Correct Passive Testing Algorithms and Complete Fault Coverage

Arun N. Netravali, Krishan K. Sabnani, and Ramesh Viswanathan

Bell Laboratories, 101 Crawfords Corner Road, Holmdel, NJ 07733,
{ann,kks,rv}@bell-labs.com

Abstract. The aim of passive testing is to detect faults in a system while observing the system during normal operation, that is, without forcing the system to specialized inputs explicitly for the purposes of testing. We formulate some general correctness requirements on any passive-testing algorithm which we term *soundness* and *completeness*. With respect to these definitions, we show that the homing algorithm, first proposed in [4], and subsequently used in [6–8], is sound and complete for passively testing the conformance of an implementation for several distinct conformance notions ranging from trace-containment to observational equivalence to even exact identity. This implies that, for some notions of conformance, there are faulty implementations that would not be detectable by any sound passive testing algorithm. We define a property to be *passively testable* as one admitting *complete fault coverage* under passive testing, *i.e.*, one for which any faulty execution can be detected through passive testing. We provide an exact characterization of passively testable properties as being a natural subclass of safety properties, namely, those that are trace-contained in sets that are *prefix-* and *suffix-closed*. For such properties, we derive efficient complete passive testing algorithms that take constant time. We demonstrate the applicability of these results to networks and network devices by considering the problem of passively testing an implementation for conformance to the TCP protocol.

1 Introduction

Today's networks are becoming larger, more heterogeneous, and are assembled by integrating equipment from multiple vendors. Consequently, managing networks and network devices is becoming increasingly difficult, making the development of automated network management tools and techniques more important. A key requirement of network management systems is detecting faults (*c.f.* [14, 18]), where a fault identifies abnormal behavior in a network or network device. Faults are detected by analyzing monitored network traffic or device state, the observed data is *mid-stream* in that the network may already have been in operation for some time, and the fault detection process is *passive* in that it cannot affect the normal operation of the network or network device, such as by injecting arbitrary traffic for the purposes of testing. Passive testing is weaker than other

H. König, M. Heiner, and A. Wolisz (Eds.): FORTE 2003, LNCS 2767, pp. 303–318, 2003.

well-known validation methods such as verification and active testing in that system implementation details are unknown, inputs are not controllable, and the system need not be in its initial state when testing begins.

While several practical monitoring tools have been described [9–11, 13], both for network management and for detecting security intrusions, the first formal and systematic treatment of passive testing [4] modeled a network as a (possibly non-deterministic) finite state machine and gave algorithms for detecting if an implementation machine conforms to a given specification machine. The fundamental idea underlying these algorithms is to perform a "homing" process which seeks to infer the current state of the network. In [6], this homing algorithm is considered for communicating finite state machines, thus providing a formal setting in which the problems of fault identification [8] and fault location [7] are addressed by adapting the homing algorithm suitably. However, this previous work does not explicitly address the precise notion of conformance for which the presented homing algorithms are correct. The algorithm in [4] is stated to be applicable for checking observational equivalence, though this is not formally established. On the other hand, the results of [7, 8] seem to be most directly applicable under the assumption that an implementation machine is faulty if it is not isomorphic to the specification machine.

In this paper, we precisely analyze the conformance relation passively tested by the homing algorithm. We first formulate general correctness conditions on any algorithm which seeks to passively test an implementation for some particular property $\mathcal{P}$. We call an algorithm *sound* (for testing property $\mathcal{P}$) if, whenever it rejects an implementation, the implementation is indeed faulty. We term the algorithm *complete* if, in addition to being sound, it rejects a faulty implementation whenever observations are made that could not have been generated by a correct implementation. Using a formalization of these correctness requirements, we prove that the homing algorithm is sound and complete for testing whether an implementation is observationally equivalent to a specification (assumed in [4]), isomorphic to a specification (assumed in [7, 8]), or conformant according to trace-containment, trace-equivalence, simulability, and even exact identity.

Given that these conformance relations are strictly distinct, it is somewhat counter-intuitive that the same homing algorithm could be complete for passively testing all of these different conformance relations. For example, there are implementations that are trace-contained in a specification machine but not observationally equivalent to it. By the soundness of the homing algorithm for trace-containment, it accepts such implementations that are faulty with respect to observational equivalence. However, the completeness of the homing algorithm with respect to observational equivalence establishes that no other sound passive-testing algorithm could detect such implementations as faulty, either. Thus, the homing algorithm's failure to reject these faulty implementations is not as much a reflection on the homing algorithm *per se*, as an intrinsic inability of the passive testing methodology to completely detect non-conformance according to observational equivalence.

We thus consider the general question of which fault notions can be completely detected using passive testing. We consider the *fault coverage* of an algorithm to be the set of implementations it can successfully reject, and deem a property to be *passively testable* if it admits a sound passive testing algorithm whose fault coverage includes all faulty implementations. We show that a passively testable property must be a safety property. This yields an alternate, more direct, proof that observational equivalence and trace equivalence are not passively testable conformance relations. However, we exhibit an example of a safety property that is not passively testable which shows that safety is a necessary but not sufficient condition. We develop an exact characterization of passively testable properties as being validity conditions on traces that are *prefix and suffix closed*. Intuitively, this means that any sub-trace of a valid trace should also be a valid trace. Applying these general results to the case of conformance testing, we obtain that conformance is passively testable exactly when: (a) the conformance relation is trace containment, and (b) the set of traces of the specification machine is suffix-closed.

Using this characterization of passively testable properties, we show that while a passive-testing algorithm for an arbitrary property must account for the unobserved initial behavior of the system, a passive-testing algorithm for a passively testable property can "pretend" as if the system is being observed from its initial state without sacrificing soundness. This allows us to derive complete passive testing algorithms for such properties that have $O(1)$ running time, in contrast to the homing algorithm which has a running time of $O(n)$ for each observation made, where n is the number of states in the specification machine.

We apply our results to passive testing of the transmission control protocol (TCP) [17], one of the most widely-deployed network protocols. We show that, except for requirements on initial conditions for establishing a TCP session, all other parts of the TCP specification can be cast as prefix and suffix-closed properties. In particular, the more complex and intricate core of the TCP protocol that is concerned with congestion control and adaptive retransmissions is well amenable to passive testing.

The remainder of the paper is organized as follows. Section 2 develops the notions of soundness and completeness for a passive testing algorithm and Section 3 analyses the homing algorithm for these properties. Section 4 investigates properties admitting complete fault coverage, identifying necessary and sufficient conditions and deriving more efficient testing algorithms for such properties. In Section 5, we apply these results to the problem of monitoring TCP implementations. We conclude in Section 6 with a summary of our contributions, comparison to other related work, and an outline of further directions.

2 Correctness of Passive Testing Algorithms

2.1 Formal Framework

In this section, we describe our formal setting for studying passive testing. There are several choices for formally modeling the system implementation under test.

The results in this paper depend on relatively few assumptions and are applicable to a broad range of these formalisms. Rather than committing to a particular formalism or demonstrating our results for each choice of a formal model separately, we develop a general axiomatization of implementation models for which our results will hold. This axiomatization provides a distillation of the key characteristics of an implementation model that enable our results and make them easily applicable to a wide class of model formalisms.

We use the following notations. For sets A, B, the set $A \times B$ denotes their cartesian product consisting of all tuples $\langle a, b\rangle$ with $a \in A, b \in B$, and A^n denotes the n-fold product $A \times \cdots \times A$. For a set A, the Kleene-closure A^* is the set of all finite sequences over the set A. The empty sequence is denoted ϵ and $\sigma \cdot \sigma'$ denotes the concatentation of the sequences σ, σ'. Our assumptions on the formal model used are captured in the following definition of *expressively sufficient* models.

Definition 1 (Implementation Model). *A formalism for modeling implementations is* expressively sufficient *if we can define functions $\mathcal{O}(\cdot)$ and $[\![\cdot]\!]$ on the class of all formal models with $\mathcal{O}(I)$ a set, and $[\![I]\!] \subseteq (\mathcal{O}(I))^*$, for any model I. Further, the following two axioms must be satisfied:*

(Axiom I) *For any $\sigma, \sigma' \in (\mathcal{O}(I))^*$, if $\sigma \cdot \sigma' \in [\![I]\!]$ then $\sigma \in [\![I]\!]$*
(Axiom II) *For any set Σ and finite sequence $\sigma \in \Sigma^*$ there is a model I_σ with $\mathcal{O}(I_\sigma) = \Sigma$ such that $[\![I_\sigma]\!] = \{\sigma' \in \Sigma^* \mid \sigma' \cdot \sigma'' = \sigma \text{ for some } \sigma'' \in \Sigma^*\}$.*

Intuitively, for an implementation model I, the set $\mathcal{O}(I)$ corresponds to the atomic observations that I can exhibit at any particular instant, such as an `ack` message being sent, or an input operation i being received, or an output operation o performed. These atomic observations are extended over time to yield trace sequences with $[\![I]\!]$ the set of all the traces that can be observed from I upto any time instant. (Axiom I) then captures the intuitive requirement that if an implementation can exhibit the observed sequence $\sigma \cdot \sigma'$, then it can also exhibit the prefix σ (namely, earlier in the same execution trace). (Axiom II) requires that the class of formal models be expressive enough so that for any trace σ there is an implementation I_σ that exhibits exactly σ and no other observations subsequently. The importance of the technical requirement of (Axiom II) will become more apparent in the development of the results of Section 4.

Popular choices for implementation models are finite automata that can be used to express the control plane of network protocols and extended finite automata that capture the data plane as well. We show that both these models are *expressively sufficient* and our results on passive testing will therefore apply to them as specific instances.

Example 1 (Finite Automata). A finite automaton M is a quadruple $\langle \Sigma, S, s_0, \delta\rangle$, where Σ is a set, S is a finite set of states, $s_0 \in S$ is the initial state, and the transition relation $\delta \subseteq S \times \Sigma \times S$. The automaton is *deterministic* if for any $s \in S, a \in \Sigma$, there is at most one s' such that $\langle s, a, s'\rangle \in \delta$. A sequence $s_0 a_1 s_1 \ldots a_n s_n$, where $s_i \in S, a_i \in \Sigma, n \geq 0, 1 \leq i \leq n$, is a run of the automaton M if $\langle s_{i-1}, a_i, s_i\rangle \in \delta$ for $1 \leq i \leq n$. We can see that finite automata satisfy the conditions of Definition 1 as follows. For an automaton $M = \langle \Sigma, S, s_0, \delta \rangle$, we take $\mathcal{O}(M)$ to be Σ and $[\![M]\!]$ to be the set of all sequences $a_1 \ldots a_n \in \Sigma^*, n \geq 0$ such that for some $s_1, \ldots, s_n \in S$, we have that $s_0 a_1 s_1 \ldots a_n s_n$ is a run of M. Clearly, (Axiom I) is satisfied by this definition. For (Axiom II), consider any sequence $\sigma = a_1 \ldots a_n$, where $a_i \in \Sigma$ for some set Σ. We define the implementation I_σ to be the automaton $\langle \Sigma, S, s_0, \delta \rangle$ where $S = \{s_0, \ldots, s_n\}$ and $\delta = \{\langle s_i, a_{i+1}, s_{i+1} \rangle \mid 0 \leq i < n\}$ which can be seen to satisfy the requirements of (Axiom II). Note that the automaton I_σ is deterministic and therefore the smaller class of deterministic finite automata is expressively sufficient, as well.

Along similar lines, variants of finite automata such as those with explicit separation between input and output symbols [4], Mealy and Moore automata, Extended Finite Automata [3], and Kripke Structures can be easily seen to also satisfy the requirement of being expressively sufficient. For the rest of the paper, we assume any ambient modeling formalism that is expressively sufficient, and refer to any models in this ambient formalism as implementations.

2.2 Acceptable Traces for Passive Testing

Given an implementation I, we are interested in checking whether I satisfies some desired correctness property, *e.g.*, that I is a faithful implementation of some protocol standard. Just as for implementation models, there is a plethora of available specification mechanisms. Since our results do not depend on the details of the particular specification method used, we abstractly view any correctness property as a collection of implementations, namely those that satisfy the property.

Definition 2. *A* property*, $\mathcal{P}$, is a collection of implementations. An implementation I satisfies a property $\mathcal{P}$, denoted $I \models \mathcal{P}$, if $I \in \mathcal{P}$.*

We now consider the question of how to passively test a black-box implementation for some property $\mathcal{P}$. In passive testing, we do not have access to the implementation itself, nor can we assume that the implementation is in its initial state of execution. Instead, we observe its execution starting from some intermediate point after which we see a trace of externally observable actions. We now define when the implementation cannot be rejected as faulty on the basis of an observed trace, σ. Since the observation was begun mid-stream, we have to assume that some unknown trace σ' preceded the observed sequence σ. Thus, the implementation cannot be deemed faulty if $\sigma' \cdot \sigma$ is a possible trace of a correct implementation, *i.e.*, an implementation I satisfying $\mathcal{P}$. We consider such traces σ to be acceptable when passively testing the property $\mathcal{P}$, and formally define $PT(\mathcal{P})$ to be the set of all such acceptable traces.

Definition 3 (Acceptable Traces under Passive Testing). *For a property $\mathcal{P}$, the set $PT(\mathcal{P})$ of* acceptable *traces is defined as $\sigma \in PT(\mathcal{P})$ if and only if there is some implementation $I \models \mathcal{P}$ and trace σ' with $\sigma' \cdot \sigma \in [\![I]\!]$.*

Conversely, if we observe a trace $\sigma \notin PT(\mathcal{P})$ at any point midstream, then we can conclude that the implementation is faulty; such traces can therefore be considered fault-symptomatic.

Proposition 1 (Fault Symptomatic Traces). *Let $\mathcal{P}$ be a property. For any implementation I, if we have $\sigma' \cdot \sigma \in [\![I]\!]$ with $\sigma \notin PT(\mathcal{P})$ then $I \notin \mathcal{P}$.*

Note that the definition of $PT(\mathcal{P})$ is based only on the property and is not dependent on any particular algorithm used to passively test the property. It can therefore be used to provide correctness criteria on algorithms for passively testing property $\mathcal{P}$. A passive testing algorithm, $\mathcal{A}$, determines an implementation to be correct or faulty on the basis of observed trace σ, which we write as $\mathcal{A}$ accepting or rejecting the trace σ, respectively. A passive testing algorithm is *sound*, if whenever it rejects an observed trace, the implementation is definitely faulty. A passive testing algorithm is *adequate* if whenever an implementation can be determined to be faulty on the basis of its observed trace, the algorithm rejects the implementation. The algorithm is *complete* if it is both sound and adequate.

Definition 4 (Passive Testing Algorithm Properties). *Let $\mathcal{A}$ be a passive testing algorithm for the property $\mathcal{P}$. Then*

Soundness: *The algorithm $\mathcal{A}$ is* sound *if whenever $\sigma \in PT(\mathcal{P})$ we have that $\mathcal{A}$ accepts the trace σ.*

Adequacy: *The algorithm $\mathcal{A}$ is* adequate *if whenever the algorithm $\mathcal{A}$ accepts the trace σ we have that $\sigma \in PT(\mathcal{P})$.*

Completeness: *The algorithm $\mathcal{A}$ is a* complete *passive testing algorithm for the property $\mathcal{P}$ if it is both sound and adequate,* i.e., *the set of traces it accepts is exactly the same as $PT(\mathcal{P})$.*

3 Homing Algorithm for Passive Testing

Most previous work on passive testing is based on the homing algorithm, given in [4], for testing for conformance to a specification given as a Mealy automaton. In this section, we analyze this algorithm for the precise conformance notion with respect to which it is complete, in the sense of Definition 4.

The homing algorithm is as follows. Because we observe the network "midstream" in its deployment, the algorithm begins by assuming that the network could be in any one of the states (as opposed to the initial state) of the specification machine. With each observation, the set of states that the network could be in is updated to include only those that are consistent with the observation just made. If we ever reach the point of not being able to find any state consistent with the observation just made, the algorithm pronounces the implementation to be faulty. We formally state the homing algorithm in the more general context of finite automata that are not necessarily complete or connected (defined in Example 1). For a specification automaton $M = \langle \Sigma, S, s_0, \delta \rangle$, the homing algorithm $\mathcal{H}_M$, defined below, tests for conformance to the automaton M.

Definition 5 (Homing Algorithm). *Let $M = \langle \Sigma, S, s_0, \delta \rangle$ be a finite automaton. Define the relation $R \subseteq S \times S$ by $\langle s, s' \rangle \in R$ iff there is some $a \in \Sigma$ with*

$\langle s, a, s' \rangle \in \delta$. For any trace $\sigma \in \Sigma^*$, the set $\mathcal{H}_M(\sigma) \subseteq S$ is defined by induction on the length of σ as follows:

$$\mathcal{H}_M(\epsilon) = \{s \mid s_0 R^* s\}$$
$$\mathcal{H}_M(\sigma \cdot a) = F_M(\mathcal{H}_M(\sigma), a)$$

where R^* is the reflexive transitive closure of R, and the function F_M is defined by $F_M(S', a) = \{s \mid$ there is an $s' \in S'$ with $\langle s', a, s \rangle \in \delta\}$. The algorithm $\mathcal{H}_M$ accepts a trace σ if and only if the set $\mathcal{H}_M(\sigma)$ is non-empty.

The running-time of $\mathcal{H}_M$ is $O(nm)$ where n is the size of the trace σ and m is the size of M (even if M is deterministic*).*

The algorithm $\mathcal{H}_M$ initially assumes that we could be in any of the states of S that are reachable from the starting state s_0, and then uses the function F_M to refine the knowledge of the current state. Intuitively, the function F_M "homes" in to the current state: if we know that the current state belongs to a set S and we make the observation a, then $F_M(S, a)$ computes the set of states we can be in after having made the observation a.

We now analyze the precise conformance relation to the specification M that the algorithm H_M tests. We use $I \equiv M$ to denote that I is isomorphic to M, and use the standard definitions of simulation, denoted $\preceq_s$, bisimilarity denoted $\approx_s$, trace-containment denoted $\preceq_{\text{tr}}$, and trace-equivalence denoted $\approx_{\text{tr}}$. Somewhat surprisingly, we can show that the algorithm $\mathcal{H}_M$ is complete for passively testing an implementation with respect to any conformance notion that is not weaker than trace-containment and that admits the specification as a conformant implementation.

Theorem 1 (Completeness of Homing Algorithm). *For any finite automaton $M = \langle \Sigma, S, s_0, \delta \rangle$, define the property $\mathcal{P}^M_{\preceq_{\text{tr}}} = \{I \mid I \preceq_{\text{tr}} M\}$. Then the algorithm $\mathcal{H}_M$ is complete for passively testing any property $\mathcal{P}$ with $\{M\} \subseteq \mathcal{P} \subseteq \mathcal{P}^M_{\preceq_{\text{tr}}}$.*

As an immediate corollary, we have that the algorithm $\mathcal{H}_M$ is complete for passively testing an implementation with respect to trace-containment, trace-equivalence, simulation, bisimulation and even isomorphism.

Corollary 1. *Let M be a finite automaton. The algorithm $\mathcal{H}_M$ is complete for passively testing each of the following properties: $\mathcal{P}^M_{\preceq_{\text{tr}}} = \{I \mid I \preceq_{\text{tr}} M\}$, $\mathcal{P}^M_{\approx_{\text{tr}}} = \{I \mid I \approx_{\text{tr}} M\}$, $\mathcal{P}^M_{\preceq_s} = \{I \mid I \preceq_s M\}$, $\mathcal{P}^M_{\approx_s} = \{I \mid I \approx_s M\}$, $\mathcal{P}^M_{\equiv} = \{I \mid I \equiv M\}$.*

In [4], the passive testing algorithm was developed to check bisimilarity (although not proven), and [7], [8] work with the implicit assumption that the homing algorithm checks for isomorphism. Corollary 1 shows that the homing algorithm is the best any passive testing algorithm can do for the conformance relations that these works respectively consider. On the other hand, of these five properties, the relation that an implementation I is not trace-contained in M is the strongest fault notion. Hence, Theorem 1 shows that if algorithm $\mathcal{H}_M$ rejects some implementation then the strongest statement we can make about the faultiness of the implementation is that it is not trace-contained in M.

4 Fault Coverage under Passive Testing

The results of Section 3 show that the homing algorithm $\mathcal{H}_M$ is a complete algorithm for passively testing the properties of trace-containment, trace-equivalence, similarity, bisimilarity, and even isomorphism or exact identity. As detailed in Section 1, this implies that, for some of these conformance relations, *e.g.*, bisimilarity, there are faulty implementations that would not be rejected by the homing algorithm. Thus, even a complete passive testing algorithm only provides a limited guarantee of the correctness of an implementation. More specifically, every rejected implementation is faulty but an accepted implementation is not necessarily correct; completeness of the algorithm only guarantees that no other sound passive testing algorithm could have rejected that implementation either.

In this section, we explore the class of properties for which we can design passive testing algorithms that, in addition to guaranteeing the faultiness of rejected implementations, also guarantee the correctness of accepted ones. In the context of passive testing, this guarantee cannot be obtained unconditionally. In particular, two aspects inherent to passive testing that we have no control over are the particular run that is exercised and observed, and the point in the trace when we begin to observe it. We factor these two uncontrollable aspects by requiring that if observations from every point of every trace of an implementation never cause it to be rejected by passive testing then it is guaranteed to be correct. A property with this characteristic *admits complete coverage under passive testing* or, more tersely, is *passively testable*.

Definition 6 (Admissibility of Complete Coverage). *A property* $\mathcal{P}$ admits complete coverage under passive testing *or is* passively testable *if and only if it has the following closure property: Suppose that* I *is an implementation such that for all traces* σ', σ *with* $\sigma' \cdot \sigma \in [\![I]\!]$*, we have that* $\sigma \in PT(\mathcal{P})$*. Then* $I \models \mathcal{P}$*.*

Definition 6 can be read, contrapositively, as stating that a passively testable property is one for which any faulty implementation $I \not\in \mathcal{P}$ manifests a fault-symptomatic trace, *i.e.*, a trace $\sigma' \cdot \sigma \in [\![I]\!]$ with $\sigma \not\in PT(\mathcal{P})$. We are interested in characterizing the class of properties that are passively testable. We begin by deriving an immediate consequence of Definition 6. Suppose that $\mathcal{P}$ is a passively testable property. Then any implementation I each of whose traces is a trace of some correct implementation $I' \models \mathcal{P}$, must also be correct, *i.e.*, included in $\mathcal{P}$.

Lemma 1. *Suppose that* $\mathcal{P}$ *is a passively testable property and* $\mathcal{P}' \subseteq \mathcal{P}$*. If* $[\![I]\!] \subseteq \bigcup_{I' \in \mathcal{P}'} [\![I']\!]$ *then* $I \models \mathcal{P}$*.*

Lemma 1 yields a necessary condition for a property to be passively testable which can be used to establish that similarity, bisimilarity, and trace-equivalence are not passively testable conformance relations.

Example 2 (Simulation, Bisimulation not Passively Testable). Consider the finite automaton $M = \langle \Sigma, S, s_0, \delta \rangle$ with $\Sigma = \{a, b, c\}$, $S = \{s_0, s_1, s_2, s_3, s_4\}$, and the transition relation $\delta = \{\langle s_0, a, s_1 \rangle, \langle s_0, a, s_2 \rangle, \langle s_1, b, s_3 \rangle, \langle s_2, c, s_4 \rangle\}$. Then the property of being simulated by M is not passively testable. To see this, consider

$I = \langle \Sigma, S', s_0', \delta' \rangle$ with $S' = \{s_0, s_1, s_2, s_3\}$, $S_0' = \{s_0\}$, $\Sigma = \{a, b, c\}$, and $\delta' = \{\langle s_0, a, s_1\rangle, \langle s_1, b, s_2\rangle, \langle s_1, c, s_3\rangle\}$. For $\mathcal{P}' = \{M\}$, we have that $[\![I]\!] \subseteq \bigcup_{I' \in \mathcal{P}'} [\![I']\!]$, but I is not simulated by M which would contradict Lemma 1 if the property of being simulated by M were passively testable. Since I is not bisimilar to M either, this example also shows that the property of being bisimilar is not passively testable also.

Example 3 (Trace Equivalence not Passively Testable). Consider the automaton $M = \langle \Sigma, S, s_0, \delta \rangle$ with $\Sigma = \{a\}$, $S = \{s_0\}$, and $\delta = \{\langle s_0, a, s_0\rangle\}$. Then the property of being trace-equivalent to M is not passively testable. To see this, consider the automaton $I = \langle \Sigma, S, s_0, \delta' \rangle$ with $\delta' = \emptyset$. For $\mathcal{P}' = \{M\}$, we have that $[\![I]\!] \subseteq \bigcup_{I' \in \mathcal{P}'} [\![I']\!]$, but I is not trace-equivalent to M which would contradict Lemma 1 if the property of being trace-equivalent to K were passively testable.

For any property $\mathcal{P}$, define the language $L_{\mathcal{P}}$ of traces as $L_{\mathcal{P}} = \bigcup_{I \in \mathcal{P}} [\![I]\!]$. An immediate consequence of Lemma 1 is the following corollary.

Corollary 2. *Suppose that $\mathcal{P}$ is a passively testable property. Then $I \in \mathcal{P}$ iff $[\![I]\!] \subseteq L_{\mathcal{P}}$.*

Corollary 2 can be restated in terms of the classical notion of being a safety property.

Definition 7 (Safety Property). *Let L be a language (set) of traces. Then L is* prefix-closed *if and only if for any traces σ, σ', if $\sigma' \cdot \sigma \in L$ then $\sigma' \in L$. A property $\mathcal{P}$ is a* safety property *if there exists a prefix-closed language L of traces such that $I \in \mathcal{P}$ iff $[\![I]\!] \subseteq L$.*

Corollary 3. *Suppose that $\mathcal{P}$ is a passively testable property. Then $\mathcal{P}$ is a safety property.*

Corollary 3 shows that the only passively testable properties are safety properties. However, the following example shows that not all safety properties are passively testable.

Example 4. Let $\Sigma = \{a, b\}$ and language $L = \{\epsilon\} \cup \{a \cdot \sigma \mid \sigma \in \Sigma^*\}$, *i.e.*, L is the set of traces where the first action or observation is always a. Then L is prefix-closed and hence $\mathcal{P} = \{I \mid [\![I]\!] \subseteq L\}$ is a safety property. Consider $I = \langle \Sigma, \{s_0\}, s_0, \{\langle s_0, b, s_0\rangle\}\rangle$. Then I does not satisfy $\mathcal{P}$ (since a is not the first observation in any of its traces). However, it can be easily seen that any trace $\sigma \in PT(\mathcal{P})$, *i.e.*, $PT(\mathcal{P}) = \Sigma^*$ and therefore for any σ', σ with $\sigma' \cdot \sigma \in [\![I]\!]$ we have that $\sigma \in PT(\mathcal{P})$. Thus $\mathcal{P}$ does not satisfy the closure condition of passive testability for I.

Example 4 shows that while safety is a necessary condition for passive testability, it is not a sufficient condition. We next identify the set of necessary and sufficient conditions under which a property is passively testable. The proof of Theorem 2 pivotally uses the requirement of (Axiom II) of Definition 1.

Definition 8 (Suffix Closure). *Let L be a language (set) of traces. Then L is* suffix-closed *if and only if for any traces σ, σ', if $\sigma' \cdot \sigma \in L$ then $\sigma \in L$.*

Theorem 2 (Characterization of Passive Testability). *A property $\mathcal{P}$ is passively testable if and only if the following two conditions hold: (1) $I \models \mathcal{P}$ iff $[\![I]\!] \subseteq L_{\mathcal{P}}$, and (2) $L_{\mathcal{P}}$ is suffix-closed.*

Using Theorem 2, we can characterize passively testable properties as a natural special subclass of safety-properties.

Corollary 4. *A property $\mathcal{P}$ is a passively testable property if and only if there is a prefix-closed and suffix-closed language L of traces such that $I \models \mathcal{P}$ iff $[\![I]\!] \subseteq L$.*

By Lemma 1, as established in Examples 2 and 3, we know that trace-containment is the only conformance relation for which we can hope to obtain complete coverage for passive testing. Using Theorem 2, we can also establish the class of specifications (with regard to any modeling formalism) with respect to which trace containment can be passively tested.

Corollary 5. *Consider any modeling formalism that is expressively sufficient and let M be a specification model in this formalism. Then the property $\mathcal{P}_{\preceq_{\mathrm{tr}}} = \{I \mid [\![I]\!] \subseteq [\![M]\!]\}$, of being trace-contained in M, is passively testable if and only if $[\![M]\!]$ is suffix-closed.*

Our next set of results derive more efficient algorithms for passive testing than the homing algorithm $\mathcal{H}_M$ of Section 3.

Theorem 3. *Suppose that $\mathcal{P}$ is a passively testable property. Then $PT(\mathcal{P}) = L_{\mathcal{P}}$.*

Theorem 3 can be used to derive an efficient algorithm for passively testing trace-containment with respect to specifications that admit complete fault coverage. Recall from Definition 4 that obtaining a complete passive testing algorithm for a property is essentially a matter of checking membership in PT of that property. By Theorem 3, this merely amounts to checking membership in the specification itself without any requirement for homing or accounting for what state the implementation could be in, as in the algorithm $\mathcal{H}_M$. The benefit of not having to keep track of a set of states we have homed in to is that in the case that the specification is a *deterministic* finite-state automaton, the reduction in complexity is by the factor of the size of the automaton.

Theorem 4 (Passive Testing Trace Containment). *Suppose that M is a specification model in a formalism is expressively sufficient, such that $[\![M]\!]$ is suffix-closed. Let $\mathcal{P}_{\preceq_{\mathrm{tr}}} = \{I \mid [\![I]\!] \subseteq [\![M]\!]\}$ be the property of being trace-contained by M. Then the algorithm $\mathcal{A}$ which checks for membership in M,* i.e., *$\mathcal{A}$ accepts a trace σ iff $\sigma \in [\![M]\!]$, is a complete passive testing algorithm for the property $\mathcal{P}_{\preceq_{\mathrm{tr}}}$, admitting complete coverage. If M is a* deterministic finite automaton, *the running-time of $\mathcal{A}$ is $O(n)$ where n is the length of the trace, or $O(1)$ in each observation seen.*

An alternative method to establish that a property $\mathcal{P}$ is passively testable is obtained by considering the complement of $L_{\mathcal{P}}$, the set of faulty executions. A property is passively testable if and only if the set of faulty executions satisfies the closure condition that any trace containing a faulty subtrace must itself also be faulty. This can be informally captured by the slogan "once a fault, always a fault": if a faulty subtrace is observed then no (unobserved) past behavior could provide mitigating circumstances to make the complete trace correct.

Corollary 6. *A property $\mathcal{P}$ is passively testable iff the set $S = \overline{L_{\mathcal{P}}}$ of faulty executions satisfies the following closure property: If $\sigma \in S$ then for any σ', σ'' we have that $\sigma' \cdot \sigma \cdot \sigma'' \in S$.*

5 An Example: Application to TCP

In this section, we apply the general results developed to testing an implementation for conformance to the Transmission Control Protocol (TCP) [17]. We choose to consider TCP because it is one of the most widely deployed IP protocols in today's internet and, as has been argued in [9], the behavior of deployed TCP implementations has a significant impact on the performance of the overall Internet.

As might be expected (*c.f.* Example 4), we can show that requirements on initial conditions for establishment of a TCP connection are not testable with complete coverage. However, the results of this section are surprisingly reassuring in that they show that all other properties of the TCP specification are prefix- and suffix-closed, and therefore passively testable with complete coverage. In particular, this includes the more complex and intricate core of the TCP protocol that is concerned with congestion control and adaptive retransmissions. Thus, the aspects of TCP that are more critical from the perspective of adversely impacting the network as a whole through faulty implementations, are well amenable to passive testing.

Our examination of TCP is structured as follows. Our overall goal is to construct a monitor that examines the packets at a TCP client (sender or receiver) to diagnose the implementation's conformance. Note that a single client can simultaneously participate in multiple TCP sessions and as both a sender and receiver — therefore the trace of packets observed by our monitor potentially includes packets belonging to different sessions. To facilitate analysis, in Section 5.1, we first consider each of several aspects of TCP in isolation and assume that all the packets being observed belong to a single session. In Section 5.2, we show how the analysis for each individual property and one session can be lifted to demonstrate passive testability of an unfiltered trace of packets belonging to multiple sessions for conformance to any boolean combination of the several aspects of TCP.

5.1 Single TCP Session and Individual Properties

TCP implements reliable, in-order stream delivery through positive acknowledgements and retransmissions while attempting to optimize throughput with a well-tuned "sliding window". The most basic parts of the TCP specification describe formats for segment headers and bodies, and how acknowledgements are sent in response to received segments. The more complex aspects of TCP deal with retransmissions of unacknowledged segments, and how the sender adjusts window size to optimize throughput while avoiding congestion. Finally, the specification details how a session is established and closed. We examine each of these aspects in turn, and state and informally argue whether they are passively testable, *i.e.*, prefix-, suffix-closed. For many properties this analysis is straightforward; other properties require careful recasting into a form that allows passive testability.

TCP Segment Formats: The specifications of what constitutes a valid segment format all take the form of being an *invariance* property. Formally, these properties can be expressed as languages of the form $\{a_0 \ldots a_n \mid \forall i.\ P(a_i)\}$, where P is a propositional condition. Invariance properties can be seen to be prefix- and suffix-closed, since if $a_0 \ldots a_n$ satisfies the property then we have that $P(a_i)$ for $i = 0, \ldots, n$ and hence any prefix $a_0 \ldots a_m$ for $m \leq n$ and any suffix $a_k \ldots a_n$ for any $0 \leq k \leq n$ also satisfies the property. An example of a segment format specification is that the checksum field of each segment sent is set correctly. This condition can be checked for each segment individually (by using its `source ip address`, `destination ip address`, `zero`, `protocol`, and `tcp length` fields as a pseudo header) and is therefore an invariance property which is passively testable.

Responsiveness of Acknowledgements: The TCP specification allows a leeway in how acknowledgements are generated. TCP prescribes that a receiver must generate an ack at least every two segments — so, an implementation may decide to not ack every single segment (although, this would meet the specification as well). Let *RAck* be the set of implementations (TCP receivers) that meet this specification, and L_{RAck} be the corresponding language of traces. The easiest way to see that *RAck* is a passively testable property is to consider the complement of L_{RAck}. This includes traces that have any subtrace σ which includes more than two segment receipts without any interleaving ack. This, trivially, satisfies the closure condition of Corollary 6.

Sequence Numbers of Acknowledgements: In TCP, acknowledgements are used to specify the sequence number of the next octet that the receiver expects to receive. This is achieved by requiring the sequence number of each acknowledgement (sent by the receiver) to be set to value one greater than the highest octet position in the contiguous prefix of the stream received. Let *SNA* be the property which holds of all TCP receivers that generate correct sequence numbers in their ack messages. The language L_{SNA} can be defined as follows. For any TCP session, the starting octet number is not necessarily 1, but specified by the sender in its first message (that associated with a `syn` message). For a trace σ, let ISn_σ, the initial sequence number of σ, be this first octet number received in σ. Then L_{SNA} consists of all traces σ such that for any σ_1, σ_2 with $\sigma = \sigma_1 a \sigma_2$ and a an ack message, we have that the sequence number sent in a, $SN(a) = m_{\sigma_1} + 1$ where m_{σ_1} is the maximum n such that octets $ISn_{\sigma_1}, ISn_{\sigma_1} + 1, \ldots, n$ have been received in σ_1. Recall that $I \models SNA$ iff $[\![I]\!] \subseteq L_{SNA}$. The language L_{SNA} is prefix-closed but not suffix-closed because octets of the transmitted stream do not necessarily arrive in increasing order. Consider a trace in which octets with sequence numbers 3, 1, and 2 are received in order after which the subsequent ack would have sequence number 4. If we consider a suffix of this trace that does not include the first message containing octet number 3, then this suffix would fail to satisfy this property. Therefore, a passive testing algorithm that tests for membership in L_{SNA} would reject some correct implementations. Our solution to passively testing whether an implementation satisfies the property *SNA* is to define a prefix- and suffix-closed language L with $L_{SNA} \subseteq L$ and to test membership in L. In other words, we check for a passively testable property that is weaker than *SNA*. The precise extent to which this is a complete test for the property *SNA* is given by the following additional properties that our definition of L will ensure.

Proposition 2. *1. For any trace σ, we have that $\sigma \in L$ iff there exists a σ' with $\sigma' \cdot \sigma \in L_{SNA}$.*

2. Suppose that $\sigma \in L \cap L_{RAck}$. Then there exists a prefix σ_1 of σ such that for any trace σ' we have that if $\sigma' \cdot \sigma_1 \in L_{SNA}$ then $\sigma' \cdot \sigma \in L_{SNA}$. If the only prefix σ' such that $\sigma' \cdot \sigma \in L_{SNA}$ is $\sigma' = \epsilon$ then the prefix $\sigma_1 = \epsilon$.

The first property states that the traces accepted are exactly those that are observable from a correct implementation. The second property states that for any accepted trace (which meets the requirement of acknowledgement responsiveness), the execution satisfies the property *SNA* provided that the unobserved prefix and a bounded initial prefix of the observed trace is correct. In other words, for an accepted trace, any faults are restricted to the unobserved part of the execution and some bounded initial segment. The bounded initial segment has the additional property that it is empty if the trace observed is complete. By the first property, a test for membership in L is a complete passive testing algorithm for *SNA*. Although L_{SNA} itself is not prefix-,suffix-closed, by the second property, we have an exact delimitation of the scope of faults (with respect to *SNA*) for any accepted trace (with respect to a test for membership in L). Due to space considerations, we omit the details of the construction of L.

The analysis of TCP aspects related to adaptive retransmission and timer backoffs, and congestion windows and congestion control are analogous to that for acknowledgement sequence numbers just detailed. Besides the standard mechanisms of multiplicative decrease and slow-start recovery or additive increase, this analysis also applies to congestion control mechanisms and TCP options more recently proposed such as TCP-Reno and SACK [2, 5] which can also be shown to be passively-testable in a manner that allows exact delineation of the scope of faults.

5.2 Multiplicity of Sessions and Properties

We have shown how all aspects of the TCP protocol, except for conditions on the initial messages for establishing a TCP session, can be passively tested with complete coverage assuming we consider each of them in isolation and assuming that a client participates in only one TCP session. In this section, we extend our results to remove this restriction on the client.

The assumption of only one TCP session is tantamount to requiring that the monitor is given a trace obtained by filtering from the actual trace only segments belonging to a particular session. The property of a segment belonging to a session is a propositional property (it is a condition on the `source port` and `destination port` fields of any individual segment) — we therefore begin by formalizing the notion of filtering a trace with respect to a propositional property, which we call *projection.*

Definition 9 (Projections). *Let φ be any propositional property over the set of atomic propositions $\mathcal{A}$. We define a projection function $|\cdot|_\varphi : (\mathcal{P}(\mathcal{A}))^* \to (\mathcal{P}(\mathcal{A}))^*$ as follows:*

$$|\epsilon|_\varphi = \epsilon$$
$$|a \cdot \sigma|_\varphi = \begin{cases} a \cdot |\sigma|_\varphi & \text{if } a \models \varphi \\ |\sigma|_\varphi & \text{otherwise} \end{cases}$$

The class of prefix-, suffix-closed languages is closed under inverse projections and positive boolean combinations.

Proposition 3. *1. Suppose that L is a language that is prefix-closed and suffix-closed. For any propositional property φ, we have that the language $L_\varphi^{-1} = \{\sigma \mid |\sigma|_\varphi \in L\}$ is prefix-closed and suffix-closed.*

2. Suppose that $\{L_i\}_{i\in\mathcal{C}}$ is a family of prefix-closed, suffix-closed languages. Then both $\bigcup_{i\in\mathcal{C}} L_i$ and $\bigcap_{i\in\mathcal{C}} L_i$ are prefix-closed and suffix-closed.

The two parts of Proposition 3 together with the results argued in Section 5.1 immediately yield a powerful array of TCP properties that can be passively tested with complete coverage. Because the property of belonging to a particular TCP session is propositional, by using projections and conjunction it follows that checking a trace for being faithful to a TCP specification for all sessions in it is passively testable. Because both conjunction and disjunction preserve the properties of being prefix and suffix-closed, we can check for arbitrary combinations of TCP implementation requirements and impose independent requirements on each session. For example, the property that all TCP sessions follow either slow-start or TCP-Reno or the property that a particular session follow multiplicative increase and another has ECN notification, is passively testable as well since each of these properties is a positive boolean combination of inverse projections of prefix-,suffix-closed properties.

6 Conclusions

Our contributions to the understanding of passive testing are as follows. Due to the nature of passive testing, one cannot, in general, detect all faulty executions with respect to some correctness requirement. For an arbitrary behavior requirement, we have captured the notion of a passive testing algorithm detecting as many faulty executions as possible through the formal criteria of soundness and completeness. Using these formulations, we have analyzed the homing algorithm and formally proven it to be the best possible algorithm for passively testing conformance to a specification automaton, with respect to several conformance relations. We have provided necessary and sufficient conditions on a fault notion for it to be completely detectable via passive testing. For such fault notions, we have presented a more efficient algorithm than the homing algorithm for passively testing them with complete fault coverage. Finally, we have analyzed the TCP specification and shown that most of it is completely checkable through passive testing.

In addition to the works already referenced in the introduction, the other closely related work is [1], where monitoring algorithms are proposed that take into account infidelity of observed traces due to buffering and packet loss. The work, however, assumes that the device under test is in its initial state when monitoring begins. The algorithms proposed are sound (*i.e.*, rejected executions are indeed faulty), but analysis of their adequacy, in the sense of Definition 4, is not addressed, and completeness of fault coverage is not considered either. On the other hand, the work presented here does not account for buffering or packet loss and it would be interesting to consider the impact of these issues in our results. Another study, conducted in a different context but close in spirit is that of [15], where it is informally argued that the class of security policies that are enforceable by monitoring system executions are exactly the safety properties. However, the monitoring mechanisms considered there also observe the system from its initial state. In this work, we have considered the setting where monitoring begins mid-stream and our characterizations have been formally proven.

Some future directions for extending this work are as follows. Our work has shown that the condition of prefix- and suffix-closure identifies an important class of languages and correctness properties. It would, therefore, be useful to provide techniques for facilitating the specification of properties satisfying this closure condition. To provide specifications declaratively as logical formulas, this would translate to formulating a logic whose expressive power is exactly the prefix and suffix closed properties. This could be done, for example, by characterizing the structure of formulas in temporal logic that yield this closure condition, as is done in [16] for safety and liveness properties. In the context of specifying the desired behavior operationally as an automaton, one could look for an algebraic characterization of prefix- and suffix-closed languages as suitable varieties of semigroups, in the style of [12]. Finally, in addition to fault detection, an important part of fault management is fault location which identifies the faulty local site which is responsible for the observed faulty global execution, and it would be interesting to examine systematic formal approaches to the fault location problem.

Acknowledgements

The authors would like to thank Richard Buskens, Patrice Godefroid, and Raymond Miller for many helpful comments on earlier drafts of this paper.

References

1. Karthikeyan Bhargavan, Satish Chandra, Peter J. McCann, and Carl A. Gunter. What packets may come: automata for network monitoring. In *Symposium on Principles of Programming Languages*, pages 206–219, 2001.
2. S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm, April 1999. RFC2582.
3. D. Lee, D. Chen, R. Hao, R. Miller, J. Wu, and X. Yin. A formal approach for passive testing of protocol data portions. In *Proceedings of the IEEE International Conference on Network Protocols*, pages 122–131, 2002.
4. D. Lee, A. Netravali, K. Sabnani, B. Sugla, and A. John. Passive testing and applications to network management. In *Proceedings of the IEEE International Conference on Network Protocols*, pages 113–122, October 1997.
5. M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgement Options, October 1996. RFC2018.
6. R.E. Miller. Passive testing of networks using a CFSM specification. In *IEEE International Performance Computing and Communications Conference*, pages 111–116, February 1998.
7. R.E. Miller and K. Arisha. On fault location in networks by passive testing. In *IEEE International Performance Computing and Communications Conference*, February 2000.
8. R.E. Miller and K. Arisha. Fault identification in networks by passive testing. In *IEEE Advanced Simulation Technologies Conferences*, April 2001.
9. Jitendra Padhye and Sally Floyd. On Inferrring TCP Behavior. In *SIGCOMM 2001*, August 2001.
10. V. Paxson. Automated packet trace analysis of tcp implementations. *Computer Communication Review*, 27(4), October 1997.
11. V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, December 1999.
12. Jean-Eric Pin. Finite semigroups and recognizable languages: An introduction. In J. Fountain, editor, *NATO Advanced Study Institute Semigroups, Formal Languages and Groups*, pages 1–32. Kluwer academic publishers, 1995.
13. M.J. Ranum, K. Landfield, M. Stolarchuk, M. Sienkiewicz, A. Lambeth, and E. Wall. Implementing a generalized tool for network monitoring. In *Proceedings of the Eleventh Systems Administration Conference (LISA XI)*, pages 1–8, 1997.
14. E. C. Rosen. Vulnerabilities of network control protocols: An example. *ACM SIGSOFT Software Engineering Notes*, 6(1), January 1981.
15. Fred B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.
16. A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6:495–511, 1994.
17. W. Richard Stevens. *TCP/IP Illustrated, Volume I: The Protocols.* Addison Wesley, Reading, Massachusetts, 1984.
18. P. Travis. Why the AT&T Network Crashed. *Telephony*, 218(4), January 1990.