Generating Integration Tests Automatically Using Frequent Patterns of Method Execution Sequences

Mark Grechanik
University of Illinois at Chicago
Email: drmark@uic.edu

Gurudev Devanla
University of Illinois at Chicago
Email: gdev2@uic.edu

***Abstract*—Integration testing is vitally important for ensuring software quality, since many serious software defects are not isolated in single components. Unfortunately, creating integration tests is often a manual and laborious effort. A fundamental problem of software testing is how to automatically create effective integration tests with oracles that find bugs efficiently.**

We created a novel approach for *Automatically SyntheSizing Integration Software Tests (ASSIST)* that automatically obtains models that describe frequently interacting components in software applications, thus reducing the number of synthesized integration tests and increasing their bug-finding power. In ASSIST, static and dynamic analyses are used along with carving runtime states to obtain test input data as well as oracles for the synthesized integration tests. We experimented with three Java applications and show that integration tests that are synthesized using ASSIST have comparable bug finding power with manually created integration tests.

***Index Terms*—software testing, pattern mining, execution trace**

I. Introduction

Many companies have adopted agile development [20], in particular continuous delivery where software is tested and released frequently, at the end of each delivery iteration. In continuous delivery, it is important to test integrated software components at the end of each short-term (i.e., ten day) iteration [21, pages 55-82]. In fact, integration testing has emerged as a major testing approach for agile and distributed software development [25], since the majority of serious software defects are no longer isolated in single components and many catastrophic problems occur in interactions among different components [23].

In *integration testing*, integrated software modules or components are evaluated as a whole to determine if they behave correctly [2, page 6] [4, page 21]. For object-oriented software, integration tests invoke methods that belong to different classes, which exchange data as a result of these invocations [11]. *Acceptance testing* is a kind of integration testing where many components of the application are integrated to implement some requirements [4]. At an extreme, *big bang testing* is an example of the coarsest granularity of the acceptance testing, where the entire application is assembled and tested in one step [33], however, its usefulness is limited, since a single debugging task is impractical [37, Section 7.2].

DOI reference number: 10.18293/SEKE2019-001

Executing acceptance and big-bang tests takes significant resources and time, since they require comprehensive and laborious installation and configuration of the *application under test (AUT)* on a testbed, and each run of these tests may take tens of minutes or hours, depending on the functionality of the AUT. Naturally, acceptance and big-bang tests are run on testbeds during major software releases, however, it is impractical to do so during and at the end of the short-term agile iterations. Clearly, finer granularity integration tests are required that are coarser than *unit tests*, where implementations of methods that belong to the same class are tested individually [4], [26]). Combining two or more methods that belong to different classes is an example of an integration test of a finer granularity that are needed for agile iterations [27]. Given that developers routinely make small incremental changes to applications to fix bugs and modify their functionalities, finer granularity tests are needed to verify that the AUT behaves as desired during and at the end of the agile iterations.

The larger the project, the more important integration testing is to find bugs efficiently in integrated components of an application [37]. Unfortunately, it is expensive, since the average cost per finding and fixing a defect is currently the highest for integration and acceptance testing [23]. This cost increases approximately by anywhere from five and up to twenty times if a defect is missed during integration testing and found at a later stage [15]. Creating effective integration tests requires significant time and effort, since it is not feasible to test all combinations of components in a software application, and the number of these combinations is enormous for nontrivial applications [5], [17]. Despite these difficulties, the demand for integration tests is high, since they are reported to have a higher defect removal efficiency when compared with other forms of testing (e.g., unit) [5], [23].

Our key idea is to use a stable release of the AUT to synthesize integration tests with oracles that will be used to test the subsequent releases of this AUT as part of agile iterative development. Creating integration tests be ASSISTed with automatically obtained models that describe frequently interacting components, and thus are viewed as strong candidates for containing integration bugs. ASSIST combines in a novel way static dataflow and dynamic analyses with pattern mining to guide the synthesis of integration tests. Our tool and experimental results are publicly available at https://www.dropbox.com/s/k1mfuzfw6r7138a/assist-release.tar.gz?dl=0.

II. The Problem

Making integration testing cheaper and more effective is very important and is equally very difficult. Constructing finely granular integration tests is a laborious effort that requires time and resources, which is difficult to justify especially in continuous delivery where software is released in short-term iterations. Combining different components blindly in integration tests reduces their effectiveness, since the total number of integration tests is exponential in the number of components in a software application and many components do not interact with one another. A fundamental problem of software testing is how to automatically create effective integration tests that have a high bug-finding power.

A main objective for software integration tests is to be effective in finding bugs. An equally important objective is to find bugs in a shorter time period without using significant amount of resources, i.e., software integration tests should also be efficient. A system that generates millions of random integration tests is unlikely to be effective and efficient, since running these tests will consume significant resources and time without any guarantees that integration bugs will be found. Thus, our main goal is to minimize the number of synthesized integration tests and their execution time and to increase their effectiveness of finding bugs at the same time.

III. Our Solution

The architecture and the workflow of ASSIST are shown in Figure 1. The input to ASSIST `(1)` is the *Application Test Suite (ATS)* that consists of the AUT, unit and acceptance tests and input test data for these tests. The AUT `(2)` is run using acceptance tests with the Profiler that collects Execution Traces that are `(3)` analyzed by the Frequent Pattern Miner that outputs `(4)` frequent patterns of method calls. The Model Learner `(5)` learns the model that correlates properties of test input data with frequently mined method calls and it produces `(6)` the Model that is used to prioritize the synthesis of integration tests, i.e., to produce more effective integration tests efficiently. An important contribution of ASSIST is that stakeholders concentrate on creating and improving unit and acceptance tests as they do it now, and effective integration tests will be synthesized automatically using models that are learned using these acceptance and unit tests.

To execute synthesized integration tests, input test data is required, i.e., all variables and fields of the objects should be initialized that are used in the methods of these integration tests. Moreover, since the first method of an integration test is the N^{th} method that is executed as part of some acceptance test, this method uses the values of different objects and their fields, which constitute the *state of the AUT*. Our idea is to generate test input data by carving the AUT state [39] before executing the N^{th} method in the acceptance test for the given integration test. In fact, since the sequence of methods in an integration test can be executed as part of two or more acceptance tests, the carved states for these acceptance tests will serve as the set of the input data for the same integration test. A rudimentary state carving method is to traverse the heap starting from the objects that are created in the main method.

The Model is used `(7)` as the input to the Execution State Carver that reruns the AUT with specific acceptance tests in order to carve `(8)` AUT States for the frequent patterns of method calls. Independently from this step, `(9)` unit tests from the ATS are inputted to the Unit Test Analyzer that uses static analyses to obtain `(10)` complex Oracle Expressions that include variables and AUT classes that are eventually used in *assert* statements in these unit tests. Next, `(11)` specific Oracle Expressions are selected for carved AUT States and then these selected Oracle Expressions are projected `(12)` onto the carved AUT States using the State Projector to obtain `(13)` the values of the oracles. As we discussed, unit tests can contain complex expressions and control flows, and its assertion statements contain variables whose values are computed as results of executing these unit tests. To determine what objects and their fields from the AUT are used in these assertions, unit tests are analyzed using a combination of backward slicing [38] and symbolic execution to obtain all expressions that contribute to computing the values of oracle expressions in assert statements. Then, these expressions are re-evaluated given the carved states and new values for the oracles for these expressions are obtained. These obtained oracles, the source code of the AUT and the Model `(14)` are used by the Integration State Synthesizer that `(15)` outputs integration tests. This concludes the description of the ASSIST architecture.

IV. Experimental Evaluation

In this section, we pose research questions (RQs), describe subject applications, explain our methodology and variables, and discuss threats to validity.

A. Research Questions

As part of evaluation, we will answer the following research questions (RQs) to assess how ASSIST meets the objectives of effectiveness and efficiency.

RQ1: How effective are synthesized integrations tests in finding bugs? The rationale for this research question is to determine whether ASSIST synthesizes integration tests that can locate more integration bugs in software when compared with certain baseline approaches, e.g., manually created integration tests and FUSION, an approach that composes integration tests from unit tests [29].

RQ2: How efficient is ASSIST in synthesizing integration tests that can help find bugs without a significant use of computing resources? The rationale for this research question is to determine if ASSIST produces fewer integration tests and their total execution time is not prohibitive, while measuring their effectiveness of finding bugs at the same time. Our goal is to show that ASSIST can synthesize a much smaller and manageable number of integration tests that have good bug-finding power.

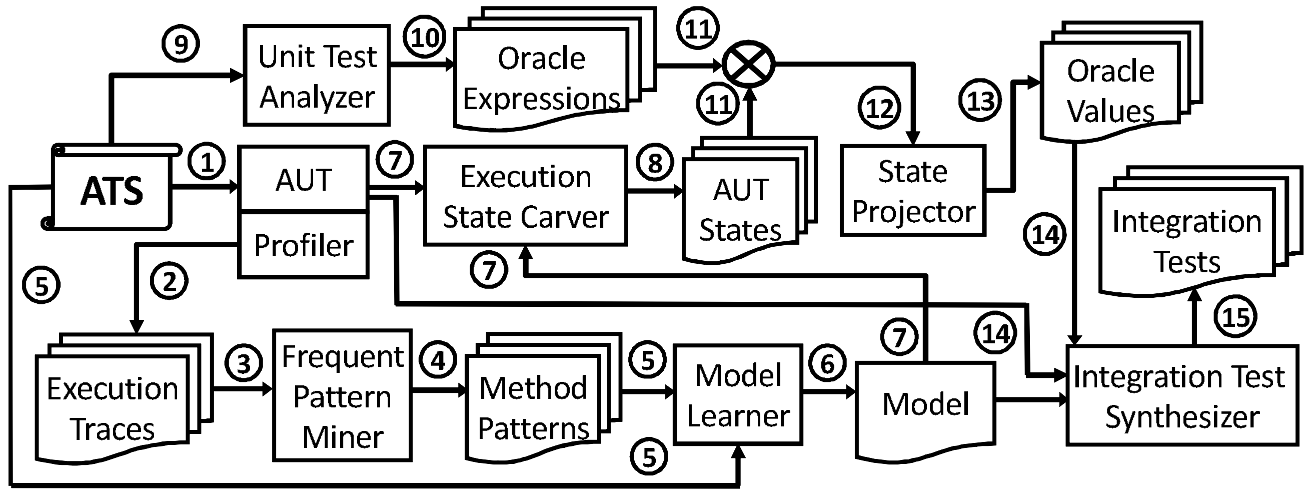

Fig. 1. The architecture and workflow of ASSIST.

TABLE I
CHARACTERISTICS OF SUBJECT APPLICATIONS.

AUT	Size, KLOC	Acc Tests	Unit Tests	Integration Tests	
				ASSIST	FUSION
NANOXML	7	36	92	48	75
SCRIBE	3.7	29	39	36	28
SHTUTXML	1.3	5	40	6	6

TABLE II
MAXIMUM VALUES OF ELAPSED EXECUTION TIME AND MEMORY CONSUMPTION FOR ASSIST.

AUT	Instrum		Pattern Mining		Synthesis		State
	Test	Mem	Test	Mem	Test	Mem	
SCR	16s	55K	0.01s	2.3KB	18s	70K	0.2GB
NAN	17s	55K	0.04s	2.4KB	7s	60K	1.5GB
SHT	10s	57K	0.18s	2.4KB	7s	52K	0.95GB

B. Subject Applications And Their Test Suites

The subject applications for evaluating ASSIST are open-source Java applications that are widely used. Their characteristics are given in Table I. SCRIBE is one of the subject applications that serves as a OAuth module for Java applications. This application has close to 50 contributors and 22 releases. NanoXML is a Java-based non-validating parser. The third subject application is ShtutXML, an XML to text mapper. All subject applications have test suites that contain mixtures of unit and acceptance tests. In addition, we used several graduate students from UIC with prior software development experience to study these applications and create additional unit and acceptance tests.

C. Methodology And Variables

To address RQ1 and RQ2 we evaluate ASSIST using the following two dependent variables: the number of synthesized integration tests and their bug-finding power. The number of synthesized integration tests is a function of the threshold value for mining frequent method sequences. The smaller the value is the more sequences can be mined, which is exponential in the limit. The larger the threshold value is, the fewer more frequent method sequences can be mined, however, the mining process may take a very long time. Unfortunately, frequent pattern mining algorithms are computationally intensive, and increasing the threshold value closer to one may take weeks or months even using powerful computers. Thus, as part of our experimentation we show the sizes of mined method sequences that we obtain for different threshold values and how they affect sizes of synthesized integration test suites.

We measure the bug-finding power of test suites using *mutation testing*, which is recognized as one of the strongest approaches for evaluating the effectiveness of test suites [14], [18]. The code of the *application under test (AUT)*, P, is modified by applying *mutation operators* to the AUT's code to create a buggy but syntactically correct version of the AUT, P', i.e., a *mutant*. Running the test suite for P on P' should fail some tests, i.e., the mutant is killed. Otherwise, the test suite is deemed not adequate to find bugs and it should be enhanced with new tests that can kill mutants that were not killed with the previous version of the test suite. A key measurement of the power of determining the adequacy of test suites is mutant killing ratio, i.e., the ratio of the number of mutants killed by a test suite to the total number of generated non-equivalent mutants. That is, our goal is to apply a mutation testing approach to the subject applications that generates mutants for determining the adequacy of integration test suites synthesized by ASSIST and the mutant killing ratio should be approximately the same when comparing to test suites that are manually created or generated using a competitive approach.

In our evaluation we compare the tests produced by our approach with FUSION [29], a state of the art tool for generating integration tests by combining unit tests using an object-relational model that it re-engineers from the source code of the application. The independent variables in our evaluation included the two mutation testing tools, jMINT [16] and JavaLanche [30], the set of subject applications, and the threshold values used in the pattern mining tool called BIDE

TABLE III
WE DESCRIBE SELECTED PARAMETERS FOR MINED FREQUENT SEQUENCES FOR SYNTHESIZING INTEGRATION TESTS USING ASSIST.

AUT	Unq Mth	Max Len Seq	BIDE Inpt	Thresh	BIDE output	Seq Used In ASSIST
SCRIBE	81	60	134	0.01	129	87
NANOXML	101	602	121	0.05	180	42
SHTUTXML	28	48	298	0.002	27	15

to extract frequent method sequences from execution traces that are obtained from the subject applications. The number of integration tests that ASSIST synthesizes is also guided by the number of tests that contain oracles that come with the subject applications.

For frequent pattern mining, we use an closed sequence frequent pattern mining tool called BIDE [34]. This tool implements a very efficient algorithm for determining closed frequent sequences. We note that ASSIST is not tied to these specific tools. Any frequent pattern mining tool that identifies closed frequent sequences can be used in place of this tool.

D. Threats to Validity

One threat to external validity is the nature of applications that we used to evaluate ASSIST – they are small and may not be good representatives of the overall population of applications. We plan to conduct more experiments with many more applications to generalize these results. Subject applications were chosen based on the availability of large and diverse tests suites that we used as acceptance tests and to extract test oracles. In addition, we are also constrained by the limitations of FUSION, since it does not produce oracles. The counter argument to this threat to validity is that the subject applications are popular and have many tests and they may be viewed as good representative of the applications that are built in industry.

Another threat to validity is that we do not address the problem of storing transient state components, such as pointers to external resources (e.g., sockets and files). Since we show that our approach works for non-transient data structures, we hope to address the challenges with transient states as the next logical step in our future work. This technical challenge does not pose a threat to the core of ASSIST.

Finally, a threat to validity is that we evaluate the mutants against the integration tests generated by only one tool, namely, FUSION. Ideally, we need more tools on generating integration mutants that contain meaningful oracles, unfortunately, little research is done in this area that resulted in tools that can be used in our evaluation. The very purpose of our research is to address this limitation.

V. RESULTS

In this section, we report the results of the experiments and state how they address our RQs. We carried out experiments using Mac OSX 10.8, 64-bit CPU 2.4 GHz Intel Core i7, 8GB of RAM, 256KB L2 Cache(per core) and 6MB L3 Cache.

Performance-related measurements are shown in Table II. Clearly, the biggest space-related expense is the the carved state that requires approximately 1.5GB for NanoXML. Instrumenting the applications and synthesizing data takes less than 20 seconds, which is a reasonable expense. We can conclude that with respect to time and memory consumption, ASSIST is a practical approach that can be used for reasonably sized software applications to generate integration tests.

We evaluate the results of our approach and compare its effectiveness compared to manually generated tests and the tests generated by FUSION. Results of our experiments are shown in Table IV. For the AUT SCRIBE the strong mutant killing ratio is the same that is achieved with manually created tests. This result shows that it is possible to achieve the same result with ASSIST as it is achieved with manually created tests. Even though the ratio for SHTUTXML is smaller with ASSIST-based integration tests when compared to manually created tests, we view it as an impressive result, since ASSIST is fully automatic.

The application NANOXML is a source of concern, since its mutant killing ratio is very low with ASSIST-based integration tests. Our investigation revealed the following. When we run the acceptance tests, that are 101 unique methods in NANOXML that are invoked and are written in execution traces. Of these, there are 27 unique methods that are mined as frequent sequential patterns. Unfortunately, there are unit tests only for five methods. For the remaining methods there are no assertions in any of the manually created tests. Moreover, those methods that are used in unit tests and are part of the mined frequent sequences are loosely integrated with other methods in these sequences, so that the tightness of integration is very poor. ASSIST is based on the assumption, that unit tests and acceptance tests are adequate to produce integration tests that can be used as the software evolves. This also exposes the characteristics of the test suite and the lack of adequate tests for methods that were identified in the set of frequently used methods. As a result, ASSIST could not synthesize enough effective integration tests.

The case with NANOXML highlights the results of consequences of deficiencies that result in poorly built tests rather than limitations of ASSIST. Indeed, if acceptance and unit tests are abundant and if they are constructed properly, ASSIST automates a difficult and laborious tasks of creating effective integration tests. Moreover, for SCRIBE, ASSIST achieve the same mutant killing ratio with fewer synthesized tests, i.e., 36 when compared with 100 manually created tests. For SHTUTXML, the mutant killing ratio is achieved with only six synthesized tests.

We also see that using weak mutation ASSIST produces a weak mutation killing ratio that is equal to the manually created tests for two applications, SCRIBE and NANOXML. In addition, our approach performs equally well compared to FUSION for both these applications. In both these cases, our approach also provides mutant killing ratios that is at least 50% of mutant killing ratio of manually created tests.

The results of evaluating ASSIST with the mutation tool

TABLE IV
EVALUATING THE EFFECTIVENESS OF ASSIST BY COMPARING IT WITH FUSION. MUTANTS ARE GENERATED USING jMINT.

AUT Name	Type of tests	Tests Created	Mutants Generated	Mutants Triggered	Mutants Killed	Trigger Ratio	Killed Ratio
SCRIBE	DEFAULT	100	115	15	6	13.04	5.21
	ASSIST	36	115	15	6	13.04	5.21
	FUSION	28	115	15	-	13.04	
NANOXML	DEFAULT	116	104	95	34	91.34	32.69
	ASSIST	48	104	5	2	4.80	1.92
	FUSION	75	104	40	-	38.46	
SHTUTXML	DEFAULT	43	12	7	7	58.3	58.3
	ASSIST	6	12	2	2	16.66	16.66
	FUSION	6	12	2	-	16.66	

TABLE V
EVALUATING THE EFFECTIVENESS OF ASSIST BY COMPARING IT WITH FUSION. MUTANTS ARE GENERATED USING JAVALANCHE.

Description of the Step	SCRIBE				NANOXML				SHTUTXML			
	FUSION		ASSIST		FUSION		ASSIST		FUSION		ASSIST	
Total mutations	356		2634		1442		7330		193		719	
Covered Mutations in Scan Step	105	29.49%	2111	80.14%	731	50.69%	6957	94.91%	0	0	403	56.05%
Covered Mutations	100	28.09%	100	3.80%	99	6.87%	99	1.35%	0	0	100	13.91%
Not Covered Mutations	256	71.91%	2534	96.20%	1343	93.13%	7231	98.65%	193	0	619	86.09%
Killed Mutants	36	10.11%	80	3.04%	47	3.26%	66	0.90%	0	0	49	6.82%
Surviving Mutants	320	89.89%	2554	96.96%	1395	96.74%	7264	99.10%	193	0	670	93.18%
Mutations Score	10.11%		3.04%		3.26%		0.90%		0.0%		6.82%	
Mutation Score for covered mutants	36.00%		80%		47.47%		66.67%		0		49%	

called Javalanche are shown in Table V. ASSIST-based synthesized integration tests are richer than ones created by FUSION, resulting in more mutants generated by Javalanche for ASSIST. As a result, ASSIST-based tests cover more mutated statements when compared to ones generates by FUSION. ASSIST-based tests kill many more mutants when compared with FUSION-generated tests. The mutation score for covered mutants is much higher for all applications for ASSIST. Based on this evidence we can conclude that **we positively answer RQ1, i.e., synthesized integrations tests are effective in finding bugs**.

To address RQ2, we will need to evaluate the results of our approach in terms of performance of the frequent mining algorithm which is dependent on the initial sequences generated during the execution of acceptance tests and the number of tests ASSIST generates. Table III provides information on the number of unique methods for each subject application, maximal sequence of method calls that was traced during execution of acceptance tests. The last three columns state the threshold values that were used to identify the frequent sequences, and the actual number of sequences that were used by ASSIST. For each of the subject applications, we note that it did not take more than a couple of seconds to identify these frequent sequences. The other result we need to evaluate to address RQ2 would be the number of integration tests ASSIST produces. Table I provides details on the number of tests we produce. It can be observed that the number of tests generated by ASSIST is close to the number of tests that were created manually or by FUSION. Ihis shows that even though there is a potential of execution trace producing a large number of sequences, using a combination frequent mining and static analysis techniques ASSIST is able to produce tests that can be executed as efficiently as manually produced tests and the tests produced by FUSION. Based on this evidence we can conclude that **we positively answer RQ2, i.e., ASSIST is efficient in synthesizing integration tests that can help find bugs without a significant use of computing resources**.

VI. RELATED WORK

Related work on automatic generation of integration tests has many branches. Different model-based approaches exist for generating integration tests using different types of formal models [9], [19], [28] and using modeling approaches for creating integration tests for distributed applications [6], [35]. Some approaches learn models and class dependencies for integration testing from application code and behavior [3] or from previous versions of the same application [7], [8]. Opposite to these approaches, ASSIST does not require models for generating integration tests, since models may be outdated, incomplete, or they may not exist at all. In addition, it is not clear how oracles are defined for these approaches. ASSIST is different from these approaches and complementary to them in that ASSIST synthesizes integration tests using models of method invocations. Unlike ASSIST, these approaches do not have explicit mechanisms to control the number of generated tests while increasing their effectiveness, and it is a question how efficiently it is achieved.

Class Integration and Test Order (CITO) approaches determine orders in which classes are composed in integration tests using graph-based and search-based solutions [1], [10], [12], [13], [22], [36]. However, the determination of a cost function, which is able to generate the best solutions, is not always a trivial task. In contrast, ASSIST does not require cost functions, and it extracts oracles from tests, while CITO does not address the oracle problem. Unlike ASSIST, CITO approaches do not reflect the frequency of executions of integrated components, and therefore it is not clear how it can increase the effectiveness of the composed integration tests.

Approaches that compose integration tests from unit tests and some software modules [24], [29], [31], [32] address a different angle of the problem than ASSIST – the latter concentrate on producing effective integration tests efficiently, while it is unclear how the former approach addresses a problem of reducing the large number of combinations of unit tests into integration tests.

VII. Conclusion

We created a novel approach for *Automatically SyntheSizing Integration Software Tests (ASSIST)* that automatically obtains models that describe frequently interacting components in software applications, thus reducing the number of synthesized integration tests and increasing their bug-finding power. In ASSIST, static and dynamic analyses are used along with carving runtime states to obtain test input data as well as oracles for the synthesized integration tests. We experimented with three Java applications and show that integration tests that are synthesized using ASSIST have comparable bug finding power with manually created integration tests.

References

[1] A. Abdurazik and J. Offutt. Coupling-based class integration and test order. AST '06, pages 50–56, New York, NY, USA, 2006. ACM.

[2] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008.

[3] L. Badri, M. Badri, and V. S. Ble. A method level based approach for oo integration testing: An experimental study. SNPD-SAWN '05, pages 102–109, Washington, DC, USA, 2005. IEEE Computer Society.

[4] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990.

[5] A. Bertolino, P. Inverardi, H. Muccini, and A. Rosetti. An approach to integration testing based on architectural descriptions. ICECCS '97, pages 77–, Washington, DC, USA, 1997. IEEE Computer Society.

[6] A. Bertolino and A. Polini. Soa test governance: Enabling service integration testing across organization and technology borders. ICSTW '09, pages 277–286, Washington, DC, USA, 2009. IEEE Computer Society.

[7] L. Borner and B. Paech. Using dependency information to select the test focus in the integration testing process. TAIC-PART '09, pages 135–143, Washington, DC, USA, 2009. IEEE Computer Society.

[8] L. Borner and B. Paech. Using dependency information to select the test focus in the integration testing process. TAIC-PART '09, pages 135–143, Washington, DC, USA, 2009. IEEE Computer Society.

[9] L. Briand, Y. Labiche, and Y. Liu. Combining uml sequence and state machine diagrams for data-flow based integration testing. ECMFA'12, pages 74–89, Berlin, Heidelberg, 2012. Springer-Verlag.

[10] L. C. Briand, Y. Labiche, and Y. Wang. An investigation of graph-based class integration test order strategies. *IEEE Trans. Softw. Eng.*, 29(7):594–607, July 2003.

[11] P. J. Clarke. *A taxonomy of classes to support integration testing and the mapping of implementation-based testing techniques to classes*. PhD thesis, Clemson, SC, USA, 2003. AAI3098273.

[12] R. Da Veiga Cabral, A. Pozo, and S. R. Vergilio. A pareto ant colony algorithm applied to the class integration and test order problem. ICTSS'10, pages 16–29, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] R. Delamare and N. A. Kraft. A genetic algorithm for computing class integration test orders for aspect-oriented systems. ICST '12, pages 804–813, Washington, DC, USA, 2012. IEEE Computer Society.

[14] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, Apr. 1978.

[15] Gartner. It key metrics data 2008: Key applications measures: Current year: Defect rates. *http://www.gartner.com/DisplayDocument?ref=g_search&id=557525*, Dec. 2007.

[16] M. Grechanik and G. Devanla. Mutation integration testing. In *2016 IEEE International Conference on Software Quality, Reliability and Security, QRS 2016, Vienna, Austria, August 1-3, 2016*, pages 353–364, 2016.

[17] M. Greiler, A. v. Deursen, and M.-A. Storey. Test confessions: a study of testing practices for plug-in systems. ICSE 2012, pages 244–254, Piscataway, NJ, USA, 2012. IEEE Press.

[18] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.*, 3(4):279–290, July 1977.

[19] J. Hartmann, C. Imoberdorf, and M. Meisinger. Uml-based integration testing. ISSTA '00, pages 60–70, New York, NY, USA, 2000. ACM.

[20] J. Highsmith and A. Cockburn. Agile software development: The business of innovation. *IEEE Computer*, 34(9):120–122, 2001.

[21] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition, 2010.

[22] Z. Jin and A. J. Offutt. Coupling-based integration testing. ICECCS '96, pages 10–, Washington, DC, USA, 1996. IEEE Computer Society.

[23] C. Jones and O. Bonsignour. *The Economics of Software Quality*. Addison-Wesley Professional, Aug. 2011.

[24] M. Jorde, S. Elbaum, and M. B. Dwyer. Increasing test granularity by aggregating unit tests. ASE '08, pages 9–18, Washington, DC, USA, 2008. IEEE Computer Society.

[25] M. Luke. How early integration testing enables agile development. *http://www.ibm.com/developerworks/rational/library/early-integration-testing-enables-agile-development*, June 2012.

[26] A. P. Mathur. *Foundations of Software Testing*. Addison-Wesley Professional, 1st edition, 2008.

[27] A. J. Offutt. Unit testing versus integration testing. pages 1108–1109, Washington, DC, USA, 1991. IEEE Computer Society.

[28] S. Ogata and S. Matsuura. A method of automatic integration test case generation from uml-based scenario. *WSEAS Trans. Info. Sci. and App.*, 7(4):598–607, Apr. 2010.

[29] M. Pezzè, K. Rubinov, and J. Wuttke. Generating effective integration test cases from unit ones. In *Proc. of 6th IEEE ICST*, 2013.

[30] D. Schuler and A. Zeller. Javalanche: efficient mutation testing for java. ESEC/FSE '09, pages 297–298, New York, NY, USA, 2009. ACM.

[31] S.-H. Shin, S.-K. Park, K.-H. Choi, and K.-H. Jung. Normalized adaptive random test for integration tests. COMPSACW '10, pages 335–340, Washington, DC, USA, 2010. IEEE Computer Society.

[32] Y. Shin, Y. Choi, and W. J. Lee. Integration testing through reusing representative unit test cases for high-confidence medical software. *Comput. Biol. Med.*, 43(5):434–443, June 2013.

[33] J. A. Solheim and J. H. Rowland. An empirical study of testing and integration strategies using artificial software systems. *IEEE Trans. Softw. Eng.*, 19(10):941–949, Oct. 1993.

[34] J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. ICDE '04, pages 79–, Washington, DC, USA, 2004. IEEE Computer Society.

[35] S. Wang, Y. Ji, W. Dong, and S. Yang. A new formal test method for networked software integration testing. ICCSA'10, pages 463–474, Berlin, Heidelberg, 2010. Springer-Verlag.

[36] Z. Wang, B. Li, L. Wang, and Q. Li. An effective approach for automatic generation of class integration test order. COMPSAC '11, pages 680–681, Washington, DC, USA, 2011. IEEE Computer Society.

[37] A. H. Watson and T. J. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. NIST Special Publication 500-235, NIST: National Institute of Standards and Technology, Gaithersburg, MD, 1996. See http://hissa.nist.gov/HHRFdata/Artifacts/ITLdoc/235/title.htm.

[38] M. Weiser. Program slicing. ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[39] G. Xu, A. Rountev, Y. Tang, and F. Qin. Efficient checkpointing of java software using context-sensitive capture and replay. ESEC-FSE '07, pages 85–94, New York, NY, USA, 2007. ACM.