% of the connections while detecting 90.48% of integration errors. The Monopoly application achieves the highest reduction in the number of tested connections (6.68%) while the PureMVC application achieves the lowest reduction (53.57%).

TABLE VII
NUMBER OF TEST CASES CREATED FOR EACH APPLICATION AND THE PERCENTAGE OF COVERED CONNECTIONS.

Project	# of connections	# of test cases	Covered connections
Monopoly	449	30	6.68%
CruiseControl	26	13	50.00%
PureMVC	28	15	53.57%
Cinema	97	13	13.40%
ApcheCli	196	17	8.67%

It is clear from the experimental results in Table I and Table VII that our test cases just cover part of the connections while at the same time detect at least 84% of seeded errors for all of the selected applications. Therefore, our approach is very effective in detecting most of the integration errors by testing the highly ranked connections. The experimental results show that the highly ranked connections contain most of the integration errors. It also shows that our approach reduces the number of test cases needed for integration testing.

V. CONCLUSION

In this paper, we propose an approach to select the test focus in integration testing. We define method-level metrics that can be used for test focus selection in integration testing. We build a tool that calculates the metrics automatically.

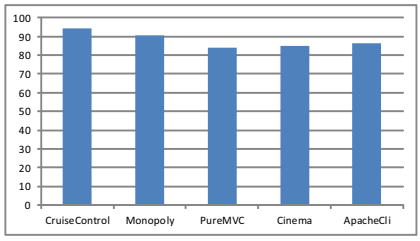

Fig. 3. Error detection rate for the selected applications.

Our approach predicts the number of test cases needed to test each method-pair dependency. We conducted experiments on several Java applications taken from different domains. The experimental results show that our proposed approach is effective in selecting the test focus in integration testing. The small number of developed test cases detect at least 80% of integration errors in all of the selected applications. In future, we are going to expand SMIT tool to work on other programming languages such as C++. In addition, we want to select the test focus for integration testing using method level metrics and integration errors history in the previous versions of the application under test.

REFERENCES

[1] G. J. Myers, *The Art of Software Testing, Second Edition*. Wiley, 2004.

[2] N. Wilde, "Understanding program dependencies," 1990.

[3] S. Benlarbi and W. L. Melo, "Polymorphism measures for early risk prediction," in *Proceedings of the 21st international conference on Software engineering*, ser. ICSE '99. New York, NY, USA: ACM, 1999, pp. 334–344. [Online]. Available: http://doi.acm.org/10.1145/302405.302652

[4] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter, "Exploring the relationship between design measures and software quality in object-oriented systems," *J. Syst. Softw.*, vol. 51, no. 3, pp. 245–273, May 2000.

[5] L. C. Briand, J. Wst, and H. Lounis, "Replicated case studies for investigating quality factors in object-oriented designs," *Empirical Software Engineering: An International Journal*, vol. 6, pp. 11–58, 2001.

[6] K. E. Emam, W. L. Melo, and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *Journal of Systems and Software*, vol. 56, no. 1, pp. 63–75, 2001.

[7] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 531–540.

[8] L. Borner and B. Paech, "Using dependency information to select the test focus in the integration testing process," in *Proceedings of the 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques*, ser. TAIC-PART '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 135–143.

[9] T. J., "Dependency finder," 2008, http://depfind.sourceforge.net/ (Online; accessed 2012).

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

[11] H. D. Mills, *On the statistical validation of computer programs*. Little Brown, Toronto: In Software Productivity.

[12] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2011, ISBN 3-900051-07-0. [Online]. Available: http://www.R-project.org

[13] H. Leung and L. White, "A study of integration testing and software regression at the integration level," in *Software Maintenance, 1990., Proceedings., Conference on*, 1990, pp. 290 –301.