Analyzing the Impact of Protocol Changes on Tests

Mahadevan Subramaniam[1] and Zoltán Pap[2]

[1] Computer Science Department,
University of Nebraska at Omaha,
Omaha, NE 68182, USA
msubramaniam@mail.unomaha.edu

[2] Department of Telecommunications and Media Informatics – ETIK,
Budapest University of Technology and Economics,
Magyar tudósok körútja 2, H-1117, Budapest, Hungary
pap@tmit.bme.hu

Abstract. Protocols governing communication among system components evolve during design and maintenance and need to be re-tested. For faster testing turnaround time, it is important that the consistency of the testing infrastructure with the protocol be preserved across changes. In this paper, we propose a state exploration based approach to identify the impacts of protocol changes on a given set of protocol tests. Protocols are modeled as a network of communicating finite state machines exchanging messages over bounded queues. Each machine denotes the behavior of an individual protocol component (*controller*). A protocol test is modeled as a sequence of inputs from the environment to the protocol controllers in an execution starting from a stable protocol state. A notion of *consistency* of a test relative to a protocol is introduced. Conditions under which a protocol change requires changing a test to preserve the consistency of the test are identified. Changes consisting of multiple atomic updates are analyzed to remove redundancies and their impact on tests is studied. A by-product of the proposed approach is a classification of tests based on how they are impacted by protocol changes, which can help users in regression test selection.

Keywords: Changes, evolution, protocol, communicating finite state machines, test consistency.

1 Introduction

Testing of protocols has been extensively studied [1,2,3,4] due to the central role played by protocols in both software and hardware. Typically, a protocol may undergo several changes during the system design and maintenance phases and may need to be repeatedly re-tested. Generating and running tests in response to every protocol change is time consuming and may also not lead to good cumulative coverage of protocol functionality across the changes. To re-test a changed protocol it is important to analyze effects of changes on existing tests.

Changes to a protocol may affect existing tests in several ways. Several existing tests may be re-usable with changed protocol. However, only some of these tests may exercise changed behaviors (in addition to original ones) whereas the rest may exercise only original behaviors. Determining whether a re-usable test exercises a changed behavior may be useful in generating new tests. Some existing tests may also become unusable for certain changes. It may be possible to use some of these tests to test the changed protocol after patching while others may have to be simply discarded. Given that a large number of tests are typically needed to validate even simple protocols, manually determining impacts of changes on the existing tests is tedious and error prone. Determining such impact may not be easy even for a single test since this may require considering all executions that can happen in the test, which may be numerous.

In this paper, we propose a white-box[1] approach to automatically determine the impact of protocol changes on protocol tests. We model protocols as a network of interacting communicating finite state machines (*CFSM*s) with bounded queues [5, 6, 7]. The transition relation of each *CFSM* describes the behavior of an individual protocol controller. The *CFSM*s interact with each other by message exchanges over the queues. They also interact with an external environment by exchanging messages over the queues to/from the environment to the controllers. We consider protocol changes at the transition level. Protocol changes add/delete/replace one or more transitions from one or more protocol controllers and are explicitly represented as change specifications [8].

We formalize a protocol test as *a protocol stable state along with a set of environment inputs*. The environment inputs are used in the executions starting from the associated protocol stable state. A simple notion of *consistency* of a test relative to a protocol is proposed. Informally, a test is consistent with a protocol if it exactly specifies the environment inputs needed in each execution that can happen in the test starting from the protocol stable state. At least one execution must happen in each consistent test. Further, since a consistent test exactly specifies the inputs used in their executions each execution can happen in at most one consistent test.

Changes can potentially produce protocols whose executions cannot happen in any consistent test. We consider such changes uninteresting. A notion of *consistently testable* protocol is introduced. In a consistent testable protocol, it is possible to design a consistent test for each execution such that the execution happens in the consistent test. Consistent testability ensures that each protocol execution can happen in exactly one consistent test.

We constrain changes to preserve consistent testability of protocols. We characterize the impact of a change on an existing test in terms of whether the change preserves the consistency of the test. To check whether a change preserves the consistency of an existing test, it is enough to just find one execution

[1] A testing environment consists of a set of protocol tests, along with a set of runtime monitors derived from protocol specifications and/or correctness properties. The runtime monitors observe sequences of global states during execution and check if each protocol execution in each protocol test conforms to the specification.

of the changed protocol that can happen in the test such that the execution uses all the inputs specified in the test. Otherwise, the test is *inconsistent*. This is because for a consistent testable protocol, if a test is consistent for a single execution that can happen in the test then the test must be consistent for all executions that can happen in the test since otherwise, that execution cannot happen in any consistent test. Consistent testability makes it easier to check whether consistency of a test is preserved by a change since it is enough to check this for just one execution of the changed protocol. For certain changes it may not be necessary to analyze even a single execution. For instance, see the discussion for changes that add transitions below.

Tests whose consistency is preserved by a change may either be *independent* of the change or they may be *re-usable*. Informally, a test is *independent* of a change if none of the executions that can happen in the test are affected by the change. Independent tests may be discarded to minimize the testing effort across changes. All other tests whose consistency is preserved by the change are re-usable tests. Re-usable tests exercise the changed behavior and can be used without any modifications to test the changed protocol.

State exploration is used to determine whether tests whose consistency is preserved by a change are independent. This is done by computing an *interaction context* for a given protocol transition. Interaction context of a given protocol transition is the set of all consistent tests such that the transition appears in some execution of each test. Informally, if a test whose consistency is preserved by a change does not belong to the interaction context of transitions being changed, then it is not affected by the change and is independent. If the test belongs to the interaction contexts of a new transition introduced by a change then it is re-usable.

If a test is made inconsistent by a change and the protocol stable state of the test is a stable state of the changed protocol then the test is patched by changing its inputs to be the external inputs used in the executions starting from that stable state of the changed protocol. A patched test is generated for each such execution with distinct environment inputs. Inconsistent tests whose protocol stable state is not a stable state of the changed protocol are not patchable and hence discarded. Interaction contexts of protocol transitions are used to patch an inconsistent test whenever possible to do so. We also describe a state exploration based procedure based on interaction contexts for checking consistent testability.

We first consider single transition changes and show that a change that adds a new transition t'_i to a controller, preserves the consistency of all existing tests. This is because such a change preserves all the executions of the original protocol in addition to preserving consistent testability. An existing consistent test γ is re-usable with respect to such a change if it belongs to the interaction context of the newly added transition t'_i, which is computed over the changed protocol. Otherwise, the test γ is independent of the change.

Consider a change replacing a transition t_i by another transition t'_i in the same controller. To determine the impact of such a change on an existing consistent test γ, the interaction context of transition t'_i and the interaction contexts of the transitions t_js in the original protocol that are distinct from t_i are computed. All the contexts are computed by state over the changed protocol. The test γ is re-usable if it belongs to the interaction context of t'_i and γ is independent if it does not belong to the context of t'_i but belongs to the context of some t_j. In this case, an execution of the original protocol without the replaced transition t_i can happen in the test γ. Otherwise, the test γ is inconsistent. The test is patched, as described above, if the protocol stable state of γ belongs to any of the interaction contexts of t_j's. The test γ is discarded otherwise.

It is in practice inevitable to deal with complex changes consisting of multiple atomic updates. Our method of analyzing the impact of multiple changes on tests is based on examining the effect of the comprised individual add / delete / replacement changes. However, the approach developed for the single transition change cases can not be used as it is, some modifications are needed to handle certain types of independence of the different atomic changes. If all the atomic updates are only additions then the effect on a test can be determined by considering the individual interaction contexts independently. However, for replacements and deletions effects on a test can not be determined independently since the inconsistency of a test whose executions contain a group of replaced protocol transitions may not be detected. To avoid this problem, the combined interaction contexts of the protocol transitions being replaced is used to determine the impact on a test.

As multiple changes are usually defined during the development process in an ad hoc manner, they are often confusing and redundant. The proposed approach also includes a novel approach to reduce complex changes and create the shortest– or in an other sense optimal – sequence of update rules inducing the required modifications.

The paper is organized as follows. After a brief discussion on the related literature in Section 2, we introduce the background of the current research including our model of protocols in Section 3. Protocol tests and the notion of test consistency are introduced in Section 4. Section 5 presents a framework for analyzing the impact of transition changes on tests and studies the effect of single changes. Section 6 describes handling and impact of multiple transition changes. Section 7 concludes the paper.

2 Related Work

Both test generation and software change impact analysis have been active areas of research in the past. Testing is an indispensable phase of a system development lifecycle, yet it has turned out to be a difficult task for the increasingly complex protocols. Because of their practical importance, significant effort has been devoted to the development of automatic test generation methods for protocols to overcome the inefficiency of manual testing [1,2,3,4]. Furthermore, with the constant increase in complexity, protocol design and implementation has become an increasingly evolutionary and iterative process. Motivated by this trend there have been some studies investigating changes and their impacts [9,10,11].

Several researchers have focused earlier on evaluating the effect of system changes on tests [10,11]. However, surprisingly, not much attention has been paid to evolution of protocols and the impact of protocol changes on tests. To the best of our knowledge, this is perhaps the first attempt towards analyzing the impact of CFSM-based protocol changes on protocol tests. Further, most of the earlier work on change impact analysis has been based on conservative static analysis and hence does not provide precise answers. In contrast, the proposed approach is based on formal approach based on state exploration that allows for accurate analysis of change impacts and also enables us to provide more useful feedback by synthesizing new tests that are guaranteed to be consistent with the changes.

The approach proposed in this paper builds on our earlier work in [8, 12]. In [8] we have developed a systematic approach to specify and consistently incorporate changes to CFSM-based protocols. This approach has been applied to several protocols including industrial-strength cache coherence protocols. In [12] we showed how the impact of consistent protocol changes on runtime monitors can be automatically evaluated and new monitors can be synthesized. Typically, in white-box testing environments, monitors are used in conjunction with tests to observe and flag errors in protocol executions. This paper shows how change impacts can be analyzed for tests and in this sense fills an important gap for evaluating impact of protocol changes on the testing infrastructure.

3 Preliminaries

A protocol $P = (P_1, \cdots, P_n, \epsilon)$ is a network of *CFSM*s [5,6], where each P_i is a protocol controller and ϵ is the environment. Communication among P_i's and ϵ is achieved by exchanging messages over a network of bounded queues.

Each protocol controller P_i is a *CFSM*, a 4-tuple, $(S_i,\ I_i,\ M_i,\ T_i)$ where S_i is a finite set of states, $I_i \in S_i$ is an initial state, M_i are the messages sent and received by P_i, and $T_i \subseteq M_i \times S_i \mapsto S_i \times 2^{M_i}$ is a deterministic transition relation. Each set $M_i = M_{i,j} \cup M_{j,i}$, for $\{i, j\} \in \{1, \cdots, n, \epsilon\}$, $i \neq j$, where $M_{i,j}$ is the set of messages sent by the controller P_i to the controller P_j and $M_{j,i}$ is the set of messages received by controller P_i from P_j. Each message in $M_{i,j}$ is written as $m_k(i, j)$ where m_k is the message label and (i, j) denotes the message queue from controller P_i to controller P_j. The set $M_{i,i}$ is empty for all i; we assume that no P_i exchanges messages with itself.

Each protocol transition t_i in T_i is of the form $m_0(j, i),\ s_i \mapsto s'_i,\ \{m_1(i, k_1), \cdots, m_l(i, k_l)\}$ where m_0 is the input message, s_i and s'_i are the input and output states respectively, and $m_1, \cdots, m_l$ are the output messages to the controllers $k_1 \cdots k_l$, $1 \leq l \leq n$, $l \neq i$. Let $S = \bigcup_{1 \leq i \leq n} S_i$ be the set of protocol states.

A *global protocol state*, $g = \langle u,\ v\rangle$ is a pair where u is an n-tuple of the individual controller states and v represents the messages, if any, in the protocol message queues. Let $u[j]$ stand for the state of the controller P_j in u and $v[i,\ j]$ stand for the messages in the queue $(i,\ j)$ in v. For ease of exposition, we will only show non-empty queues in v in a global protocol state. We write $v = \langle\rangle$ when we all queues are empty in the state g.

The *initial global protocol state* $g_0 = \langle u_0, v_0 \rangle$ is a pair where $u_0[i] \in I_i$ for all i, and $v_0[i, j] = \langle\rangle$ if $i \neq \epsilon$ and $j \neq \epsilon$. So, the only non-empty queues in an initial global protocol state are those containing messages from/to the environment ϵ.

A *stable global protocol state* is a global protocol state $g = \langle u, \langle\rangle\rangle$ where all the queues are empty. We call the elements of S in u in a stable global protocol state as *stable controller states.* Note that the initial global protocol state is a stable global protocol state but not vice versa.

A transition t_i: $m_0(j, i)$, $s_i \mapsto s'_i$, $\{m_1(i, k_1), \cdots, m_l(i, k_l)\}$ is *enabled* in global state $g = \langle u, v \rangle$ if $u[i]$ is the input state s_i and the top of queue $v[j, i]$ is the input message $m_0(j, i)$, and all the output messages $m_1(i, k_1)$, $\cdots$, $m_l(i, k_l)$ can be enqueued (there is space to do so) in the corresponding queues in v.

An *execution step*, $g \rightarrow^{t_i} g'$ using the transition t_i enabled in the global protocol state g produces a global protocol state g' with the controller P_i in the state s'_i; other controller states are unchanged. The input message $m_0(j, i)$ is dequeued from the queue $v[j, i]$ and the output messages $m_1(i, k_1)$, $\cdots$, $m_l(i, k_l)$ are enqueued to the appropriate queues in v to produce g'. If t_i is enabled in g then we say that the global protocol state g' is enabled by t_i. The state $g' = g$ if t_i is not enabled in g.

An *executable* path of P, $r_1 = g_0 \rightarrow^{t_0} g_1 \cdots \rightarrow^{t_k} g_{k+1}$. We will simply represent the path r_1 by a sequence of transitions $[t_0, \cdots, t_{n-1}]$.

A protocol *run* is an executable path starting and ending in stable global protocol states. Each protocol run starts from a stable global protocol state by processing the environment input messages leading to further message exchanges until a stable global protocol state is reached.

Protocol P is *consistent* if and only if *i*) no two protocol transitions t_i and t_j have the same input state and same input message, *ii*) every t_i appears in a protocol run, and *iii*) every executable path from a stable global state reaching a state enabling a transition t_i is a prefix of a protocol run.

Henceforth, we will assume that protocols are consistent. Note that in our model, protocol controllers are not input-enabled since they communicate with each other by exchanging messages over bounded queues. Our model thus more closely relates to the process algebraic than the I/O model of concurrency.

Changes to protocols are specified as a finite set of rules, *update rules* (*ur*), that may add, delete, or replace one or more transitions in one or more controllers. ur_j: $t_i \Rightarrow t'_i$ replaces a transition t_i by a new transition t'_i, ur_j: t_i deletes a transition t_i, and ur_j: t'_i adds a transition t'_i in controller P_j. We consider changes with both single as well multiple rules.

It is assumed that all changes preserve the consistency of a protocol.

4 Tests and Consistently Testable Protocols

We primarily focus on white-box testing in this paper [13]. In our testing model, a protocol implementation under test is stimulated by the environment ϵ by inputting messages. The implementation processes the messages one at a time, starting from a stable global protocol state resulting in a protocol test run.

To ensure conformance of the implementation, a runtime monitor [14, 15] is added to the implementation. The monitor observes certain global protocol states in each run in a test. Starting from a conforming monitor state, the monitor transitions with each observed global protocol state ending in a conforming monitor state if the test run is behaviorally conforming to the protocol specification; the monitor ends in an error state for non-conforming test runs. The global protocol states observed by the monitor in a test run depend on the conformance properties being checked.

A protocol *test* $\gamma = \langle u,\, v\rangle$ is a pair where u is an n-tuple of controller stable states and v is an n-tuple of queues containing input messages from environment ϵ to the n individual protocol controllers. For ease of exposition, we will only show non-empty queues in v while describing a test γ.

A *test run* of the test γ is a protocol run starting from the stable global protocol state $\langle u, \langle\rangle\rangle$ that processes all the environment input messages appearing in v. In general, test γ may have several test runs based on the different interleavings of the protocol transitions.

A test γ *exercises* a transition t_i if t_i appears in a test run of γ.

A test γ is *consistent* relative to a protocol if there is at least one test run for γ and every executable path starting from γ is extensible to a test run of γ.

Note that if the test γ is consistent then $v \neq \langle\rangle$ since protocol runs always start with an environment input message. Further, the consistency of a test ensures that every execution path starting from the stable state corresponding to that test can be extended to a run that uses all the environment input messages specified in test γ. More importantly, consistency of a test ensures that consistent tests that share a test run must be identical.

Proposition 1. *Consistent test $\gamma_1 = \gamma_2$ if test run of γ_1 is a test run of γ_2.*

Proof Sketch. Let r_1 be the common test run that starts from the stable global stable state u. Let v be the external environment input messages used in run r_1. Let $\gamma_1 = \langle u_1,\, v_1\rangle$ and $\gamma_2 = \langle u_2,\, v_2\rangle$. Since r_1 is a test run of both γ_1 and γ_2, $u_1 = u_2 = u$ and since both γ_1 and γ_2 are both consistent test, $v_1 = v_2 = v$. $\square$

From Proposition 1, it follows that a protocol run is a test run of at most one consistent test. An example of a consistent and an inconsistent test can be found in Section 5.2 and Section 4.1, respectively.

In our testing model, the possible outcomes of a test are all the observable sequences of global protocol states that can happen in all the test runs. Since the global protocol states are directly accessible to the runtime monitor they are not a part of a test. Our notion of consistency of a test is closely related to the notion of *valid tests* that has been extensively studied earlier in the testing of protocols and FSMs. An overview of FSM and protocol testing may be found in [13]. A valid test is a consistent test whose outcome conforms to the protocol specification. However, a consistent test need not be a valid test since consistency does not require a conforming test outcome. A consistent test however, precludes invalid or inconclusive tests [13] with inputs that cannot be processed in any run.

Further, in our model, the verdict associated with a test is determined by the runtime monitor, based on the global states observed by the monitor. Hence on changing an implementation, it may be necessary to change both the monitor as well as the tests. In this paper, we assume that any change to the protocol implementation is preceded by a change to a specification and leads to an appropriate change to the monitor[2]. We only focus on impact of implementation changes on the test itself. Henceforth, in this paper, we do not explicitly distinguish a protocol implementation under test from its specification and simply refer to the former as the protocol.

Our choice of the above white-box testing model is largely motivated by the experience of first author in designing real cache coherence, network, and I/O protocol products. In practice, many protocols work over bounded resources and hence are not input-enabled. For protocols with input-enabled components, every test is trivially consistent. In such cases, the approach in [12] may be used to determine how the change affects the test purpose.

4.1 Consistently Testable Protocols

As mentioned above, consistency of protocol tests ensures that any protocol run is a test run of at most one consistent test. To test the conformance of a protocol, we must also make sure that every protocol run is a test run of some consistent test, *i.e.*, it must be possible to devise a consistent test to check each protocol run. A protocol is *consistently testable* if each protocol run is a test run of a consistent test. Consequently, each protocol run of a consistently testable protocol is a test run of exactly one consistent test.

A protocol may be consistent but it may not be consistently testable.

Example 1: Consider the following protocol with the controllers P_1 and P_2 with respective controller stable states $\{s_0\}$ and $\{t_0\}$ with transitions,

P_1 : 1. $m_0(\epsilon, 1),\ s_0 \mapsto s_1,\ m_1(1, 2)$, 2. $m_2(2, 1),\ s_1 \mapsto s_0,\ m_3(1, 2)$,
3. $m_9(2, 1),\ s_1 \mapsto s_0,\ m_4(1, \epsilon)$.
P_2 : 4. $m_1(\epsilon, 2),\ t_0 \mapsto t_2,\ m_9(2, 1)$, 5. $m_1(1, 2),\ t_0, \mapsto t_1,\ m_2(2, 1)$,
6. $m_3(1, 2),\ t_1 \mapsto t_0,\ m_4(2, \epsilon)$, 7. $m_1(1, 2),\ t_2 \mapsto t_0,\ m_5(2, \epsilon)$,
8. $m_5(\epsilon, 2),\ t_1 \mapsto t_1,\ m_6(2, \epsilon)$.

It can be verified that the above protocol is consistent. However, the protocol is not consistently testable since the protocol run r_1 = [1, 4, 3, 7] is not a test run of any consistent test. This is because for r_1 to be a test run, the test must contain the environment input messages $m_0(\epsilon,\ 1)$ and $m_1(\epsilon,\ 2)$. Such a test is not consistent since it can lead to the run r_2 = [1, 5, 2, 6], which does not use the input message $m_1(\epsilon,\ 2)$ and cannot be further extended to a run ending in a stable global protocol state while doing so.

[2] In an earlier paper [12], we have shown how impact of protocol changes on runtime monitors can be automatically evaluated. The procedure described there uses selective state exploration guided by protocol states observed by the monitor to identify and automatically synthesize monitors for protocol changes. For more details the reader may please refer to that paper.

In general, consistent testability imposes additional constraints on the runs of a consistent protocol, which can be used to check whether a given protocol is consistently testable.

In a consistent testable protocol, for any two protocol runs r_1 and r_2 that start by processing the same environment input message in the same stable state, the set of environment inputs processed by r_1 (r_2) should be a subset of those processed by r_2 (r_1). If the environment messages used in r_1 (r_2) is a proper subset of r_2 (r_1) then the extra messages in r_2 (r_1) should only transition the protocol from one stable controller state to another.

In principle, to verify consistent testability of a given protocol, it suffices to find two runs r_1 and r_2 that violate the above condition. For instance in the above protocol, the runs r_1 and r_2 start from the same global stable state $\langle\langle s_0, t_0\rangle, \langle\rangle\rangle$, and process the same environment input message $m_0(\epsilon, 1)$ in that stable state. However, the environment messages processed by r_2 is a proper subset of those processed by r_1; run r_1 additionally processes the environment input message $m_1(\epsilon, 2)$, which does not transition the protocol from one stable state to another. Hence the protocol is not consistently testable as shown above.

A more feasible approach for verifying consistent testability of a given protocol is to compute the interaction contexts of the transitions of the protocol by state exploration and analyze these contexts for the above described condition by considering messages of context elements with the same stable state. This is described in the next section.

5 Impact of Single Transition Changes on a Test

In this section, we describe how to determine the impact of addition, replacement, and deletion of single protocol transitions on an existing protocol test. Conditions under which the consistency of a test is preserved by a change are identified. If such a test exercises the change then it may be re-used without any additional modifications. Otherwise, the test is independent of the change and may not be included to test the changed protocol[3].

In this paper, we assume the changes are incorporated into protocols only if they preserve the consistent testability. In principle, this can be checked by considering the runs of the changed protocol starting from the same controller stable state and checking whether their environment inputs are a subset of each other as described above.

5.1 Interaction Context of Transitions

To determine the effect of changes on tests an *interaction context* is associated with each protocol transition.

The interaction context of a transition t_i, $IC(t_i) = \{\langle u_j, v_j\rangle\}$ is the set of all tests $\langle u_j, v_j\rangle$ exercising the transition t_i.

[3] Of course, such tests may be included in regression testing of the changed protocol based on several coverage criteria and these are not considered here.

The interaction context $IC(t_i)$ is computed by doing repeated backward and forward image computations over the global protocol state space starting respectively from the global state enabling and the state enabled by the transition t_i. The computations stop once all reachable global states with controller stable states and only environment inputs are obtained. The set of global stable protocol states produced by the forward and the backward computations are then matched to produce global stable states and the environment inputs and form one pair of the interaction context. The context $IC(t_i)$ is the set contains all such pairs produced by the matching stable global protocol states.

For instance, consider computing the interaction context $IC(5)$ of the transition 5 in the protocol example described in the previous section. The backward image computation starts from the global state enabling transition 5, $g_p = \langle\langle xs_1, t_0\rangle, \langle m_1(1, 2)\rangle\rangle$ where xs_1 is a symbolic state variable denoting the controller $P_1's$ state. One step image computation of state g_p with transition 1 gives the global state $g_1 = \langle\langle s_0, t_0\rangle, \langle m_0(\epsilon, 1)\rangle\rangle$ and this step also instantiates the variable xs_1 to value s_1 to give the instantiated state $g_p = \langle\langle s_1, t_0\rangle, \langle m_1(1, 2)\rangle\rangle$. Since g_1 is a stable global protocol state with only environment inputs and there are no other predecessors of g_p, the backward image computation stops.

Similarly, the forward image computation starts with the state $g_s = \langle\langle xs_2, t_1\rangle, \langle m_2(2, 1)\rangle\rangle$, and after two image computation steps using the transitions 2 followed by transition 6 stops with the global state $g_2 = \langle\langle s_0, t_0\rangle, \langle\rangle\rangle$. The variable xs_2 is instantiated with value s_1 to produce the instantiated $g_s = \langle\langle s_1, t_1\rangle, \langle m_2(2, 1)\rangle\rangle$. The states g_1 and g_2 match since $g_p \rightarrow^5 g_s$ is an execution step for the instantiated states g_p and g_s. Hence the initial state g_1 is included in the context to produce $IC(5) = \{g_1 = \langle\langle s_0, t_0\rangle, \langle m_1(\epsilon, 1)\rangle\rangle\}$.

Interaction contexts may be used to check whether a protocol is consistently testable. To do so, we union the interaction contexts of the protocol transitions. Then, for each pair $\langle u_1, v_1\rangle$ and $\langle u_2, v_2\rangle$ such that $u_1 = u_2$ if v_1 and v_2 have the same prefix of environment inputs then we check that v_1 (v_2) is a subset of $v_2(v_1)$. If one is proper subset of the other then it is ensured that each extra message m_i is processed by a transition whose input and output states are controller stable states. This guarantees that the extra messages only transition the protocol among global stable protocol states.

For each change, we consider the transitions appearing in the change and compute their interaction contexts. The interaction context may be computed either based on the original or computed based on the changed protocol depending on whether we are adding, deleting or replacing a protocol transition. As explained below, for replacement changes the interaction context of the transition being replaced is computed based on the original protocol whereas that of the transition being added is computed using the changed protocol.

5.2 Adding a Transition

Consider an update rule ur: t'_i, that adds transition t'_i: $m(j, i)$, $s_i \mapsto s'_i$, $m'(i, k)$ to controller P_i of a protocol P. The update ur allows the controller P_i to process the input message $m(j, i)$ in state s_i, which is not possible in the original protocol, and produces a consistent and consistent testable changed protocol. It should be clear that every protocol run of the original protocol is a run of the changed protocol since all transitions of P are also present in P'. Further, since the changed protocol must be consistent it also follows that the transition t'_i appears in at least one changed protocol run.

Let $\gamma = \langle u, v\rangle$ be any consistent protocol test of the original protocol. Since γ is consistent, a run of the original protocol and therefore, a run of the changed protocol is a test run of γ. Now, since the changed protocol is consistently testable, there is exactly one consistent test for each changed protocol run. Hence it follows that γ is a consistent test of the changed protocol.

The effect of such a change on the test γ is determined by computing the interaction context $IC(t'_i)$ of the newly added transition t'_i, by performing state exploration over the changed protocol as described above. If the test γ belongs to $IC(t'_i)$ then the test γ exercises the newly added transition t'_i. In this case γ is re-usable. If the test does not appear in $IC(t_i)$ then none of the executions of γ contain the transition t'_i and hence γ is independent of the change.

Example 2: Consider the following protocol with controllers P_1 and P_2 with stable states $\{s_0,\ s'_0\}$ and $\{t_0\}$ respectively, with the transitions,

$$\begin{array}{lll} P_1: & 1.\ m_1(2,1),\ s_0 \mapsto s_1,\ m_2(1,2), & 2.\ m_3(2,1),\ s_1 \mapsto s_2,\ m_4(1,2), \\ & 3.\ m_5(2,1),\ s_2 \mapsto s'_0,\ m_6(1,\epsilon), & 4.\ m_0(\epsilon,1),\ s'_0 \mapsto s_0,\ m_0(1,e). \\ P_2: & 5.\ m_0(\epsilon,2),\ t_0 \mapsto t_0,\ m_1(2,1), & 6.\ m_2(1,2),\ t_0, \mapsto t_0,\ m_3(2,1), \\ & 7.\ m_4(1,2),\ t_0 \mapsto t_0,\ m_5(2,1). & \end{array}$$

A consistent protocol test is $\gamma = \langle\langle s'_0,\ t_0\rangle,\ \{\ m_0(\epsilon,\ 2),\ m_0(\epsilon,\ 1)\}\ \rangle$; a test run for γ is $r_1 = [4,\ 5,\ 1,\ 6,\ 2,\ 7,\ 3]$. Suppose we add transition 8: $m_1(2,1),\ s'_0 \mapsto s_1,\ m_2(1,2)$ to controller P_1. It can be verified that this change preserves both the consistency and the consistent testability of the protocol. The interaction context of this new transition, $IC(8) = \{\langle\langle s'_0,\ t_0\rangle,\ \{\ m_0(\epsilon,\ 2),\ m_0(\epsilon,\ 1)\},\ \langle\langle s'_0,\ t_0\rangle,\ \{m_0(\epsilon,\ 1)\}\}$, includes the test γ and hence the test γ is re-usable for the changed protocol. A test run of γ with the new transition is [5, 8, 6, 2, 7, 3, 4].

Alternatively, we can add transition, $9: m_7(\epsilon,2),\ t_0 \mapsto t_0,\ m_8(2,\epsilon)$ to the controller P_2 of the above protocol while preserving its consistency and consistent testability. Then, the interaction context $IC(9) = \{\langle\langle s_0, t_0\rangle, \langle m_7(\epsilon,2)\rangle\rangle, \langle s'_0, t_0\rangle, \langle m_7(\epsilon,2)\rangle\}$, does not include the test γ and it can be verified that test γ does not exercise transition 9 and hence is independent of this change.

Note that each pair in the interaction context $IC(t'_i)$ corresponds to a consistent test that exercises the newly added transition t'_i. For changes that add a single transition, we simply determine, which of these tests already exist for the original protocol. The remaining pairs may be used as new tests.

5.3 Replacement and Deletion of Transition

Consider an update rule ur: $t_i \Rightarrow t'_i$ that replaces a transition t_i in controller P_i with a new transition t'_i in the same controller. The update ur produces a consistent changed protocol in which there are runs containing the newly added transition t'_i. It also ensures that every transition t_j that appears in an original protocol run with the replaced transition t_i appears in some other runs not containing t_i. The changed protocol is also consistently testable.

To determine the effect of the update ur on a consistent test γ of the original protocol, we first determine whether γ exercises the new transition t'_i in the changed protocol. To do so, the interaction context $IC(t'_i)$ is computed over the changed protocol. If γ belongs to $IC(t'_i)$ then a test run of γ containing t'_i is a protocol run of the changed protocol. In this case, the update ur must preserve the consistency of γ since there is a run with t'_i in the changed protocol that can be tested only by using the test γ. Hence γ is re-usable.

However, if γ does not belong to $IC(t'_i)$ then it may no longer be a consistent test of the changed protocol. As an example, consider changing the protocol described in the previous subsection, using the rule ur: 4. $m_0(e,1)$, $s'_0 \mapsto s_0$, $m_0(1,e)$, $\Rightarrow$ 8: $m_1(2,1)$, $s'_0 \mapsto s_1$, $m_2(1,2)$ that replaces transition 4 by transition 8. The changed protocol is consistent and consistently testable. However, the existing test $\gamma = \langle\langle s'_0, t_0\rangle, \langle m_0(\epsilon, 2), m_0(\epsilon, 1)\rangle\rangle$ is no longer consistent since no transition in the changed protocol can process the input message $m_0(\epsilon, 1)$. Note that a consistent test that exercises the new transition 8 is $\langle\langle s'_0, t_0\rangle, \langle m_0(\epsilon, 2)\rangle\rangle$.

In general, the replacement change ur makes a test γ inconsistent only if every test run of γ in the original protocol contains the replaced transition t_i. We can determine this by extending the interaction context computation to include the executable paths. Then, it can be checked that every executable path in the context $IC(t_i)$ computed over the original protocol contains the transition t_i.

Alternatively, we can consider each transition t_j distinct from t_i in the original protocol and compute the contexts $IC(t_j)$ and check that the test γ belongs to one of these contexts. This ensures that γ exercises a transition t_j different than t_i. To ensure that γ does not exercise t_i we simply compute these contexts over the changed protocol, where transition t_i has been replaced.

If the test γ does not belong to any $IC(t_j)$ then it is inconsistent for the changed protocol. In this case, we consider each pair $\langle u_j, v_j\rangle$ in each context $IC(t_j)$. If $u = u_j$ for some pair then we patch γ to generate a new test $\gamma_j = \langle u, v_j\rangle$. If no such pair is found then γ cannot be patched and is discarded.

Obviously, if γ belongs to neither $IC(t_i)$ nor $IC(t'_i)$ then no runs in the original protocol containing transition t_i are test runs of γ and no runs in the changed protocol containing t'_i are test runs of γ. As these are the only runs affected by the change, the test γ is consistent with respect to the changed protocol. In this case, the test γ is independent of the replacement change and may be discarded from the tests used for the changed protocol.

As an example, consider the protocol obtained from the protocol previous subsection after addition of the transition 8. This protocol can be changed using the replacement rule ur: 8 $\Rightarrow$ 9, that replaces transition 8 by the transition 9 described there. The change produces a consistent and consistently testable protocol. It can be verified that this replacement change preserves the consistency of the existing test $\gamma = \langle\langle s'_0, t_0\rangle, \langle m_0(\epsilon, 2), m_0(\epsilon, 1)\rangle\rangle$, since [4, 5, 1, 6, 2, 7, 3] is a run of the original protocol from γ not containing the replaced transition 8.

The impact of changes that perform deletion of a transition t_i is also determined by computing the interaction contexts of transitions t_j distinct from t_i as described above. The test γ is re-usable across such a change if it belongs to some context $IC(t_j)$; otherwise, γ is inconsistent. An inconsistent test γ is patched in the same way as described above.

6 Impact of Multiple Transition Changes on a Test

It is in practice inevitable to deal with multiple updates simultaneously. Some of the more general changes – for example the introduction of new states – are too complex to be specified by a single update rule, thus multiple rules – sequences of atomic rules – have to be used to define these changes. Furthermore, in any development process it is not practical to modify and/or analyze the test suite for the given protocol at each atomic update. Instead, test suites are revised at certain stages of the development, typically after some substantial changes have been introduced to the system.

Let ur: $\{t'_1,\ t'_2,\ \cdots,\ t'_i,\ t_{i+1} \Rightarrow t'_{i+1},\ \cdots\ t_m \Rightarrow t'_m\}$ be any protocol update, denoting the set of transition changes to a given protocol P, where (primed) transitions t'_j's are added and (unprimed) transitions t_k's are deleted.

The effect of change ur on a protocol test γ is determined by considering the interaction contexts of the transitions in ur. The contexts of the (primed) transitions t'_j's are computed over the changed protocol and that of (unprimed) transitions t_k's are computed over the original protocol.

The protocol transitions t'_j's being added may be considered individually and the effect on the test γ may be determined as described in the previous subsection on addition of single transitions. However, determining the effect of deletion of transitions by considering transitions t_k's individually does not work since inconsistencies arising due to test runs containing multiple deleted transitions may be missed. For instance, let t_1 and t_2 be any two transitions in ur that are being deleted. Assume that every test run of γ contains either t_1 or t_2 but not both. Hence in this case, removal of both t_1 and t_2 must make the test γ inconsistent.

However, this will not be the case if we individually consider deletions since while considering deletion of t_1, the test γ will be consistent since there is a run with t_2. Similarly, individually considering t_2 will also lead to γ being consistent since there is a test run with t_1.

To handle this problem the procedure for handling individual replacements is modified to consider all the interaction contexts of the protocol transitions being replaced. Let $IC_r = \bigcup_{t_l \notin ur} IC(t_l)$ be the union of all the interaction contexts of the transitions t_l's in the original protocol that are not being replaced. The multi-controller ur preserves the consistency of a test γ only if it belongs to IC_r.

6.1 Handling Redundancies in Multiple Updates

According to the discussion above, we sometimes have to consider significant changes to a controller involving large sequences of individual update rules. As the changes are defined in an ad hoc development process, they are often unnecessarily complex and contain redundancies. In such cases the impact of the given update on tests can not be analyzed efficiently based on the original change specification. Instead, an equivalent multiple update is constructed, which is optimal in the sense that it is the best suitable for the analysis.

The essence of our approach is as follows: Let us consider that we are given a redundant update specification with multiple update rules, and we have to analyze its impact on a given test. We apply the specified update to compute a changed protocol and ensure that this protocol is consistent and consistently testable. But then we do not immediately move on to analyze the impact of the change based on the original update specification. Instead we first apply a method to reduce the update, i.e., to determine equivalent set of atomic changes that are producing the same changed protocol and that are more appropriate for evaluating the impacts. The reduction in the most straightforward case brings on the removal of redundancies, but in a more sophisticated approach it creates an update that is optimal with respect to the cost of evaluating the impacts. Finally, we apply the method described in the first part of this section to determine the impact of the change on the test considering each atomic change of the optimized multiple update.

We consider the previously discussed three types of atomic update rules: Addition, deletion and replacement of transitions. The problem of determining the best equivalent update can be stated as follows: Let us consider an update δ with multiple rules turning CFSM P_i to CFSM P_i'. Identify the shortest equivalent sequence of update rules changing CFSM P_i to P_i'.

In a more sophisticated approach – if some update rules are preferred over others – a cost function may be assigned to update rules. Let ρ be a cost function that assigns a nonnegative real number $\rho(ur)$ to each update rule. We constrain ρ to be a distance metric. That is, it satisfies the following three properties: $\rho(ur) \geq 0$ and $\rho(ur^0) = 0$ (nonnegative definiteness); $\rho(ur) = \rho(ur^{-1})$ (symmetry); $\rho(ur_{13}) \leq \rho(ur_{12}) + \rho(ur_{23})$ for any three operations with the following property: $P_i \rightharpoondown P_i'$ via ur_{12}, $P_i' \rightharpoondown P_i''$ via ur_{23} and $P_i \rightharpoondown P_i''$ via ur_{13} (triangle inequality). Furthermore, let the cost of an update with multiple rules $\delta = \{ur_1, ur_2, ..., ur_k\}$ be $\rho(\delta) = \sum_{i=1}^{k} \rho(ur_i)$.

The cost of an update rule – in general – may represent any practical property of the given atomic change. In our case costs reflect the impact of the given atomic update rule on the test set; updates that are likely to induce inconsistent tests are assigned a higher cost than others. For instance, consider a multiple controller change that includes – among others – the addition of two interdependent transitions t_1' and t_2', such that t_2' occurs in a run in the changed protocol iff t_1' also occurs. Obviously, the two update rules have the same interaction contexts, thus we only have to consider one of them to analyze the impacts. This interdependency can be taken into account for example by setting the cost of one of the update rules to 0.

As our costs are defined as distance metrics, the problem of optimizing multiple updates can be restated as finding the (edit) distance between two CFSMs

P_i and P_i', where the distance between P_i and P_i' is defined to be the minimum cost of all sequences of edit operations that change P_i to P_i':[4]

Definition 1. $dist(P_i, P_i') = min\{\rho(\delta) \mid \delta \text{ is an update changing } P_i \text{ to } P_i'\}$.

With this approach we have turned the problem of reducing multiple updates to an approximate graph matching problem [16]. Thus the tools and algorithms of the graph matching theory can be used to generate an update with the following properties:[5] It is equivalent to the original update specification, i.e., it induces the required modifications; it is the lowest-cost update, i.e., the impact of the given change on tests can be most effectively calculated based on it.

7 Conclusion

An automatic approach for determining impact of protocol changes on existing protocol tests in a white-box testing model is proposed. Protocols are modeled as a network of CFSMs that interact by message passing over bounded queues. Protocol changes add/replace/delete one or more protocol transitions in one or more controllers. Protocol tests are formalized as a protocol stable state along with a set of external environment inputs. Notions of consistent tests and consistently testable protocols are introduced. Changes must preserve consistently testability of protocols so that it is still possible to test all the runs of the changed protocol by using consistent tests. It is shown how symbolic state exploration over the changed protocol can be used to ensure that the protocol is consistently testable. The impact of a change on a test is characterized in terms of whether the consistency of the test is preserved by the change. For tests, whose consistency is preserved, we further show how state exploration can be used to determine whether the test exercises the changed behavior in which case it is re-usable; otherwise, the test is independent of change. We showed that single transition additions always preserve consistency of existing tests. Single transition replacements may make tests inconsistent if every test run exercises the deleted transition. We have shown how the approach can be extended to deal with more complex changes where protocol transitions in multiple controllers are simultaneously changed. We also describe a novel approach to reduce complex changes and create the shortest – or in an other sense optimal – sequence of changes inducing the required modifications. To the best of our knowledge, this is perhaps the first paper to formally address the effect of changes to CFSM-based protocols on protocol tests. We plan to extend this approach to use static analysis to analyze the protocol transition dependencies [19] to make it more useful in practice. We also plan to investigate augmenting existing tests with addition information such as states where inputs are issued to further facilitate the change impact analysis in practice.

[4] The original problem identifying the shortest sequence of update rules is a special case of the latter with all update rules having equal costs.

[5] For the algorithms and their application considering CFSMs see [17] and our earlier paper [18].

References

1. Bochmann, G.V., Petrenko, A.: Protocol testing: review of methods and relevance for software testing. In: ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis, New York, NY, USA, ACM Press (1994) 109–124
2. Linn, R.J., Uyar, M.., eds.: Conformance testing methodologies and architectures for OSI protocols. IEEE Computer Society Press, Los Alamitos, CA, USA (1995)
3. Lee, D., Yiannakakis, M.: Principles and methods of testing finite state machines – a survey. Proceedings of the IEEE **84**(8) (1996) 1090–1123
4. Duale, A.Y., Uyar, M..: A method enabling feasible conformance test sequence generation for efsm models. IEEE Trans. Comput. **53**(5) (2004) 614–627
5. D. Brand, A.M., Zafiropulo, P.: On communicating finite state machines. In: Journal of Associating Computing Machinery, JACM. Volume 30(2). (1983)
6. Peng, W., Purushothaman, S.: Data flow analyses of communicating finite state machines. In: Transactions on Programming Languagaes and Systems TOPLAS. Volume 13. (1991)
7. Holzmann, G.J.: Design and validation of computer protocols. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1991)
8. Subramaniam, M., Chundi, P.: Preserving consistency and executability of protocols across updates. In: Proceedings of the 6th International Conference on Formal Engineering Methods, ICFEM. Volume LNCS. (2004)
9. Arnold, R.S.: Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos, CA, USA (1996)
10. Ryder, B.G., Tip, F.: Change impact analysis for object-oriented programs. In: Proceedings of PASTE-01. (2001)
11. Rothermal, G., Harrold, M.J.: A safe, efficient regression test selection technique. In: ACM Transactions on Software Engineering and Methodology. Volume 6(2). (6(2), 1997)
12. Subramaniam, M.: Preserving consistency of runtime monitors across protocol changes. In: Proc. of Tenth IEEE International Conference on Engineering of Complex Computer Systems ICECCS. (2005)
13. Schmitt, M.: Automatic Test Generation Based on Formal Specifications. Ph.d., Georg-August-University of Goettingen (2003)
14. M. Kaufmann, A.M., Pixely, C.: Design constraints in symbolic model checking. In: Proc. of Intl. Conference on Computer-Aided Verification CAV. Volume LNCS. (1998)
15. K. Shimizu, D. L. Dill, A.J.H.: Monitor-based formal specification of pci. In: Proc. of Intl. Conference on Formal Methods in Computer-aided design, FMCAD. Volume LNCS 1954. (LNCS 1954, 2000)
16. Bunke, H.: Graph matching: Theoretical foundations, algorithms, and applications. In: Proceedings of Vision Interface 2000, Montreal. (2000) 82–88
17. Wang, J.T.L., Zhang, K., Chirn, G.W.: Algorithms for approximate graph matching. Information Sciences **82**(1-2) (1995) 45–74
18. Pap, Z., Csopaki, G., Dibuz, S.: On the theory of patching. In: Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods, SEFM. (2005) 263–271
19. Subramaniam, M., Shi, J.: Using dominators to extract protocol contexts. In: Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods, SEFM. (2005)