Journal of Software Engineering and Applications, **2013, 6, 500-510**
http://dx.doi.org/10.4236/jsea.2013.69060 Published Online September 2013 (http://www.scirp.org/journal/jsea)

Test Selection on Extended Finite State Machines with Provable Guarantees

Bo Guo, Mahadevan Subramaniam

Department of Computer Science, University of Nebraska Omaha, Omaha, USA.
Email: bguo@unomaha.edu, msubramaniam@unomaha.edu

Received July 22nd, 2013; revised August 21st, 2013; accepted August 29th, 2013

ABSTRACT

Building high confidence regression test suites to validate new system versions is a challenging problem. A model-based approach to build a regression test suite from a given test suite is described. The generated test suite includes every test that will traverse a change performed to produce the new version, and consists of only such tests to reduce the testing costs. Finite state machines extended with typed variables (EFSMs) are used to model systems and system changes are mapped to EFSM transition changes adding/deleting/replacing EFSM transitions and states. Tests are a sequence of input and expected output messages with concrete parameter values over the supported data types. An invariant is formulated to characterize tests whose runtime behavior can be accurately predicted by analyzing their descriptions along with the model. Incremental procedures to efficiently evaluate the invariant and to select tests for regression are developed. Overlaps among the test descriptions are exploited to extend the approach to simultaneously select multiple tests to reduce the test selection costs. Effectiveness of the approach is demonstrated by applying it to several protocols, Web services, and model programs extracted from a popular testing benchmark. Our experimental results show that the proposed approach is economical for regression test selection in all these examples. For all these examples, the proposed approach is able to identify all tests exercising changes more efficiently than brute-force symbolic evaluation.

Keywords: Formal Methods; Model-Based Software Testing; Regression Testing; Extended Finite State Machines

1. Introduction

Maintenance and evolution of complex systems is a challenging problem since all the modifications made to obtain a new system version must be thoroughly tested to gain confidence that the new system functions correctly. Usually, this requires users to select tests that exercise the changes from the original test suite, evaluate the adequacy of selected tests, and possibly design new tests. This is a time consuming and error prone activity.

Usually, regression test suites are built by selecting relevant tests from an available test suite. Automatically identifying relevant tests from a test suite to validate the new system version is called the *regression test selection* problem. This is an active area of research with several earlier works involving software programs ([1-3] are excellent surveys) as well as executable system models [4, 5], which use finite state machines extended with variables (EFSMs). All of these works require the test traces on original versions to be maintained to select tests, which can be impractical for real systems using large test suites. The methods also tend to be safe, leading to the selection of many irrelevant tests that can increase the regression test costs.

In this paper we present a model-based approach for building regression test suites consisting of all and only the relevant tests and without using any additional information about prior test executions. Such an approach has the potential to reduce the maintenance overheads across versions and regression costs due to more compact test suites while increasing tester confidence due to the absence of irrelevant tests. We consider EFSMs whose variables range over a rich set of commonly used data types. System tests are derived from EFSM tests, which are a sequence of input and expected output messages with concrete parameter values over the supported data types. System changes are mapped to EFSM state and transition changes that add/delete/replace one or more EFSM transitions.

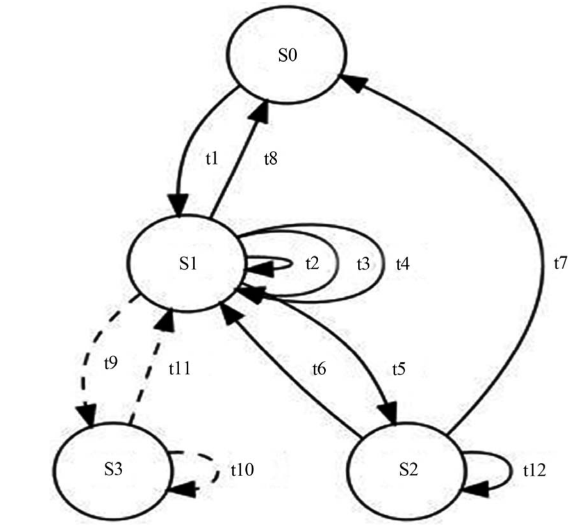
Given a change, and a test suite, the proposed approach automatically analyzes each test description to *a priori* determine the transitions that will be executed when the test is run on the EFSM. We formulate an invariant for each test that describes all the plausible EFSM execution paths that the test can potentially take when it is run on the EFSM. If the invariant is satisfiable, the transitions in the corresponding path are guaranteed to be executed by the test when it is run. The invariant for a test is automatically built using the transitions (and their post-images) *matching* the test description. Informally, a transition is a match for a test description if it can process some test input in the description. A theorem prover is used in a push-button way to determine if the invariant is satisfiable and identifies the associated transitions.

Often, several tests in the given test suite have overlapping descriptions. For instance, it is typical for tests to use the same inputs to bring an EFSM to a common state and then exercise other specific behaviors. Such tests as well as others can be selected simultaneously whenever a given change matches these tests at the overlapping portions of their descriptions. To enable analysis of a group of tests, a test suite is organized into a test forest whose each tree represents a group of tests with overlapping descriptions. We describe a procedure to simultaneously select and discard groups of tests. Such a procedure alleviates regression test selection costs in many cases.

The proposed approach has been implemented and applied to several EFSM models representing protocols, web services, and other applications with encouraging results. Our experimental results show that the proposed approach is economical for regression test selection in all these examples. For all the examples, the proposed approach selects every test that exercises a change while discarding all their relevant tests. Our experiments use a powerful theorem prover called *Simplify* [6] extended with rewrite rules [7].

Overview of the approach using a simple example follows next. Section 2 gives a brief overview of EFSM model. EFSM changes, tests, and change exercising tests are described in Section 3. In Section 4, we describe the formulation of invariant based on test descriptions. Section 5 describes incremental procedures for selecting tests exercising changes and identifying unusable tests. Section 6 extends the approach to handle multiple tests. Section 7 describes related work. Section 8 describes our experiments and Section 9 concludes the paper.

Simple Example: Consider a bank web service EFSM depicted in **Figure 1(a)** whose transitions are given in **Figure 2**. Users start by opening an account with a cash amount greater than or equal to a minimum balance amount (*min*), and are given a unique number as account *id*. The current balance in each account is represented by an array *B*[] mapping the account id to a non-negative



(a)

λ_1: Open(100)/ack(1), deposit(1,50)/ack(B[1]), wdraw(1,160)/ack(B[1]), wdraw(1,10)/ack(B[1]), close(1)/ack(B[1])

λ_2: Open(100)/ack(1),deposit(1,50)/ack(B[1]),wdraw(1,110)/ack(B[1]), wdraw(1,10)/ack(B[1]),close(1)/ack(B[1])

λ_3: Open(100)/ack(1),deposit(1,50)/ack(B[1]),wdraw(1,110)/ack(B[1]), wdraw(1,10)/ack(B[1]),deposit(1,20)/ack(B[1]),close(1)/ack(B[1])

(b)

Figure 1. Bank web service EFSM and tests.

```
t1open(v),(v>=min),s0→s1,{id+=1; B[id]=v;ack(id)}
t2deposit(i,v),(i==id   v>0   T1>0),s1→s1,
{B[id]+=v;T1-= 1; ack(B[id])}
t3wdraw(i,v),(i==id   v>0   (B[i]–v)>=min   T1>0),s1→s1,
                                        {B[id]-=v;T1-= 1;ack(B[id])}
t4wdraw(i,v),(i==id   v>0   (B[i]–v)<0   T1>0),s1→s1,{ack(B[id])}
t5wdraw(i,v),(i==id   v>0   (0<=(B[i]–v)<min)   T1 >0),s1→s2,
                                                  {B[id]-=v;ack(B[id])}
t6deposit(i,v),(i==id   v>0   (B[i]+v)>=min),s2→s1,
                                     {B[id]+=v; T1=max;ack(B[id])}
t7 close(i),(i== id),s2→s0,{ack(B[id])}

t8 close(i),(i== id),s1→s0,{ack(B[id])}

t9 (T1== 0),s1→s3,{bonus=B[id];}

t10(bonus>0),s3→s3,{B[id]+= 1;bonus-= 1;}

t11(bonus== 0),s3→s1,{T1=max;}

t12wdraw(i,v),(i==id ∧ v>0),s2 →s2,{ack(B[id])}

t12'wdraw(i,v),(i==id ∧ v>0 ∧(B[i]–v)>= 0),s2→s2,{B[id]-=v; ack(B[id])}
```

Figure 2. Bank web service transitions.

number. Users operate the account by performing deposits and withdrawals. Withdrawals exceeding the current balance are ignored. That leading to a balance lesser than the *min* value results in a state where withdrawals are ignored and a deposit that brings the balance above the *min* value is only allowed. Accounts accrue a bonus that doubles the current balance provided a specified maximum *max* number of withdrawals (that are not ignored) and deposits succeed in maintaining a balance greater than or equal to the *min* value. The bonus amount is transferred incrementally to the account and no operations are processed during this period. The solid arcs in **Figure 1(a)** are ***external*** transitions that can be observed by their messages. Others (dashed arcs in **Figure 1(a)**) are ***internal*** transitions.

A test suite used to validate the EFSM is given in **Figure 1(b)**. Each test starts the EFSM in the state with *id* = 0; *min* = 50; *max* = 2; (Timer) *T*1 = *max*; *bonus* = 0.

Now, suppose that we delete transition t_{12} and add transition t'_{12} to allow withdrawals when the balance is below min but is non-negative. Usually, tests in the original test suite that are will execute (or executed) the added (or deleted) transition are said to *exercise the change* and selected for regression. Test $\lambda 1$ is not selected because it does not exercise the change whereas tests $\lambda 2$ and $\lambda 3$ are selected because they exercise the change.

The descriptions of these tests can be automatically analyzed to accurately predict whether they will exercise the change when run. So these tests can be selected and/or discarded without running them. The proposed approach develops procedures that analyze test descriptions to select and discard tests to build a high confidence regression test suite.

2. Preliminaries

This section is mostly derived from earlier works [7-11], where more details can be found.

Extended Finite State Machines: An EFSM $E = (I, O, S, V, T)$ [9], is a 5-tuple where I, O, S, V, and T are finite sets of parameterized input and output messages, states, variables, and transitions respectively. Each message in I and O has a finite number of parameters; finite set of variables, $V = X \cup \{IQ, OQ\}$, is the union of the set of data variables X and two message queue variables IQ and OQ denoting the input and output queues from and to the environment respectively. A transition, t: $im(\mathrm{p}[])$, P_t, $s_t \rightarrow q_t$, $om(e[])$, A_t, where $p[] = p_1, ..., p_n$ are distinct, typed parameter variables, P_t is the guard, A_t is an ordered sequence of assignments, and $e[] = e_1, ..., e_w$ is a list of expressions over V and parameters $p[]$. Transitions having an input and an output message are called **external** transitions; others are **internal** transitions.

The **global state** of an EFSM consists of a state from S and a set of constraints over variables. The state is **concrete** if each variable is constrained to a constant value. A state is **initial**, if the constraints satisfy the initial conditions of the system. A transition is **enabled** in a system global state if the message in the global state is an instance of the input message of the transition and the global state satisfies the guard of the transition. **Execution step**, $g \rightarrow {}^{t}g'$, transitions from global state g to g' using t enabled in g. A **run** is a sequence of consecutive steps starting and ending in initial global states.

The **most general post-image** of a transition t, *Mgpos* (t), is a global state representing all the concrete global states that can result after executing the transition t. The concrete post-image of transition t from a global state g, *Cpos*(t, g), is the concrete global state produced by executing transition t in the global state g. If t is not enabled in g then *Cpos*(t, g) has the value *false*. Concrete post-images to deal with the dynamic behavior of tests. We relate the static and the dynamic behaviors of a transition by relating their most-general and concrete post-images with respect to a concrete a global state as: $Cpos(t, g) = Mgpos(t) \wedge g$.

3. EFSM Changes and Tests

3.1. Changes

Changes to the EFSM are specified at the transition level. An addition change, $\delta = <+, t_a>$, adds a new external transition t_a to an EFSM. A deletion change, $\delta = <-, t_d>$, deletes an existing external transition t_d from an EFSM. A replacement change, $\delta = <-/+, (t_d, t_a)>$, replaces an existing external transition t_d in an EFSM by a new external transition t_a. Certain transition changes may have larger impacts and can modify the EFSM interface itself. For instance, an addition change can introduce new EFSM messages and states. Similarly, a deletion change can result in the removal of existing messages and states. Other EFSM changes such as addition or deletion of states and/or variables can all be expressed in terms of changes to the transitions. All the EFSM changes are assumed to produce a new EFSM that is deterministic and consistent.

3.2. EFSM Tests

An EFSM test, $\lambda = <g_0, [i_1/o_1, \ldots, i_n/o_n]>$, is a pair, whose first component is a concrete global state $g0$ and the second component is a finite sequence of test elements of the form: test input/expected test output. Each input (and output) is a sequence of assignments to the message queue and/or data variables. Only constants appear in an input. Both constants and data variables can appear in an output. For brevity, our test inputs and outputs only refer to the messages (queues are implicit) and not to the data variables.

Example: The EFSM tests for the bank example are depicted in **Figure 1(b)**. Test inputs of all these tests assign a single message having constant parameter values to the input message queue and expected outputs assign constant values and variables to the output message queue.

Now, we extend EFSM executions to handle test inputs.

A test input **extended concrete global state** of EFSM is a global state of the EFSM in which the message parameters are bound to the constant values specified by the test input. Transition t **processes** the test input i in the concrete global state g if the transition t is enabled in the extended concrete global state g_i.

Test λ is **applied** to EFSM E by starting E in concrete global state g_0. Transition t_0 enabled in g_0 is executed to generate concrete global state, say, g_1, and the process repeated until no more transitions are enabled in the last generated concrete global state, say, g_m. Extend g_m with the first test input i_1 and execute the enabled transition tm to generate the state g_{m+1}. Transition enabled in g_{m+1} is executed to produce the next state and the process is repeated. The process terminates on either reaching the initial concrete global state after processing all test inputs or if no progress can be made.

4. Structural Test Invariant

Transitions matching test inputs and sequences of sets of transitions matching a test description are described. Test extended most general pre- and post-images are then defined and used to formulate the invariant for tests.

4.1. Matching Transitions and Sequences

Definition: A transition matches a test input if 1) the test input is an instance of the input message of the transition and 2) the instantiated transition guard is satisfiable.

Example: In **Figure 2**, t_5 matches the input *wdraw* (1,110) of test λ_2 since *wdraw*(1,100) is an instance of *wdraw*(i, v) with i = 1, and v = 110, and the instantiated guard of t_5: $(1 == id) \wedge (110 > 0) \wedge 0 <= (B[1] - 110) \wedge (B[1] - 110) < min \wedge (T1 > 0)$, is a satisfiable formula. □

The match operation only checks the input message and the guard but ignores the input state of the transition. Hence the operation is conservative *i.e.*, a transition may match a test input but may not able to process that input when the test is actually applied. However, as shown below, the operation will include all the transitions that can process the test input.

Several EFSM transitions can match a test input. Let $T(i_k)$ be the (possibly empty) set of all transitions matching the test input i_k. A **matching sequence**, $\phi(\lambda) = [T(i_1), ..., T(i_n)]$, of a test λ is a sequence of sets of transitions constructed by point-wise matching of the inputs of the test λ. A transition sequence, $\rho = [t_1, ..., t_n] \in \phi(\lambda)$ if each $t_k \in T(i_k)$, $k = 1n$.

Example: Matching sequence $\phi(\lambda_2) = [\{t_1\}, \{t_2, t_6\}, \{t_3, t_4, t_5, t_{12}\}, \{t_3, t_4, t_5, t_{12}\}, \{t_7, t_8\}]$ for the test λ_2 in **Figure 1(b)**. A transition sequence of $\phi(\lambda_2)$ is $\rho = [t_1, t_2, t_3, t_3, t_7]$. □

4.2. Deriving Invariant from a Test Description

Recall from Section 2 that the most general post-image, of a transition t_w, $Mgpos(t_w)$,is a global state representing all the concrete global states that can result after executing the transition. The **most general pre-image** of transition t_w, $Mgpre(t_w)$, is a global state representing all the concrete global states in which the transition is enabled.

Let the transition t_w match the test input i_w. The test **extended most general post-image of** t_w with respect to input i_w, denoted as $Emgpos(t_w)$, is the set of all global states that can be produced by executing transition t_w instantiated with input *iw*. $Emgpos(t_w) = false$ (empty set of global states) if t_w does not match i_w.

The test **extended most general pre-image of** *tw* with respect to input i_w, denoted as $Emgpre(t_w)$, is the set of all global states where transition t_w instantiated with input i_w is enabled. $Emgpre(t_w) = false$, if t_w does not match *iw*.

A **structural invariant** for a test $\lambda = \langle g_0, [i1/o1, \ldots, in/on]\rangle$, can be formulated using test description and test extended post-images of the matching transitions as:

$$\psi(\lambda) = \bigvee_{\rho \in \phi(\lambda)} \text{Init}(g_0 \wedge \bigwedge_{k=1n}^{n} \text{Emgpos}(t_k)),$$

where the predicate *Init* checks if its argument is an initial global state and ρ is the transition sequence $[t_1, \ldots, t_n]$. Each disjunct of $\psi(\lambda)$ corresponds to a transition sequence ρ from the matching sequence $\phi(\lambda)$ and is made of $n + 1$ conjunctions. First conjunct is the concrete global state g_0 in which the test λ starts and the remaining n conjuncts are the test extended post-images of the n transitions in ρ.

The invariant $\psi(\lambda)$ simply checks that the matching sequence $\phi(\lambda)$ contains at least one feasible run from the concrete global state g_0 in which the test λ is applied. Each disjunct in $\psi(\lambda)$ considers a transition sequence from the matching sequence and incrementally checks its feasibility. The transition sequence is a feasible run if the disjunct is satisfiable. Since EFSMs are deterministic, each test has at most one test run and therefore, at most one of the disjuncts is satisfiable. Hence the invariant is satisfiable if and only if the test run consists of the transitions appearing in the corresponding disjunct. Therefore, all the transitions that will appear in a test run can be accurately predicted (without running the test) if the invariant of the test is satisfied.

An efficient approach to check the satisfiability of the invariant employs a directed *transition control graph* $TCG(\lambda)$ consisting of transitions belonging to the matching sequence $\phi(\lambda)$. The graph has a special *start* node and one node for each occurrence of each transition in the matching sequence $\phi(\lambda)$. Node t_{wj} denotes the occurrence of transition $t_w$in the j^{th} matching set $T(i_j)$ in $\phi(\lambda)$. $TCG(\lambda)$ is a levelized graph with level 0 having the *start* node and level k, $k \geq 1$, consisting of nodes representing all transitions in matching set $T(i_k)$.

Edges in the graph connect a node in a level to zero or more nodes in the next level. Edges can have a label s

(single successor) or a label c (a potential successor). Let t_v and t_w be nodes at levels k and $k+1$ respectively. Edge (tvk, $tw(k+1)$, s) belongs to the graph if the transition t_w must follow t_v in all runs. Otherwise, the edge (tvk, $tw(k+1)$, c) belongs to the graph. An edge from *start* toa node $tw1$ with label s is added if tw is enabled in the initial state g_0.

Example: The **Figure 3** depicts the transition control graph $TCG(\lambda 2)$ for the test $\lambda 2$. □

The procedure to check the invariant is described in **Figure 4**. It takes a test λ and the graph $TCG(\lambda)$ as inputs and outputs *Success* if the transitions executed by the test can be accurately predicted and *Fail* otherwise. The graph is traversed level by level starting at the level 0 to see if any path in the graph forms a feasible run from the concrete global state g_0. At each level, a node is marked by the procedure if the sequence of transitions in the path from the *start* node to the marked node in the graph forms a feasible path from the concrete state $g_0$of the test λ. A label, $L(t_k)$, is associated with the marked node $t_k$in each level k. Let $\rho = [t_1, \ldots, t_w]$ be the transition sequence that forms a feasible path from $g0$ when the procedure reaches the current level (*clevel* in **Figure 4**). The label associated with a node t_k belonging to the sequence ρ is $L(t_k) = g_0 \wedge Emgpos(t_1) \wedge \ldots \wedge Emgpos(t_{k-1})$.

To extend the feasible path to the $w + 1^{th}$level, first, the candidate immediate successor nodes of the marked node at the current level are identified. If the marked node corresponding to t_w has an immediate successor linked by

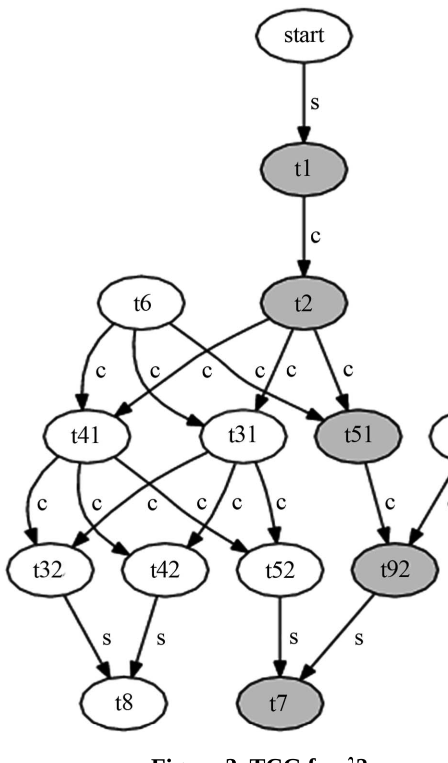


Figure 3. TCG for λ2.

```
Inputs: Test $\lambda = <g_0, [i_1/o_1, \ldots, i_n/o_n]>$,
        $g_0 = (x_s == s_0)$ Λpred;
        Graph $TCG(\lambda)$ with $n$ levels
Output: Success if $\psi(\lambda)$ is satisfiable;
        Fail otherwise
Method:
  L(start) = true;
  Emgpos(start) = $g_0$;
  clevel = 0; Mark start
  while(clevel < n) do
      for the marked node, $t_w$, in clevel do:
      Clean-up:
        if ($t_w$, $t_{w+1}$, s) ϵ TCG then
          TCG = TCG – ($t_w$, $t_v$, c),
                  $t_v$ ϵ TCG
          Delete dangling node and edges
          until none remain
      Extend to next level:
        foreach$t_{w+1}$ ϵ immedsucc($t_w$) do
          F = (L($t_w$) ΛEmgpos($t_w$))
                =>Emgpre($t_{w+1}$))
          if valid(F)
          L($t_{w+1}$) = L($t_w$) ΛEmgpos($t_w$);
          Mark $t_{w+1}$;
          clevel = clevel + 1
        if no immedsucc($t_w$) marked, Fail
  Let $t_n$ be the last marked node
  If Init(L($t_n$) ΛEmgpos($t_n$))
  Return Success
  Else return Fail
```

Figure 4. Check procedure.

an s edge, then this is the only candidate immediate successor. In this case, the other immediate successors of the node corresponding to t_w, if any, are deleted from the graph. Deleting these edges from the graph may make certain nodes dangling *i.e.*, nodes that do not have any successor and/or a predecessor. Such nodes cannot participate in the feasible run, if any, and hence are deleted from the graph along with the resulting dangling edges. Such deletion of dangling nodes and edges is repeated until no more such nodes and edges remain. If the marked node corresponding to t_w has no immediate success or linked by an s edge, then all the nodes are candidate immediate successors.

Then, we propagate the label $L(t_w)$ to each of the candidate immediate successors t_{w+1} and compute a formula $F = (L(t_w) \wedge Emgpos(t_w)) \Rightarrow Emgpre(t_{w+1})$. The formula F states that processing the test input w using transition $t_w$in the concrete global state $g_0 \wedge Emgpos(t_1) \wedge \ldots \wedge Emgpos(t_{w-1})$ will result in a concrete global state in which the immediate successor transition t_{w+1} processing the test input i_{w+1} is enabled. If F is valid then that immediate successor node is marked and the label $L(t_{w+1})$ is set. The current level is incremented and the procedure is continued. If the propagated formula F is not valid for any of the candidate successors then this implies that no transition in the matching set $T(i_{w+1})$ can process the test input i_{w+1} in the concrete global state obtained after processing the first w test inputs. And, the procedure fails. If the path can be extended up to the last level (level n) of the graph and the $L(t_n) \wedge Emgpos(t_n)$ is an initial concrete global state then the procedure returns *Success*. Otherwise, the procedure returns *Fail*.

5. Selecting Tests

The model-based regression test selection problem using the EFSMs is analogous to that for programs [10]. It takes as inputs–a deterministic, consistent EFSM E_1 with a test suite T, and a change δ that produces a modified, deterministic and consistent EFSM E_2. It outputs a test suite $T' \wedge T$, consisting of subset of tests of T guaranteed to exercise the change δ on E_2.

Test $\lambda = <g_0, [i_1/o_1, ..., i_n/o_n]>$ exercises an addition change $\delta = <+, t_a>$ on E_2 if there exists a feasible path $\rho = [t_1, ..., t_a]$ from g_0 on E_2. Test λ exercises a deletion change $\delta = <-, t_d>$ on E_2 if there exists a feasible path $\rho = [t_1, ...,t_d]$ from g_0 on E_1. Test λ exercises a replacement change, $\delta = <-/+, (t_d, t_a)>$ on E_2 if it either exercises the addition change $<+, t_a>$ on E_2 or it exercises the deletion change $<-, t_d>$ on E_2.

For test selection, we slightly generalize the satisfiability of the invariant. Let λ be any test for an EFSM E.

Definition: Let $\rho = [t_1, …,t_k]$ be a transition sequence obtained from a path: $start \rightarrow t_1 \rightarrow ... \rightarrow t_k$ of $TCG(\lambda)$.

The invariant $\psi(\lambda)$ is *k-satisfiable* if ρ is a feasible path over E for the test λ.

5.1. Selecting Tests for Addition Changes

The main steps of an incremental procedure to determine if the test λ is a candidate for an addition change δ are given below. The procedure takes the original model E_1, the matching sequence, the graph $TCG(\lambda)$, and the added transition t_a as its inputs and returns 1 if λ is a candidate test for δ and returns 0 otherwise. It also outputs the updated compatibility graph to be used for future changes.

1) Update the original matching sequence $\phi(\lambda)$ by adding t_a to the appropriate matching sets.

2) Suppose that t_a occurs exactly once in the k^{th} set of $\phi(\lambda)$. Check if $\psi(\lambda)$ is *k-satisfiable* on E_1using the original graph $TCG(\lambda)$. If so then not t_a, but some original transition, will process the k^{th} test input on new EFSM E_2. If $\psi(\lambda)$ is only(k − 1)-*satisfiable*, let $F = (L(t_{k-1}) \wedge Emgpos (t_{k-1})) \Rightarrow Emgpre(t_a)$, where t_{k-1} is the node marked at level $k - 1$. If F is not a valid, t_a will not process the k^{th} input on E_2 (this is also the case when $\psi(\lambda)$ is satisfiable to a level less than $k - 1$). So, λ is not a candidate; the graph is unchanged.

If $\psi(\lambda)$ is $(k - 1)$-*satisfiable*, but not *k-satisfiable* and F is a valid then λ is a candidate and the transition t_a will process the k^{th} input on E_2. Update $TCG(\lambda)$ by adding node t_a at level k and edge (t_{k-1}, t_a, s). Attempt to extend the feasible path to level $k + 1$ using the procedure described in **Figure 4**, using the updated graph. If successful and t_{k+1} is the marked, add edge, (t_a, t_{k+1}, s), to the updated graph and output the resulting graph.

3) If t_a occurs in many sets of sequence $\phi(\lambda)$, all in the interval $[l, m]$ and $\psi(\lambda)$ is satisfiable up to level m or higher, then λ is not a candidate and the graph is unchanged. Otherwise, process each level in the interval, starting at level l, as described above. Go to next level only if t_a will not process inputs at the previous levels.

4) Finally, if t_a does not occur in sequence $\phi(\lambda)$, the test λ is not a candidate and the graph is unchanged.

The above incremental procedure uses the original graph to identify candidate tests. The graph is locally updated so that they can be similarly used in selecting tests for future changes. Such an incremental procedure can be effective in practice since it is bounded by the size of the matching sequence elements affected by the change and is independent of the overall size of the EFSM. Further, focusing on the earliest occurrence of change transition can reduce the analysis time, especially for long tests.

5.2. Selecting Tests for Deletion Changes

The main steps of the procedure are similar to those of the procedure for the addition change described above. The only difference is in the updating of the compatibility graph. While handling a deletion change, if a node corresponding to transition t_d in the graph is marked by the invariant checking procedure run on E_1, the corresponding test is selected. In this case, the updated graph is obtained by deleting every node and the resulting dangling edges from the graph corresponding to transition t_d. The process is repeated with all the nodes that do not have an immediate predecessor or an immediate successor until no more such nodes can be found and the resulting graph is returned as the updated graph.

Replacement changes are viewed as a pair of addition and deletion changes. Tests are selected if they are chosen for either.

6. Simultaneous Test Selection

Often, several tests in the given test suite start in the same concrete global state and have overlapping test inputs. For instance, it is typical for tests to use the same inputs to bring the EFSM to a common state and then exercise other specific behaviors. Such tests can be selected simultaneously whenever a change matches these tests at the overlapping portions of their test inputs. To simultaneously select and discard tests, the test suite $T =$

$\{\lambda_1, ..., \lambda_n\}$ of the original EFSM $E1$ is partitioned into groups of tests. Each group $G = \{\lambda_1, ..., \lambda_k\} \subseteq T$ consists of tests that are applied in the same concrete global state g_0 and share a non-empty prefix of their test inputs, *i.e.*, they have at least the same first test input.

Each group G is represented using a test suite tree (TST). Each node of the TST tree denotes a test input occurring at a particular position in the non-empty prefix of the test inputs of the tests in G. The root node of TST denotes test input of the tests in G occurring at the first position of the prefix. Node v is a child of node u in TST if in some test in G, the test input i_u at a position p in the prefix is an immediate predecessor of the test input iv at position $p + 1$. The edge between a parent node u and child node v is labeled by the set consisting of all tests where this is the case. Further, the set of tests labeling an edge between a parent and a child node is the union of all tests appearing in the subtree rooted at the child node.

Example: TST in **Figure 5** represents the test suite of **Figure 1(b)**.

Below, we describe a procedure to simultaneously select and discard tests from T to build a test suite T' for the new EFSM E_2 for addition change $\delta = \langle +, t_a \rangle$.

Consider a TST comprised of a group of tests G all starting in the concrete global state g_0. To select tests from this TST, for each node u, the set of transitions of the EFSM $E2$ matching the test input i_u, $T(i_u)$, is computed. Each matching set $T(u)$ of the node u is maintained at that node. If t_a does not appear in any matching sets, none of the tests in the TST are chosen.

Suppose that t_a appears in some matching set $T(i_u)$ at node u of the TST. Let the sequence of matching sets from the root node of the TST to the node u be the matching sequence $\alpha = [T(i_1), ..., T(i_n), T(i_u)]$. We build a control graph using the transitions appearing in α and use

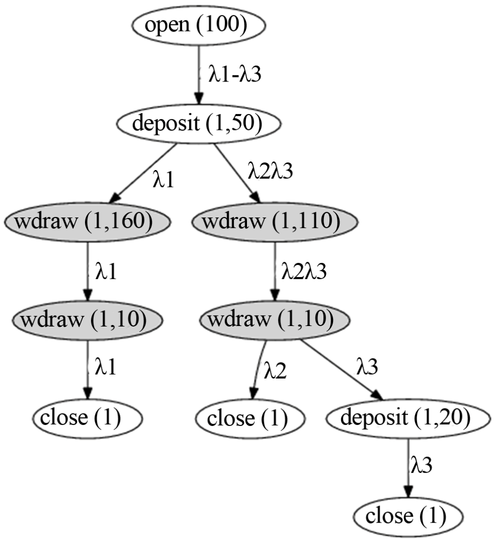


Figure 5. TST for bank example.

the procedure in **Figure 4** with the graph and g_0 as its inputs to determine if some transition sequence belonging to α forms a feasible path from g_0 on E_2. If the procedure returns *Success* and transition t_a is the transition marked by the procedure in the matching set $T(i_u)$, all tests labeling the edge between i_n and i_u in the TST are chosen. Node i_u and its descendants are removed.

If the procedure returns *Fail* because only a prefix of the matching sequence α, say, $[T(i_1), ..., T(i_m)]$ contains a feasible path then the TST is updated by setting the sets $T(i_1), ..., T(i_m)$ to the respective transitions marked by the procedure. The subtree of TST rooted at the node i_{m+1} is removed (all tests in this subtree are discarded since they do not exercise the addition change).

The procedure can also return *Success* but t_a may not be the marked transition in the matching set $T(i_u)$. In this case, we update the matching sets with the transitions marked by the procedure to incrementally continue analyzing the extensions of the matching sequence α reaching other nodes of the TST whose sets include t_a.

The left-right traversal of the updated TST (and the test suite T) is continued until all nodes in the TST whose sets contain t_a have been analyzed. All the tests chosen in are included in T' and the same process is repeated with each TST. Procedures to select multiple tests for deletion and replacement changes are similar.

Example: The matching nodes u for the bank example are highlighted in **Figure 5**. A left-to-right traversal of this tree selects tests $\lambda2$, $\lambda3$ at level 4 after which the nodes at the lower levels can be removed. □

7. Related Work

Approaches for regression testing have been broadly classified as being code-based or model-based. Yoo and Harman [3] is a recent survey on regression test selection and related problems. Rothermal and Harrold [2] and Harrold and Orso [1] are two other surveys emphasizing code-based approaches.

Code-Based Approaches: Code-based approaches work on Programs and have been extensively studied earlier. The above surveys discuss many of these approaches in detail. Most of these approaches perform control and data flow analysis to determine the difference between original and modified programs and use available test traces to determine if the test should be selected for regression. However, these methods are conservative do not target precise selection of tests [12]. On the other hand, precisely selecting tests is a primary goal of this paper, which is crucial for high-confidence test suites.

Model-Based Approaches: Model-based approaches use executable models and model-programs instead of actual code to select regression tests. Recently, there has been a lot of interest in this area due to the advent of embedded systems such as automobiles [13] and complex component based systems [14], where early testing of models can alleviate the validation costs of actual systems. Earlier works on model-based regression testing include works by Briand *et al.* [15,16] using UML models and that by Korel *et al.* [5,17,18] using EFSM models. The approach described in [15,16] extracts changes by comparing two versions of a class, use case/sequence diagrams in a UML design. These changes are then used to classify a test as obsolete, retestable, re-usable by mapping a test to a complete message sequence in a sequence diagram. Earlier works using EFSMs have focused on regression test minimization, and prioritization problems, unlike test selection considered in this paper.

The proposed approach differs from the above model-based approaches in its focus on constructing accurate regression test suites based on models. Unlike the earlier approaches, in the proposed approach no test exercising a change in the EFSM is missed and every test that does not exercise any change in the EFSM is selected. Second, our EFSM models and the analysis procedures handle a rich set of data types and allow test cases to have constant values involving all these data types including aggregate data types. Finally, the proposed approach exploits the overlap in test descriptions to simultaneously select or discard tests for building a regression test suite.

8. Experiments

We refer to the proposed approach as SPG (selection with provable guarantees) in this section. We have implemented SPG and applied it several web services, protocols, as well as many model programs taken from a well-known testing benchmark [19]. Our objectives for these experiments are: study the efficiency of SPG for regression test selection for EFSMs. Our prototype is coded in Perl and C on a Linux server with 4GB memory and employs built-in graph libraries. It uses the reasoning framework, *SAIL*, implemented based on the theorem prover *Simplify* [6] extended with queues [11].

Change and Test Generation: We use the changes provided by the applications whenever they are available. In addition, changes are synthetically generated using the following simple process. Given an EFSM model (text files), the number of transitions to be changed, and the overall number of mutated EFSMs, the input EFSM is first compiled into a graph. A new graph corresponding to each new EFSM is got from the original graph by marking the number of transitions given as input with the change actions *a* (addition) or *d* (deletion), chosen randomly. The process is repeated to generate the new EFSMs.

The test suite for the original EFSM is hand crafted wherever possible, such as those for model programs from [19]. Tests were also automatically generated using the model-based test generator ParTeg [20].

8.1. Case Studies

We have applied the prototype to ten examples from the literature: completion (*Cmp*), two-phase commit (*Tcp*), and conference (*Cnf*), and third-party call (*Thp*), Cruise Control (*Con*), Printtokens (*Pri*)2, automatic teller machine (*Atm*) [4,5], bank (*Bnk*),vending machine (*Ven*), and a Microwave oven (*Mic*) [21]. The completion, two-phase commit, and conference protocols described on the web-site3 have been used earlier to evaluate formal testing approaches. The completion protocol *Cmp* is used by an application to tell the coordinator to try, commit, or abort an atomic transaction. Two-phase commit *Tcp* is a coordination protocol that defines how participants reach an agreement on the outcome of an atomic transaction.

The conference protocol, *Cnf*, is a chatbox-like protocol. The EFSM models for *Cmp* and *Tcp* were manually created using their graphical and textual descriptions and contain 7 and 14 transitions respectively. Their test suites have 300 and 800 tests respectively. For the *Cnf* protocol, we used the EFSM description (c) available from the website referred above. This model has 19 transitions and the test suite has 723 tests. The website gives two EFSMs called (c) and (d) and describes four changes to transform EFSM (c) to EFSM (d). The four changes specified there are all additions that allow members to send data before joining the conference. The third-party call (*Thp*) is a protocol from Praxis with 15 transitions and 837 tests [22].

Cruise Control (*Con*) and Printtokens (*Pri*) are programs from the popular test benchmark [19]. These programs are manually translated to obtain the EFSM models. The EFSM for *Con* has 13 transitions and 1000 tests. EFSM for *Prn* has 89 transitions and 1439 tests. Microwave Oven (*Mic*) is originally described as a Kripke structure [21]. We simply modified the labels on the arcs to obtain EFSM transitions by adding input and output messages. The model has 12 transitions and 1160 tests. The remaining examples web automatic teller machine (*Atm*) (6 transitions and 800 tests), bank (*Bnk*) (9 transitions and 1124 tests), a vending machine example (*Ven*) (8 transitions and 87 tests) all appear as EFSMs in an earlier testing paper [23] and were used as such.

8.2. Results

Overview: Our results for the SPG approach are summarized in a table in **Table 1**. The first column of the table lists the ten examples along with the number of EFSM transitions. The second column lists the test suite size and the average test length for each example.

Columns three, four, and five show *C*1, the cost for running the full test suite, *C*2, the cost for running the selected tests, and *C*3, the cost for performing analysis respectively. These columns list the average costs per

Table 1. Regression test selection costs for SPG.

Case	TestSuite		C1 sec	C2 sec	C3 sec	Select Tests	Avg time Save (%)
	Size	Avg length					
Con (13)	1000	78	2152	837	629	386	32
Prn (89)	1439	102	6345	3511	2035	798	13
Atm (6)	800	50	1210	842	214	486	13
Mic (12)	1160	12	289	92	7	614	66
Bnk (9)	1124	35	2483	738	1041	364	28
Thp (15)	837	66	1249	367	232	203	52
Ven (8)	87	37	92	23	50	20	20
Cnf (19)	723	47	629	328	187	257	18
Cmp (7)	800	59	532	252	132	316	28
Tcp (14)	311	18	147	48	27	102	49

change. Next column is the average number of selected tests per change. Finally, the last column lists the average time savings per change, defined as the percentage ($C1$ - ($C2$ + $C3$))/$C1$, based on the cost model of [24].

As seen from **Table 1**, SPG achieves an average time savings of around 30% while achieving an average reduction of around 40% in the size of the test suite selected for all attempted examples.

Time Savings: Varying amount of time savings obtained across these examples can be mainly attributed to three reasons: complexity of the data values, and data types used in the EFSMs and the tests, exploiting overlap in the test descriptions, and the compatibility of transitions. Time savings higher than 50% for *Mic* are mainly due to simple data types such as boolean and integers whereas *Atm*and *Bnk* do not have as much time savings since their EFSMs and tests involve arrays. Time savings are significant for *Thp* because the input messages in its EFSM have no parameters. Consequently, its tests do not involve any concrete values and allow for lot more overlaps in the test descriptions. These overlaps are effectively exploited by our simultaneous test selection procedures using the TST trees.

Reduction in Test Suites: The variance in the number of selected tests largely depends on the number of transitions in the EFSM models, and those appearing in loops. Consider the examples *Atm* with 6 transitions and high (486) average number of selected tests and *Ven* with 8 transitions and a very low (20) average number of selected tests. The difference in the average number of selected tests in these examples can be attributed to the number of transitions appearing in loops. Almost all transitions of example *Atm* appear together in one or more loops. Hence a feasible run corresponding to each test is likely to contain all of the transitions and this leads to high number of selected tests. This is in contrast with *Ven* where loops contain at most one or two transitions. Reduction in test suite size is directly related to time savings in examples such as *Prn*, *Atm*, and *Thp*. However, in *Con*, almost 70% of the tests are discarded but time savings are not as much. This is because discarded do not take much time to run. Similarly, in *Mic* few tests are eliminated but the time savings are higher.

Fault Detection Using SPG: We also studied whether faults in the system under test can be detected by SPG.

We considered faults that are caused solely due to the changes in the model. The criterion for test selection used by SPG is a necessary condition for detecting such faults. We used the popular TCAS example from the test benchmark [19] with 41 versions and 1590 tests. We chose the faulty versions *Ver*1, *Ver*2, *Ver*6, *Ver*7, *Ver*8, and *Ver*9 produced by mutation analysis. We created models from each of the faulty versions of the code and translated the code-based tests to model based tests. SPG was used to select the model-based tests. The corresponding code-based tests were run on the original and modified code to identify the model-based tests revealing faults. Our results are depicted in **Table 2**. The first column gives the faulty versions; the second column gives the tests selected by SPG; the third column gives the number of selected tests that reveal faults.

As shown, SPG was able to identify a non-zero number of fault-revealing tests for each version. To further check if SPG missed any of the fault-revealing tests, we ran all the code-based tests in the original test suite on both the original and modified C programs and collected the tests producing different outputs. These tests were the same as those identified using SPG in all cases. Hence SPG selects all the fault-revealing tests in all the versions.

SPG vs. SYM: The first and third bars in **Figure 6** depict the results of our comparison of the analysis costs ($C3$) of the SPG and brute-force symbolic execution (SYM) [25]. The X-axis plots the examples and the Y-axis plots the analysis cost in seconds. Results show that cost of SYM is higher than that of SPG in all examples. This is because, first, SYM does not exploit the test

Table 2. Regression test selection costs for SPG.

TCAS (1590 tests)	Selected tests	Fault Reveal tests
*Ver*1	432	130
*Ver*2	527	61
*Ver*6	331	12
*Ver*7	1560	36
*Ver*8	1560	1
*Ver*9	508	9

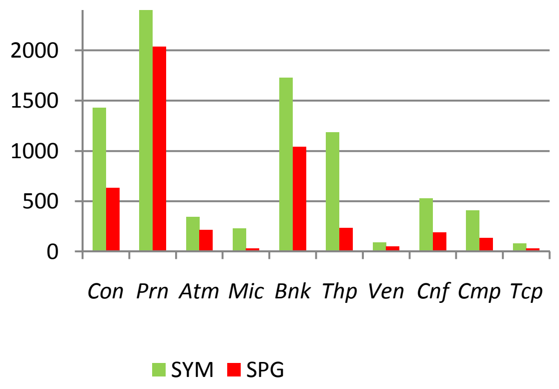


Figure 6. Regression test selection costs for SPG.

description and hence considers every transition in each execution step and second, SYM analyzes non-modification traversing tests in their entirety.

9. Conclusion

An EFSM model-based formal approach to accurately select tests for regression testing is described. Test descriptions are statically analyzed using the model to formulate a structural invariant such that the transitions that will be executed by the test can be provably predicted whenever the invariant is satisfiable. Bounded, incremental procedures for selecting tests guaranteed to exercise addition/deletion/replacement changes and identify unusable tests are described. We also extend the procedures to simultaneously select and discard multiple tests from a test suite by exploiting the overlap in the test descriptions. The effectiveness of the approach is illustrated on several examples including programs from a popular benchmark, web services, and protocols. Our results show that our approach achieves an efficiency of around 30% in all these examples and reduces test suit sizes up to 40%, while being fully inclusive and precise.

REFERENCES

[1] M. J. Harrold and A. Orso, "Retesting Software during Development and Maintenance," *Frontiers of Software Maintenance*, Beijing, 28 September-4 October 2008, pp. 99-109.

[2] G. Rothermel and M. J. Harrold, "Analyzing Regression Test Selection Techniques," *IEEE Transactions on Software Engineering*, Vol. 22, No. 8, 1996, pp. 529-551. doi:10.1109/32.536955

[3] S. Yoo and M. Harman, "Regression Testing Minimization, Selection, and Prioritization: A Survey," *Software Testing*, *Verification*, *and Reliability*, Vol. 22, No. 2, 2010, pp. 67-120. doi:10.1002/stvr.430

[4] Y. Chen, R. L. Rrobert and H. Ural, "Regression Test Suite Reduction Using Extended Dependence Analysis," *4th Workshop on Software Quality Assurance*, 2007, pp. 62-69.

[5] B. Korel, L. Tahat and B. Vaysburg, "Model Based Regression Test Reduction Using Dependence Analysis," *International Conference on Software Maintenance*, 2002, pp. 214-223.

[6] D. Detlefs, G. Nelson and J. B. Saxe, "Simplify: A Theorem Prover for Program Checking," *Journal of the ACM*, 52, 3, 2005, pp. 365-473. doi:10.1145/1066100.1066102

[7] D. Kapur and H. Zhang, "An Overview of Rewrite Rule Laboratory (RRL)," *Conference on Rewriting Techniques and Applications*, Chapel Hill, 3-5 April 1989, pp. 559-563.

[8] B. Daniel and Z. Pitro, "On Communicating Finite-State Machines," *Journal of the ACM*, Vol. 30, No. 2, 1983, pp. 323-342.

[9] D. Lee and M. Yiannakakis, "Principles and Methods of Testing Finite State Machines-Survey," *IEEE Computers*, Vol. 84, No. 8, 1996.

[10] M. Subramaniam and P. Chundi, "An Approach to Preserve Protocol Consistency and Executability across Updates," *Conference on Formal Engineering Methods*, Seattle, 8-12 December 2004, pp. 341-356.

[11] M. Subramaniam and B. Guo, "A Rewrite-Based Approach for Change Impact Analysis of Communicating Systems Using a Theorem Prover," Tech Report, Department of Computer Science, University of New Orleans, New Orleans, 2008.

[12] G. Rothermel and M. J. Harrold, "A Safe, Efficient Regression Test Selection Technique," *ACM Transactions on Software Engineering and Methodology*, Vol. 6, No. 2, 1997, pp. 173-210.

[13] E. Bringmann and A. Kramer, "Model-Based Testing of Automotive Systems," *Conference on Software Testing*, *Verification and Validation* (*ICST*'08), Lillehammer, 9-11 April 2008, pp. 485-493.

[14] Y. Chen, R. L. Probert and D. P. Sims, "Specification-Based Regression Test Selection with Risk Analysis," *Conference of the Centre for Advanced Studies on Collaborative Research*, 2002.

[15] L.C. Briand, Y. Labiche and S. He, "Automating Regression Test Selection Based on UML Designs," *Information and Software Technology*, Vol. 51, No. 1, 2009, pp. 16-30. doi:10.1016/j.infsof.2008.09.010

[16] L. C. Briand, Y. Labiche and G. Soccar, "Automating Impact Analysis and Regression Test Selection Based on UML Designs," *Conference on Software Maintenance*, 2002, pp. 252-261.

[17] B. Korel, L. H. Tahat and M. Harman, "Test Prioritization Using System Models," *Conference on Software Maintenance*, 26-29 September 2005, pp. 559-568.

[18] B. Vaysburg, L. H. Tahat and B. Korel, "Dependence Analysis in Reduction of Requirement Based Test Suites," *Symptoms on Software Testing and Analysis*, Vol. 27, No. 4, 2002, pp. 107-111.

[19] H. Do, S. Elbaum and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and Its Potential Impact," *Empirical Software Engineering*, Vol. 10, No. 4, 2005, pp. 405-435. doi:10.1007/s10664-005-3861-2

[20] S. Weibleder, "Parteg". http://parteg.sourceforge.net/

[21] E. M. Clarke, O. Grumberg and D. A. Peled, "Model Checking," MIT Press, Cambridge, 1999.

[22] C. Keum, S. Kang, I. Ko, J. Baik and Y. Choi, "Generating Test Cases for Web Services Using Extended Finite State Machine," *Conference on Testing of Software and Communication Systems*, New York, 16-18 May 2006, pp. 103-117.

[23] M. Subramaniam, L. Xiao, B. Guo and Z. Pap, "Approach for Test Selection for EFSMs Using a Prover," *Conference on Testing of Software and Communications Systems*, Eindhoven, 2-4 November 2009, pp. 146-162. doi:10.1007/978-3-642-05031-2_10

[24] H. K. N. Leung and L. White, "A Cost Model to Compare Regression Test Strategies," *Conference on Software Maintenance*, Sorrento, 15-17 October 1991, pp. 201-208.

[25] J. C. King, "Symbolic Executed and Program Testing," *Communications of the ACM*, Vol. 19, No. 7, 1976, pp. 358-394. doi:10.1145/360248.360252