Substra: A Framework for Automatic Generation of Integration Tests

Hai Yuan
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
hyuan3@ncsu.edu

Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
xie@csc.ncsu.edu

ABSTRACT

A component-based software system consists of well-encapsulated components that interact with each other via their interfaces. Software integration tests are generated to test the interactions among different components. These tests are usually in the form of sequences of interface method calls. Although many components are equipped with documents that provide informal specifications of individual interface methods, few documents specify component interaction constraints on the usage of these interface methods, including the order in which these methods should be called and the constraints on the method arguments and returns across multiple methods. In this paper, we propose Substra, a framework for automatic generation of software integration tests based on call-sequence constraints inferred from initial-test executions or normal runs of the subsystem under test. Two types of sequencing constraints are inferred: shared subsystem states and object define-use relationships. The inferred constraints are used to guide automatic generation of integration tests. We have implemented Substra with a tool and applied the tool on an ATM example. The preliminary results show that the tool can effectively generate integration tests that exercise new program behaviors.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Experimentation, Reliability, Verification.

Keywords: Software Testing, Integration Testing, Test Generation.

1. INTRODUCTION

In component-based software development [6], a software system is built out of well-encapsulated components each of which has a set of well-defined interfaces. Components interact with each other by invoking methods through these interfaces. One important means to verifying the correctness of component interactions is through integration testing. In integration testing, integration tests are generated to test interactions among different components. These integration tests are usually in the form of sequences of method calls in the interfaces. However, syntactically correct sequences do not necessarily result in valid tests. Specifications can be used to create valid method-call sequences as tests. Various approaches have been developed for generating feasible method-call sequences from specifications [1,3,10]. However, specifications are often not available, especial for those legacy systems. Even when some components are equipped with specifications on their interfaces, many of these specifications only specify individual method calls – a method call's name, parameters, and return value. A component's document usually does not specify sequencing constraints on the method calls, including the order in which these interface methods should be called and the constraints among method arguments and returns across multiple methods.

These sequencing constraints are important for helping assure correct component usages. For example, `cli`, one of the jakata common components[1], is a component that is used to parse command line parameters. When using `cli`, programmers must at first provide a pattern of options by calling the `Option` class' constructor: `opt = new Option(...)`. Then a parser is called trying to match the command line input with the option pattern: `commandline = parser.parse(opt, input)`. It is crucial that the option pattern must be constructed before it can be used to parse a command line input.

Sequencing constraints are also useful in guiding automatic generation of component integration tests. Without any guidance, random test generation would result in method-call sequences that do not appear in real applications or erroneous sequences, e.g., using an object before defining it. With the guidance of sequencing constraints, we can avoid generating invalid tests and focus on generating meaningful tests that emulate the real usages of multiple components within a software system.

In this paper, we propose Substra, a framework for automatic generation of integration tests based on call-sequence constraints dynamically inferred from initial-test executions or normal runs of the subsystem under test. These constraints are inferred based on two types of information: shared subsystem states and define-use relationships. The first type of information describes that if a method call m_1's exit state is equivalent [15] to another method call m_2's entry state, then m_1 and m_2 may be invoked consecutively. The second type of information is as follows: if the return value r of a method call m_1 is the receiver or one of the arguments of another method call m_2, that is, in the form $r.m_2(\cdots)$ or $v.m_2(\cdots, r, \cdots)$ (v is an arbitrary object), then m_2 must appear after m_1 in a method call sequence. Substra uses an object state machine (OSM) to model these constraints. A subsystem's states are represented as nodes in the state machine, method calls as tran-

[1] `http://jakarta.apache.org/commons/cli/`

AST'06, May 23, 2006, Shanghai, China.

sitions, and define-use relationships as guard conditions of transitions. Substra proposes an iterative process that exploits initial-test executions or normal runs of the system to dynamically infer sequencing constraints, and then uses these constraints to help generate new tests. One iteration includes six steps: collect execution traces, find boundary calls, infer define-use relationships, construct basic object state machines, construct a subsystem state machine, and generate new tests. We have developed a tool for Substra and applied it on an ATM [4] program. The preliminary results show that the tool can effectively generate integration tests that exercise new program behaviors.

The rest of the paper is organized as follows. Section 2 presents a formal description of Substra. Section 3 introduces the implementation of the approach. Section 4 shows our preliminary results of applying the tool on an example. Section 5 discusses issues of the approach and its current implementation. Section 6 presents related work and Section 7 concludes.

2. FRAMEWORK

In this section, we give a formal description of the Substra framework, including a formal definition of an object state machine (OSM) (Section 2.1), subsystem state machine (SSM) (Section 2.2), and the concepts and the procedure used by Substra (Section 2.3).

2.1 Object State Machine

Xie and Notkin [16] have defined an object state machine for a class:

DEFINITION 1. *An* object state machine *(OSM) M of a component c is a sextuple $M(c)$ = (I, O, S, δ, λ, $INIT$) where I is a nonempty set of method calls as c's interface; O is the set of the return values of these method calls; and S represents the set of states of c's objects. $INIT \in S$ is the initial state that the machine is in before calling any constructor method of c. $\delta : S \times I \rightarrow P(S)$ is the state transition function and $\lambda : S \times I \rightarrow P(O)$ is the output function in which $P(S)$ and $P(O)$ are the power sets of S and O, respectively. When the machine is in a current state s and receives a method call $i \in I$, it moves to one of the next states specified by $\delta(s, i)$ and produces one of the method returns $r \in O$ given by $\lambda(s, i)$.*

There are various ways to define an object's states. By default, our framework defines the state of an object based on the values of all the fields transitively reachable from the object [15].

2.2 Subsystem State Machine

DEFINITION 2. *A subsystem S is a triple (SC, SS, SI), where $SC = C_i$ is the set of components included in the subsystem; SS is the set of states of the subsystem, $SS \subseteq \prod CS_i$, where CS_i is the set of states of components C_i; and $SI \subseteq \bigcup CI_i$, where CI_i is the set of interfaces of component C_i. Each method call $mc_i \in SI$ consists of two parts: method signature MS_i and guard conditions GC_i that must be satisfied before mc_i can be invoked.*

Based on the definition of a subsystem, we can extend the definition of an OSM to the definition of a subsystem state machine (SSM):

DEFINITION 3. *A subsystem state machine for a subsystem S: (SC, SS, SI) is a sextuple SM = (SI, SO, SS, δ, λ, $INIT$) where SI and SS are nonempty sets of interface and states of S, respectively. SO is the set of return values of interface method calls. $\delta : SS \times SI \rightarrow P(SS)$ is the state transition function and $\lambda : SS \times SI \rightarrow P(SO)$ is the output function, $P(SS)$ and $P(SO)$ are the power sets of SS and SO. $\delta = \bigcup M(C_i).\delta$ and $\lambda = \bigcup M(C_i).\lambda$. $INIT \in SS$ is the initial state of the state machine.*

Note that we consider only the return values within the scope of a subsystem. This constraint helps us to simplify the procedure of identifying define-use relationships among boundary calls, which we shall define later.

An integration test case can be described as a triple $< Sub, MS, A >$, where Sub is the subsystem under test, MS is the integration test input, which consists of a sequence of method calls ($MS = (m_1, m_2, \cdots, m_k), m_i \in SI$), and A is a set of assertions that are used as test oracles. In this research context, we focus on generating integration test inputs MS (in short as integration tests).

One objective of integration-test generation is to generate meaningful method-call sequences to reveal possible bugs in the system or increase our confidence on the system correctness. However, it is often infeasible to test all method call combinations in practice. Then guidelines and constraints could be used to help generate focused integration tests.

2.3 Substra

We propose Substra, a framework that guides automatic generation of integration tests with call-sequence constraints inferred from dynamic executions. Two types of sequencing constraints are inferred and used by Substra: shared state constraints and define-use constraints. Based on these constraints, Substra generates integration tests in the form of sequences of boundary method calls.

DEFINITION 4. *A boundary method call is a call $c_i.m$ whose receiver $c_i \in SC$ while its caller $c_j \notin SC$.*

The reason for us to define boundary calls is that we only concern about how a subsystem is manipulated by other subsystems through its interface, but are not interested in other non-interface related interactions, such as intra-subsystem interactions. With the notion of boundary calls, we can reduce the size of subsystem interface SI, thus reducing the number of method-call combinations.

DEFINITION 5. *A shared state constraint is a constraint for two transitions $\delta_1 : (ss_1, si_i) \rightarrow ss_2$ and $\delta_2 : (ss_2, si_2) \rightarrow ss_3$, where transitions δ_1 and δ_2 share state ss_2.*

In an SSM, a shared state constraint is expressed as a node ss_2 with incoming transition δ_1 and outgoing transition δ_2. The constraint prescribes that methods si_1 and si_2 be called consecutively.

The d-u (define-use) constraint is a common constraint used in test generation. The constraint prescribes that an entity must be defined before it can be used. According to the constraint, any used variable (parameter or object field variable) in a method call must be defined before the method call can be actually invoked. The method signature of a method call can be characterized by three parts: the receiver r, method name n, and parameter list p.

DEFINITION 6. *A define-use constraint between method calls m_i and m_j is a constraint that the return value ret_i of m_i is equal to the receiver r_j of m_j ($ret_i = r_j$), or ret_i is used as one of m_j's argument ($ret_i \in p$).*

The define-use constraint prescribes that method call m_j must appear after the method call m_i in a test. There are different ways

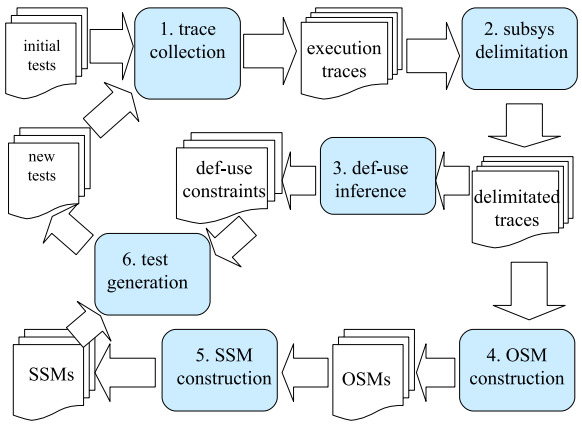

Figure 1: An Overview of Substra Process.

to define a variable, such as assignment and state-modify method calls. In our current work, we use only the return value of a method call as definition. We plan to add other ways of definition into the framework in our future work.

Figure 1 shows an overview of the Substra process that infers call-sequence constraints from dynamic executions and uses these constraints to guide automatic generation of integration tests. In the iterative process, each iteration consists of six steps: collect execution traces, find boundary calls, infer define-use relationships, construct basic object state machines, construct a subsystem state machine, and generate new tests.

1. From program executions, we collect execution traces including executed method calls, object fields' value at entry and exit points of method calls, etc.
2. We identify boundary calls by examining the method call information included in the execution traces.
3. We infer define-use constraints by examining the boundary call information included in the execution traces.
4. We construct OSMs for individual objects in the subsystem according to the preceding definition of OSMs.
5. We construct a subsystem state machine (SSM) by combining the states of individual OSMs according to related boundary calls. That is, if we want to construct the starting state S of transition T, and we already know that T exists in both OSM_1 and OSM_2 with starting states $O_1.s_1$ and $O_2.s_2$, respectively, then S is defined as follows: $S = (O_1.s_1, O_2.s_2)$.
6. We traverse the SSM, collect the transition information along the traversal, and build a skeleton of method-call sequences based on the visited transitions and the inferred sequencing constraints. To generate executable tests, we generate random values for method arguments in the skeleton of method-call sequences. Alternatively the skeleton can be fed into some test-generation tools such jCUTE [13].

3. IMPLEMENTATION

We have developed a tool to implement our Substra framework. The tool performs all the steps of the framework except that we use a third-party tool to collect execution traces and optionally use a third-party test-generation tool to generate tests based on the generated test skeleton.

At first, we execute the initial test suite on the system under test. During the execution, we use the Java front-end of Daikon [5] to collect object states. Daikon is a tool that dynamically detects likely program invariants in program executions. It can collect object-field values during program executions and report properties that hold true on these fields during the executions. In our approach, we use Daikon to collect method call sequences and an individual object's states at the entry and exit points of a method call. Users can also specify the object fields that they are not interested in by providing a file containing fields of no interests. Given the execution information, we construct basic object state machines that describe the state transitions of each object.

After the execution traces are collected, our tool starts to identify boundary calls. As defined in the previous section, a boundary call is a method call whose caller is outside of the monitoring scope while the callee is inside. We recognize caller-callee relationships by traversing method-call entry and exit points.

When constructing basic object state machines, we also store the reference of each object constructed during the executions. We use these object references to identify define-use relationships. By sequentially going through the boundary calls, we put return value r of each boundary call m_i into a monitored object list. Whenever we identify either a boundary method call m_j's receive object or an object assigned as its argument is in the list, a define-use relationship is constructed in the form of *caller = return* m_i or *parameter name = return* m_i. The relationship is used as a guard condition of the transition in the constructed subsystem state machine, corresponding to method call m_j.

Based on the preceding information, we can construct the subsystem state machine (SSM). A transition in the SSM is a boundary method call, using the inferred define-use relationship as its guard condition. A state of the SSM is the combination of the currently monitored objects' states. For example, if there are two objects o_1 and o_2 in the monitored object list at the exit point of transition T, each in states s_1 and s_2, respectively, then the subsystem's state at the exit point of transition T is $\{s_1, s_2\}$.

Finally the tool generates tests, which are method-call sequences. The tool deems the constructed SSM as a graph, and uses the depth-first search (DFS) algorithm to traverse the SSM. However, unlike standard DFS algorithm, the tool does not mark the visited node. This mechanism makes it possible to traverse loops for more than one time. A maximum length of method sequence is used to constrain how many edges the tool should visit. When the tool reaches a node with several outgoing edges, it randomly picks one. After traversing each edge, the tool uses the edge's information (method name and parameter list) to build a method call, and then fill the callee and parameter list with local variable names based on the guard conditions.

The abstraction technique we used makes the resulting SSM a more general description of the program's behavior other than just the summation of the execution traces taken by all the initial tests or existing normal runs. For example, if the execution of test A produces trace $s_1s_2s_3s_4s_5$ (s_i represents a state in the SSM), and the execution of test B produces trace $s_0s_4s_2s_6$, we can find out in the resulting SSM there exists a loop $s_2s_3s_4$. Based on the resulting SSM, we can generate tests such as $s_1s_2s_3s_4s_2s_6$, which is a new trace different from any of the existing traces produced by the execution of tests A and B. Moreover, we can also manipulate the values of arguments of a method call to make it possible to expose more states. For example, in the initial test, v_1 is passed as an argument of method call m_1, resulting in subsystem state s_1. However, given the same method call, when we supply another value v_2 as the argument, the method execution may lead to a new subsystem state s_2, and may also produce a new trace.

4. PRELIMINARY RESULTS

We applied our tool on an ATM [4] example, which contains a main program and three packages: the `banking` package models the entities and procedures used by a bank, the `simulation` package provides GUI and passes user inputs to the logic part of the program, and the `atm` package simulates the behaviors of an ATM. The `atm` package includes two sub-packages: `atm.physical` models the physical parts of an ATM, such as a card reader and a customer console, and `atm.transaction` implements the logical functionality of an ATM, such as withdrawal and deposit.

Because ATM users may be more concerned about the correctness of an ATM's capability of performing transactions, in our evaluation, we selected the `atm.transaction` package as our primary subject. We also included helper classes such as `atm.ATM`, and `banking.Card` in our subsystem scope. Objects of these classes are passed as arguments in making and performing transactions. The original ATM example is a concurrent program. Its inputs are passed only through GUI. To simplify the evaluation procedure, we made slight changes to the original program. First, we changed the program to non-concurrent. Because the concurrent mechanism is employed only by the `atm.ATM` class without affecting the `atm.transaction` package, we consider that this change will not affect the functionality of performing transactions. Second, we modified some methods in the `atm.transaction` package. The modifications include removing the method calls that accept inputs from GUI when making and performing transactions. Instead, we used method arguments to pass in these inputs. This modification separates the application logic from its interface and the modification helps us focus on the functionality under test. However, this modification also brings some side effects. For example, there are only four valid types of transaction (withdrawal, deposit, transfer, inquiry) represented as four integers in the ATM example. With GUI, users can select only one of these four types of transactions. However, by passing arguments, we may be able to select an invalid transaction type when we do not infer constraints on argument value ranges.

After the modifications, we developed an initial test that withdraws from an existing account with a correct ATM card number and PIN. We applied our tool on the modified ATM example with the developed test. Figure 2 shows the subsystem state machine constructed by our tool from the execution of the initial test. Subsystem states are displayed in nodes and state transitions caused by interface calls are expressed as edges. In the state representation inside each node, we show only the fields whose values have been changed by the node's incoming transitions. This simplified view helps developers understand how interface calls change subsystem states.

In the figure, we can observe from the $tr1$ transition that the constructor of `atm.ATM` takes five arguments with the types of `int`, `String`, `String`, `SimulatedBank`, and `InetAddress`. Among them, the type of the forth argument is within our subsystem scope. From the guard condition of the constructor ($tr2$) `arg3 = return SimulatedBank.SimulatedBank()`, we can observe that this argument must be the return value of a previous `SimulatedBank()` constructor. There are no constraints on other arguments. Similarly, for the transition $tr6$ `Transaction.makeTransaction(ATM, Card, PIN, TType)`, its guard conditions also show that these arguments must be the return values of $tr1$, $tr3$, $tr4$, and $tr5$, respectively. For $tr7$, the guard condition `Caller = return Transaction.makeTransaction(ATM,Card,PIN,TType)` shows that the receiver of the method call `performTransaction(int, int, int)` must be the return value of $tr6$.

Then we apply our test generation tool on the inferred SSM to automatically generate new tests. For the arguments whose types are within the scope of subsystem, the tool fills them based on the return-use constraints. For primitive types such as `int`, the tool randomly generates a value within a predefined scope; for the ATM example, we specified the value range for `int` as from 0 to 2. For non-primitive types that are not in the subsystem scope, we currently copy the method sequences for producing objects of these non-primitive types from the initial test. In future work, we plan to dynamically capture and regenerate the method sequences for producing objects of these non-primitive types in the initial test.

Based on the program behaviors exercised by the automatically generated tests, we can divide them into three groups. The first group includes the tests that exercise new program behavior including erroneous behavior. This group of tests is the most valuable one. For the ATM example, our automatically generated tests reveal new program behaviors such as "withdrawal with incorrect account information", "deposit with incorrect account information", and "transfer with incorrect account information". These behaviors are not exercised by the initial test. The second group of new tests are redundant tests, which do not exercise new program behaviors. The third group of tests are invalid tests that may include invalid arguments; these tests could be useful in robustness testing. For the ATM example, some tests generate incorrect transaction types beyond the valid range. When being supplied with incorrect types, `makeTransaction` returns a null reference and thus the subsequent `performTransaction` throws an `NullPointerException`. Some tests include incorrect account numbers. In the GUI of the ATM example, only two types of accounts are allowed: checking and saving, represented by 1 and 2, respectively. In our automatic test generation, 0 is also generated as an account type and this invalid account type causes the program execution to exit abnormally. In future work, we plan to extend our tool to allow users to configure the preconditions or simply value ranges for method arguments.

5. DISCUSSION

There are several major limitations of our current tool. First, our current tool uses offline dynamic analysis, which makes the tool often not scalable for large programs. In particular, we store Daikon traces in files. Our tool then reads all the information in the trace files into memory before the tool does any analysis on the content in the file. This mechanism makes our tool unable to handle large trace files collected from real large applications. One way to address the problem is to use online dynamic analysis, which analyzes execution information immediately after it is collected at runtime. Another way that may alleviate the problem is to collect only information of interest, e.g., the values of fields or arguments of interest.

Second, our approach assumes that all the tests in the initial test suite are valid and correct, so that the behaviors exhibited by the existing program executions are correct according to user's requirements. If the initial test suite exercises erroneous behaviors, subsequently generated tests based on the erroneous behaviors may not be meaningful or valid.

Third, our tool currently relies on the Daikon frontend [5] to collect subsystem states and the collected states may not be complete. Usually a data structure may have nested fields with several levels in depth. By default, the Daikon frontend collects the values of fields that are within the first three levels of depth. We expect that the instrumentation depth of three would be sufficient to reveal system state changes in many cases.

Figure 2: The SSM for an ATM subsystem produced from performing a successful withdrawal transaction.

6. RELATED WORK

Xie and Notkin [18] developed the operational violation approach for generating and selecting unit tests based on operational abstractions [5]. An operational abstraction is program behaviors observed from program executions. In the approach, a programmer runs a set of automatically generated tests and verifies the outputs with the existing operational abstractions. If there is a violation, the programmer will determine if it is the application's desired behavior or a program error. Our Substra framework targets at testing the interactions among multiple classes rather than the behavior of a single class. The notion of operational violation can be similarly applied by treating inferred subsystem state machines as operational abstractions.

Kung et al. [7] extract object state models from a class's source code via static analysis and use them to guide test generation. States in an object state model are defined by value intervals over object fields. These value intervals are derived from path conditions of method source; the transitions are derived by symbolically executing methods. Different from their approach, Substra focuses on testing multiple classes and uses dynamic analysis to infer state models (subsystem state machines) rather than using static analysis.

There exist several approaches for abstracting concrete state machines. Xie and Notkin [17] proposed to use only individual fields to represent an object's states. They also proposed the observer abstraction approach [16] that represents a state of an object by using the return values of observers invoked on the object. Our previous Brastra [19] approach abstracts concrete states based on branch coverage information of methods invoked on the concrete states. All these state abstraction approaches can be integrated into Substra for abstracting states in the subsystem state machines.

From system-test executions, both Whaley et al. [14] and Ammons et al. [2] mine protocol specifications for component interfaces. They use sequencing order among method calls in the interfaces without using internal object-field values. Both approaches usually require a good set of system tests for exercising component interfaces. Substra focuses on sequencing constraints among methods of multiple classes rather than a single class. Substra additionally infers the constraints related to method arguments and returns across multiple methods.

Briand et al. [8] developed a method that reverse engineers UML sequence diagrams from program executions. The method adapts the UML sequence diagram metamodel and defines a trace metamodel that models the collected execution traces. Control flow related positions in the source code (method entry and exit, branch, loop) are instrumented to help collect execution traces. Three consistency rules: MethodCall, Return, and ConditionStatement expressed in OCL are defined to map execution traces to sequence diagrams. In contrast, Substra uses state information to construct subsystem state machines rather than sequence diagrams.

Rountev et al. [11] presented an algorithm to infer UML sequence diagrams from intra-method control flow graphs (CFGs) with static analysis. The approach defines branch successors and loop successors that delineate the scope of branches and loops in CFGs. Armed with this information, the algorithm generates interactive fragments from CFGs and assembles them into sequence diagrams. In contrast, Substra uses dynamic analysis to infer subsystem state machines rather than sequence diagrams.

Saff et al. [12] developed automatic test factoring techniques to generate focused unit tests out of system test executions in order to reduce the cost of running expensive system tests. Orso and Kennedy [9] selectively capture and replay the interactions between the unit under test and its environment during system-test or in-field executions. Unlike Substra, both approaches replay existing program behaviors that have been exercised in the past without generating new tests to exercise new program behavior.

There are a number of approaches for generating integration tests based on specifications. Ali et al. [1] propose an approach to generate integration tests from UML collaboration diagrams and statecharts. The approach constructs SCOTEM (State COllaboration TEst Model), an intermediate test model that represents all the paths that a message sequence may pass through, from UML collaboration diagrams and statecharts. The SCOTEM is fed to a path generator to build test paths. Based on the test paths, tests are manually created and executed to verify the implementations.

Basanieri et al. [3] developed the Cow Suite (Cowtest pluS UIT Environment) tool to generate test suites from UML sequence diagrams and use cases. Cow Suite is the integration of two techniques: UIT (Use Interaction Test) and Cowtest (Cost-Weighted Test Strategy). UIT accepts a set of UML sequence diagrams and generates tests consisting of procedures corresponding to message passing in the sequence diagrams. Cowtest prioritizes and selects generated tests based on the weight of UML diagram elements. The select strategy can be based on a fixed number of tests or fixed functional coverage. The generated test suites can be used as inputs of some test drivers to generate actual tests to test if the sequence diagrams are correctly implemented.

Paradkar [10] developed the Specification and Abstraction Language for Testing (SALT) environment to help test design and automatic test generation. SALT is a modeling language that formally specifies a method under test. The language describes the relationship between method's inputs and outputs as well as the context in which the method is called. Context variables can be single valued, or aggregation of simple values. This context information makes it feasible to automatically generate meaningful tests.

All of the preceding integration test generation techniques require specifications that help generate tests automatically. In contrast, Substra can generate integration tests without requiring specifications.

7. CONCLUSIONS

We have proposed Substra, a framework for automatic generation of integration tests when there are no sequencing constraint specifications for the subsystem under test. Substra infers subsystem state machine (SSM) dynamically from initial-test executions or normal runs of the subsystem under test. The inferred SSM not only shows how the subsystem's state is changed by each method call, but also gives define-use constraints, which help determine the order in which methods should be called and help construct method arguments in a method-call sequence. Given the information, we automatically generate new tests for the subsystem under test. We have developed a tool to implement Substra and applied it on an ATM example. The preliminary results show that the tool can effectively generate integration tests that exercise new program behaviors. Our current implementation focuses on sequential programs; in future work, we plan to investigate techniques for extending our approach to test concurrent programs.

8. REFERENCES

[1] S. Ali, M. J. ur Rehman, L. C. Briand, H. Asghar, Z. Zafar, and A. Nadeem. A state-based approach to integration testing for object-oriented programs. Technical Report SCE-05-08, Department of Systems and Computer Engineering, Carleton University, May 2005.

[2] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proc. 29th ACM SIGPLAN-SIGACT*

Symposium on Principles of Programming Languages, pages 4–16, 2002.

[3] F. Basanieri, A. Bertolino, and E. Marchetti. The Cow-Suite approach to planning and deriving test suites in UML projects. In *Proc. 5th International Conference on the Unified Modeling Language*, pages 383–397, 2002.

[4] R. C. Bjork. ATM example introduction. `http://courses.knox.edu/cs292/ATMExample/Intro.html`.

[5] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123, 2001.

[6] G. T. Heineman and W. T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

[7] D. Kung, N. Suchak, J. Gao, and P. Hsia. On object state testing. In *Proc. 18th International Computer Software and Applications Conference*, pages 222–227, 1994.

[8] L.C.Briand, Y.Labiche, and J.Leduc. Towards the reverse engineering of UML sequence diagrams for distributed real-time Java software. Technical Report SCE-04-04, Carleton University, September 2004.

[9] A. Orso and B. Kennedy. Selective Capture and Replay of Program Executions. In *Proc. 3rd International ICSE Workshop on Dynamic Analysis*, pages 29–35, St. Louis, MO, May 2005.

[10] A. Paradkar. SALT - an integrated environment to automate generation of function tests for APIs. In *Proc. 11th International Symposium on Software Reliability Engineering*, pages 304–316, 2000.

[11] A. Rountev, O. Volgin, and M. Reddoch. Static Control-Flow Analysis for Reverse Engineering of UML Sequence Diagrams. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 96–102, September 2005.

[12] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *Proc. 21st Annual International Conference on Automated Software Engineering*, pages 114–123, Long Beach, CA, November 2005.

[13] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *Proc. International Conference on Fundamental Approaches to Software Engineering (FASE 2006)*, 2006.

[14] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. International Symposium on Software Testing and Analysis*, pages 218–228, 2002.

[15] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th IEEE International Conference on Automated Software Engineering*, pages 196–205, Sept. 2004.

[16] T. Xie and D. Notkin. Automatic extraction of object-oriented observer abstractions from unit-test executions. In *Proc. 6th International Conference on Formal Engineering Methods*, pages 290–305, Nov. 2004.

[17] T. Xie and D. Notkin. Automatic extraction of sliced object state machines for component interfaces. In *Proceedings of the 3rd Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2004)*, pages 39–46, October 2004.

[18] T. Xie and D. Notkin. Tool-assisted unit-test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, 2006.

[19] H. Yuan and T. Xie. Automatic extraction of abstract-object-state machines based on branch coverage. In *Proc. 1st International Workshop on Reverse Engineering To Requirements at WCRE 2005*, pages 5–11, November 2005.