An Approach for Test Selection for EFSMs Using a Theorem Prover

Mahadevan Subramaniam[1], Ling Xiao[1], Bo Guo[1], and Zoltan Pap[2]

[1] Computer Science Department
University of Nebraska at Omaha
Omaha, NE 68182, USA
msubramaniam@mail.unomaha.edu
[2] Ericsson, Hungary
pap@tmit.bme.hu

Abstract. This paper describes an automatic approach for selecting tests from a test suite to validate the changes made to an extended finite state machine (EFSM). EFSMs supporting variables over commonly used data types including booleans, numbers, arrays, queues, and records, and communicating with the environment using parameterized messages are considered. Changes to the EFSM add/delete/replace one or more transitions. Tests are described using a sequence of input and output messages with parameter values. We introduce a class of *fully-observable* tests. The description of a fully-observable test contains all the information to accurately determine the transitions executed by the test. Interaction among the EFSM transitions captured in terms of a *compatibility* relation is used along with a given test description to automatically identify fully-observable tests. A procedure is described for selecting a test for a given change based on accurately predicting if the test executes the change transition. We then describe how several tests can be simultaneously selected by grouping them based on overlap of their descriptions. The proposed approach has been implemented using a theorem prover and applied to several examples including protocols and web services with encouraging results.

1 Introduction

Evolution and maintenance of communication systems is a challenging problem. Comprehensive testing of the changes in each evolution step is essential to gain confidence that the modifications behave as intended without any adverse consequences. Test suites used to validate these systems are usually very large, comprised of both hand-crafted and generated tests. Re-running the entire test suite at each evolution step is impractical [8].

Regression test selection addresses this problem by identifying tests in a given test suite that are relevant to validate the changes performed in an evolution step. This is an active area of research with earlier works involving the evolution and maintenance of software programs (see [9] for an excellent survey) as well as state-based communication system models [2, 5, 6].

In this paper, we describe a novel and automatic approach for test selection for extended finite state machines(EFSMs) supporting a rich set of commonly used data types including booleans, numbers, arrays, queues, and records. Changes to the EFSM in each evolution step are performed at the transition level. Changes can add/delete/replace one or more transitions. Test descriptions contain a sequence of input and output messages with parameter values over the supported data types. EFSMs have been extensively used to model communication systems and also serve as formal models underlying several state-based concurrent specification languages. More recently, there has been a lot of interest in EFSMs supporting various data types to model web services[1].

Given a change and a test description, the proposed approach automatically analyzes the test description to determine whether the test will exercise the change in which case the test is selected. A class of tests called **fully-observable tests** is identified. The description of a fully-observable test contains adequate information to accurately determine if the test exercises a given change without actually executing the test. Interaction among the transitions, captured in terms of compatibility of transitions, is used to identify fully-observable tests. A procedure for identifying fully-observable tests that uses a theorem prover in a push-button manner to reason about data types is described. A given change is **matched** with a test description and it is shown that it suffices for a test to be fully-observable up to the point of the match to accurately predict whether a change will be exercised. A procedure to select tests is described. We also describe how several tests can be simultaneously selected by grouping them based on overlap of their descriptions.

The proposed approach has been implemented using the theorem prover *Simplify* extended with an in-house rewrite engine and has been applied to several EFSMs. Our initial results from preliminary experiments on seven examples including protocols and web services from the literature are highly promising. They show that the time for test selection together with the running of the selected tests is always lesser than re-running the entire test suite [8].

An overview of the approach using an example follows next. Section 2 describes related work. Section 3 defines a compatibility relation among transitions. Section 4 introduces fully-observable tests and a procedure for identifying these tests. Section 5 discusses how tests are selected for a given change and extends this approach to handle multiple tests in a test suite. Section 6 describes experiments and Section 7 concludes the paper.

1.1 A Simple Example

Consider a simple bank web service EFSM depicted in Figure 1. 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 account id (a positive integer). They can then perform deposits and withdrawals using the *id* until closing the account. Withdrawals exceeding the current account balance represented by the

[1] Please see Testcom2006, Testcom2007, Testcom 2008 for relevant papers.

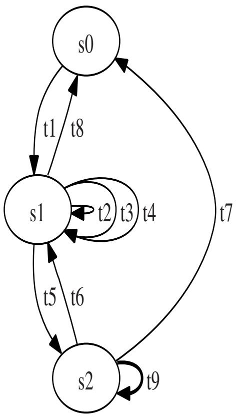

Fig. 1. Bank web service with Overdraft Update

t1	open(v), (v >= min), s0 → s1, {id += 1; bal[id] = v; ack(id)}
t2	deposit(i, v), (i == id ∧ v > 0), s1 → s1, {bal[i] += v; ack(bal[i])}
t3	wdraw(i, v), (i == id ∧ v > 0 ∧ (bal[i] - v) >= min), s1 → s1, {bal[i] -= v; ack(v)}
t4	wdraw(i, v), (i == id ∧ v > 0 ∧ (bal[i] - v) < 0), s1 → s1, {ack(0)}
t5	wdraw(i, v), (i == id ∧ v > 0 ∧ 0 <= (bal[i] – v) < min), s1 → s2, {bal[i]-= v; ack(v)}
t6	deposit(i, v), (i == id ∧ v > 0 ∧ (bal[i] + v) >= min), s2 →s1, {bal[i] += v; ack(bal[i])}
t7	close(i), (i == id), s2 → s0, {ack(bal[i])}
t8	close(i), (i == id), s1 → s0, {ack(bal[i])}
t9	wdraw(i, v), (i == id ∧ v > 0 ∧ (bal[i] - v) >= 0) , s2 → s2, {bal[i] -= v; ack(bal[i])}

Tests

λ1	open(100)/ack(1), deposit(1,50)/ack(150), close(1, 50)/ack(150).
λ2	open(100)/ack(1), deposit(1,50)/ack(150), wdraw (1,75)/ack(75), wdraw(1,50)/ack(50), close(1)/ack(25).
λ3	open(100)/ack(1), deposit(1, 50)/ack(150), wdraw (1, 110)/ack(110), wdraw(1, 40)/ack(40), close(1)/ack(0).
λ4	open(100)/ack(1), deposit(1, 50)/ack(150), wdraw(1, 10)/ack(10), wdraw (1, 100)/ack(100), wdraw(1, 40)/ack(40), close(1)/ack(0).
λ5	open(100)/ack(1), deposit(1, 50)/ack(150), wdraw (1, 110)/ack(110), wdraw(1, 40)/ack(40), deposit(1,40)/ack(40), wdraw(1, 40)/ack(40), close(1)/ack(0).

Fig. 2. Transitions and tests

array *bal*[] are returned and a withdrawal resulting in balance falling below the *min* amount suspends future withdrawals until the balance becomes at least the *min* amount.

The EFSM has 8 transitions t_1-t_8, depicted in Figure 2. Suppose we modify this EFSM by adding a transition t_9 to allow withdrawals not exceeding the balance even when the balance falls below *min*[2].

Consider the test suite in Figure 2, with tests λ_1-λ_5 for validating this new EFSM. Each test in this suite starts the EFSM with global variables $min = 50$; (account id) $id = 0$; $bal[] = initarray$ (array with all accounts having an undefined value) and serially applies the test inputs and compares the generated outputs with the corresponding outputs in the test to assign pass/fail verdict. Each test input is processed after executing all the transitions enabled by the previous test input. Such a test suite may include tests used for original EFSM and new tests generated using approaches like constrained path selection [5].

[2] Note that unspecified inputs received in an EFSM state are simply ignored. For e.g., withdrawals exceeding balance in state s_1 return 0 whereas they are ignored in state s_2 without affecting the balance.

We want to find which tests from the test suite in Figure 2, if any, are to be re-run to check the newly added transition t_9. Usually, tests which execute t_9, are the candidates[3].

First, considering each test one by one, the input conditions (and the output message) of t_9 are matched with those in the test description[4]. Transition t_9 is likely to be executed by a test to process its matching test input(s).

Test λ_1 where there is no match need not be re-run.

The next test λ_2 is matched by transition t_9 at its third and fourth inputs and is further analyzed to determine if t_9 is guaranteed to be executed by this test. A sequence of sets of transitions, $\phi = [\{t_1\}, \{t_2,t_6\}, \{t_3,t_4,t_5,t_9\}, \{t_3,t_4,t_5,t_9\}, \{t_7,t_8\}]$, point-wise matching the inputs of λ_2 is obtained. Note that, if any of the sets in ϕ are empty then the test has unspecified inputs and is discarded. Pairs of transitions from consecutive sets of ϕ are analyzed to see if ϕ has all the information about the transitions that the test λ_2 will execute. If so, we form an executable path up to (and including) the processing of the test input(s) matched by t_9 and choose the test as a candidate to be re-run if this path includes t_9.

The first test input of λ_2 must be processed by the transition t_1 in the first set in ϕ. Hence the description of λ_2 has all the information about the transitions that λ_2 will execute to process the first test input.

Now, we check if the second input of λ_2 can be processed by t_2 or t_6 from the second set in ϕ, immediately following t_1. Transition t_6 cannot immediately follow t_1 since the input state of t_6 differs from the output state of t_1. However, transition t_2 can process the second test input immediately following t_1[5].

Note that it may be possible to execute a transition such as t_6 after executing some other transition following transition t_1. Since we cannot find this intermediate transition based on the test description, we rule out t_6. If all the transitions in some set in ϕ of a test are ruled out then we can stop since this means that the test description does not contain all the information about the transitions that the test will execute. Such tests are not re-run and may either involve unspecified inputs or may execute transitions not appearing in ϕ.

For the third input of λ_2, we consider each transition in the corresponding set of ϕ with t_2. Transition t_9 cannot immediately follow t_2. Nor can t_4 immediately follow t_2 since t_4's input guard instantiated with the test input: $id == 1 \wedge 75 > 0 \wedge (bal[1]$ - $75) < 0$, requires that the formula: $(bal[1]$ - $75 + 50) < 0$ to be true after executing t_2 which requires: $(bal[1]$ - $75 + 50) < 0 \wedge (bal[1] == 100)$ to be true after executing t_1, which is not possible. Similarly, we can rule out transition t_5. The only remaining transition t_3 can immediately follow t_2 and can process the third input and hence we go to the next input of λ_2.

[3] These tests are called modification-traversing tests [9].

[4] Input conditions match if the messages in the description are instances of those in the transition and the transition guard is satisfied by the test input values. More details are in Section 3.

[5] Interaction among transitions is analyzed using post-images and pre-images computed from the transitions. For more details see Section 4.

Transition t_9 cannot immediately follow t_3. However, t_5 can immediately follow and process the fourth test input. Since the description of λ_2 contains all the information about the transitions executed up to (and including) the processing of the last test input matched by t_9, we can guarantee that t_9 will not be executed by λ_2. Therefore, test λ_2 need not be re-run to test t_9.

Test λ_3 has the same sequence of matching transitions as λ_2, except that the parameters in its third and fourth inputs are different. Analysis of λ_3 as described above, shows that t_1 and t_2 will process the first two test inputs. The third input will be processed by t_5 since its instantiated guard: $id == 1 \wedge 110 > 0 \wedge (0 <= bal[1]$ - $110 < min)$ can be satisfied after executing t_1 and t_2. Similarly, it can be shown that the next test input will be processed by t_9, which guarantees that t_9 will appear in its test run. Hence test λ_3 will be re-run and is a candidate to test t_9. It can be similarly verified that t_9 will be executed by the last two tests λ_4 and λ_5 to process their fifth and sixth inputs respectively making them candidates to test t_9 as well. Changes that delete and replace transitions can also be handled by the proposed approach and are discussed in Section 4.

The reader would have noticed the repetitive analysis of tests λ_3-λ_5 due to the sharing of test inputs in their descriptions. Tests can be grouped together based on the overlapping inputs in their descriptions to reduce selective re-testing costs. Section 5 gives more details.

2 Related Work

There is a lot of earlier work on regression test selection (see [9] for an excellent survey). Regression test selection for EFSMs has been studied earlier [2,5,6] for model-based regression test selection and test prioritization. In [5], Korel et. al, discuss a method for model-based test selection of EFSMs involving addition and deletion of transitions. Using control and data dependencies between the change transition and the rest of the model they identify equivalent tests to reduce the test suite. Since a test may cover multiple changes, tests are included only if their changes are not covered by others. In [2], Chen et. al, extend the work in [5] to consider new type of replacements called change transitions and refine control and data dependencies proposed in [5]. The work in [6] focusses on test prioritization and proposes several heuristics for the same.

The proposed approach extends these earlier works in several ways. First, we support more expressive, executable EFSMs with a rich set of data types including booleans, numbers, arrays, queues, and records. In this sense, this work bridges the gap between the code-based [9] and the model-based [2, 5, 6] approaches. Second, unlike model-based approaches, we do not consider tests to be explicit sequence of transitions. Our tests are a sequence of input assignments to the data variables (much like programs), with expected outputs and a verdict of pass/fail. We use a theorem prover to analyze a test to determine if it executes a given modified transition and select such tests. In this sense our notion of affected tests is similar to that of modification-traversing tests of [9]. Finally, we use a theorem prover to support richer set of data types than that have been traditionally considered by model-based approaches. Most importantly, this allows us to be more precise in comparison to the conservative data flow techniques used by model-based approaches.

3 Preliminaries

Extended Finite State Machines: An extended finite state machine (EFSM) [1,7,10] is a finite state machine extended with variables and communicates with the environment by exchanging parameterized messages using (possibly infinite) FIFO queues. An EFSM $E = (I, O, S, V, T)$, 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. Message m has typed, distinct, parameters $p_1, \cdots,$ p_k, written as $\overrightarrow{p}$; types can be one of – boolean, number, array, queue, and record. The set $V = X \cup \{IQ, OQ\}$, is the union of the global variables X and the queue variables IQ and OQ denoting the input and output queues from (to) the environment respectively. A transition t in T is of the form: $m_k(\overrightarrow{p})$, P_t, $s_t \mapsto q_t$, $m_l(\overrightarrow{e})$, A_t where the predicate P_t, action list A_t, and $\overrightarrow{e} = (e_1, \cdots,$ $e_w)$ respectively are, a conjunction of atomic predicates, an ordered sequence of assignments, and a series of expressions over the data and queue variables and parameters $p_1, \cdots\ p_k$. The input(output) messages are optional in a transition.

Transitions having both input and output messages are **explicit**[6] transitions.

Semantics of EFSM: A **global state**, $g = (\langle s \rangle, \mathit{pred})$ is a pair whose first element $s \in S$; second element *pred* is a conjunction of atomic predicates over V including a (possibly empty) set of equalities representing the actions executed by the transitions to reach the global state g[7]. An atomic predicate is formed by using relational operators (*and*, *or*, *not*, $==$, $\neq$, $<$, $\leq$, $>$, $\geq$) over expressions over the different data types (booleans, numbers, arrays, queues, and records). An **initial global state** of E is a global state g in which state s belongs to the initial state of S, the second element *pred* is the **initial predicate**, a conjunction of atomic predicates over queue variables stating that the output queue has the initial value *initq*, the input queue has environment messages, and the data variables have their initial values. Transition is **enabled** in a global state if its input message matches the head of IQ in the state and its input condition is satisfied at that state. An **execution step**, $g \rightarrow_t g'$, executes the transition t enabled in g resulting in the global state g'. A **run** $r = g_0 t_0 g_1 \cdots\ t_{n-1} g_0$ is a sequence of consecutive execution steps starting and ending in the initial global state.

Simplify Prover. Theorem prover Simplify [3] extended with rewrite rules is used to analyze tests in a push-button manner. Quantified formulas called verification conditions are generated by automatically translating the EFSM predicates and assignments [10] and input to the prover. The prover returns *valid* if input formula F is true under all the assignments to the variables in F

[6] Analogous to explicit transitions in SDL.

[7] Equalities in *pred* are represented as rewrite rules to aid equality simplification. More details are in [10].

and returns *invalid*, otherwise. To check if F is satisfiable, its negation is input to the prover. Simplify contains decision procedures for numbers, booleans, equality, partial-orders, and the theory of maps [3]. The theory of maps is used to reason about data types such as arrays, records including message queues.

A global state whose variables are fully instantiated (with constant values) is a **concrete global state**. Each global state represents a (possibly infinite) set of concrete global states obtained using satisfying assignments to the predicates (including message queues). Tests and test runs deal with executability and hence use concrete global states.

EFSMs are deterministic, i.e., in each concrete global state (that includes message queues) at most one transition is enabled.

3.1 Tests, Changes, and Affected Tests

An EFSM **test** (description) $\lambda = \langle g_0, [i_1/o_1,\cdots,i_n/o_n]\rangle$ is a concrete global state g_0 along with a finite sequence of input/output elements where each element is a sequence of assignments to the data and the message queue variables. Test λ is **applied** by starting the machine in the concrete global state g_0 and repeatedly executing the enabled EFSM transitions and serially processing the test inputs. The **test run** $r_\lambda = g_0 t_0, \cdots t_m g_m \cdots g_0$, is an EFSM run produced by applying λ to the EFSM in state g_0; all the global states in r_λ are concrete global states. Each test λ has a unique run r_λ since the EFSM is deterministic, i.e., at most one transition is enabled in any concrete global state.

In this paper, we focus on tests that are designed to execute specified behaviors. Tests that check the EFSM behavior by providing unspecified or inappropriate inputs are not considered. However, the approach applies equally to these as well. Please see [11] for identifying such failing tests.

Changes to the EFSM are done at the transition level. A change $\delta = \langle sign, t\rangle$, $sign \in \{+,\text{-}\}$ either adds or deletes the transition t respectively. Change can add explicit transitions having the same input and output messages with different input states and/or input conditions. Newly added transitions can also refer to new messages and/or new states. In such cases, we will assume that the test suite from which tests are to be selected already contains tests with these new messages [5]. Replacement of transitions can be modeled using addition and deletion of transitions. Change $\delta = \langle t_d, t_a\rangle$ denotes a replacement deleting t_d and adding t_a. More complex updates are specified by using a set of order-independent [11] transition changes.

Test λ is *affected* by a change δ if the test run executes the transition t in δ. Then, λ is a candidate to be re-run for testing the change δ.

4 Analyzing Interaction among Transitions

Pre-image of t, $Pre(t) = \langle s_t, (nP_t \wedge IQ.\text{hd} == m_j(\overrightarrow{p}) \wedge OQ \neq full^{8})\rangle$, is a symbolic global state that includes every global state g (possibly none) where the

[8] For brevity, we assume output queues to be unbounded, henceforth.

transition is enabled. It is generated using transition t; nP_t is got from predicate P_t of t by renaming variables to refer to their latest instances [10].

Example: Transition t_5 in Figure 2, $Pre(t_5) = \langle s_1, (i == id \wedge v > 0 \wedge 0 <= (bal_0[i] - v) \wedge (bal_0[i] - v) < min_0 \wedge IQ_0.\text{hd} = wdraw(i, v))\rangle$ where bal_0, min_0, and IQ_0 refer to the latest instances of these variables. $\Box$

The **post-image** of t, $Pos(t) = \langle q_t, (nP_t \wedge IQ_1 == deq(IQ_0) \wedge IQ_0.\text{hd} == m_j(\overrightarrow{p}) \wedge OQ_1 == \langle OQ_0, m_l(\overrightarrow{ne})\rangle \wedge nA_t)\rangle$, is a symbolic global state that includes every global state g' (possibly none) that can be obtained after executing t. It is automatically generated using t; $I(O)Q_0$ and $I(O)Q_1$ stand for the queues before and after executing t; $\overrightarrow{ne}$ denotes the parameter expressions in the output message using the latest instances of variables; nA_t is similarly obtained from the action statements A_t after translating them into equalities [10].

Example: $Pos(t_5) = \langle s_2, (i == id \wedge v > 0 \wedge 0 <= (bal_0[i] - v) \wedge (bal_0[i] - v) < min_0 \wedge IQ_1 == deq(IQ_0) \wedge IQ_0.\text{hd} == wdraw(i, v) \wedge OQ_1 == \langle OQ_0, ack(v)\rangle \wedge bal_1[i] == bal_0[i] - v)\rangle$. $\Box$

Below, we view $Pre(t)$ and $Pos(t)$ as formulas with the EFSM state being an equality predicate over a predefined state variable st.

Compatibility of Transitions. Compatibility relation among transitions determines if a transition can immediately follow another in the EFSM runs. Given transitions t_i and t_j with input messages m_i and m_j respectively, let $\psi = (IQ_0 == \langle m_i(\overrightarrow{p_i}), m_j(\overrightarrow{p_j})\rangle$). Formula ψ provides the context with the input queue having the two input messages and the output queue capacity being set so that both transitions can enqueue their outputs.

Transition t_j is **incompatible** with t_i if t_j cannot immediately follow t_i in any EFSM run. Transition t_j is found to be incompatible with t_i by checking that $\psi \wedge Pos(t_i) \wedge Pre(t_j)$ is an unsatisfiable formula.

Example: In Figure 2, transition t_9 is incompatible with transition t_6. Context $\psi = (IQ_0 == \langle\ deposit(i_1, v_1), wdraw(i_2, v_2)\ \rangle)$.
$Pos(t_6) = \langle s_1, (i_1 == id \wedge v_1 > 0 \wedge (bal_0[i_1] + v_1) >= min_0 \wedge IQ_1 == deq(IQ_0) \wedge IQ_0.\text{hd} == deposit(i_1, v_1) \wedge OQ_1 == \langle OQ_0, ack(bal_1[i_1])\rangle \wedge bal_1[i_1] == bal_0[i_1] + v_1)\rangle$.
$Pre(t_9) = \langle s_2, (i_2 == id \wedge v_2 > 0 \wedge (bal_0[i_2] - v_2) >= 0 \wedge IQ_1.\text{hd} = wdraw(i_2, v_2))\rangle$. It is verified using the prover that $\psi \wedge Pos(t_6) \wedge Pre(t_9)$ is an unsatisfiable formula. $\Box$

Note that t_j being incompatible with t_i is independent of the input parameters in t_i and t_j since they are renamed and are disjoint.

Transition t_j is **strongly compatible** with t_i if t_j can immediately follow t_i in all EFSM runs. Transition t_j is found to be strongly compatible with t_i if ($\psi \wedge Pos(t_i)) \implies PreElim(t_j)$ is a valid formula. $PreElim(t_j)$ is obtained from $Pre(t_j)$ by eliminating conjuncts that involve only the input parameters of t_j. This ensures that t_j immediately follows t_j regardless of the values of the input parameters.

Example: Transition t_8 is strongly compatible with transition t_6 in Figure 2. Context $\psi = (IQ_0 == \langle\ wdraw(i_1, v_1), close(i_2)\ \rangle)$; $Pre(t_8) = PreElim(t_8) = \langle s_1, IQ_1.\text{hd} = close(i_2)\rangle$; $Pos(t_6)$ is given above. It is verified using the prover that $(\psi \wedge Pos(t_6)) \implies PreElim(t_8)$ is a valid formula.

Note also that t_6 is also strongly compatible with the t_2 in Figure 2 since $(\psi \wedge Pos(t_6)) \implies PreElim(t_2)$ is a valid formula. However, it is evident $(\psi \wedge Pos(t_6) \implies Pre(t_2)$ is not a valid formula since the conjunct in $Pre(t_2)$ derived from the condition $v > 0$ of t_2 is not valid. □

More than one transition can be strongly compatible with a transition provided these transitions have mutually exclusive predicates over the input parameters. Consider a deterministic EFSM[9] with transitions, t_1: m_1, $true$, $s_0 \mapsto s_1$, ack; t_2 $m_2(v)$, $v > 0$, $s_1 \mapsto s_1$, ack; t_3: $m_3(v)$, $v < 0$, $s_1 \mapsto s_1$, ack. It is easily verified that both t_2 and t_3 are strongly compatible with t_1. Note however, that only one of t_2 or t_3 can immediately follow t_1 in any concrete global state.

Transition t_j is **compatible** with t_i if t_j can immediately follow t_i in some EFSM runs but not in others. Transition t_j is found to be compatible with t_i if t_j is neither incompatible nor strongly compatible with t_i.

Example: Transition t_3 is compatible with t_1 in Figure 2, since $Pos(t_1) \wedge Pre(t_3)$ is not unsatisfiable and $Pos(t_1) \implies PreElim(t_3)$ is not a valid formula. □

Many transitions can be compatible with a given transition. It is also possible to have transitions t_j and t_k where t_j is strongly compatible and t_k is compatible with a given transition t_i. Consider a deterministic EFSM whose transitions include, t_1: $m_1()$, $s_0 \mapsto s_1$, ack, $\{x = 1;\}$; t_2: $m_2()$, $x > 0$, $s_1 \mapsto s_1$, ack, $\{\}$; t_3: $m_3()$, $y > 0$, $s_1 \mapsto s_1$, ack, $\{\}$. Transition t_2 is strongly compatible with t_1 since it can always immediately follow t_1. However, since the value of y is unknown after executing t_1, t_3 can immediately follow t_1 in some runs but not in others. In this case, for deterministic behavior, the EFSM must ensure that t_1 is only executed in a concrete state in which the value of y is not positive.

The pre-image and post-image of EFSM transitions and the compatibility of transitions can be pre-computed using a theorem prover before test selection and used to identify fully-observable tests as described next.

5 Selecting Fully-Observable Tests

Test λ is **fully-observable** if every transition in the test run r_λ is an explicit transition. Recall from Section 2, that transitions having both input and output messages are explicit transitions. We can identify these tests by using their descriptions as described below.

Consider test (description) $\lambda = \langle g_0, [i_1/o_1, \cdots, i_n/o_n]\rangle$. Transition t_k **matches** i_k/o_k in λ if input i_k (output o_k) is an instance of the input (output) message of t_k and the input condition of t_k instantiated with parameter values from i_k is a satisfiable formula. As an example, the transition t_5 in Figure 2, matches

[9] Please see Section 3.

the input $wdraw(1,\ 110)/ack(110)$ of λ_3 since the input and output messages $wdraw(i,\ v)$ and $ack(v)$ are instances of the test input and test output respectively and the condition of t_5 instantiated with the parameters: $(1 == id) \wedge (110 > 0) \wedge (bal[1]$ - $110) >= 0 \wedge (bal[1]$ - $110) < min$ is a satisfiable formula.

Let $T(i_k/o_k)$ denote the set of transitions matching i_k/o_k and ϕ=[$T(i_1/o_1),\cdots, T(i_n/o_n)$] denote the sequence of sets of transitions that point-wise match the sequence $[i_1/o_1,\ \cdots,\ i_n/o_n]$. As an example, $\phi = [\{t_1\},\ \{t_2,t_6\},\ \{t_3,t_4,t_5,t_9\},\ \{t_3,t_4,t_5,t_9\},\ \{t_7,t_8\}]$ for the test λ_2 in Figure 2.

Note that if transitions t_k and t'_k both belong to some set $T(i_k/o_k)$ then they cannot both be strongly compatible with a transition t_i. If this were to happen then both of them can immediately follow t_i in every EFSM run. Since this is not possible in a deterministic EFSM they must have mutually exclusive predicates involving their input parameters in which case they both cannot both match the input i_k and hence cannot both belong to $T(i_k/o_k)$. However, if transition t_k is strongly compatible with t_i whereas t'_k is only compatible with t_i then we can conclude that t_k must immediately follow t_i.

Transition Compatibility Graph. Compatibility information among the transitions in the sequence ϕ for a test λ is represented by a directed acyclic graph TCG. In addition to *start* node, TCG has one node for each occurrence of each transition in the sequence ϕ. Transitions in the set $T(i_k/o_k)$ in ϕ appear at level k; *start* is at level 0. There are labeled directed edges between nodes in consecutive levels. Edge with label s from t_k to t_{k+1} is present if t_{k+1} is strongly compatible with t_k; edge with label c from t_k to t_{k+1} is present if t_{k+1} is compatible with t_k. Edge from *start* to a node t_1 with label s is added if $pred \implies Pre(t_1)$ is a valid formula where *pred* is the predicate in the concrete initial global state g_0 of the test λ.

Note that there can be at most one outgoing edge with label s from any node since a node can be strongly compatible with at most one element from a set of transitions matching a test input as discussed above. Further, since all the variables are fully instantiated in g_0 there is exactly one outgoing edge from *start* to a node in level 1 and this has the label s.

As an example, the TCG for the test λ_3 is given in Figure 3.

5.1 Identifying Fully-Observable Tests

Test λ is **initial** if every transition in $T(i_1/o_1)$ is incompatible with every non-explicit transition i.e., no transition can precede any transition from $T(i_1/o_1)$. Similarly, test λ is **final** if every non-explicit transition is incompatible with every transition in $T(i_n/o_n)$ i.e., no non-explicit transition can follow any transition in $T(i_n/o_n)$.

A test must be both initial and final for the test to be fully-observable. Then, the TCG is used to check if the test is fully-observable as described below.

1. Terminate with success if there is path from *start* to some node in the last level labeled by s.

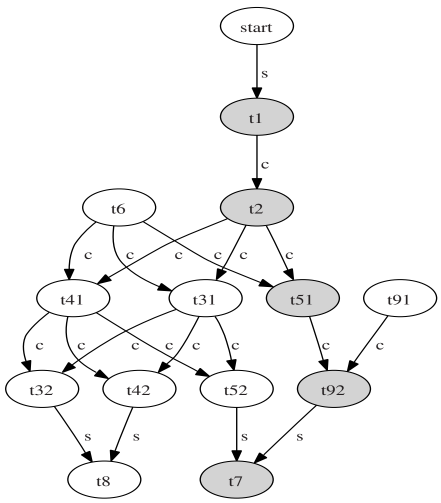

Fig. 3. TCG for Test λ_3

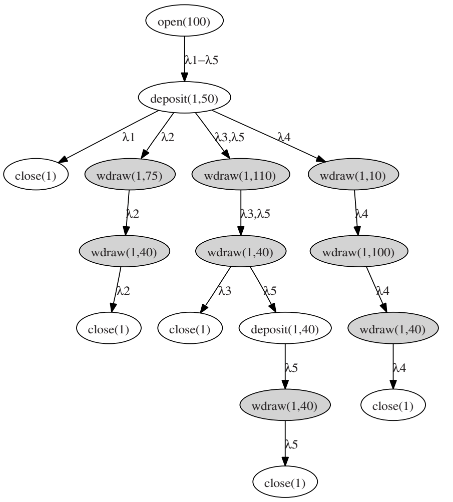

Fig. 4. Test Suite Tree for Bank Example

2. Remove nodes (and corresponding edges) that do not either have any predecessors (except *start*) or do not have any successors (except those at the last level) until no more such nodes exist.
3. Remove all outgoing edges from node t_k with a label c if there is an outgoing edge from t_k with label s. Repeat previous step to remove the resulting unconnected nodes.
4. The resulting graph is then processed iteratively level by level in an attempt to construct an executable path from *start* to some node in the last level. At each step, a formula labeling a node in the previous level is propagated to all the successors. A new formula is generated at each successor. If it is a valid formula then it is assigned to be the label of that node and the process moves to the next level until reaching the last level, in which case, an executable path from *start* to a node in the last level has been found and the test is declared fully-observable; the path comprises of nodes that have been assigned a label. If none of the formulas generated at the successors at a level are valid formulas then the process terminates declaring test to be not fully-observable. At level 1, the label assigned to the node t_1, $L(t_1) = pred \implies Pre(\sigma(t_1))$[10] where *pred* is the predicate of state g_0 and $\sigma(t_1)$ is an instance of t_1 obtained by uniformly replacing all occurrences of the input parameters by the corresponding values from the test input i_1. At the k^{th} step, given $L(t_k)$, the formula generated at successor t_{k+1} is: $L(t_k) \wedge Pos(\sigma_k(t_k)) \implies Pre(\sigma_{k+1}(t_{k+1}))$, which if valid is assigned as $L(t_{k+1})$. Note that the process can be terminated with success at a level k if all the subsequent levels are connected by s label edges.

[10] σ is a substitution from input parameters of t_1 to the input values in the corresponding test input generated by matching.

Example: Test λ_3 is both initial and final. The TCG in Figure 3 is constructed and analyzed level by level since there is no path from *start* to either t_7 or t_8 labeled s. Unconnected nodes t_6 and t_{91} and the associated edges are removed. The formula generated at node t_1 is:$\langle s_0$, $(min == 50) \wedge IQ_0 == \langle open(100)$, $deposit(1, 50)$, $wdraw(1, 110)$, $wdraw(1,40)$, $close(1)\rangle\rangle \implies \langle s_0$, $IQ_0.\text{hd} == open(100) \wedge 100 > 0\rangle$ is a valid formula and hence is assigned to be $L(t_1)$. The formula generated at node t_2 using $L(t_1)$ is again found to be valid and assigned to be $L(t_2)$.

At the third level, t_2 has three successors t_{31}, t_{41} and t_{51}. The formula generated at level t_{31} is of the form $(L(t_2) \wedge Pos(\sigma_2(t_2))) \implies Pre(\sigma_3(t_{31}))$ where σ_2 uniformly substitutes i by 1 and v by 50 in t_2 and σ_3 uniformly substitutes i by 1 and v by 110 in t_3. Since a conjunct in the consequent: $((bal_1[1]$ - $110) >= 50)$ cannot be established from the conjuncts: $(bal_1[1] == bal_0[1] + 50)$ and $bal_0[1] == 100$ in the antecedent, the formula is not valid and the node t_{31} is skipped. Similarly, node t_{41} is also skipped. The relevant conjunct in the formula generated for t_{51} : $(bal_1[1]$ - $110 < 50)$, follows from those in the antecedent, resulting in a valid formula. Hence t_{51} is labeled with this formula as its label $L(t_{51})$. The next two levels have a single successor, nodes t_{92} and t_7 respectively. The formula generated at t_{92} using $L(t_{51})$ is a valid formula and is assigned to $L(t_{92})$. The formula generated at t_7 is also valid and is assigned to $L(t_7)$, resulting in the executable path that is highlighted in Figure 3. Therefore, the test λ_3 is declared to be fully-observable.

5.2 Selecting Tests

Given a fully-observable test λ and a change $\delta = (sign, t)$, the above procedure can be used in a straightforward way to accurately determine whether λ is affected.

Addition Change. Consider a change $\delta = (+, t_a)$ that adds a single transition t_a to the EFSM.

To determine if δ affects a test λ, the sequence ϕ of matching transitions is extracted from the description of λ. If t_a is an explicit transition and does not appear in ϕ then the test λ is unaffected. Suppose that the new transition t_a appears exactly once at the k^{th} set in ϕ. Using the compatibility information of the original transitions, we construct TCG and determine whether λ is fully-observable up to (and including) level k of the graph. Then, λ can be declared unaffected without even analyzing the compatibility of transition t_a with other transitions since λ will execute a transition other than t_a at the k^{th} level, the only level where t_a can appear.

Suppose that λ is fully-observable only up to level k - 1 with node t_{k-1} being labeled with formula $L(t_{k-1})$. Now, compatibility of t_a with t_{k-1} is determined. If t_a is strongly compatible then the test is declared affected. If t_a is compatible then we generate: $(L(t_{k-1}) \wedge Pos(\sigma_{k-1}(t_{k-1}))) \implies Pre(\sigma_k(t_a))$, and check if it is a valid formula. If so, λ is declared affected. If t_a is incompatible with

t_{k-1} or if λ is not fully-observable up to level k - 1 then the approach fails and cannot accurately determine whether the test is affected. In such cases, λ may be conservatively selected as being affected since t_a matches some element in λ.

Suppose that transition t_a matches λ at multiple positions covered by an interval $[i, k]$ then if λ is fully-observable up to a level m that is greater than or equal to k then λ can be unaffected regardless of compatibility t_a. If m equals a level that immediately precedes a matching level, say, level k - 1, which contains node t_{k-1} such that $L(t_{k-1})$ is a valid formula then we compute compatibility of t_a with t_{k-1} and determine whether λ is affected as described above. In all other cases, λ may be conservatively selected to test t_a.

Deletion and Replacement Changes. To determine if a change $\delta = (-, t_d)$ affects a test λ the above procedure is slightly modified to handle unspecified behaviors that may arise due to missing transitions. To do this, we allow test suite to have tests with unexpected inputs to indicate adverse impact of deletion.

Suppose that transition t_d matches the description of test λ at a level k and the node t_d is assigned a label as described above. If test λ ignores its k^{th} input due to absence of t_d (and the following inputs) and belongs to the test suite of the original EFSM then λ is not selected since it results in an expected unspecified behavior caused by deletion of t_d. However, if λ is newly generated then it is selected to indicate the adverse impact of deleting t_d.

We also select tests which indicate that deleting t_d does not cause any adverse effects. Effects of t_d are first nullified by removing its output actions and making output state identical to its input state. A test λ is selected to indicate that no adverse effect happens due to deleting transition t_d that matches λ provided t_d is assigned label and the test is found to be fully-observable. Even if t_d does not match a test λ, we can check if the conditions enabling t_d can occur while executing λ by adding the modified t_d to each k^{th} set of transitions of ϕ whenever t_d is compatible or strongly compatible with at least one of the transitions in previous set. Then the analysis is repeated and test λ is selected as done for addition with multiple matches as described above.

In situations where the updates contain many changes the above approach for deletion may miss certain tests because their runs cannot reach a concrete global state where the deleted transition t_d is enabled without the new transitions. In such cases, an interim EFSM [11] can be created by applying all the changes in the update except for δ and the analysis performed using the transitions available in this EFSM.

To determine if a replacement change $\delta = (t_d, t_a)$ affects a test λ, we view it as a deletion change involving t_d followed by addition of t_a. A test λ is affected by δ if it is either affected by the deletion of t_d or the addition of t_a. The effect of deleting t_d is computed using both the original system and the intermediate system obtained by adding t_a. The effect of addition of t_a is similarly computed using both the original system as well as the intermediate system obtained after deleting t_d.

5.3 Handling Multiple Tests

Consider a test suite $TS = \{\lambda_1, \cdots, \lambda_n\}$ comprised of fully-observable tests. To determine the tests in TS that are affected by an addition or deletion change $\delta = (sign, t)$, TS is modeled as a forest with each tree (TST) comprised of all tests starting with the same state g_0 and having same starting test input(outputs are not used to determine affected tests and are ignored). Nodes in each TST are test inputs. Edges are labeled by sets of tests. Node u is a parent of node v if the input of u is applied before that of v at some test λ_i of TS. Then, the test λ_i is added to the set of tests labeling the edge between nodes u and v.If inputs i_1 and i_2 can appear in any order then we break the cycle by creating separate trees. In each TST, the set of tests labeling the edge to each parent is a union of the sets of tests labeling the edges to its children. So, the set of tests in TST are refined as we go down the tree. For example, the bank example from Section 1, has a single test suite tree depicted in Figure 4.

Given a change transition t, and a test suite tree TST, we first find the nodes u matching t_d and annotate the paths to each u with matching sets of transitions. Each node has a single matching set of transitions. Let $T(u)$ stand for the matching set corresponding to a node u. A left-right traversal of the tree picks the first matching node u in each TST path $p = v_1 v_2 \cdots v_n u$ and checks if the sequence $\phi = [T(v_1), \cdots, T(v_n), T(u)]$ is fully-observable and t_d is the labeled transition in the set $T(u)$. If so, all the tests on the edge incident on node u are marked affected, the node and its descendants are removed and we analyze the remaining TST paths. Otherwise, we update transition sets $T(v_1) = \{t_1\}, \cdots, T(v_n) = \{t_n\}$, and $T(u) = \{t_u\}$ where t_i are the labeled transitions found while checking full-observability of p and continue analyzing the extensions of the path p reaching another matching node u. The process terminates once every path reaching a matching node is analyzed and all the unmarked tests at the end of the process are declared unaffected. Note that if node u appears at level k of p which is fully-observable only up to a level $m < k$ then we can ignore all siblings of u whose common ancestor is at level l, $m < l < k$.

The matching nodes u for the bank example are highlighted in Figure 4. A left-to-right traversal of this tree detects affected tests λ_3-λ_5 at level 4 after which the matching node at level 6 can be removed.

6 Preliminary Experiments

We have implemented the proposed approach in Perl using a package supporting graph subroutines. Given an EFSM, modified EFSMs are generated by instrumenting the transitions as added(`+`), deleted(`-`) and replaced(`r`)[11]. Addition and deletion of every transition in the original EFSM is covered. Then, by using the instrumentations in the modified EFSM files, changes are extracted, test description templates with uninstantiated parameter values are generated for each

[11] Conference protocol also includes more specific changes as discussed below.

Example	Ntsc	Stsc	C_1(secs)	C_2(secs)	C_3(secs)
Atm (9)	9	3	35	8	17
Bnk (9)	46	26	1108	285	621
Cmp (7)	9	1	15	2	4
Cnf (19)	40	11	390	63	134
Tcp (14)	9	3	41	7	8
Thp (15)	16	8	134	28	76
Ven (8)	10	4	41	11	27

Fig. 5. Regression Test Selection Costs

change by using transition coverage of the modified (as well as the original EFSMs). Test descriptions are produced from these templates by randomly assigning constant values to the parameters and included in the test suite. Non-executable test descriptions are removed by performing symbolic execution. We also added hand-crafted tests. We then analyzed each test in the test suite for each change using the proposed approach. During analysis, EFSM expressions are automatically translated into the language of the prover, and the prover is invoked in a push-button manner to check satisfiability of the generated formulas.

We have studied seven simple examples from the literature: completion, two-phase commit, and conference protocols (*Cmp*, *Tcp*, *Cnf*)[12], third-party call (*Thp*), automatic teller machine (*Atm*) [2,5], bank example (*Bnk*) web services, and a vending machine (*Ven*). The main goal was to compare the effectiveness of the proposed approach with the retest-all approach based on the cost model of [8]. According to this model test selection is more economical than using the entire suite if the cost of selection(C_3) is less than the cost of running tests that the selection lets us omit (C_1 - C_2); C_1 is the cost of running the full test suite; C_2 is the cost of running the selected tests.

Our results are summarized in the table in Figure 5. For each example, first column gives the total number of transitions without instrumentation; second column `Ntsc` is the number of test cases in the test suite; third column `Stsc` is the average number (rounded) selected tests *per change*; and the next three columns give the costs C_1, C_2 and C_3 in terms of the running times. These times were averaged (rounded) by considering a set of changes that cover every transition.

Conference protocol *Cnf* above, is a chatbox-like protocol and has been used earlier by formal testing approaches. We used the EFSM description (c) available from the web site and changed to the description (d) in the same web site. The four changes specified there to do this are additions that allow members to send data before joining the conference. Our use of the more expressive parameterized EFSMs allowed us to specify conference id as a parameter because of which two additions suffice to evolve EFSM (c) to EFSM (d).

[12] http://schemas.xmlsoap.org/ws/2004/10/wsat/Completion;
http://schemas.xmlsoap.org/ws/2004/10/wsat/Volatile2PC(Durable2PC);
http://fmt.cs.utwente.nl/ConfCase/

As evident from the table, in each case, we were able to select a non-empty set of tests for the chosen change. The average number of tests selected per change results in a smaller test suite in all cases. Further, our results show that the cost of regression selection is economical $C_3 < (C_1 - C_2)$ in all of our examples. For the examples, Atm and Ven, the reduction in test suite size does not reflect on the economy of regression test selection. Even though a lot of tests are eliminated, the cost of regression test selection is high in these examples since the tests removed do not take much time to execute. However, the proposed approach has the potential to perform highly economical regression test selection since it can eliminate long running tests by only partially analyzing them. But our initial experiments did not reflect that since our test descriptions were restricted to be sequences of no more than size 20 and we averaged over all changes.

7 Conclusion and Future Work

A novel approach for regression test selection for EFSMs supporting a rich set of commonly used data types including booleans, numbers, arrays, queues, and records is proposed. Changes to the EFSM are performed at the transition level by adding/deleting/replacing one or more transitions. Test descriptions supporting input and output values from these data types are automatically analyzed using a theorem prover to identify tests that exercise a given change. A class of fully-observable tests is identified. It is shown that the description of a fully-observable test contains adequate information to accurately predict whether the test will exercise a given change. Procedures to identify fully-observable tests and select tests to exercise changes that add/delete/replace transitions are described. We also extend the proposed approach to simultaneously analyze several tests in a test suite to reduce analysis costs. Initial results of our experiments on 3 web services and 4 protocols based on a well-known cost model for regression testing [8] are promising. They show that the cost to select tests is smaller than the difference in execution times between running all vs. running only the selected tests.

Acknowledgements. The first author would like to thank Gregg Rothermal, for fruitful discussions and help on earlier works on regression test selection.

References

1. Brand, D., Zafiropulo, P.: On Communicating Finite State Machines. JACM 30(2) (1983)
2. Chen, Y., Probert, R., Ural, H.: Regression test suite reduction using extended dependence analysis. In: SOQUA 2007, September 3-4 (2007)
3. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A Theorem Prover for Program Checking. Journal of the ACM 52(3) (2005)
4. Kapur, D., Zhang, H.: An Overview of Rewrite Rule Laboratory (RRL). In: Dershowitz, N. (ed.) RTA 1989. LNCS, vol. 355, pp. 559–563. Springer, Heidelberg (1989)

5. Korel, B., Tahat, L., Vaysburg, B.: Model Based Regression Test Reduction Using Dependency Analysis. In: Proceedings of the International Conference on Software Maintenance (ICSM 2002), October 03-06, p. 214 (2002)
6. Korel, B., Koutsogiannakis, G., Tahat, L.H.: Application of system models in regression test suite prioritization. In: IEEE International Conference Software Maintenance, ICSM 2008, September 28-October 4, vol. (2008)
7. Lee, D., Yiannakakis, M.: Principles and Methods of Testing Finite State Machines – A Survey. Proceedings of the IEEE 84(8) (1996)
8. Leung, H.K.N., White, L.: A Cost Model to Compare Regression Test Strategies. IEEE Conf. on Software Maintenance, ICSM (1991)
9. Rothermel, G., Harrold, M.J.: Analyzing Regression Test Selection Techniques. IEEE Transactions on Software Engineering (1996)
10. Subramaniam, M., Guo, B.: A Rewrite-based Approach for Change Impact Analysis of Communicating Systems Using a Theorem Prover, CS Dept. University of Nebraska-Omaha (cst-2008-3) Technical Report (Work in progress paper in Testcom 2008) (2008)
11. Subramaniam, M., Pap, Z.: Updating Tests Across Protocol Changes. In: Proc. of IFIP Conference on Testing of Communicating Systems (2006)