Regression Test Suite Prioritization Using System Models

Luay H. Tahat
Computer Science Department
Gulf University for Science & Technology
Hawally 32093, Kuwait
tahaway@iit.edu

Bogdan Korel
Computer Science Department
Illinois Institute of Technology
Chicago, IL 60616, USA
korel@iit.edu

Mark Harman
University College London
Computer Science Department
Malet Place, London
WC1E 6BT, UK
m.harman@cs.ucl.ac.uk

Hasan Ural
School of Info Tech & Eng
Faculty of Engineering
University of Ottawa
Ottawa, Ontario, K1N 6N5, Canada
ural@site.uottawa.ca

Abstract

During regression testing, a modified system is often retested using an existing test suite. Since the size of the test suite may be very large, testers are interested in detecting faults in the modified system as early as possible during this retesting process. Test prioritization attempts to order tests for execution so that the chances of early detection of faults during retesting are increased. The existing prioritization methods are based on the source code of the system under test. In this paper, we present and evaluate two model based selective methods and a dependence based method of test prioritization utilizing the state-based model of the system under test. These methods assume that the modifications are made both on the system under test and its model. The existing test suite is executed on the system model and information about this execution is used to prioritize tests. Execution of the model is inexpensive as compared to execution of the system under test; therefore the overhead associated with test prioritization is relatively small. In addition, we present an analytical framework for evaluation of test prioritization methods. This framework may reduce the cost of evaluation as compared to the framework that is based on observation. We have performed an empirical study in which we compared different test prioritization methods. The results of the empirical study suggest that system models may improve the effectiveness of test prioritization with respect to early fault detection.

1. Introduction

During maintenance of evolving software systems, their specification and implementation are changed to fix faults, to add new functionality, and to change the existing functionality. Regression testing is the process of validating that the changes introduced in a system do not adversely affect the unchanged parts of the modified system. There has been a significant amount of research on regression testing that resulted in a variety of regression testing methods. These methods are generally classified as code-based [e.g., 1-7] or specification-based regression testing methods [e.g., 8-13].

Although new tests are generated and used to test the changed parts of a modified system, previously developed tests in an existing test suite are often employed to retest the modified system to ensure that unchanged parts of the system are not adversely affected by the changes made to the system. Depending on the size of the existing test suite, this system retesting may be very expensive. Test prioritization orders tests from the existing test suite, for "execution" in such a way that faults in the modified system are uncovered early during the retesting process. Test prioritization methods [14-18, 53-55] order tests according to some criterion, e.g., a code coverage is achieved at the fastest rate. Tests are then executed in this prioritized order: tests with higher priority, based on the prioritization criterion, are executed first, whereas tests with lower priority are executed later. The existing test prioritization techniques use the source code and information gathered during previous executions of the system to prioritize the test suite for system retesting. Notice that test prioritization is appropriate for software systems for which execution of a test suite is expensive in terms of time and resources. For systems for which execution of a test suite is very fast, test prioritization may have limited benefits.

System models are often used during the development of a software system, e.g., in partial code generation and in test generation for model-based testing. Several modeling languages have been developed to model state-based software systems, e.g., State Charts [19], Extended Finite State Machines (EFSM) [20], and Specification Description Language (SDL) [21]. In recent years, several model-based test generation [20, 22, 23, 24-26] and test suite reduction [9, 13] techniques have been developed based on these modeling languages.

In this paper, we present a model-based test prioritization approach. Our approach prioritizes tests using information from the original and modified system models together with information collected during the execution of the modified model on an existing test suite. Based on this approach, we present two model-based test prioritization methods: *model-based selective test prioritization* and *model dependence-based test prioritization*. We have performed an empirical study in which we compared effectiveness of the presented test prioritization methods. In addition, we present an analytical framework for evaluation of some test prioritization methods with respect to the effectiveness of early fault detection.

The rest of the paper is organized as follows: Section 2 provides an overview of the EFSM-based system modeling. Section 3 gives the preliminaries on the problem of test prioritization and presents the model based test prioritization. Section 4 presents the proposed model-based selective test prioritization method. Section 5 presents the proposed model dependence-based test prioritization method. In Section 6, a framework for comparison of test prioritization methods is discussed. In Section 7, the results of an empirical study are presented. Section 8 outlines the related work on test prioritization. In Section 9, conclusions and future research directions are discussed.

2. EFSM-based System Modeling

System models for state-based systems describe the system behavior by a set of states and transitions between these states. The most popular formal description techniques (languages) used for modeling of state-based systems are: Extended Finite State Machines (EFSMs), Specification Description Language (SDL), and State Charts. These languages are often graphical, which makes them easy to comprehend and utilize. They have received wide industry acceptance, especially in the fields of telecommunications, embedded systems, and computer networking, where state-based systems are prevalent.

In this paper, we concentrate on system models given as EFSMs; the underlying model for other modeling languages for state based systems such as SDL. An EFSM consists of a set of states (including a start state and an exit state) and transitions between states. A transition is triggered at its originating state when an event occurs (i.e., an input is received) and an enabling condition (i.e., a Boolean expression) associated with the transition is satisfied. When the transition is triggered, a sequence of actions may be performed (which may manipulate variables and produce an output) and the system is transferred to the terminating state of the transition.

An EFSM M is expressed formally as a 7 tuple: $M = (\Sigma, Q, Start, Exit, V, O, R)$ where:

- Σ is the set of events,
- Q is the set of states,
- $Start \in Q$ is the start state,
- $Exit \in Q$ is the exit state,
- V is a finite set of variables,
- O is the set of actions,
- R is the set of transitions, where each transition T is represented by the tuple: $T = (E, C, A, S_b, S_e)$ where:
  - $E \in \Sigma$ is an event,
  - C is an enabling condition defined over V,
  - A is a sequence of actions, $A = \langle a_1, a_2, \ldots, a_j \rangle$, where $a_i \in O$,
  - $S_b \in Q$ is the transition's originating state,
  - $S_e \in Q$ is the transition's terminating state.

In addition, the following notation related to a transition T is introduced:

- $S_b(T)$ is the originating state of transition T,
- $S_e(T)$ is the terminating state of transition T,
- $C(T)$ is the enabling condition (a Boolean expression) associated with transition T,
- $E(T)$ is the event associated with transition T,
- $A(T)$ is a sequence of actions associated with transition T.

In M, Σ is a set of events, each of which is an external stimulus (input) that may be associated with a list of arguments; i.e., an event $E \in \Sigma$ is represented by $E(\arg_1, \arg_2, \ldots, \arg_k)$. States in Q are passive elements in the EFSM model. States are just snapshots of the system and they are not involved in any kind of decision making or computation. The states *Start* and *Exit* are where the system starts and terminates, respectively. The variables in V provide storage for values that is accessible by enabling conditions and actions in transitions. An action $a_i \in O$ is one of the following types: assignment action, *output* action, or function call. An *assignment* action assigns a value to a variable. An *output* action displays a variable or a constant to the external environment. A *function call* to some function $f(v_1, v_2, \ldots, v_k)$ returns the evaluated value.

A transition T in R is triggered when the system is in the originating state $S_b(T)$, the event $E(T)$ occurs, and the enabling condition $C(T)$ is evaluated to TRUE. When transition T is triggered, the $A(T)$ sequence of actions is performed and the system is transferred to the terminating state $S_e(T)$. If a transition T is specified at a state with no enabling condition, no other transition from that state can be associated with $E(T)$.

EFSM models may be depicted as graphs where states are represented by nodes and transitions by directed edges between states. A simplified EFSM model of an Automated Teller Machine (ATM) system is shown in Figure 1 [9, 17, 27]. This

ATM system supports three types of transactions: balance inquiry, withdrawal and deposit represented by transitions. Before transactions can be performed, an ATM user must enter a valid PIN that is matched against the PIN stored in the ATM card. A user is allowed a maximum of four attempts to enter the valid PIN. For example, the transition labeled T_2 is triggered when the system is in state $S1$, event PIN(p) is received, the value of parameter p does not equal to variable *pin,* and the value of variable *attempts* is less than 3. When the transition is triggered, an error message is displayed, the value of variable *attempts* is incremented, and the user is prompted to enter another pin. Notice that in this example, for transition T_2, $S_b(T_2) = S1$, $S_e(T_2) = S1$, $C(T_2)$ = (p != pin) and (attempts < 3), $E(T_2)$ = PIN(p).

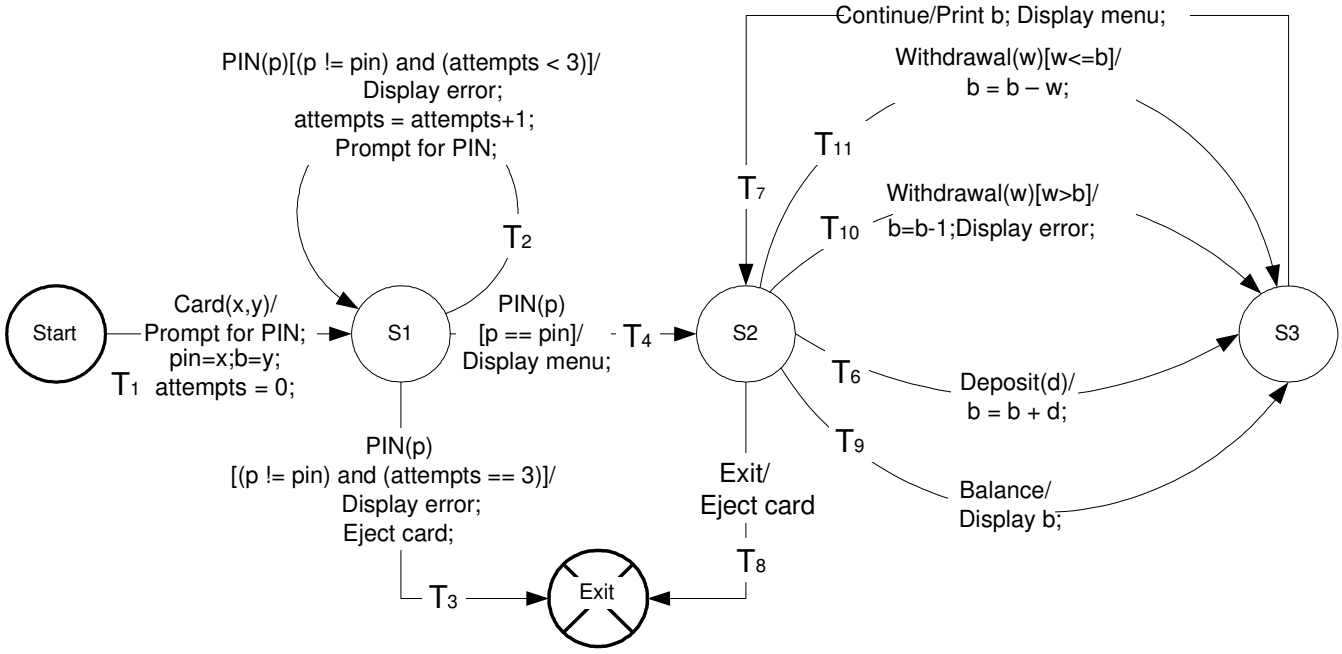


Figure 1. EFSM model of ATM system

In this paper, we assume that the EFSM model is executable, i.e., enough detail is provided in the model so that the model executor can execute the model based on the model specification (or an executable program corresponding to the model can be generated from the model specification). In order to support model execution, some actions may not be implemented (they are represented by "empty" actions). However, all actions are implemented during the development of the system.

A *test* is a sequence of events with values for arguments associated with the events. For example, consider the following sequence of events t with input values:

t: Card *(1234,100)*, PIN (1234), Continue, Withdrawal (20), Continue, Exit.

When the model of Figure 1 is executed on the sequence of events t *above*, the following sequence of transitions is executed: $\boldsymbol{\tau}(t) = \langle T_1, T_4, T_{11}, T_7, T_8 \rangle$. Notice that for event *Continue* in state $S2$, no transition is executed, thus, the number of transitions executed in $\boldsymbol{\tau}(t)$ is 5, whereas the number of events in t is 6. Let $\boldsymbol{\tau}(t) = \langle T_{i_1}, T_{i_2}, \ldots, T_{i_k}, T_{i_{k+1}}, \ldots, T_{i_m} \rangle$, be a sequence of transitions traversed (executed) during execution of the model M on t. Let $\boldsymbol{\tau}(t)[k]$ be a transition in sequence $\boldsymbol{\tau}(t)$ at position k, *i.e.,* $\boldsymbol{\tau}(t)[k] = T_{i_k}$. For example, for $\boldsymbol{\tau}(t) = \langle T_1, T_4, T_{11}, T_7, T_8 \rangle$, $\boldsymbol{\tau}(t)[3] = T_{11}$, i.e., transition T_{11} is at position 3 in $\boldsymbol{\tau}(t)$.

In this paper, we assume that the EFSM model is deterministic, i.e., for every event $E_i(x_i)$ where $x_i = \arg_1, \arg_2, \ldots, \arg_k$, in t there is one and only one possible execution of model M (at most one transition is executed for a given event $E_i(x_i)$). When model M is executed for a given sequence of events $t = \langle E_1(x_1), E_2(x_2), \ldots, E_n(x_n) \rangle$, a sequence of transitions $\boldsymbol{\tau}(t) = \langle T_{i_1}, T_{i_2}, \ldots, T_{i_m} \rangle$ is executed. Notice that the number of events n in t is not necessarily equal to the number of executed transitions m in $\boldsymbol{\tau}(t)$.

3. Test Prioritization

Test prioritization tries to order tests for execution, so the chances of early detection of faults during retesting of the modified system are increased. In this paper, we define the test prioritization problem with respect to early fault detection [14, 17]. The goal is to increase the likelihood of revealing faults earlier during execution of the prioritized test suite. Let $TS = \{t_1, \ldots, t_N\}$ be a test suite of size N, where t_i is a test. Let $D(TS) = \{d_1, \ldots, d_L\}$ be a set of L faults in the system that are detected

by test suite *TS*. Let $TS(d) \subseteq TS$ be a set of tests that fail because of fault *d*. Let $\theta = \langle t_{i_1}, t_{i_2}, \ldots, t_{i_N} \rangle$ be a prioritized sequence of tests of test suite *TS*, where the subscript indicates the position of a test in the sequence, e.g., test t_{i_1} is in position 1, test t_{i_2} is in position 2. Let $t_{i_k} \in TS(d)$ be the first failed test in sequence θ caused by fault *d*, i.e., all tests $t_{i_1}, t_{i_2}, \ldots, t_{i_{k-1}}$ in θ between position 1 and *k*-1 do not fail because of *d*. Let $p_\theta(d) = k$ be the position of t_{i_k}, i.e., the first position of the failed test in θ caused by fault *d*. Let $rp_\theta(d)$ be the first relative position of the failed test in θ caused by fault *d*, where $rp_\theta(d)$ is computed as follows:

$$rp_\theta(d) = \frac{p_\theta(d)}{N} \tag{3.1}$$

Notice $rp_\theta(d)$ represents the test suite fraction at which *d* is detected and its values range between $0 < rp_\theta(d) \leq 1$.

The rate of fault detection [15, 17] is a measure of how rapidly a prioritized test sequence detects faults. This measure is a function of the percentage of faults detected in terms of the test suite fraction, i.e., the relative position in the test suite. More formally, let $P(\theta) = \langle rp_\theta(d_1), \ldots, rp_\theta(d_L) \rangle$ be a list of relative positions of first failed tests for all faults in $D(TS)$[1]. Let $F(\theta) = \langle rp_1, \ldots, rp_q \rangle$, $q \leq L$, be an ordered (in ascending order) sequence of all unique first relative positions from $P(\theta)$, where rp_i represents the test suite fraction at which at least one fault is detected in θ. $F(\theta)$ represents an order in which faults are uncovered by test sequence θ. The *rate of fault detection* $RFD(\theta)$ for sequence θ can be defined as a sequence of pairs (rp_i, fd_i), $RFD(\theta) = \langle (rp_1, fd_1), \ldots, (rp_q, fd_q) \rangle$, where rp_i is an element of $F(\theta)$, and fd_i is the cumulative fraction of faults detected at position rp_i in $F(\theta)$ and is computed as follows:

$$fd_i = \frac{\sum_{j=1}^{i} nd_\theta(rp_j)}{|D(TS)|} \tag{3.2}$$

where, $nd_\theta(rp_j)$ is the number of faults detected at the relative position rp_j in θ.

For example, suppose the test suite $TS=\{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}\}$ consists of 10 tests that detect four faults $D(TS) = \{d_1, d_2, d_3, d_4\}$ in a system. The following tests fail because of individual faults: $TS(d_1) = \{t_5, t_7\}$, $TS(d_2) = \{t_3, t_7\}$, $TS(d_3) = \{t_5\}$, and $TS(d_4) = \{t_3, t_9\}$. Let $\theta_1 = \langle t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10} \rangle$ and $\theta_2 = \langle t_{10}, t_6, t_4, t_1, t_9, t_2, t_5, t_7, t_3, t_8 \rangle$ be two prioritized test sequences. The rates of fault detection for θ_1 and θ_2 can be graphically represented as in Figure 2.

Figure 2 shows that, after executing 30% (test suite fraction rp=0.3) of the tests in sequence θ_1, 50% of the faults were detected (fd=0.5), and after executing 50% of the tests (rp=0.5) in sequence θ_1, all the faults were detected (fd=1.0). For sequence θ_2, after executing 50% of the tests (rp=0.5), 25% of the faults were detected (fd=0.25), after executing 70% of the tests (rp=0.7), 75% of the faults were detected (0.75), and after executing 80% of the tests (rp=0.8), all the faults were detected (fd=1.0). Thus, sequence θ_1 leads to a faster rate of fault detection.

Rate of fault detection for θ_1 and θ_2				
θ_1	*fd*: fraction of faults detected	0.5	1.0	
	rp: Test suite fraction	0.3	0.5	
θ_2	*fd*: fraction of faults detected	0.25	0.75	1.0
	rp: Test suite fraction	0.5	0.7	0.8

Figure 2. Rate of fault detection for θ_1 and θ_2

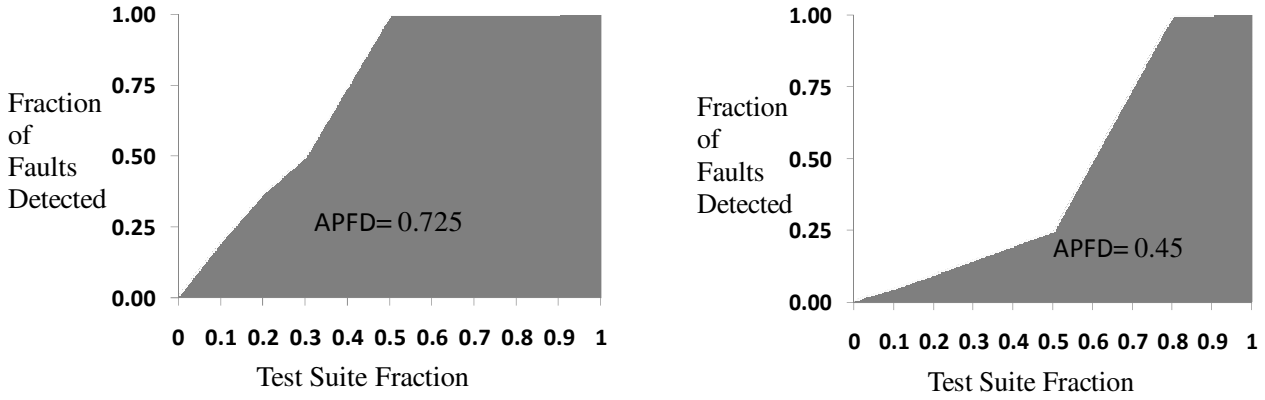
In order to measure how rapidly a prioritized test sequence detects faults during the execution of sequence θ, a weighted average of the percentage of faults detected, $APFD(\theta)$, was introduced [15]. For a given rate of fault detection $RFD(\theta) = \langle (rp_1, fd_1), \ldots, (rp_q, fd_q) \rangle$, $APFD(\theta)$ is computed as:

$$APFD(\theta) = \frac{\sum_{i=0}^{q} (fd_{i+1} - fd_i)(2 - rp_{i+1} - rp_i)}{2} \tag{3.3}$$

where $(rp_0, fd_0) = (0,0)$ and $(rp_{q+1}, fd_{q+1}) = (1, 1)$.

[1] At the same position in θ more than one fault may be detected, therefore, some positions in $P(\theta)$ may have the same value.

The values of $APFD(\theta)$ range from 0 to 1, $0 \leq APFD(\theta) < 1$, where higher $APFD(\theta)$ value means a faster (better) fault detection rate. Note: $APFD(\theta) < 1$, because we assume that $fd_0 < fd_1$, where fd_0=0. For the two sequences θ_1 and θ_2 presented earlier, Figures 3.a and 3.b show the percentage of faults detected versus the fraction of the test suite used for these two sequences. The area under the curves represents the weighted average of the percentage of faults detected over the life of the test suite. The resulting APFDs for the two sequences θ_1 and θ_2 are: $APFD(\theta_1) = 0.725$ and $APFD(\theta_2) = 0.45$. As a result, sequence θ_1 leads to a higher rate of fault detection than θ_2.



a) APFD for sequence θ_1 b) APFD for sequence θ_2

Figure 3. APFD for θ_1 and θ_2

The simplest test prioritization method is random test prioritization where tests are ordered randomly. For a test suite of size N, there are N ! possible test sequences. Random test prioritization selects randomly one of these sequences and thus may be viewed as a "no test prioritization" approach and, therefore, used as a base-line for comparison with other test prioritization methods.

The goal of model-based test prioritization is early fault detection in the modified system. This is achieved in our approach by using the original and modified system models, the difference between these two models, and the information collected during execution of the modified model on the test suite. The collected information is used to prioritize the test suite. The prioritized test suite is then used for retesting the modified system. Notice that execution of the model is very fast compared to the execution of the actual system. Therefore, execution of the model for the whole test suite is relatively inexpensive, with the result that the overhead associated with test prioritization is relatively small.

Changes in specifications frequently lead to changes in system models and system implementations. Model-based test prioritization uses the original model M_o and the modified model M_m and automatically identifies the difference between these two models [17, 28], as a set of elementary model modifications. There are two types of elementary modifications: a transition addition and a transition deletion. As a result, the difference between M_o and M_m is represented by a set R_a of added transitions and a set R_d of deleted transitions. Any complex modification to the model can be expressed as sets R_a and R_d of these two types of elementary modifications.

A straightforward algorithm for identifying model differences, hence modifications in terms of added and deleted transitions can be devised as follows. Let R_o be a set of transitions of M_o and R_m be a set of transitions of M_m. The algorithm then amounts to taking two set differences between R_o and R_m provided that the state and transition names are preserved across versions of the models. That is, only previously unused state and transition names appear in the modified model M_m for the added states and transitions, Then, the algorithm becomes the following:

$$R_a = R_m - R_o \quad (3.4)$$

$$R_d = R_o - R_m \quad (3.5)$$

The complexity of this algorithm is at most, $4*(|R_o| + |R_m|)-1$ comparisons provided that the sets R_o and R_m are sorted in the same order over the transition names.

Notice that the sequence in which these elementary modifications is applied to the original model is not relevant. A transition addition may occur between existing states or may involve an introduction of a new state when a transition is added to the model. Similarly, a transition deletion may, in some cases, result in the deletion of a state. Notice, however, that an addition of a new state and a deletion of a state are always associated with a transition addition and a transition deletion. Therefore, addition of a new state or a deletion of a state is not considered to be an elementary modification. Note that the approach of identifying the differences between the modified model and the original model is straightforward. Even when the name of a state is changed, one can identify all incoming and outgoing transitions in the original model as deleted transitions and all incoming and outgoing transitions to the state with the new name in the modified model as added transitions.

For example, the difference between the original model of Figure 1 and the modified model of Figure is: deletion of transition T_{11} and addition of transition T_{12}, i.e., $R_a = \{T_{12}\}$ and $R_d = \{T_{11}\}$. Transition T_{11} no longer exists in the modified model, and it is shown as a dashed line in Figure , merely to aid presentation.

In this paper, based on the above approach, we propose two methods of model-based test prioritization: *model-based selective test prioritization* and *model dependence-based test prioritization* in Sections 4 and 5, respectively.

4. Model-based selective test prioritization

Model-based selective test prioritization (henceforth called *selective test prioritization*) assigns a high priority to tests that execute the modified (added or deleted) transitions in the modified model. A low priority is assigned to tests that do not execute any modified transition. Let TS_H be a set of high priority tests in a test suite TS and TS_L be a set of low priority tests. Sets TS_H and TS_L are disjoint and test suite $TS = TS_H \cup TS_L$. Notice that information about executed added/deleted transitions may also be used in regression test selection [9, 13, 26, 27, 29]. However, in this paper, we concentrate only on using this information for test suite prioritization. We present two versions of the selective test prioritization and investigate their effectiveness in early detection of faults.

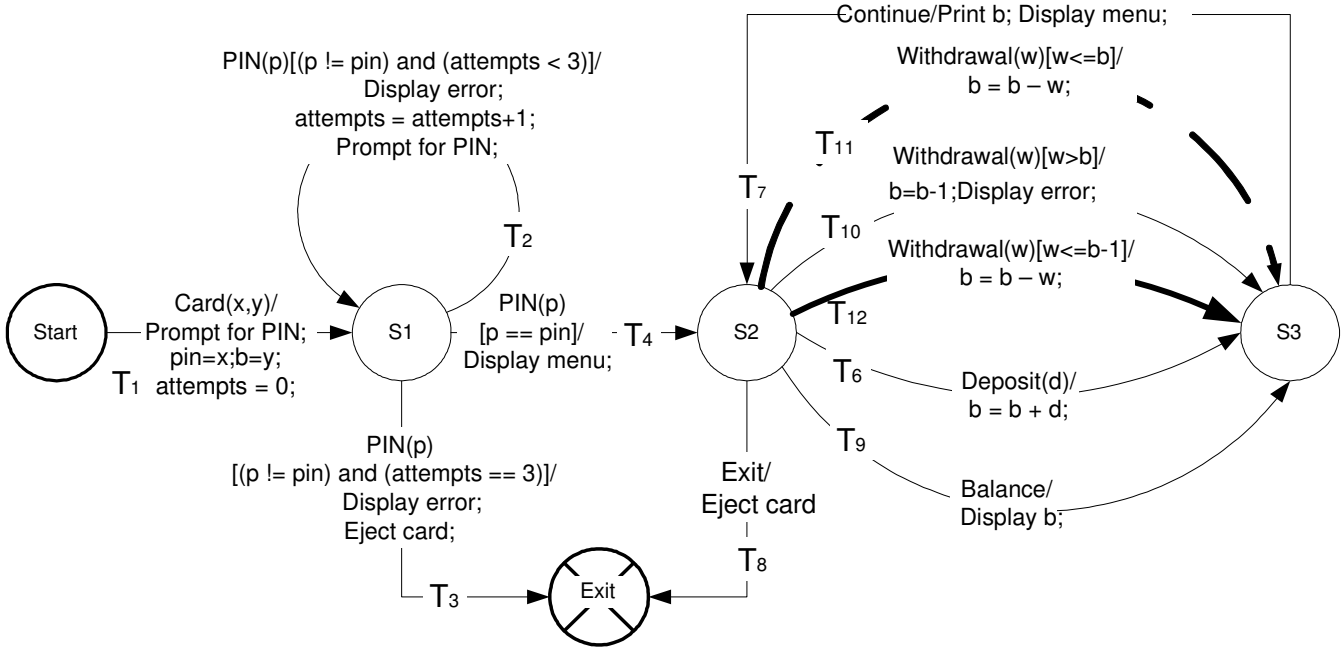


Figure 4. A modified model of Figure 1

Version I: In this version, modified transitions of M_m are represented only by added transitions of R_a. Since deleted transitions of R_d do not exist in the modified model M_m, they are ignored. Every transition $T \in R_a$ is monitored during execution of M_m on test suite TS. Let t be a test and $\boldsymbol{\tau}(t) = \langle T_{i_1}, T_{i_2}, \ldots, T_{i_m} \rangle$ be a sequence of transitions traversed during execution of M_m on t. If, during execution of M_m on test t, transition $T \in R_a$ is executed, a high priority is assigned to t, i.e., $t \in TS_H$. Otherwise, a low priority is assigned to t, i.e., $t \in TS_L$.

Example 1. Consider the following three tests and the corresponding sequences of transitions traversed during execution of the modified model of Figure :

t_1: Card(12,10), PIN(12), Withdrawal(5), Continue(), Exit()
t_2: Card(12, 10), PIN(12), Withdraw(15), Continue(), Exit()
t_3: Card(12, 10), PIN(12), Deposit(20), Continue(), Withdraw(10), Continue(), Exit()
t_4: Card(15, 10), PIN(15), Deposit(20), Continue(), Withdraw(10), Continue(), Exit()

$\boldsymbol{\tau}(t_1) = <T_1, T_4, T_{12}, T_7, T_8>$
$\boldsymbol{\tau}(t_2) = <T_1, T_4, T_{10}, T_7, T_8>$
$\boldsymbol{\tau}(t_3) = <T_1, T_4, T_6, T_7, T_8>$
$\boldsymbol{\tau}(t_4) = <T_1, T_4, T_6, T_7, T_{12}, T_7, T_8>$

A set of added transitions for the modified model of Figure 4 is $R_a = \{T_{12}\}$. Based on the execution of these tests, the following high and low priority tests are identified: $TS_H = \{t_1, t_4\}$ and $TS_L = \{t_2, t_3\}$. Notice that since T_{12} is executed on test t_1 and t_4, a high priority is assigned to these tests.

***Version II*:** In this version, the modified transitions of M_m are represented by added and deleted transitions, i.e., transitions of set R_a and set R_d. These transitions are monitored during execution of the modified model M_m on test t. In Version II additional instrumentation of M_m is required to capture the execution of deleted transitions because deleted transitions no longer exist in M_m. When M_m, during its execution, is in a state from which the deleted transition was outgoing, it is possible to capture traversal of a deleted transition (imitation of execution of a deleted transition) when the event associated with the deleted transition is received and the enabling condition of the deleted transition evaluates to TRUE. This is considered as "execution" of the deleted transition.

The following governs the execution of the EFSM model M on a sequence of events $t = <E_1(x_1), E_2(x_2), \ldots, E_n(x_n)>$ that may capture traversal of a deleted transition (imitation of execution of a deleted transition): suppose a modified model M_m is in some state S and event $E_i(x_i)$ occurs. If there exists a deleted transition $T_d \in R_d$ such that:

a. $E(T_d) = E_i$
b. $C(T_d)$ evaluates to TRUE
c. $S_b(T_d) = S$

then, we say that the deleted transition T_d is "executed".

Notice that "execution" of a deleted transition means that the modified model M_m remains in the state S, no action is performed, and it does not consume the event. When M_m is executed for a given sequence of events $t = < E_1(x_1), E_2(x_2), \ldots, E_n(x_n)>$, a sequence of transitions $\boldsymbol{\tau}(t) = < T_{i_1}, T_{i_2}, \ldots, T_{i_m}>$ is executed, where some of the executed transitions in the sequence may be "deleted" transitions of set R_d. Notice that $\boldsymbol{\tau}(t)$ may or may not contain "executed" deleted transitions.

Example 2. Consider the following three tests and the corresponding sequences of transitions traversed during execution of the modified model of Figure 4 on these tests, in which transition T_{11} is deleted and transition T_{12} is added:

t_1: Card(12,10), PIN(12), Withdrawal(15), Continue(), Exit()
t_2: Card(12, 10), PIN(12), Withdraw(10), Continue(), Deposit(10), Continue() Exit()
t_3: Card(12, 10), PIN(12), Withdraw(5), Continue(), Exit()

$\boldsymbol{\tau}(t_1) = <T_1, T_4, T_{10}, T_7, T_8>$
$\boldsymbol{\tau}(t_2) = <T_1, T_4, (T_{11}), T_6, T_7, T_8>$
$\boldsymbol{\tau}(t_3) = <T_1, T_4, (T_{11}), T_{12}, T_7, T_8>$

Notice that, in sequence $\boldsymbol{\tau}(t_2)$, transition T_4 is executed followed by deleted transition (T_{11}) (indicated in parentheses). The deleted transition T_{11} is executed because after executing the two events: *Card*(12, 10) and *PIN*(12), the model moves to state *S*2, event *Withdrawal(10)* occurs, and an enabling condition associated with transition T_{11} (i.e., "*w* <= *b*") evaluates to TRUE. Note that in sequence $\boldsymbol{\tau}(t_3)$, T_{12} and T_{11} are "executed". However, only T_{12} is actually executed whereas we only capture a potential execution of deleted transition T_{11}.

If a transition T in R_a or R_d is executed during execution of modified model M_m on test t, a high priority is assigned to test t, i.e., $t \in TS_H$. Otherwise, a low priority is assigned to test t, i.e., $t \in TS_L$. For example, the set of added transitions for the modified model of Figure is $R_a = \{T_{12}\}$ and the set of deleted transitions is $R_d = \{T_{11}\}$. Based on the execution of the three tests discussed in Example 2, the following high and low priority tests are identified: $TS_H = \{t_2, t_3\}$ and $TS_L = \{t_1\}$. Notice that since T_{11} is executed on test t_2 and T_{12} is executed on test t_3, a high priority is assigned to these two tests.

During system retesting based on Version I or Version II, tests with high priority are executed first followed by execution of low priority tests. High priority tests and low priority tests are ordered using random ordering. The algorithm for selective

test prioritization is shown in Figure . In the first step, high priority tests are ordered randomly (lines 1-4), then low priority tests are ordered randomly (lines 5-8) in the prioritized test sequence θ.

Input: A set of high priority tests: TS_H
A set of low priority tests: TS_L
Output: Prioritized test sequence: θ

1. **for** p=1 to $|TS_H|$ **do**
2. select randomly and remove test t from TS_H
3. insert t into θ at position p
4. **endfor**
5. **for** p=1 to $|TS_L|$ **do**
6. select randomly and remove test t from TS_L
7. insert t into θ at position $p + |TS_H|$
8. **endfor**
9. output θ

Figure 5. Selective test prioritization algorithm

In the next section, we present our model dependence-based test prioritization method in which high priority tests TS_H are prioritized using model dependence analysis.

5. Model dependence-based test prioritization

The selective test prioritization method presented in the previous section may improve the effectiveness of test prioritization with respect to the fault detection capability when compared to random test prioritization. In this section, we propose model dependence-based test prioritization to further improve the effectiveness of test prioritization with respect to fault detection capability. This dependence based approach uses model dependence analysis [9, 12, 13, 17, 26, 28] to prioritize high priority tests TS_H identified by *Version II* of selective test prioritization. This improvement is achieved by identifying different ways in which added and deleted transitions interact with the remaining parts of the model and using this information to prioritize high priority tests. The model dependence analysis is based on two types of dependences that may exist in the model: data dependence and control dependence. These model dependences are between transitions and they are used to identify potential "interactions" between transitions. The goal of model dependence-based test prioritization is to identify unique patterns of interactions between model transitions and added/deleted transitions that are present during execution of the modified model on tests in a given test suite. The interaction pattern and the test suite are used to prioritize the test suite for retesting the modified system.

5.1. Model Dependence Analysis

Before we present the model dependence-based test prioritization technique, we introduce dependences that may exist in the EFSM model. We extend the existing code-based dependence analysis, which is commonly used for white box testing [30] to model dependence analysis [9, 17]. We define two types of dependences between transitions: data dependence and control dependence. Data dependence captures the notion that one transition defines a value of a variable and another transition may potentially use this value. Control dependence captures the notion that one transition may affect traversal of another transition. These dependences capture the notion of potential "interactions" between transitions in the model.

5.1.1. Data Dependence

Model dependence analysis with respect to data dependence focuses on occurrences of variables within the system model. Each variable occurrence is classified as being a variable definition or a variable use. We refer to these as *definition* and *use*, respectively. A definition of a variable v in a transition is any occurrence of v at which v is assigned a value. A transition can define a variable v by defining v as a part of the action(s) (e.g., $v = x + 5$). A use of a variable v in a transition is any occurrence of v that references the value of v. A transition can reference a variable v in a Boolean expression associated with the transition (*e.g.*, $[v < 0]$) or by using v in action(s) associated with the transition (*e.g.*, $x = v + 5$).

Let T be a transition. The following concept related to transition T is introduced:

- D(T) is a set of variables defined by transition T, i.e., variables defined by an action(s) of T.
- U(T) is a set of variables used in transition T, i.e., variables used in a condition and an action(s) of T.

For example, in the EFSM model of Figure 1, for transition T_1, D(T_1) = {*pin*, *b*, *attempts*}and U(T_1) = {x, y}.

Data dependence captures the notion that one transition defines a value of a variable and another transition may potentially use this value. There exists a *data dependence* between transitions T_i and T_k if transition T_i modifies the value of variable v, transition T_k uses v, and there exists a path (transition sequence) in the model from T_i to T_k along which v is not modified [9]. More formally, there exists *data dependence* between transitions T_i and T_k if there exists a variable v such that: (1) $v \in D(T_i)$, (2) $v \in U(T_k)$, and (3) there exists a path (transition sequence) in the EFSM model from T_i to T_k along which v is not modified; such a path is referred to as a *definition-clear path.* For example, there exists a data dependence between transitions T_6 and T_{11} in the model of Figure 1 because transition T_6 assigns a value to variable b in the action "$b = b + d$", transition T_{11} uses variable b in condition "[$w <= b$]", and there exists a path (sequence of transitions (T_6, T_7, T_{11})) from T_6 to T_{11} along which b is not modified.

5.1.2. Control Dependence

Control dependence was originally defined for a program's Control Flow Graph (CFG) [31]. Control dependence captures the notion that one node in the control graph may affect the execution of another node. In this paper, we extend the concept of program control dependence to the EFSM model [9]. Control dependence in an EFSM exists between transitions and it captures the notion that one transition may affect traversal of another transition. Control dependence between transitions is defined similarly to control dependence between nodes of a CFG [31], i.e., in terms of the concept of post-dominance. Let Y and Z be two states (nodes) and T be an outgoing transition (edge) from Y. State Z *post-dominates* state Y iff Z is on every path from Y to the exit state of the EFSM. State Z post-dominates transition T iff Z is on every path from Y to the exit state of the EFSM through transition T. Transition T_k is control dependent on transition T_i iff: (1) $S_b(T_k)$ does not post-dominate $S_b(T_i)$ and (2) $S_b(T_k)$ post-dominates transition T_i. Less formally, transition T_k is control dependent on T_i if (1) there exists another transition T that if executed instead of T_i prevents T_k from being executed, and (2) T_k cannot be executed without execution of T_i, i.e., in every path in which T_i is executed T_k is also executed. Notice that the definition of control dependence presented in this paper captures the same view as the definition of control dependence between nodes in a CFG [31]. For example, transition T_{11} is control dependent on T_4 in the model of Figure 1 because (1) $S_b(T_4)$ does not post dominate $S_b(T_{11})$ (condition 1 of control dependence definition is true) and (2) state $S_b(T_{11})$ post dominates transition T_4 (condition 2 is TRUE). Note that $S_b(T_4)$ is $S1$ and $S_b(T_{11})$ is $S2$. The issue of control dependence in EFSMs is discussed in more detail elsewhere [37, 38].

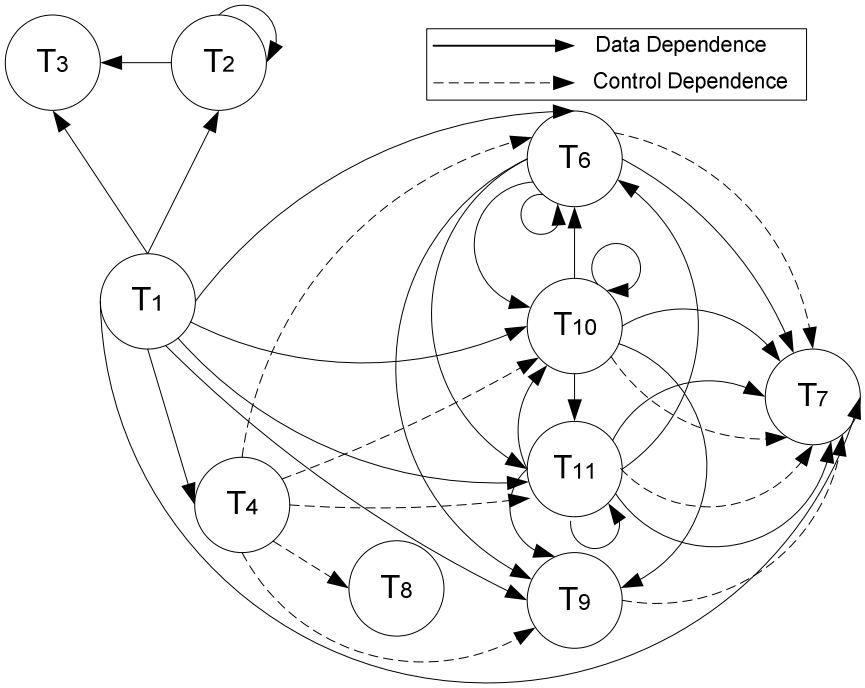


Figure 6. Static EFSM dependence graph of the model in Figure 1

5.1.3. Static EFSM Dependence Graph

Static data and control dependences in an EFSM model can be depicted by a graph referred to as a *static dependence graph*, where nodes represent transitions and directed edges represent data and control dependences. More formally, let $M = (\Sigma, Q, Start, Exit, V, O, R)$ be an EFSM model and let $G = (R, E)$ be a model dependence graph of model M where:

R is a set of nodes (set of transitions)

E is a binary relation on R, $E \subseteq R \times R$, representing data and control dependences by a set of directed edges where:

$(T_i, T_k) \in E$, if there exists a data or control dependence from transition T_i to transition T_k.

Figure 6 shows a dependence graph of the model of Figure 1. Note that data dependences are shown as solid edges and control dependences are shown as dashed edges.

5.1.4. Dynamic Dependence

In order to prioritize tests, we are interested in data and control dependences that are present during execution of the modified model M_m on each test t in test suite TS. We refer to these dependences as *dynamic data and control dependences*. Let t be a test and $\boldsymbol{\tau}(t) = < T_{i_1}, T_{i_2}, \ldots, T_{i_m} >$ be a sequence of transitions traversed during execution of the modified model M_m on test t of a test suite TS. Notice that some of the executed transitions in sequence $\boldsymbol{\tau}(t)$ may be "deleted" transitions of set R_d.

There exists a dynamic data dependence between transitions T_i and T_k in $\boldsymbol{\tau}(t)$ if transition T_i modifies the value of variable v, transition T_k uses v, and for all transitions T_j in the modified model M_m, $i < j < k$, v is not modified. More formally, there exists a dynamic data dependence between transitions $\boldsymbol{\tau}(t)[i]$ and $\boldsymbol{\tau}(t)[k]$ in $\boldsymbol{\tau}(t)$, $i < k$, if there exists a variable v such that: (1) $v \in D(\boldsymbol{\tau}(t)[i])$, (2) $v \in U(\boldsymbol{\tau}(t)[k])$, and (3) and for all j, $i < j < k$, $v \notin D(\boldsymbol{\tau}(t)[j])$ such that $\boldsymbol{\tau}(t)[j] \in R_m$ where R_m is the set of transitions in M_m.

There exists a *dynamic control dependence* in $\boldsymbol{\tau}(t)$ between transitions $\boldsymbol{\tau}(t)[i]$ and $\boldsymbol{\tau}(t)[k]$, $i < k$, if there exists a control dependence between $\boldsymbol{\tau}(t)[i]$ and $\boldsymbol{\tau}(t)[k]$, and for all j, $i < j < k$, there is no control dependence between $\boldsymbol{\tau}(t)[j]$ and $\boldsymbol{\tau}$(t)[k] such that $\boldsymbol{\tau}(t)[j] \in R_m$.

For example, consider the following test t for the modified model of Figure :

t: Card(5,6), PIN(5), Deposit(1), Continue(), Withdrawal(2), Continue(), Withdrawal(90), Continue(), Exit().

On test t, the following transitions are executed in sequence $\boldsymbol{\tau}(t) = <T_1, T_4, T_6, T_7, (T_{11}) T_{12}, T_7, T_{10}, T_7, T_8>$. There exists a dynamic data dependence between T_6 and T_{12} with respect to variable b and a dynamic control dependence between T_4 and T_6 in $\boldsymbol{\tau}(t)$.

5.1.5. Dynamic Dependence Graph

Dynamic data and control dependences in $\boldsymbol{\tau}(t)$ can be depicted by a graph, referred to as a *dynamic dependence graph*, where nodes represent transitions and directed edges represent dynamic data and control dependences in $\boldsymbol{\tau}(t)$. Formally, let t be a test and $\boldsymbol{\tau}(t) = < T_{i_1}, T_{i_2}, \ldots, T_{i_m} >$ be a sequence of transitions traversed during execution of the modified model M_m on test t. Let $G'(\boldsymbol{\tau}(t)) = (R_m, E_m)$ be a dynamic model dependence graph of $\boldsymbol{\tau}(t)$ where:

R_m is a set of transitions in M_m

E_m is a binary relation on R_m, $E_m \subseteq R_m \times R_m$, representing data and control dependences where $(\boldsymbol{\tau}(t)[i], \boldsymbol{\tau}(t)[k]) \in E_m$, if there exists a dynamic data or control dependence from transition $\boldsymbol{\tau}(t)[i]$ to transition $\boldsymbol{\tau}(t)[k]$ in $\boldsymbol{\tau}(t)$.

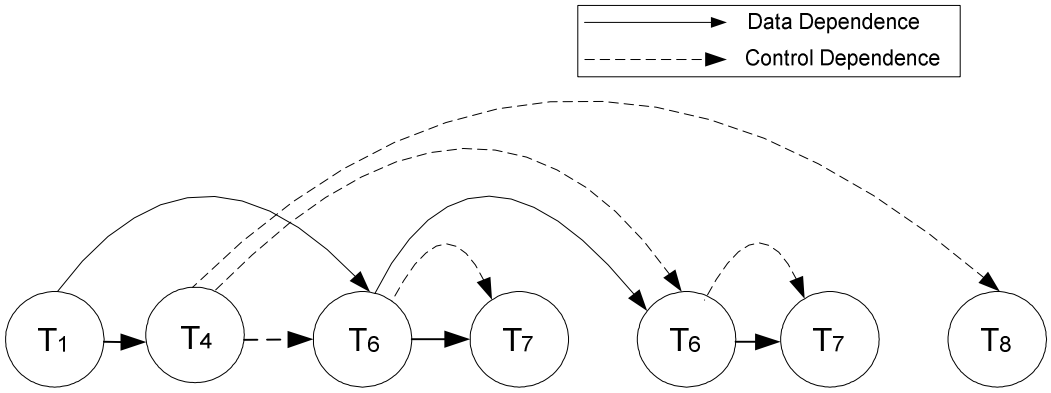


Figure 7. Dynamic dependence graph of $\tau(t)$

For example, consider the following test t for the modified model of Figure :

t: Card(5,6), PIN(5), Deposit(1), Continue(), Deposit (6), Exit().

On test t, the sequence of transitions $\tau(t) = \langle T_1, T_4, T_6, T_7, T_6, T_7, T_8 \rangle$ is executed. Figure shows the dynamic dependence graph of $\tau(t)$ where data dependences are shown as solid edges and control dependences are shown as dashed edges.

5.2. Interaction patterns

In order to prioritize high priority tests TS_H identified by *Version II* of the selective test prioritization method, the dependence analysis is used to identify different ways added and deleted transitions interact with the remaining parts of the model. The principle of model dependence-based test prioritization is to identify unique patterns of interactions between the model and the added/deleted transitions that are present during execution of the modified model on tests in TS_H. This information can be used to guide the choice of priorities. During execution of the modified model M_m on a test t, we identify three types of interactions between a modified part of the model and the remaining parts of the model: (1) the effect of the model on the modification (affecting transitions), (2) the affect of the modification on the remaining part of the model (affected transitions), and (3) the side effects of transitions caused by the modification. These interactions may be viewed as computing a model slice [28].

Since there are three types of interactions between a modified part of the model and the remaining parts of the model, in this paper, we introduce three types of interaction patterns related to each modification (i.e., an added or deleted transition) as shown in Figure : (1) an affecting interaction pattern, (2) an affected interaction pattern, and (3) a side-effect interaction pattern. The affecting interaction pattern captures interactions between model transitions that affect the modification. The affected interaction pattern captures transitions that are affected by the modification. Finally, the side-effect interaction pattern captures interactions that occur because of side effects introduced by the modification. In this context, we consider a side-effect to be an introduction of a new dependence or a removal of an existing dependence between other transitions.

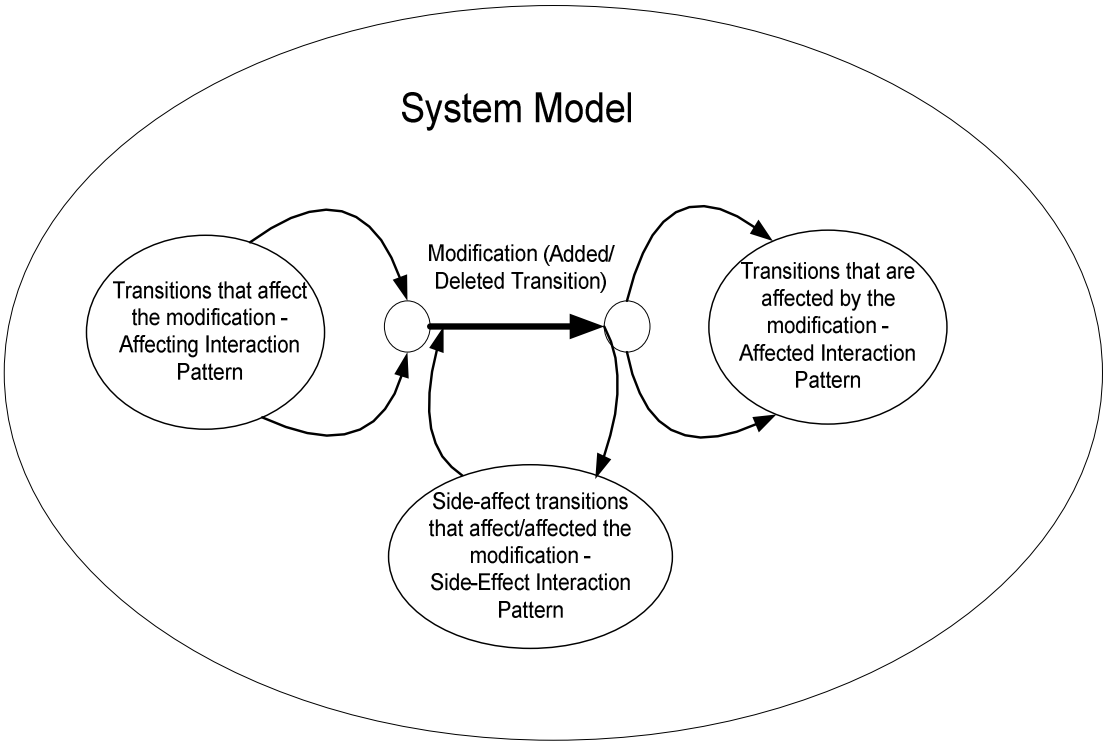


Figure 8. Interaction patterns

Interactions between model transitions are represented as model dependences between transitions. Consequently, the affecting interaction pattern, affected interaction pattern, and side-effect interaction pattern are represented as model dependence subgraphs (derived from a model dependence graph) with respect to added and deleted transitions.

5.2.1. Affecting Interaction Pattern

An affecting interaction pattern is formed by identifying transitions that affect an added or deleted transition T during execution of the modified model on a test t of test suite TS. These transitions are identified by traversing backwards within $G'(\tau(t)) = (R_m, E_m)$, dynamic model dependence graph of $\tau(t)$ of test t, starting from the added or deleted transition T and by marking all transitions that affect T. Then, an affecting interaction pattern, IP(t, T), is formed as a subgraph of $G'(\tau(t))$ by keeping only those edges of $G'(\tau(t))$ that are within paths starting at marked transitions and terminating at T.

Example 3. Consider the following tests for the modified model of Figure in which transition T_{12} is added to the model and transition T_{11} is deleted:

t_1: Card(5,6), PIN(5), Deposit(1), Continue(), Withdrawal(2), Continue(), Withdrawal(90), Continue(), Exit()

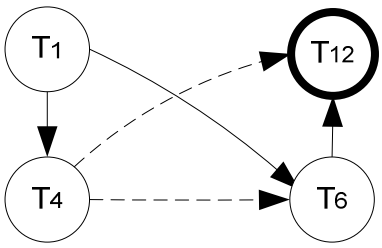
t_2: Card(5,6), PIN(5), Deposit(11), Continue(), Withdrawal(17), Continue(), Withdrawal(19), Continue(), Exit()

On these tests the following sequences of transitions are executed in the model of Figure 4:

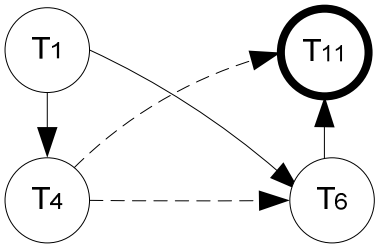
$\tau(t_1) = \langle T_1, T_4, T_6, T_7, (T_{11}), T_{12}, T_7, T_{10}, T_7, T_8 \rangle$

$\tau(t_2) = \langle T_1, T_4, T_6, T_7, (T_{11}), T_{10}, T_7, T_8 \rangle$

where added transition T_{12} and deleted transition T_{11} are executed in $\tau(t_1)$ and deleted transition T_{11} is executed in $\tau(t_2)$. The affecting interaction pattern with respect to added transition T_{12} for test t_1 and deleted transition T_{11} for test t_2 is shown in Figure .



Affecting interaction pattern for T_{12} for test t_1



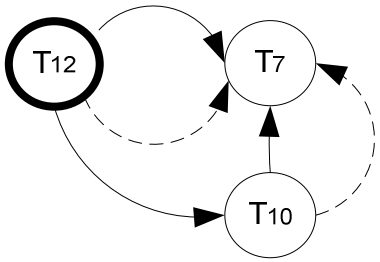
Affecting interaction pattern for T_{11} for test t_2

Figure 9. Affecting interaction pattern for tests t_1 and t_2

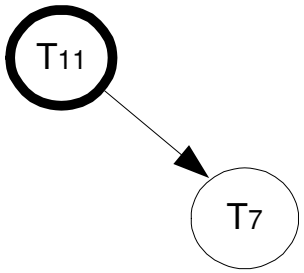
5.2.2. Affected Interaction Pattern

An affected interaction pattern is formed by identifying transitions that are affected by an added or deleted transition T during execution of the modified model on a test t of test suite TS. These transitions are identified by traversing forwards within $G'(\tau(t)) = (R_m, E_m)$, dynamic model dependence graph of $\tau(t)$ of test t, starting from the added or deleted transition T and by marking all transitions that are affected T. Then, an affected interaction pattern, IP(t, T), is formed as a subgraph of $G'(\tau(t))$ by keeping only those edges of $G'(\tau(t))$ that are within paths starting at T and terminating at marked transitions.

For example, the affected interaction pattern for the two tests discussed in Example 3 is shown in Figure .



Affected interaction pattern for T_{12} for test t_1



Affected interaction pattern for T_{11} for test t_2

Figure 10. Affected interaction pattern for test t_1 and t_2

5.2.3. Side-Effect Interaction Pattern

A side-effect interaction pattern captures interactions between transitions that occur because of side effects introduced by a transition of interest (an added or deleted transition). An addition or deletion of a transition may introduce in the modified model new dependences that do not exist in the original model or it may cause the removal of some dependences that do exist in the original model [9, 17, 12, 13]. During execution of the modified model on a test, new or removed data and control dependences that are present during model execution are identified. These dependences are referred to as a Side-Effect Interaction Pattern. For example, deleting transition T_9 from the ATM model of Figure 1 eliminates the existing data dependence between transitions T_1 and T_7 with respect to variable b in the modified model because there is no definition-clear path with respect to variable b from T_1 to T_7 in the modified model.

5.3. Model Dependence-Based Test Prioritization Algorithm

Given a set TS_H of high priority tests determined in *Version II* of the selective test prioritization algorithm, during the execution of the modified model M_m on a test t in TS_H, a set of interaction patterns is computed for each added or deleted transition T, i.e., $\{IP_1(t, T), \ldots, IP_q(t, T)\}$. Let $TS(IP_i(t, T)) \subseteq TS_H$ be a set of tests where for a test t in TS_H and for an added or deleted transition T, the following holds: (1) an added or deleted transition T is executed in M_m on test t, and (2) interaction

pattern $IP_i(t, T)$ is computed with respect to T in $\tau(t)$. We refer to $IPS = \{TS(IP_1(t, T)),\ldots,TS(IP_q(t, T))\}$ as an *interaction pattern test distribution* set. Notice that each test $t \in TS_H$ belongs to at least one $TS(IP_i(t, T))$, and the same test may belong to different $TS(IP_i(t, T))$ sets.

The algorithm that computes a prioritized test sequence using the interaction patterns is shown in Figure 11. The input to the algorithm is a set of interaction patterns test distribution $IPS = \{TS(IP_1(t, T)), \ldots, TS(IP_q(t, T))\}$, a set of high priority tests TS_H, and a set of low priority tests TS_L. The output of the algorithm is the prioritized sequence of tests θ.

The algorithm in the first step (lines 1-13) prioritizes tests that are associated with interaction patterns, by iteratively selecting (lines 3-12) one test from each interaction pattern $TS(IP_i(t, T))$ and inserting them into the prioritized sequence. After selecting one test from each interaction pattern, the algorithm repeats this process (lines 2-13) until all tests in *IPS* are selected. In the next step (lines 14-17), the algorithm continues with the prioritization with low priority tests by ordering them randomly. Notice that the algorithm also selects $TS(IP(t, T))$ from *IPS* and tests from $TS(IP(t, T))$ randomly. In addition, no assumption is made about the order in which interaction patterns are processed, i.e., interaction patterns are randomly ordered for test prioritization.

Input: a set of interaction pattern test distribution $IPS = \{TS(IP_1(t, T)),\ldots,TS(IP_q(t, T))\}$
a set of high priority tests: TS_H
a set of low priority tests: TS_L

Output: Prioritized test sequence: θ

```
1   p=0
2   while true do
3     for every TS(IP(t, T)) ∈ IPS do //select randomly TS(IP(t, T)) from IPS
4       if TS(IP(t, T)) ≠ ∅ then
5         p=p+1
6         select randomly test t from TS(IP(t, T))
7         remove t from every TS(IP(t, T)) to which t belongs
8         insert t into θ at position p
9         if p=|TS_H| then exit while loop
10      endif
11      if TS(IP(t, T)) = ∅ then IPS = IPS - {TS(IP_1(t, T))}
12    endfor
13  endwhile
14  for p=1 to |TS_L| do
15    select randomly and remove test t from TS_L
16    insert t into θ at position p + |TS_H|
17  endfor
18  output θ
```

Figure 11. Model dependence-based test prioritization algorithm

Example 4. Consider the test suite *TS* consisting of seven tests, $TS = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7\}$, that detect two faults $D(TS) = \{d_1, d_2\}$ in a system and the following tests fail because of individual faults: $TS(d_1) = \{t_5\}$ and $TS(d_2) = \{t_5, t_7\}$. Suppose that the following high and low priority tests are determined: $TS_H = \{t_1, t_4, t_5, t_7\}$ and $TS_L = \{t_2, t_3, t_6\}$, and three interaction patterns are computed with the following distribution of tests among them:

$TS(IP_1(t, T)) = \{t_4, t_5\}$
$TS(IP_2(t, T)) = \{t_1, t_4, t_7\}$
$TS(IP_3(t, T)) = \{t_5, t_7\}$

The algorithm in Figure 11 may output the following sequences out of many possible sequences: $\theta = \{t_4, t_1, t_5, t_7, t_3, t_2, t_6\}$.

6. Measuring effectiveness of early fault detection

In order to compare different test prioritization methods an empirical study may be performed with different systems that contain known faults. In this paper, the rate of fault detection [15] is used as a measure of the effectiveness of early fault detection to evaluate the effectiveness of test prioritization methods for a given system(s) with known fault(s). Notice that the

rate of fault detection is not used during the process of prioritizing tests by test prioritization methods, but it is used only during an empirical study to measure the effectiveness of individual test prioritization methods.

Test prioritization methods may generate many different solutions (prioritized test sequences) for a given test suite. For example, for test suite *TS* of size *N*, random test prioritization generates a prioritized test sequence out of *N*! possible test sequences (all possible permutations of tests in *TS*). A factor that may influence the resulting prioritized test sequence is, for example, an order in which tests are processed during the prioritization process. As a result, a test prioritization method under study may generate different prioritized test sequences with different rates of fault detection.

To compare test prioritization methods under study, we need to determine an average rate of fault detection for each test prioritization method for a given system, known faults, and a test suite. Notice that the major assumptions for the empirical study are that (1) all system faults that are used in the empirical study are known, e.g., by seeding faults into the system, and (2) one is able to identify the mapping between faults and failed tests, i.e., for each failed test it is possible to determine which fault(s) caused the failure.

Let $TS = \{t_1, \ldots, t_N\}$ be a test suite of size N and let $D(TS) = \{d_1, \ldots, d_L\}$ be a set of L known faults in the system that are detected by TS. Let $TS(d)$ be a set of failed tests caused by fault $d \in D(TS)$. Let $\theta = <t_{i_1}, t_{i_2}, \ldots, t_{i_N}>$ be a prioritized sequence of tests of test suite TS, and let $P(\theta) = <rp_\theta(d_1),\ldots, rp_\theta(d_L)>$ be a list of relative positions of the first failed tests for all faults in $D(TS)$ for test sequence θ. The rate of fault detection for θ can be determined based on $P(\theta)$ as discussed in Section 3. Since the rate of fault detection is based on the concept of a relative position of the first failed test, we introduce the concept of the *average relative position*, $RP(d)$, of the first failed test that detects fault d. Notice that $rp_\theta(d)$ represents a relative position of the first failed test that detects fault d in test sequence θ, whereas $RP(d)$ represents an average relative position of the first failed test that detects d for a test prioritization method.

Let $AP(d)$ be the average (expected) position of the first failed test that detects fault d for a test prioritization method under study. The following formula is used to compute $AP(d)$:

$$AP(d) = \sum_{i=1}^{N} i \cdot pr(i,d) \tag{6.1}$$

where $pr(i,d)$ is a probability that a test prioritization method under study selects the first failed test caused by fault d at position i, i.e., the first failed test $t \in TS(d)$ caused by fault d is in the i^{th} position.

$RP(d)$, the average relative position of the first failed test that detects d, is computed from $AP(d)$ as follows:

$$RP(d) = \frac{AP(d)}{N} \tag{6.2}$$

For a test prioritization method under study, $RP(d)$ can be determined analytically or it can be estimated by sampling. In this paper, we discuss how $RP(d)$ can be determined analytically for some simple test prioritization methods discussed in this paper. The analytical approach may reduce the cost of evaluation of test prioritization methods as opposed to evaluation by sampling. However, for many test prioritization methods, determining $RP(d)$ analytically may be very difficult. Therefore, estimation by sampling is used. In order to estimate $RP(d)$ by sampling, one needs to develop a simulator for a test prioritization method under study. The simulator generates prioritized test sequences according to the test prioritization method. For each test sequence, the position of the first failed test for each fault is determined. After a large number of test sequences are generated, $RP(d)$ for each fault is computed using formula 6.3:

$$RP(d) = \frac{\sum_{i=1}^{N} i \cdot R(i,d)}{W \cdot N} \tag{6.3}$$

where

W is the number of prioritized test sequences generated by a simulator during sampling,

$R(i,d)$ is a number of prioritized test sequences that are generated by a simulator for which $p_\theta(d) = i$, i.e., the first failed test $t \in TS(d)$ caused by fault d is in the i^{th} position.

In order to have a precise estimation of $RP(d)$, a large number of prioritized test sequences, W, needs to be generated by the simulator. The simulator needs to take into account the randomness factors that affect test prioritization for each test prioritization method under study.

6.1. Random prioritization

In random test prioritization, tests are ordered in random order. For a test suite of size N, there are $N!$ possible test sequences. $AP_R(d)$, the average position of the first failed test that detects d, for random test prioritization can be precisely computed by the following formula where $\mu = |TS(d)|$:

$$AP_R(d) = \frac{\mu \cdot \sum_{i=1}^{N-\mu+1} i \cdot \binom{N-\mu}{i-1} \cdot (i-1)!\cdot(N-i)!}{N \cdot N!} \quad (6.4)$$

The expression inside of the summation, except i, represents the number of random test sequences for which the first failed test caused by d is in position i. $RP_R(d)$, the average relative position of the first failed test that detects d, is computed from $AP_R(d)$ by Formula 6.2. For example, suppose that test suite $TS = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7\}$ detects two faults $D(TS) = \{d_1, d_2\}$ in a system and the following tests fail because of individual faults: $TS(d_1) = \{t_5\}$ and $TS(d_2) = \{t_5, t_7\}$. $RP_R(d_1) = 0.57$ and $RP_R(d_2) = 0.38$ are the average relative positions for random test prioritization for faults d_1 and d_2.

6.2. Selective prioritization

In selective test prioritization tests are divided into two categories: high priority tests and low priority tests. During prioritization, first all high priority tests are selected for execution and then low priority tests are selected. High priority tests are ordered randomly. Similarly, low priority tests are ordered randomly. The effectiveness of selective test prioritization depends on whether failed tests are high priority tests or not. More formally, let TS_H be a set of high priority tests and TS_L be a set of low priority tests. Let p, $p \le \mu$, be a number of failed tests in TS_H caused by fault d, where $\mu = |TS(d)|$. Let $AP_R(d, N, q)$ be the average position of the first failed test that detects fault d for random test prioritization for a test suite TS of size N where the test suite contains q failed tests caused by fault d (Formula 6.4).

The average position $AP_s(d)$ for the selective test prioritization is computed as follows:

Case I: $p \ge 1$ $\quad AP_s(d) = AP_R(d, K, p)$

Case II: $p = 0$ $\quad AP_s(d) = K + AP_R(d, N\text{-}K, \mu)$

where $K = |TS_H|$ and $N = |TS|$.

In Case I, it is assumed that TS_H contains at least one failed test caused by fault d. The average position for the selective test prioritization method is equivalent to the average position of random test prioritization for test suite TS_H with p failed tests, i.e., $AP_R(d, K, p)$. In Case II, it is assumed that TS_H does not contain any failed test caused by fault d, i.e., TS_L contains all μ failed tests. Executing all high priority tests (K tests) does not uncover fault d. Only when low priority tests are executed, fault d is detected. The average position in Case II is equivalent to the average position of random test prioritization for test suite TS_L with μ failed tests after all K high priority tests are executed, i.e., $K + AP_R(d, N\text{-}K, \mu)$. $RP_s(d)$, the average relative position of the first failed test that detects d, is computed from $AP_s(d)$ by Formula 6.2.

For example, suppose the following high and low priority tests are determined for test suite TS of Example 4: $TS_H = \{t_1, t_4, t_5, t_7\}$ and $TS_L = \{t_2, t_3, t_6\}$. $RP_s(d_1) = 0.36$ and $RP_s(d_2) = 0.24$ are the average relative positions for the selective test prioritization for faults d_1 and d_2.

6.3. Model dependence-based prioritization

For model dependence-based test prioritization, $RP(d)$, the average relative position of the first failed test that detects fault d, is determined by sampling. This sampling works as follows: During execution of the modified model on the test suite as presented in Section 4.2, we identify a set of tests associated with each interaction pattern and a set of failed tests. Then we randomly generate prioritized test sequences according to the model dependence-based test prioritization technique of Figure 11. For each prioritized test sequence, the position of the first failed test for each fault is determined. After a large number of prioritized test sequences, W, are generated, $RP(d)$ for each fault is computed using Formula 6.3.

Consider Example 4. Suppose that the following high priority selective tests are identified $TS_H = \{t_1, t_4, t_5, t_7\}$, and three interaction patterns are computed with the following distribution of tests among them: $TS(IP_1) = \{t_4, t_5\}$, $TS(IP_2) = \{t_1, t_4, t_7\}$, $TS(IP_3) = \{t_5, t_7\}$. $RP_s(d_1) = 0.31$ and $RP_s(d_2) = 0.21$ are the average relative positions for the model dependence-based test prioritization technique computed by the randomized estimation we outlined above.

6.4. Average rate of fault detection

In Section 3, the rate of fault detection $RFD(\theta)$ was discussed for a prioritized test sequence θ. Computation of $RFD(\theta)$ depends on a list $P(\theta) = <rp_\theta(d_1),\ldots, rp_\theta(d_L)>$ of positions of first failed tests in θ for all L faults in $D(TS)$. In this section, we introduce the *average rate of fault detection ARFD* for a test prioritization method. The average rate of fault detection is based on the average relative positions $RP(d)$. More formally, let $P = <RP(d_1),\ldots, RP(d_L)>$ be a list of the average relative positions of first failed tests determined for a test prioritization method for all faults in $D(TS)$. Let $F = <RP_1,\ldots,RP_q>$ be an ordered (in ascending order) sequence of all unique average relative positions from P, where $q \leq L$. The average rate of fault detection $ARFD$ for the test prioritization method is defined as a sequence of pairs (RP_i,fd_i), $ARFD = <(RP_1,fd_1), \ldots,(RP_q,fd_q)>$, where RP_i is an element of F, and fd_i represents the cumulative percentage of faults detected at position RP_i as discussed in Section 3 (Formula 3.2).

For example, suppose test suite $TS = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}\}$ consists of 10 tests that detect four faults $D(TS) = \{d_1, d_2, d_3, d_4\}$ in a system. The following tests fail because of individual faults: $TS(d_1) = \{t_5, t_7\}$, $TS(d_2) = \{t_3, t_7, t_9\}$, $TS(d_3) = \{t_6\}$ and $TS(d_4) = \{t_3, t_9\}$. $RP_R(d_1) = 0.37$, $RP_R(d_2) = 0.28$, $RP_R(d_3) = 0.55$ and $RP_R(d_4) = 0.37$ are the average relative positions for random test prioritization. Suppose that during the execution of the model on TS, the following high priority tests are identified for the selective test prioritization: $TS_H = \{t_1, t_3, t_4, t_6, t_7, t_9\}$. $RP_s(d_1) = 0.35$, $RP_s(d_2) = 0.18$, $RP_s(d_3) = 0.35$ and $RP_s(d_4) = 0.23$ are the average relative positions for the selective test prioritization. The average rates of fault detection for random test prioritization and the selective test prioritization are shown in Table 1 below:

Table 1. Average rates of fault detection

Random	*fd*: fraction of faults detected	0.25	0.75	1.0
	RP: Test suite fraction	0.28	0.37	0.55
Selective	*fd*: fraction of faults detected	0.25	0.5	1.0
	RP: Test suite fraction	0.18	0.23	0.35

In order to compare average rates of fault detection for different test prioritization methods, we may use a weighted average of the percentage of faults detected, *APFD*, as discussed in Section 3 (Formula 3.3). For two average rates of fault detection shown in Table 1, $APFD_R = 0.688$ and $APFD_s = 0.781$. In this example, selective test prioritization leads to a higher average rate of fault detection than random test prioritization.

7. Empirical Study

The goal of the empirical study is to compare the effectiveness of early fault detection of test prioritization methods presented in this paper: random test prioritization, selective test prioritization (Version I and II), and model dependence-based test prioritization. We used $RP(d)$, the average relative position of the first failed test that detects fault d, as the measure of the effectiveness of early fault detection. In the empirical study we concentrate on model faults.

Due to the unavailability of system models for real world commercial software, we used EFSM system models that are in the public domain for the empirical study. These EFSM models are: an ATM model [9, 17], a cruise control model [32], a fuel pump model [33], the Transfer Control Protocol-communication dialer (TCP) [12], and the Integrated Service Digital Network (ISDN) protocol [34]. The size of these system models ranges from 5 to 20 states and 20 to 87 transitions. Table 2 presents a summary of the models used in our study.

Table 2. System models used in the empirical study

Model Name	# of Transitions	# of States	# of Events	# of Variables	Size of LOC Implementation	Size of *TS*
ATM	25	8	10	8	609	834
Cruise Control	20	5	8	18	633	1000
Fuel Pumps	28	13	16	12	795	922
TCP-Dialer	49	17	10	30	1025	1000
ISDN	87	20	37	1	1416	900

LOC: Lines of code

For each model under study, we implemented the corresponding prototype system in the C language. The sizes of these implementations ranged from 609 to 1416 lines of source code. For each implementation, we created test suites using specification-based testing methods, i.e., equivalence class partitioning and boundary-value analysis. In addition, tests were derived based on model-based testing [12, 26, 27], i.e., transition coverage, and constrained-path coverage. The sizes of test suites range from 800 to 1000 tests. Each implementation was tested and debugged for its test suite until all tests are passed. As a result, these implementations were considered to be correct implementations in this empirical study.

In order to measure the effectiveness of early fault detection of different test prioritization methods, we created incorrect models (faulty models). We seeded one fault into a model at a time and then made appropriate changes to the corresponding system (implementation). Note that for this study, the oracle is the correct system implementation not the model. In this study, we used the mutation testing techniques to manually seed faults in system models. This approach is the one that is used to evaluate the adequacy of a test suite of a program [35], but we adapt it to seed faults in the model. Mutation testing techniques [56] introduce faults by inserting simple syntactic code changes into the program and check whether the test suite can detect these changes. In general, mutants represent likely faults the programmer could have made.

In a model, a fault may be seeded in $E(T)$, $C(T)$, and $A(T)$ of a transition T. For the purpose of this study, we considered seeding faults in an enabling condition and an action(s) only. We did not consider seeding faults in events, since faults of this type do not occur often. Notice that a fault seeded in a transition does not change the beginning and terminating states of a transition. A fault in an action associated with a transition is manually injected in assignment statements, output actions, and constants (operators used: +, -, changing a variable/constant, etc.). For a fault injected in the enabling condition associated with a transition, we assume the condition in the model is represented in the form of an expression followed by a relational operator followed by an expression i.e. exp_1 *op* exp_2. We injected faults in an expression or in a relational operator (some operators used: +, -, <, >, <=, >=, etc.). This type of fault seeding is similar to source-code based fault seeding. In addition, we used fault seeding that is specific only to models (model-specific fault seeding), e.g., addition/deletion a transition, splitting a transition into two or more transitions, or merging two or more transitions into one transition.

As mentioned, we seeded only one fault into the model at a time. For each model, we randomly selected a transition into which the faults are seeded. Transitions that are executed by each test are not selected for seeding faults. We manually created a set of faults (mutants) for each model. Some faults caused a large number of tests to fail. Such faults were rejected because there will not be a significant difference in the effectiveness of test prioritization methods when a large number of tests fail. We were interested in faults that cause a small number of tests to fail. Therefore, we selected only those faults for which the number of failed tests ranges from 1 to 10 tests. As a result, for each model, we have identified 10-12 seeded faults, i.e., for each model 10-12 incorrect model versions have been created. Table 3 shows a summary of selected faults seeded in each model, where the model name is in column 1 and the number of seeded faults in the model is in column 2. Column 3 shows the number of faults selected for which the number of failed tests ranges from 1 to 10. Column 4 shows the number of transitions in the model where the seeded fault causes 1-10 tests to fail.

For every fault seeded in the model, we executed the test suite *TS* on the corresponding system implementation to identify which tests passed and which tests failed. In addition, for each seeded fault in the model, the test suite *TS* is executed on the model to collect the execution traces *ETS*(*TS*). For each model with a seeded fault and corresponding implementation, we measured the average relative position (*RP*) of the first failed test for each test prioritization method under study.

Table 3. Summary of seeded faults in each model

Model name	# of Faults seeded	# of Faults selected	# of transitions involved in seeding faults
ATM	19	12	7
Cruise Control	17	10	6
Fuel Pumps	15	10	7
TCP	20	11	8
ISDN	18	11	8

The results of the empirical study are shown in Figure 12, which present the boxplots of the *RP* values for the four test prioritization methods, the five models, and all models in total. The presented results indicate that our model-based test prioritization approach may improve the effectiveness of test prioritization for Version II of selective test prioritization and the model dependence-based test prioritization. However, the results for Version I of selective test prioritization are mixed. In several cases, this test prioritization method performs much worse than random test prioritization. This is caused by the fact that monitoring only "added" transitions in the modified model may not be sufficient for effective test prioritization. On the other hand, Version II of selective test prioritization monitors also execution of "deleted" transitions that results in a signifi-

cant improvement in effectiveness of test prioritization. Although the model dependence-based test prioritization technique has a little more overhead compared to Version II of selective test prioritization, it may lead to further improvement in the effectiveness of test prioritization. This may be attributed to the fact that more information about the model behavior is collected that may improve the effectiveness of test prioritization.

Another goal of the empirical study is to compare the cost of each test prioritization method presented in this paper. We determined the cost of each model-based test prioritization method by measuring the time required to prioritize a test suite *TS* for each model under study. For random test prioritization, which is used as a baseline, this includes only the time required to prioritize a test suite *TS* for each model. For selective test prioritization (Versions I and II), this includes the time required to execute the test suite on the model and to prioritize a test suite *TS*. For model dependence-based test prioritization, this includes the time required to: (1) execute a test suite *TS* on the model, (2) compute interaction patterns test distribution (interaction patterns analysis), and (3) prioritize a test suite *TS* for each model.

Table 4 shows the time required to prioritize the test suite for each model and the average for all models for each test prioritization method. The experiment was performed on a PC workstation with Intel® Core™ Due CPU-2.00GHz processor running under Windows XP professional.

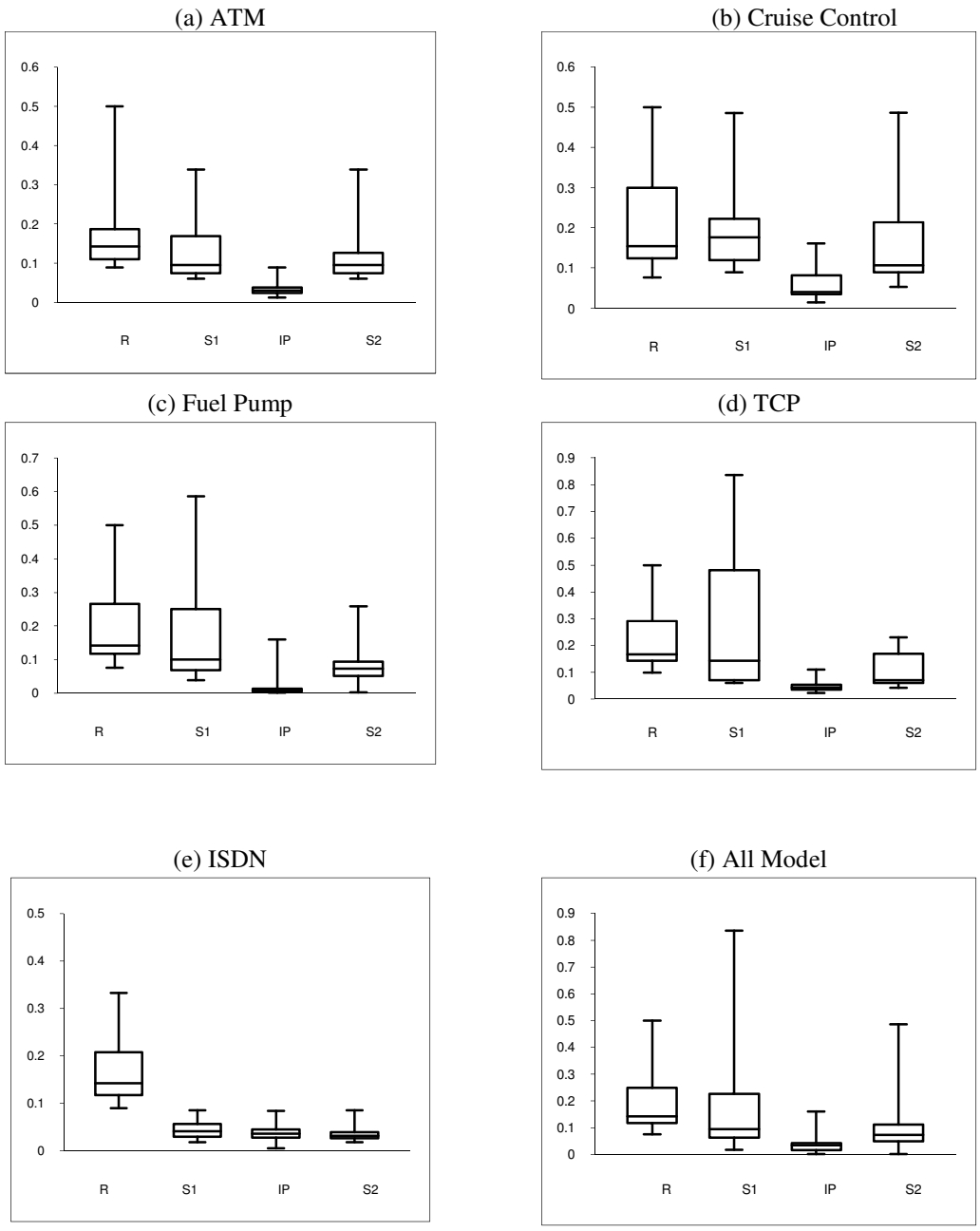
Table 4. The time required to prioritize a test suite by test prioritization methods in mseconds

	R	S1 & S2			IP		
Model Name	P1	ESa	P2	Total 1	ESb	P3	Total 2
ATM	15	328	31	359	391	1,893	2,266
Cruise Control	16	468	32	500	578	3,506	4,084
Fuel Pump	15	297	31	328	343	1,781	2,124
TCP	15	453	31	484	578	3,422	4,000
ISDN	15	500	31	531	625	265,016	265,641
Average for all Models	15	409	31	440	503	55,123	55,623

R: Random prioritization.
S1: Selective prioritization – Ver. I.
S2: Selective prioritization – Ver. II
IP: Model dependence prioritization

P1: The time required to prioritize *TS* for Random prioritization
P2: The time required to prioritize *TS* for S1 & S2
P3: The time required to prioritize *TS* for IP
ESa: Time of model execution for all tests in *TS* for S1& S2
ESb: Time of model execution for all tests in *TS* for IP
Total 1: Total time required to prioritize *TS* for S1&S2
Total 2: Total time required to prioritize *TS* for IP

We observed that the model dependence-based test prioritization method on average is more expensive than selective test prioritization (Versions I & II) and random prioritization. This is because model dependence-based test prioritization requires more analysis and collects more information from the model to prioritize a test suite *TS*. In addition, the model dependence-based test prioritization method computes the interaction pattern test distribution (interaction pattern analysis) to prioritize a test suite *TS* for each model. Notice that execution of models for all test suites is very fast, and it is comparable for both the selective and model dependence-based test prioritization methods. The selective prioritization is faster than the model dependence-based prioritization because the latter needs to compute the interaction pattern test distribution. However, for four models (and their test suites) the model dependence-based prioritization was able to prioritize tests within 4 seconds. For one model (*ISDN*) it took over 4 minutes to prioritize tests because this model is the largest among all models that were investigated in this study, but we believe that this time is still very reasonable as far as test prioritization is concerned.



R: Random prioritization.
S1: Selective prioritization – Version I
S2: Selective prioritization – Version II.
IP: Model dependence-based prioritization

Figure 12. Data ranges of *RPs* for (a) the ATM model, (b) the cruise control model, (c) the fuel pump model, (d) the TCP model, (e) the ISDN, and (f) for all models in total

Threats to validity are concerned with issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of an empirical study. We have identified the following threats to validity: In our empirical study, threats to validity could be due to low statistical power, resulting from the relatively small number of seeded faults per model. To limit the impact of this threat to validity, in Section 6 we introduced a concept of an average relative position of the first failed test, $RP(d)$, to compare different test prioritization methods. In addition, in our study we concentrated on types of faults for which only a small number of tests fail. We believe that these types of faults are very difficult to detect early. We were only interested in seeded faults that caused 1-10 tests to fail. Notice that when the number of failed tests is relatively large then there is no significant difference between prioritization methods. For example, when 20 tests fail because of defect d, in a test suite of 1,000 tests, $RP(d)$=0.04 for random prioritization. This clearly indicates that random approach can on average uncover fault d early when many tests fail. Therefore, we believe that using the concept of an average relative position of the first failed test and "hard to detect" faults to compare different prioritization methods should limit the impact of this threat. Another threat to validity is related to the use of mutation analysis in an empirical study, i.e., the types of faults seeded may not be representative of real faults. However, the existing literature suggests that faults seeded using mutation operators can be representative of real faults [57]. The fact that we seeded faults in different models (and in different transitions) with different characteristics leading to different types of faults should limit the impact of this threat. Moreover, for each EFSM model under study, we implemented the corresponding prototype system. This may be considered to be a potential threat to validity. However, this is due to the unavailability of the real commercial implementations. In addition, notice that EFSM models used in this study have been taken from independent sources and our implementations represent possible implementations of these systems (notice that, in total, five different implementations were developed). Therefore, we believe that this should limit the impact of this threat.

8. Related Work

Approaches to the optimization and efficient management of regression testing can be divided into three distinct activities: Selection, Minimization, and Prioritization of tests in a test suite. The goal of Test Suite Minimization is to identify tests that are redundant for a particular test adequacy criterion. These redundant tests may be avoided, thereby reducing the overall cost of regression testing activity.

In selection, the goal is to identify those tests that are redundant with respect to a particular set of changes made since the previous version of the system was tested. Minimization can be thought of as selection with an arbitrary set of changes. That is, a test can be removed by a minimization algorithm which deems it redundant if it is already covered by some other test. A test that is removed by selection is only redundant for the particular set of recent changes in question and, therefore, the removal is only safe with respect to these changes. Regression Test Selection seeks to identify that subset of tests that are relevant to the changes; their execution is required for safe regression testing.

Finally, Regression Test Prioritization seeks to order tests so that the maximum benefit is achieved from regression testing should it be stopped at some arbitrary point. The goal is to order the tests in rank order so that progress towards test goal fulfilment is achieved at the highest rate. In Regression Test Prioritization no test is ever removed completely from consideration. Rather, the tests are ordered so that, resources permitting, all will be executed, but when resources do not permit, there is minimal loss arising from the forced decision to ignore those that remain unexecuted.

At the implementation level, there has been much work on all three regression test optimization problems, leading to several recent surveys [51, 52]. However, there has been comparatively little attention paid to the problems of regression testing at the model level, which is surprising considering the widespread interest in Model Driven Development.

This paper concerns the problem of Regression Test Prioritization at the model level, guided by dependence analysis. There is evidence to suggest that interest in regression test prioritization is increasing more rapidly than interest in selection and minimization [51]. However, this growing interest remains largely confined to the implementation level. By contrast, of the small fraction of the literature that concerns model based regression testing, only a further small fraction addresses model based regression test prioritization. Most work on regression testing at the model level is concerned with the problem of test selection.

Briand et al. [47] introduce RTST, a Regression Test Selection Tool. More recently, this work has been developed and extended into a comprehensive study of the regression test selection problem for the UML [46]. This work can be viewed as complementary to ours in two ways. First, it focuses on aspects of the UML other than state based models, whereas we are concerned with state based modeling notations. Second, Briand et al. are concerned with Regression Test Selection, whereas our work is concerned with prioritization.

Orso et al. [39] present two regression test selection techniques for component based systems, using component metadata to support the identification of selective subsets of tests to be used in efficient re-testing strategies. Their approach is illustrated in terms of component based systems specified as UML statecharts and is evaluated using two real world java systems. Their approach is applied to state based models, like ours. However, Orso et al. focus on the use of component metadata for selection, whereas our approach is based on a dependence analysis for prioritization.

Farooq et al. [44] also recently presented a regression test selection approach based on changes identified in both the statecharts and class diagrams of the UML, classifying tests into the three classes: obsolete, re-usable and re-testable. They illustrate the application of their approach with a student enrolment state machine. Like other work on model level regression testing, this work is concerned with selection rather than prioritization.

Several authors have considered the application of regression testing to entire UML models, rather than to statecharts, which merely denote a single submodel within the overall UML system. This work has focused on all aspects of the UML and therefore, like our work, this previous work involves state based models (since these are part of the UML). However, once again, the focus of this work is on the problem of test selection rather than test prioritization.

Wu and Offutt [41] present retesting strategies for UML, based on a differencing approach that identifies the modified parts of the new model. Similar to our work, Wu and Offutt also incorporate the tracing of data dependence based changes in their work.

Pilskalns et al. [40] present a safe and efficient re-test strategy based on UML model level changes, illustrating their approach with a case study of an open source system called Batik. Their work is concerned with regression testing the model itself, rather than to using the model to select tests to be applied to the implementation.

Other authors have also considered regression testing for non-state-based UML models (such as Activity diagrams [48] and Class diagrams [49]) and have considered regression testing for other non-UML modeling approaches [50]. Deng et al. [45] present an overview of an approach to whole life-cycle model based testing and maintenance, briefly mentioning the issues of selective regression testing of models.

Fraser and Wotawa [42] adapt Rothermel et al's APFD metric for use with tests generated from model checkers, illustrating with the well-known cruise control case study [43]. Unlike the previously mentioned work, this work is concerned with prioritization.

Korel et al. presented methods of test prioritization based on the state-based model of the system under test [17]. These methods assume that the modifications are made both on the system under test and its model. The existing test suite is executed on the system model and information about this execution is used to prioritize tests. This paper is an extension to Korel et al's work on the model-based test prioritization method [17]. In this paper we have extended the empirical study and introduced formal definitions of test prioritization methods. In addition, we presented an analytical framework for evaluation of test prioritization methods. This framework may reduce the cost of evaluation as compared to the framework that is based on observation.

Korel et al. extended their research on model-based prioritization for a class of modifications for which models are not modified (only the source code is modified) [29, 36]. Several model-based test prioritization heuristics were introduced. Their major motivations for these heuristics were simplicity and effectiveness in early fault detection. The results of their study suggest that system models may improve the effectiveness of test prioritization with respect to early fault detection. This extension is semi-automated as opposed to the fully automated model-based prioritization presented in this paper because of a difficulty in automated identification of a mapping between source-code changes to the corresponding model elements.

Korel et al. [18] proposed simple model-based test prioritization heuristics. The major stress was on simplicity. These simple heuristics have shown promise when a large number of transitions is modified. However, for small modifications the performance of these heuristics can be equivalent to the selective prioritization -Version II.

It should be noted that code-based test prioritization methods [15, 16, 51, 53] are dependent on information relating the tests of the test suite to various elements of a system's code of the original system (before modification). For example, a particular code-based technique can utilize information about the number of statements executed by a test. The system code then executes the test suite and information about executed code elements is collected for each test. Different types of test prioritization heuristics can then be applied to analyze the information collected and prioritize the tests accordingly. The major weakness of code-based prioritization is that only the original code is used, not the modified version. In addition, the instrumented original system needs to be executed in order to collect information for prioritization. As mentioned earlier, this process may be very expensive for many types of systems. On the other hand, model-based prioritization presented in this

paper uses the modified model of the system for prioritization. In addition, execution of the modified model for the whole test suite is very fast as compared to execution of the whole system.

9. Conclusions

In this paper, we have presented model-based test prioritization methods in which the information about the system model and its behavior is used to prioritize the test suite for system retesting. In addition, we presented an analytical framework for comparison of test prioritization methods with respect to the effectiveness of early fault detection. In an empirical study, we investigated the test prioritization methods we introduced with respect to their effectiveness of early fault detection. The results from the empirical study are promising and suggest that using system models may improve the effectiveness of test prioritization.

In future research, we plan to perform an experimental study on larger models and systems to have better understanding of the advantages and limitations of model-based test prioritization. In addition, we plan to perform an experimental study in which we will investigate the effectiveness of model-based test prioritization for faults in implementations of model changes in the system (these are code-based faults related to implementation of model changes). An initial study [36] comparing the model-based and source-code based prioritization indicates that model-based prioritization may be an attractive alternative to the source-code based prioritization.

The model-based test prioritization method we introduced in this paper is only one way that tests can be prioritized based on interaction patterns. One may develop other algorithms to prioritize tests based on interaction patterns, e.g., tests that "cover" larger numbers of IPs are assigned a higher priority.

10. References

[1] Y. Chen, D. Rosenblum, K. Vo, *Testube: A System for Selective Regression Testing*, Proc. IEEE International Conference on Software Engineering, pp. 211-220, 1994.

[2] R. Gupta, M. Harrold, M. Soffa, *An Approach to Regression Testing Using Slices*, Proc. IEEE International Conference on Software Maintenance, pp. 299-308, 1992.

[3] B. Korel, A. Al-Yami, *Automated Regression Test Generation*, Proc. ACM International Symposium on Software Testing and Analysis, pp. 143-152, 1998.

[4] G. Rothermel, M. Harrold, *Selecting Tests and Identifying Test Coverage Requirements for Modified Software*, Proc. IEEE International Conference on Software Maintenance, pp. 358-367, 1994.

[5] G. Rothermel, M. Harrold, *A Safe, Efficient Regression Test Selection Technique*, ACM Transactions on Software Engineering & Methodology, 6(2), pp. 173-210, 1997.

[6] B. Sherlund, B. Korel, *Modification Oriented Software Testing*, Proc. Quality Week, pp. 1-17, 1991.

[7] L. White, Test Manager: *A Regression Testing Tool*, Proc. IEEE International Conference on Software Maintenance, pp. 338-347, 1993.

[8] S. Beydeda, V. Gruhn, *An Integrated Testing Technique for Component-Based Software,* Proc. IEEE International Conference on Computer Systems and Applications, pp. 328 –334, 2001.

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

[10] J. Loyall, S. Mathisen, P. Hurley, J. Williamson, *Automated Maintenance of Avionics Software*, Proc. IEEE Aerospace and Electronics Conference, pp.508-514, 1993.

[11] W. Tsai, X. Bai, R. Paul, L. Yu, *Scenario-Based Functional Regression Testing*, Proc. IEEE International Computer Software and Applications Conference, pp. 496-501, 2001.

[12] Y. Chen, R. Probert, H. Ural. *Model-Based Regression Test Suite generation Using Dependence Analysis.* Proc. the 3rd ACM Workshop on Advances in Model Based Testing (A-MOST), pp. 62-67, 2007.

[13] Chen Y, Probert RL, Ural H. *Regression test suite reduction using extended dependence analysis*. Proc. International Workshop on Software Quality Assurance (SOQUA 2007). ACM: New York NY, 2007; 62–69.

[14] G. Rothermel, R. Untch, C. Chu, M. Harrold, *Test Case Prioritization: An Empirical Study*, Proc. IEEE International Conference on Software Maintenance, pp. 179-188, 1999.

[15] G. Rothermel, R. Untch, M. Harrold, *Prioritizing Test Cases For Regression Testing*, IEEE Transactions on Software Engineering, vol. 27, No. 10, pp. 929-948, 2001.

[16] W. Wong, J. Horgan, S. London, H. Agrawal, "*A Study of Effective Regression Testing in Practice*," Proc. International Symposium on Software Reliability Eng., pp. 230-238, 1997.

[17] B. Korel, L. Tahat, M. Harman. *Test Prioritization Using System Models*. Proc. IEEE International Conference on Software Maintenance. Budapest, Hungary, pp. 559-568, 2005

[18] B. Korel, G. Koutsogiannakis**,** L. Tahat. *Prioritization Algorithms for Regression Testing in Model Based Systems*. Proc. the 3rd ACM Workshop on Advances in Model Based Testing (A-MOST), 2007.

[19] D. Harel, M. Politi. *Modeling reactive systems with statecharts: the STATEMATE approach*. McGraw-Hill, New York, 1998.

[20] K. Cheng, A. Krishnakumar, *Automatic Functional Test Generation Using The Extended Finite State Machine Model*, Proc. ACM/IEEE Design Automation Conf., pp. 86-91, 1993.

[21] ITU-T. *Recommendation Z.100 Specification and description language (SDL).* International Telecommunications Union, Geneva, Switzerland, 1999.

[22] J. Dick, A. Faivre, *Automating the Generation and Sequencing of Test Case from Model-Based Specification*, Proc. International Symposium on Formal Methods, pp. 268-284, 1992.

[23] R. Dssouli, K. Saleh, E. Aboulhamid, A. En-Nouaary, C. Bourhfir, *Test Development For Communication Protocols: Towards Automation*, Computer Networks, 31, pp.1835-1872, 1999.

[24] H. Ural, K. Saleh, A.W. Williams. *Test generation based on control and data dependencies within system specifications in SDL*, Computer Communications 2000; **23**(7): 609–627.

[25] A. Y. Duale, M. U. Uyar. *A method enabling feasible conformance test sequence generation for EFSM models*. IEEE Transactions on Computers, 53(5):614–627, 2004.

[26] B. Vaysburg, L. Tahat, B. Korel, *Dependence Analysis in Reduction of Requirement Based Test Suites*, Proc. ACM International Symposium on Software Testing and Analysis, pp. 107-111, 2002.

[27] L. Tahat, B. Vaysburg, B. Korel. *Requirement-Based Automating Black-Box Test Generation.* Proc. the 25th Annual IEEE International Computer Software and Applications Conference, pp: 489-496, 2001.

[28] B. Korel, S. Inderdeep, L. Tahat, B. Vaysburg. *Slicing of State-Based Models.* Proc. IEEE International Conference on Software Maintenance, ICSM 2003. Amsterdam, Netherlands. (September 2003): 34-43.

[29] B. Korel, G. Koutsogiannakis, L. Tahat. *Model-Based Test Prioritization Heuristic Methods and Their Evaluation*. IEEE International Conference on Software Maintenance. ICSM 2008, Beijing, China, September. 2008, pp: 247-256.

[30] J. Laski, and B. Korel. *A Data Flow Oriented Program Testing Strategy.* IEEE Transactions on Software Engineering. 9.3 (May 1983): 347-354.

[31] J. Ferrante, K. Ottenstein and J. Warren. *The Program Dependence Graph and its Use in Optimization.* ACM Transactions on Programming Languages and Systems. 9.5 (1987): 319-349.

[32] Stevens, W. *TCP/IP Illustration, volume I, the Protocol*. Addison Wesley Reading, MA. 2001.

[33] Mueller C., Ph.D. Thesis. *Automated Interface Propping Applied to Cost Component Evaluation*. Illinois Institute of Technology. Chicago, IL. 2003.

[34] ITU-T Recommendation Standard, Q.931, *Digital Subscriber Signaling System No. 1 – Network Layer*, ISDN User Network Interface Layer 3 Specifications for Basic Call Control. May 1998.

[35] A. J. Offutt, A. Lee, G. Rothermel, R. Untch, C. Zapf. *An experimental determination of sufficient mutation operators*. ACM Transactions on Software Engineering and Methodology. 5.2 (1996):99-118.

[36] B. Korel, G. Koutsogiannakis. *Experimental Comparison of Code-Based and Model-Based Test Prioritizatio*n, 5th Workshop on Advances in Model Based Testing, A-MOST 2009, Denver, April 2009, IEEE digital library."

[37] K. Androutsopoulos, D. Clark, M. Harman, Z. Li, and L. Tratt. *Control Dependence for Extended Finite State Machines.*

Proc. Fundamental Approaches to Software Engineering (FASE '09) , York, UK, 22nd-29th March, 2009. Springer LNCS volume 5503, pages 216-230.

[38] K. Androutsopoulos, N. Gold, M. Harman, Z. Li, and L. Tratt. *A Theoretical and Empirical Study of EFSM Dependence*. Proc. 25th IEEE International Conference on Software Maintenance (ICSM 2009), Edmonton, Alberta, Canada, 23rd-26th September 2009. To appear.

[39] Alessandro Orso, Hyunsook Do, Gregg Rothermel, Mary Jean Harrold, David S. Rosenblum. *Using component metadata to regression test component-based software*. Software Testing Verification and Reliability (STVR). 17(2):61-94, June 2007.

[40] Orest Pilskalns, Gunay Uyan, Anneliese Andrews. *Regression Testing UML Designs*. 22nd IEEE International Conference on Software Maintenance, pages: 254 – 264, 2006

[41] Ye Wu and Jeff. Offutt. *Maintaining evolving component-based software with UML*. Seventh European Conference on Software Maintenance and Reengineering(CSMR), pages 133- 142, 2003.

[42] Gordon Fraser and Franz Wotawa. *Test-case prioritization with model-checkers*. 25th IASTED International Multi-Conference: Software Engineering, Innsbruck, Austria, pages 267-272, 2007

[43] J. Kirby. *Example NRL/SCR Software Requirements for an Automobile Cruise Control and Monitoring System*. Technical Report TR-87-07, Wang Institute of Graduate Studies, 1987.

[44] Qurat-ul-ann Farooq, Muhammad Zohaib Z. Iqbal, Zafar I Malik andAamer Nadeem. *An approach for selective state machine based regression testing.* Proceedings of the 3rd International ACM Workshop on Advances in Model-Based Testing, London, United Kingdom, pages: 44 – 52, 2007

[45] D. Deng, P. C. -Y. Sheu, T. Wang and A. K. Onoma. *Model-Based Testing and Maintenance*. Proceedings of the IEEE Sixth International Symposium on Multimedia Software Engineering (ISMSE), pages: 278 – 285, 2004

[46] Lionel C. Briand, Yvan Labiche and S. He. *Automating regression test selection based on UML designs*. Information and Software Technology. 51(1):16-30, 2009

[47] Lionel C. Briand, Yvan Labiche and G. Soccar. *Automating impact analysis and regression test selection based on UML designs*. International Conference on Software Maintenance (ICSM), pages 252-261, 2002.

[48] Y. Chen, R.L. Probert and D.P. Sims. *Specification based regression test selection with risk analysis*. Proceedings of Conference of the Center for Advance Studies on Collaborative Research, 2002.

[49] Y. Le Traon, T. Jéron, J.-M. Jézéquel and P. Morel. *Efficient object-oriented integration and regression testing*. IEEE Transactions on Reliability 49(1):12-25, 2000.

[50] H. Muccini, M.S. Dias and D.J. Richardson. *Reasoning about software architecture-based regression testing through a case study*. International Computer Software and Applications Conference (COMPSAC), pp. 189-195, 2005.

[51] Shin Yoo and Mark Harman. *Regression Testing Minimisation, Selection and Prioritisation - A Survey*. Department of Computer Science, King's College London, Technical Report: TR-09-09, October, 2009. Also to appear in revised form in Software Testing Verification and Reliability (STVR).

[52] Mary Jean Harrold and Alessandro Orso. *Retesting software during development and maintenance*. Frontiers of Software Maintenance (FoSM 2008), International Conference on Software Maintenance. pp 99-108. Beijing, China, 2008.

[53] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. IEEE Transactions on Software Engineering, vol: 28, pp. 159:182, 2002.

[54] G. Fraser and F. Wotawa. *Ordering coverage goals in model checker based testing*. ICST, pp. 31-40, 2008.

[55] J. A. Jones and M. J. Harrold. *Test-suite reduction and prioritization for modified condition/decision coverage*. IEEE Transactions on Software Engineering, vol 29, pp. 92-101, 2003.

[56] Yue Jia and Mark Harman. *An Analysis and Survey of the Development of Mutation Testing*. IEEE Transactions on Software Engineering. To appear, 2010.

[57] Hyunsook Do and Gregg Rothermel. On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques, IEEE Transactions on Software Engineering, Vol 32, issue 9, pp:733-752, September 2006.