Passive Testing with Asynchronous Communications

Robert M. Hierons, Mercedes G. Merayo, Manuel Núñez

▶ To cite this version:

Robert M. Hierons, Mercedes G. Merayo, Manuel Núñez. Passive Testing with Asynchronous Communications. 15th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOOODS) / 33th International Conference on Formal Techniques for Networked and Distributed Systems (FORTE), Jun 2013, Florence, Italy. pp.99-113, 10.1007/978-3-642-38592-6_8 . hal-01515240

HAL Id: hal-01515240
https://inria.hal.science/hal-01515240

Submitted on 27 Apr 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Passive Testing with Asynchronous Communications*

Robert M. Hierons[1], Mercedes G. Merayo[2], and Manuel Núñez[2]

[1] Department of Information Systems and Computing, Brunel University
Uxbridge, Middlesex, UB8 3PH United Kingdom,
rob.hierons@brunel.ac.uk
[2] Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Madrid, Spain,
mgmerayo@fdi.ucm.es,mn@sip.ucm.es

Abstract. Testing is usually understood to involve the tester interacting with the studied system by supplying input and observing output. However, sometimes this *active* interaction is not possible and testing becomes more *passive*. In this setting, passive testing can be considered to be the process of checking that the observations made regarding the system satisfy certain required properties. In this paper we study a formal passive testing framework for systems where there is an asynchronous communications channel between the tester and the system. We consider a syntactic definition of a class of properties and provide a semantic representation, as automata, that take into account the different observations that we can expect due to the assumption of asynchrony. Our solution checks properties against traces in polynomial time, with a low need for storage. Therefore, our proposal is very suitable for real-time passive testing.

1 Introduction

Testing is widely used to increase the reliability of complex systems. Traditionally, testing of software had a weak formal basis, in contrast with testing of hardware systems where different formalisms and formal notations were used from the beginning [23, 10]. However, it has been recognised that *formalising* the different aspects of testing of software is very beneficial [7]. The combination of formal methods and testing is currently well understood, tools to automate testing activities are widely available, there are several surveys on the field [14, 13], and industry is becoming aware of the importance of using formal approaches [8].

Usually, testing consists of applying stimuli (inputs) to the system and deciding whether the observed reactions (outputs) are those expected. However, we might be required to assess a particular system but without having access to it. The reasons for these limitations might be due to security issues or because the

* Research partially supported by the Spanish projects TESIS and ESTuDIo (TIN2009-14312-C02 and TIN2012-36812-C02) and the UK EPSRC project Testing of Probabilistic and Stochastic Systems (EP/G032572/1).

system is running 24/7 and our interaction might produce undesirable changes in the associated data. In these situations, testing can still play a role, but becomes more *passive* since interaction will be replaced by observation. Formal passive testing is already a well established line of research and extensions of the original frameworks [21, 4, 2] have dealt with issues such as security and time [24, 22, 1]. Essentially, in passive testing we have a property and we check that the trace being observed satisfies that property. Ideally, we want the process of checking whether the property is satisfied to be quick and to take very little storage since this can allow passive testing to occur in real-time. The application of passive testing in real-time has an important benefit: a detected error can be notified to the operators of the system almost immediately and they can then take appropriate measures. If the trace needs to be saved and processed off-line, then the time between the detection of the error and the corresponding notification will significantly increase.

One possible approach to work with properties and traces is to represent the property P as an automaton $M(P)$ such that a trace satisfies P if and only if it is not a member of the language defined by $M(P)$: it does not reach a final (error) state of $M(P)$. If $M(P)$ is deterministic then the process of checking whether a trace satisfies P takes linear time and is an incremental process: every time we observe a new input or output we simply update the state of $M(P)$. Even if $M(P)$ is non-deterministic, the process of checking whether a trace satisfies P takes time that is linear in the size of $M(P)$ and the length of the trace and the process is still incremental. This makes such an automaton based approach desirable if we can find efficient ways of mapping properties to automata.

Previous work on passive testing has assumed that the monitor that observes and checks the behaviour of the System Under Test (SUT) observes the actual trace produced. However, the monitor might not directly observe the interface of the SUT and instead there may be an asynchronous channel/network between the monitor and the SUT. Where this is the case, the trace observed by the monitor might not be the one produced by the SUT: input is observed before it is received by the SUT and output is observed after it is sent by the SUT. Suppose, for example, that the monitor observes the trace $?i?i!o$ in which $?i$ is an input and $!o$ is an output. In this case it is possible that the SUT actually produced either $?i!o?i$ or $!o?i?i$ and that the observation of $?i?i!o$ was due to the delaying of output. Thus, if we have properties that the SUT should satisfy and we directly apply them in such a context then we may obtain false positives or false negatives. We therefore require new approaches to passive testing in such circumstances and in this paper we focus on the case where the asynchronous channels are first in first out (FIFO).

There are several ways of applying passive testing when observations are through FIFO channels. One approach creates a model of our property and adds queues to this. However, the addition of queues can lead to the model requiring more storage space. In particular, if the queues are not bounded then it leads to there being an infinite number of states while if there is a bound on the queue length then the number of states increases exponentially with this

bound. An alternative is to transform the trace ρ observed to form an automaton M_ρ that represents all traces that might lead to ρ being observed where there is asynchronous communications. However, under this approach we have to analyse the entire, potentially very long, trace that has been observed and the monitor has to store this. This approach thus mitigates against real-time uses since it can significantly increase the storage and processing requirements. In this paper we explore a third alternative, in which for a property P we produce an automaton $\mathcal{A}(P)$ such that we check whether a trace ρ observed is a member of the language defined by $\mathcal{A}(P)$.

We are not aware of previous work on passive testing where there is an asynchronous communications channel between the system and the monitor. In contrast, there has been some work on active testing and several approaches have appeared in the literature for models where there is a distinction between inputs and outputs [11, 20, 25, 12]. Passive testing is a monitoring technique and as such it is related to runtime verification since they share the same goal, checking the correctness of a system without interacting with it, but use different formalisms and methodologies. In runtime verification it is not usual to distinguish between inputs and outputs, since their observation makes them events of the same nature, and therefore it is difficult to compare the work from that area with ours. While some work has investigated asynchronous runtime monitoring, the problems considered in this context are different: this line of work does not distinguish between input and output and does not explore potential reorderings of traces. Instead, it looks at the situation in which the monitor and system do not synchronise on actions: actions engaged in by the system might instead be recorded and analysed later, with a compensation phase being used to undo any later actions if an error is found [5].

The rest of the paper is structured as follows. In Section 2 we introduce notation to define systems and traces that will be used throughout the paper. Section 3 introduces the notion of an ideal that will be used in creating automata from observations. Section 4 is the bulk of the paper and presents how properties can be translated into automata and provides our theory to check traces against properties. Finally, in Section 5 we present our conclusions and provide some lines for future work.

2 Preliminaries: systems and observations

In this section we introduce the basic notion used in this paper to define systems as well as concepts associated with the traces that a system can perform and with the traces that a monitor can actually observe in an asynchronous setting.

Definition 1. *An* input-output transition system (IOTS) $M = (Q, I, O, T, q_{in})$ *is a tuple in which Q is a countable set of states, $q_{in} \in Q$ is the initial state, I is a countable set of inputs, O is a countable set of outputs, and $T \subseteq Q \times (I \cup O) \times Q$ is the transition relation. A transition $(q, a, q') \in T$ means that from state q it is possible to move to state q' with action $a \in I \cup O$. We use the following notation concerning the performance of (sequences of) actions.*

- *$\mathcal{A}ct = I \cup O$ is the set of actions.*
- *If $(q, a, q') \in T$, for $a \in \mathcal{A}ct$, then we write $q \xrightarrow{a} q'$ and $q \xrightarrow{a}$.*
- *We write $q \xRightarrow{\sigma} q'$ for $\sigma = a_1 \ldots a_m \in \mathcal{A}ct^*$, with $m \geq 0$, if there exist $q_0, \ldots, q_m$, $q = q_0$, $q' = q_m$ such that for all $0 \leq i < m$ we have that $q_i \xrightarrow{a_{i+1}} q_{i+1}$. Note that $q \xRightarrow{\epsilon} q$, where ϵ is the empty sequence.*
- *If there exists q' such that $q_{in} \xRightarrow{\sigma} q'$ then we say that σ is a* trace *of M and we write $M \xRightarrow{\sigma}$. We let $L(M)$ denote the set of traces of M.*

We have an asynchronous setting and, therefore, we do not have to consider only the traces that can be performed by a system but also how these traces can be observed. Intuitively, if a system performs a certain trace then we can observe a variation of this trace where the outputs appear later than they were actually performed. Next we formally define this idea and given a system M and a trace σ we let $\mathcal{L}(\sigma)$ denote the set of traces that might be observed by a monitor if M produces trace σ and communications between the monitor and the SUT are asynchronous and FIFO.

Definition 2. *Let I and O be sets of inputs and outputs, respectively, and $\sigma, \sigma' \in \mathcal{A}ct^*$ be sequences of actions. We say that σ' is an observation of σ, denoted by $\sigma \rightsquigarrow \sigma'$, if there exist sequences $\sigma_1, \sigma_2 \in \mathcal{A}ct^*$, $!o \in O$ and $?i \in I$ such that $\sigma = \sigma_1 !o ?i \sigma_2$ and $\sigma' = \sigma_1 ?i !o \sigma_2$. We let $\mathcal{L}(\sigma)$ denote the set of traces that can be formed from σ through sequences of transformations of the form $\rightsquigarrow$, that is, $\mathcal{L}(\sigma) = \{\sigma' | \sigma \rightsquigarrow^* \sigma'\}$, where $\rightsquigarrow^*$ represents the repeated application of $\rightsquigarrow$. We overload this to say that given an IOTS M, $\mathcal{L}(M) = \cup_{\sigma \in L(M)} \mathcal{L}(\sigma)$ is the set of traces that might be observed when interacting with M through asynchronous FIFO channels.*

Example 1. Assume that the SUT has produced the trace $\sigma = ?i_1 !o_1 !o_2 ?i_2 !o_1$. Due to the asynchronous nature of the system, the monitor might observe any of the traces in the set $\mathcal{L}(\sigma) = \{?i_1 !o_1 !o_2 ?i_2 !o_1, ?i_1 !o_1 ?i_2 !o_2 !o_1, ?i_1 ?i_2 !o_1 !o_2 !o_1\}$.

3 Sets of events from observations and ideals

In line with previous work in formal passive testing [2], we will consider properties of the form (σ, O_σ) for $\sigma \in \mathcal{A}ct^*$ and $O_\sigma \subseteq O$. Such a property says that if the SUT produces the sequence σ then the next output must from the set O_σ. It is straightforward to devise an automaton $M(P)$ that accepts only the traces that do not satisfy such a property P: we define $M(P)$ such that it accepts the regular language $\mathcal{A}ct^*\{\sigma\}(\mathcal{A}ct \setminus O_\sigma)\mathcal{A}ct^*$. It is easy to check that $M(P)$ accepts a trace ρ if and only if ρ does not satisfy this property: ρ contains a subsequence that has σ followed by an action that is not in O_σ. It is well known that an automaton that represents a regular expression can be produced in quadratic time [3]. This process can be further improved to achieve sub-quadratic complexity and can be efficiently parallelised to work in $O(\log(|\sigma|))$ time [9]. Such an automaton $M(P)$ has $O(|\sigma|)$ states and $O(|\sigma| \cdot \log(|\sigma|)^2)$ transitions [18]. In the next section we adapt the above approach for the case where communications are asynchronous.

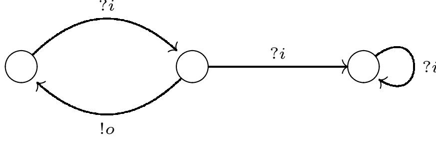

Fig. 1. An IOTS M such that $\mathcal{L}(M)$ is not regular

Before outlining our solution, we briefly comment on some alternatives. Previous work has described a delay operator that takes a trace σ of an IOTS and returns the set of traces that might be observed if the SUT produces σ and interacts asynchronously through FIFO channels with its environment [19]. However, the delay operator cannot be applied directly since it applies to a single trace rather than an automaton $M(P)$. While it has been shown how a test purpose can be adapted to incorporate the delay into the verdict [25], this approach assumes that the tester waits for output before sending the next input and so does not apply input in a state where output can be produced. Since the input is not supplied by the tester in passive testing, we cannot make such an assumption. We might instead aim to define a general method that takes an IOTS M (with finite sets of states and transitions) and produces IOTS M' (with finite sets of states and transitions) with $L(M') = \mathcal{L}(M)$. If we can achieve this then M' can be used. However, the following shows that there is no such general method.

Proposition 1. *Given an IOTS M with finite sets of states and transitions, there may be no IOTS M' with finite sets of states and transitions such that $L(M') = \mathcal{L}(M)$.*

Proof. An IOTS with finite sets of states and transitions defines a regular language so it is sufficient to find some such M where $\mathcal{L}(M)$ is not a regular language. Let M be the IOTS with three states shown in Figure 1 (the initial state is represented by the leftmost vertex). We will use proof by contradiction, assuming that $\mathcal{L}(M)$ is a regular language. Thus, since $\mathcal{L}(M)$ is regular and I^*O^* is regular we have that $\mathcal{L}(M) \cap (I^*O^*)$ is regular. However, $\mathcal{L}(M) \cap (I^*O^*)$ contains all sequences of the form of n inputs followed by n or fewer outputs and this is not a regular language. This provides a contradiction as required.

While this result shows that there is no general method that takes a property P defined by an IOTS $M(P)$ with finite sets of states and transitions and returns a suitable property for use when communications are asynchronous, we will see in the next section that we can take advantage of the structure of the properties we consider. First we will show how, for trace σ, we can produce an automaton $\mathcal{A}^T(\sigma)$ that gives the set of traces that might be observed if the SUT produces σ. Since we are applying passive testing, a trace σ of interest might not be the start of the overall trace observed and so we will then adapt $\mathcal{A}^T(\sigma)$ to produce the automaton $\mathcal{A}(\sigma, O_\sigma)$ that will be used.

Given a sequence σ, we will define a partial order $\ll$ on the inputs and outputs in σ to represent which actions must be performed before other ones if the SUT produces σ. In order to distinguish between repeated actions in the trace, events are constructed from actions by labelling each action in σ with the occurrence of the symbol in the trace.

Definition 3. *Let $\sigma = a_1 \dots a_n \in \mathcal{A}ct^*$ be a sequence of actions. We let $E(\sigma)$ denote the set of events of σ, where $e = (a_i, k)$ belongs to $E(\sigma)$ if and only if there are exactly $k-1$ occurrences of a_i in $a_1 \dots a_{i-1}$. This says that the ith element of σ is the kth instance of a_i in σ.*

Example 2. Consider the trace $\sigma = ?i_1!o_1!o_2?i_2!o_1$. The corresponding set of events is $E(\sigma) = \{(?i_1, 1), (!o_1, 1), (!o_2, 1), (?i_2, 1), (!o_1, 2)\}$.

Definition 4. *Let $\sigma \in \mathcal{A}ct^*$ be a sequence of actions. Given two events $e_i = (a_i, k_i)$ and $e_j = (a_j, k_j)$ belonging to $E(\sigma)$, we write $e_i \ll e_j$ if either $i = j$ or $i < j$ and one of the following conditions hold: a_i and a_j are inputs, or a_i and a_j are outputs, or a_i is an input and a_j is an output.*

The first two cases in the definition of $\ll$ result from channels being FIFO. The last case results from the observation of outputs being delayed, while an input is observed before it is received by the SUT. Essentially, $(a_i, k_i) \ll (a_j, k_j)$ does not hold for $i < j$ if a_i is an output and a_j is an input since in this case it is possible that the observation of output a_i is delayed until after input a_j has been sent. Given a trace $\sigma \in \mathcal{A}ct^*$ it is straightforward to prove that $(E(\sigma), \ll)$ is a partially ordered set. Next we introduce the notions of an *ideal* and anti-chains of a set of events.

Definition 5. *Let $\sigma \in \mathcal{A}ct^*$ be a sequence of actions and $E(\sigma)$ be the set of its events, possibly annotated to avoid repetitions. A set $\mathcal{I} \subseteq E(\sigma)$ is said to be an* ideal *of $(E(\sigma), \ll)$ if for all $e_i, e_j \in E(\sigma)$, if $e_i \ll e_j$ and $e_j \in \mathcal{I}$ then $e_i \in \mathcal{I}$. An ideal $\mathcal{I}$ is a* principal ideal *if there is some e_j such that $\mathcal{I}$ contains only e_j and all elements below it under $\ll$, that is, $\mathcal{I} = \{e_i \in E(\sigma) | e_i \ll e_j\}$. Finally, a set $E' \subseteq E(\sigma)$ is an* anti-chain *if no two different elements of E' are related under $\ll$.*

The essential idea is that if the SUT produces σ and e_i is a maximal element of ideal $\mathcal{I}$, then $\mathcal{I}$ includes all events that *must* be observed before e_i is observed by the monitor.

Example 3. Consider again the trace $\sigma = ?i_1!o_1!o_2?i_2!o_1$. We have that the following sets of events $\mathcal{I}_1 = \{(?i_1, 1), (!o_1, 1), (!o_2, 1)\}$, $\mathcal{I}_2 = \{(?i_1, 1), (?i_2, 1)\}$ and $\mathcal{I}_3 = \{(?i_1, 1), (!o_1, 1), (?i_2, 1)\}$ are ideals of $(E(\sigma), \ll)$. However, only $\mathcal{I}_1$ and $\mathcal{I}_2$ are principal ideals. The ideal $\mathcal{I}_3$ contains the events $(!o_1, 1)$ and $(?i_2, 1)$ that are not related under $\ll$, therefore, $\mathcal{I}_3$ is not a principal ideal. Finally, the sets $E_1 = \{(!o_1, 1), (?i_2, 1)\}$ and $E_2 = \{(!o_2, 1), (?i_2, 1)\}$ are anti-chains of $(E(\sigma), \ll)$.

Next we present an alternative characterisation of the notion of ideal that shows that an ideal is defined by its maximal (under $\ll$) elements.

Lemma 1. *Let $\sigma \in \mathcal{A}ct^*$ be a sequence of actions. We have that $\mathcal{I} \subseteq E(\sigma)$ is an ideal if and only if one of the following conditions holds:*

- *$\mathcal{I}$ contains an input a_i and all earlier inputs;*
- *$\mathcal{I}$ contains an output a_j and all earlier inputs and outputs; or*
- *$\mathcal{I}$ contains an input a_i, an output a_j, all inputs before a_i, and all inputs and outputs before a_j.*

The following classical result [6] relates ideals and anti-chains.

Proposition 2. *The set of ideals is isomorphic to the set of anti-chains, by associating with every anti-chain E' the ideal which is the union of the principal ideals generated by the elements of E'. Vice versa, the anti-chain corresponding to a given ideal $\mathcal{I}$ is the set of maximal elements of $\mathcal{I}$.*

The following result provides a measure, in the worst case, on the number of ideals contained in a set of events. This result will be relevant since it will be used to calculate the complexity of the algorithm that computes the automaton associated with a certain property P.

Proposition 3. *Let $\sigma \in \mathcal{A}ct^*$ be a sequence of actions with length m. There are $O(m^2)$ ideals in $E(\sigma)$.*

Proof. By Proposition 2 we know that the number of ideals is the same as the number of anti-chains. However, we also know that any two inputs in $E(\sigma)$ are related under $\ll$. Similarly, any two outputs in $E(\sigma)$ are related under $\ll$. Thus, an anti-chain can have at most two elements (one input and one output) and so there are $O(m^2)$ anti-chains. The result therefore holds.

An ideal $\mathcal{I}$ is a set of elements from $E(\sigma)$ such that all 'earlier' elements, under $\ll$, are contained in $\mathcal{I}$. Ideal $\mathcal{I}$ of $(E(\sigma), \ll)$ is therefore one possible set of events that might be observed, as the prefix of a trace from $\mathcal{L}(\sigma)$, if the SUT produces σ. Thus, an ideal $\mathcal{I}$ defines a set of events in one or more prefixes of a trace from $\mathcal{L}(\sigma)$. Similarly, the events in a prefix of a trace from $\mathcal{L}(\sigma)$ form an ideal of $E(\sigma, \ll)$. As a result, we can reason about prefixes of traces in $\mathcal{L}(\sigma)$ by considering the ideals of $(E(\sigma), \ll)$.

4 Creating automata for properties

In this section we show how the ideals associated with a certain trace can be used to construct appropriate automata. More specifically, given a sequence of actions σ, we will use the ideals of $(E(\sigma), \ll)$ to represent states of a finite automaton $\mathcal{A}^T(\sigma)$ that accepts the set of sequences in $\mathcal{L}(\sigma)$. We will study properties of these automata and the time complexity of using them in passive testing.

Definition 6. *Given non-empty $\sigma \in \mathcal{A}ct^*$ we let $\mathcal{A}^T(\sigma)$ denote the finite automaton with state set S that is equal to the set of ideals of $(E(\sigma), \ll)$, alphabet $\mathcal{A}ct$, initial state $\{\}$ and the following set of transitions: given ideal $\mathcal{I}$ and $a \in \mathcal{A}ct$, there is a transition $t = (\mathcal{I}, a, \mathcal{I}')$ for ideal $\mathcal{I}'$ if and only if $\mathcal{I}' = \mathcal{I} \cup \{a\}$. In addition, $\mathcal{A}^T(\sigma)$ has one final state, which is the ideal $E(\sigma)$.*

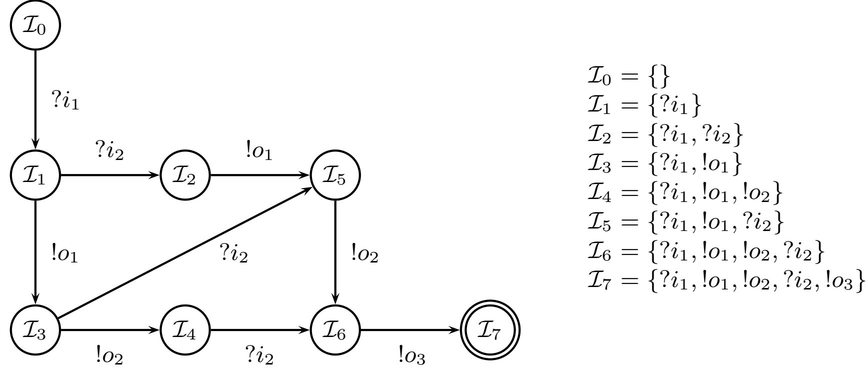

Fig. 2. Automaton $\mathcal{A}^T(\sigma)$ for the trace $\sigma = ?i_1!o_1!o_2?i_2!o_3$

A finite automaton A is essentially an IOTS with finite sets of states, inputs and outputs and a set of final states. Then A defines the regular language $L(A)$ of labels of walks from the initial state of A to final states of A. Note that an IOTS with finite sets of states and actions can be seen as a finite automaton where all the states are final.

Example 4. Let $\sigma = ?i_1!o_1!o_2?i_2!o_3$ be a trace. Figure 2 depicts the automaton $\mathcal{A}^T(\sigma)$ that accepts the set of sequences in $\mathcal{L}(\sigma)$.

We have that $\mathcal{A}^T(\sigma)$ defines the set of behaviours, $\mathcal{L}(\sigma)$, that can be observed if the SUT produces σ.

Proposition 4. *Given $\sigma \in \mathcal{A}ct^*$ we have that $L(\mathcal{A}^T(\sigma)) = \mathcal{L}(\sigma)$.*

Proof. We will prove a slightly stronger result, which is that ρ labels a walk from the initial state of $\mathcal{A}^T(\sigma)$ if and only if ρ is a prefix of a sequence in $\mathcal{L}(\sigma)$.

We first prove the left to right implication by induction on the length of ρ. The result clearly holds for the base case in which ρ is the empty sequence. Now assume that it holds for all sequences of length less than k, $k \geq 1$, and ρ has length k. Thus, $\rho = \rho_1 a$ for some $a \in I \cup O$. By the inductive hypothesis we have that ρ_1 is a prefix of a sequence in $\mathcal{L}(\sigma)$. In addition, by the definition of $\mathcal{A}^T(\sigma)$, we have that the set of events in ρ_1 forms an ideal $\mathcal{I}_1$ and $\mathcal{I}_1 \cup \{a\}$ is an ideal. Thus, since $\mathcal{I}_1 \cup \{a\}$ is an ideal, there does not exist $b \in E(\sigma) \setminus (\mathcal{I}_1 \cup \{a\})$ such that $b \ll a$. By the definition of $\mathcal{L}(\sigma)$ we have that ρ_1 can be followed by a and so $\rho_1 a$ is a prefix of a sequence in $\mathcal{L}(\sigma)$ as required.

We now prove the right to left implication, again, by induction on the length of ρ. The result clearly holds for the base case in which ρ is the empty sequence. Now assume that it holds for all sequences of length less than k, $k \geq 1$, and ρ has length k. Thus, $\rho = \rho_1 a$ for some $a \in I \cup O$. By the inductive hypothesis we have that ρ_1 is the label of a walk of $\mathcal{A}^T(\sigma)$ and assume that this walk reaches a state representing ideal $\mathcal{I}_1$. By the definition of $\mathcal{L}(\sigma)$ there cannot exist an action in σ that is not in ρ_1 and that must be observed before a and so precedes a under $\ll$.

Algorithm 1 Producing $\mathcal{A}(\sigma, O_\sigma)$

1: Input (σ, O_σ).
2: Let $\mathcal{A}(\sigma, O_\sigma) = \mathcal{A}^T(\sigma)$, let s_0 denote the initial state of $\mathcal{A}(\sigma, O_\sigma)$, and let s_f denote the final state of $\mathcal{A}(\sigma, O_\sigma)$.
3: For all $a \in \mathcal{A}ct$ add the transition (s_0, a, s_0). *These transitions ensure that we are considering all possible starting points in a trace ρ' observed.*
4: For every state s of $\mathcal{A}(\sigma, O_\sigma)$ that represents an ideal that does not contain output and for all $!o \in O$, add the transition $(s, !o, s)$. *These transitions correspond to the possibility of earlier output being observed after input from σ.*
5: For every state s of $\mathcal{A}(\sigma, O_\sigma)$ that represents an ideal that contains all of the input from σ and for all $?i \in I$, add the transition $(s, ?i, s)$. *These transitions correspond to the possibility of later input being observed before some of the output from σ.*
6: Add a new state s_e to $\mathcal{A}(\sigma, O_\sigma)$ and for all $!o \in O \setminus O_\sigma$ add the transition $(s_f, !o, s_e)$. *If we have observed the input and output from σ and the next output is not from O_σ then go to the final (error) state.*
7: Make s_e the only final state of $\mathcal{A}(\sigma, O_\sigma)$.
8: Complete A: if there is no transition from a state $s \neq s_e$ with label $a \in \mathcal{A}ct$ then add the transition (s, a, s_0).
9: Output $\mathcal{A}(\sigma, O_\sigma)$.

Thus, $\mathcal{I}_2 = \mathcal{I}_1 \cup \{a\}$ is an ideal and so $\mathcal{A}^T(\sigma)$ contains a transition from the state representing $\mathcal{I}_1$ to the state representing $\mathcal{I}_2$ with label a, concluding that $\rho = \rho_1 a$ is the label of a walk of $\mathcal{A}^T(\sigma)$ as required. The result therefore follows.

We now have to adapt $\mathcal{A}^T(\sigma)$ to take into account two points: σ might be preceded by other actions and the observation of earlier outputs might be delayed; and σ might be followed by later actions and the outputs from σ might not be observed until after later inputs. Algorithm 1 achieves this.

Example 5. Let $\sigma = ?i_1!o_1!o_2?i_2!o_3$ and consider the automaton $\mathcal{A}^T(\sigma)$ depicted in Figure 2. Given a set of outputs O_σ, Figure 3 shows the automaton $\mathcal{A}(\sigma, O_\sigma)$ constructed by using Algorithm 1.

Before proving that Algorithm 1 returns the correct result, we define what it means for an automaton A to be sound: if the SUT produces a trace that does not satisfy property P then the trace observed by the monitor is in $L(A)$.

Definition 7. *Let P be a property and A be a finite automaton. We say that A is* sound *for P if and only if whenever the SUT produces a trace σ_1 that does not satisfy property P and the trace $\sigma_1' \in \mathcal{L}(\sigma_1)$ is observed we have that $\sigma_1' \in L(A)$.*

This essentially corresponds to the automaton not being able to produce false positives: if the SUT fails property P, then the automaton will produce an 'alarm'. The automaton produced by Algorithm 1 is sound.

Theorem 1. *Given property $P = (\sigma, O_\sigma)$, the automaton $\mathcal{A}(P)$ returned by Algorithm 1 when given P is sound for P.*

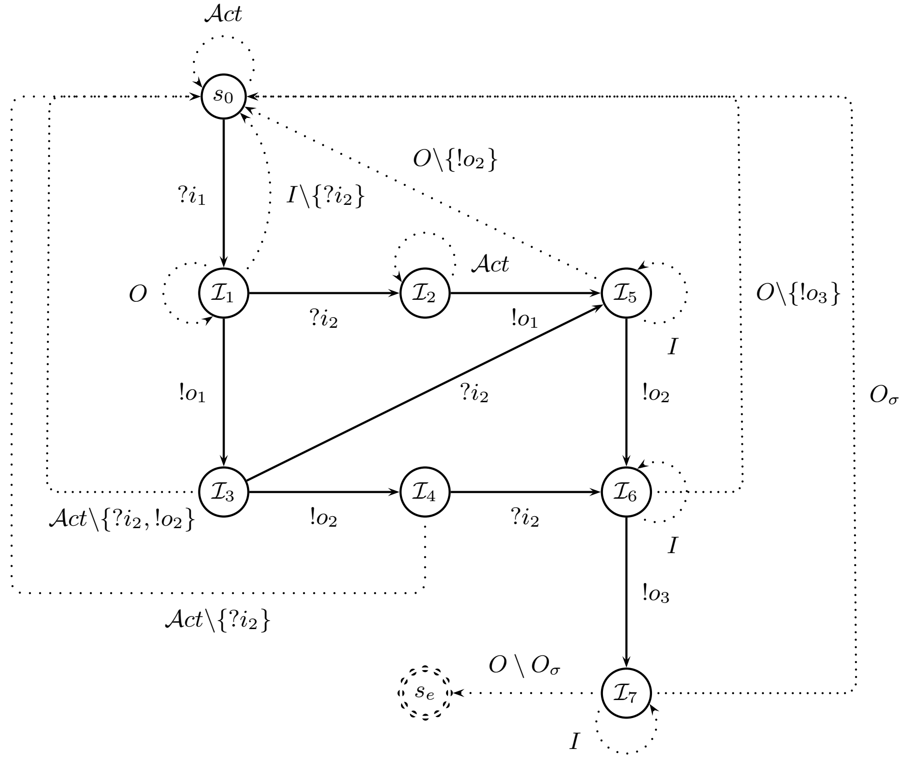

Fig. 3. Automaton $\mathcal{A}(\sigma, O_\sigma)$ for the trace $\sigma =?i_1!o_1!o_2?i_2!o_3$ and a set of outputs O_σ

Proof. Recall that we label actions using their occurrence, if necessary, so that they are unique and this labelling is preserved by the delay of output. We assume that the SUT has produced a trace σ_1 that does not satisfy P, that this led to the observation of the trace $\sigma_1' \in \mathcal{L}(\sigma_1)$ and we are required to prove that $\sigma_1' \in L(\mathcal{A}(\sigma, O_\sigma))$. Since σ_1 does not satisfy P we have that $\sigma_1 = \sigma_2 \sigma a \sigma_3$ for some $a \in O \setminus O_\sigma$. Since $\sigma_1' \in \mathcal{L}(\sigma_1)$ we have that $\sigma_1' = \sigma_2' \sigma' a \sigma_2'$ for some $\sigma', \sigma_2', \sigma_3'$ such that σ' satisfies the following.

- σ' starts with the first action of σ_1' that is from σ;
- σ' may contain outputs not in σ (delayed from σ_2);
- σ' may contain inputs not in σ (due to outputs from σ or a being delayed past inputs from σ_3); and
- σ' can have outputs from σ being delayed past inputs from σ.

By the definition of $\mathcal{A}(\sigma, O_\sigma)$ we have that the state of $\mathcal{A}(\sigma, O_\sigma)$ after σ_1' can be the initial state of $\mathcal{A}(\sigma, O_\sigma)$. Further, by Proposition 4 we know that σ' with the extra initial outputs and final inputs removed can take $\mathcal{A}(\sigma, O_\sigma)$ to the final state s_f. Consider the corresponding path ρ of $\mathcal{A}(\sigma, O_\sigma)$.

The additional outputs in σ' that are not in σ (and so come from σ_2) are all before the first output in σ' that was from σ and so, by construction, we can define a path ρ' that includes these by adding self-loops to ρ.

Similarly, the additional inputs in σ' that are not in σ (and so come from σ_3) are all after the last input in σ' that was from σ and so we can define a path ρ'' that includes these by adding self-loops to ρ'. Path ρ'' thus takes $\mathcal{A}(\sigma, O_\sigma)$ to state s_f and has label σ'. The result now follows from observing that a takes $\mathcal{A}(\sigma, O_\sigma)$ from state s_f to the final state and this final state cannot be left.

An automaton A being sound for P denotes an absence of false positives. However, we might also want the absence of false negatives: if the SUT produces a trace and the resultant observation is in $L(A)$ then the trace produced by the SUT must not have satisfied P. This is captured by the notion of *exact*.

Definition 8. *Let P be a property and A be a finite automaton. We say that A is* exact *for P if and only if whenever the SUT produces some trace σ_1 and the observed trace $\sigma'_1 \in \mathcal{L}(\sigma_1)$ is such that $\sigma'_1 \in L(A)$ we must have that σ_1 does not satisfy P.*

The automaton $\mathcal{A}(\sigma, O_\sigma)$, produced by our algorithm, need not be exact for (σ, O_σ) as the following example shows.

Example 6. Consider the property $P = (?i, \{!o\})$ and the observed trace $\sigma'_1 = ?i!o'!o$ which is in the language defined by the automaton $\mathcal{A}(?i, \{!o\})$. We can consider two examples for the trace σ_1 produced by the SUT.

- $\sigma_1 = !o'?i!o$ and so σ_1 satisfies P.
- $\sigma_1 = ?i!o'!o$ and so σ_1 does not satisfy P.

The above shows that an observation made might be consistent with both traces that satisfy P and traces that do not. Therefore, our automata might not be exact. Note that this is not a drawback of our theory, but a consequence of working within a framework where the available information does not allow us to reconstruct the trace that was originally performed by the system. Similar situations appear in other frameworks such as when testing systems with distributed interfaces [15, 16]. We are currently working on approaches that allow us to make stronger statements regarding the failure of a property. We discuss our preliminary ideas in the part of the next section devoted to future work.

Even though we cannot expect *exactitude*, we should be able to ensure that if the observed trace is one that might have resulted from a trace of the SUT that does not satisfy the property P then the observed trace is in $L(A)$. This is captured by the following notion.

Definition 9. *Let P be a property and A be a finite automaton. We say that A is* precise *for P if and only if whenever a trace σ'_1 is in $L(A)$ there is some trace σ_1 that does not satisfy P such that $\sigma'_1 \in \mathcal{L}(\sigma_1)$.*

The following result shows that Algorithm 1 returns an automaton that is precise for the considered property.

Theorem 2. *Given property $P = (\sigma, O_\sigma)$, the automaton $\mathcal{A}(P)$ returned by Algorithm 1 when given P is precise for P.*

Proof. Suppose that trace σ_1' is in $L(\mathcal{A}(\sigma, O_\sigma))$. By definition it is sufficient to prove that there exists some trace σ_1 that does not satisfy P such that $\sigma_1' \in \mathcal{L}(\sigma_1)$. Note that σ_1 does not satisfy P if and only if it has a prefix that ends in σa for some $a \in O \setminus O_\sigma$.

By the construction of $\mathcal{A}(\sigma, O_\sigma)$, since $\sigma_1' \in L(\mathcal{A}(\sigma, O_\sigma))$, we have that σ_1' has prefix $\sigma_2'\sigma_3'a$ such that σ_3' takes $\mathcal{A}(\sigma, O_\sigma)$ from state s_0 to s_f and $a \in O \setminus O_\sigma$. In addition, we must have that σ_3' differs from σ in only three ways:

– the addition of outputs before the outputs of σ, through self-loops in states that correspond to ideals that contain no output;
– the addition of inputs after the inputs of σ, through self-loops in states that correspond to ideals that contain all of the inputs from σ; and
– the delay in output from σ.

Thus, the input projection of σ_3' is the input projection of σ followed by some sequence σ_I' of inputs and the output projection of σ_3' is some sequence σ_O' of outputs followed by the output projection of σ. As a result, $\sigma_3'a \in \mathcal{L}(\sigma_O'\sigma\sigma_I'a)$. Thus, σ_1' has prefix $\sigma_2'\sigma_3'a$ such that $\sigma_2'\sigma_3'a \in \mathcal{L}(\sigma_2'\sigma_O'\sigma\sigma_I'a)$ for some $a \in O \setminus O_\sigma$. By definition, since σ_I' contains only inputs and a is an output, we have that $\mathcal{L}(\sigma_2'\sigma_O'\sigma\sigma_I'a) \subseteq \mathcal{L}(\sigma_2'\sigma_O'\sigma a\sigma_I')$. Therefore σ_1' has prefix $\sigma_2'\sigma_3'a$ such that $\sigma_2'\sigma_3'a \in \mathcal{L}(\sigma_2'\sigma_O'\sigma a\sigma_I')$ for some $a \in O \setminus O_\sigma$. Since $\sigma_2'\sigma_O'\sigma a\sigma_I'$ does not satisfy property P, we can set $\sigma_1' = \sigma_2'\sigma_O'\sigma a\sigma_I'$ and the result follows.

By Proposition 3 we know that $\mathcal{A}(\sigma, O_\sigma)$ has $O(|\sigma|^2)$ states. In addition, we can construct the relation $\ll$ and the set of anti-chains in $O(|\sigma|^2)$ time. The following is therefore clear.

Proposition 5. *Given property (σ, O_σ), the process of generating the automata $\mathcal{A}(\sigma, O_\sigma)$ takes $O(|\sigma|^2)$ time.*

In passive testing we can update the current state of $\mathcal{A}(\sigma, O_\sigma)$ whenever we make a new observation and thus the complexity of applying passive testing is linear in the length of the trace that the SUT is producing. The following shows that the process is polynomial in the length of σ, which suggests that it can be applied in real-time since properties, and so $|\sigma|$, are usually relatively small.

Proposition 6. *Given property (σ, O_σ), the process of updating the state of $\mathcal{A}(\sigma, O_\sigma)$ takes $O(|\sigma|^4)$ time when a new input or output is observed.*

Proof. At each point in the process of simulating $\mathcal{A}(\sigma, O_\sigma)$ with a trace we have a current set of states. Consider that a new action a is observed and the set of states before this is S'. We know that $\mathcal{A}(\sigma, O_\sigma)$ has $O(|\sigma|^2)$ states. Assume that we have a list of transitions sorted by label. Using a binary search we can locate the start of the transitions with label a in time that is of order of the logarithm of the number of transitions and so in $O(\log|\sigma|)$ time. We can then determine the set of states after a by including those that are reached from states in S' by transitions with label a. We can find the transitions that leave states in S' in $O(|\sigma|^2)$ time and for each such state we can determine in $O(|\sigma|^2)$ time which states can be reached by a. Thus, the overall time complexity is of $O(|\sigma|^4)$ time.

5 Conclusions and future work

Testing is widely used to increase the confidence regarding the correctness of a system. Testing activities can be *formalised* when the tester has a formal definition of the properties that the system must have. Testing is usually interactive: the tester provides inputs to the SUT, receives outputs, and determines whether the outputs match the expected result. However, in some situations the tester cannot interact with the SUT and testing becomes passive. In this case, passive testing techniques replace classical (active) testing ones. In this paper we studied a formal methodology to passively test systems where communication is asynchronous. This is a topic that, as far as we know, has received little attention in the formal passive testing community.

In order to use our methodology in real-time, increasing performance and decreasing the needs for storage, we transform properties into automata that *approximately* capture the original property. We proved that in general an asynchronous framework it is not possible to construct a finite automata that accepts only the traces matching the original property. In order to construct our automata we used a classical mathematical structure, called *ideals*, so that the observation of an action changes the current state of the automaton from one of the associated ideals to another. We proved that the main operations, updating automata and checking traces, can be done in polynomial time, reinforcing the suitability of our methodology for use in real-time. We also defined notions of soundness and precision. The automaton A is sound for property P if whenever the SUT produces a trace that does not satisfy P, the observation made leads the automaton A to a final state (indicating a failure). The automaton A is precise for property P if whenever the observation made leads the automaton A to a final state (indicating a failure) we must have that the observation was one that could be made by the SUT failing to satisfy P. It transpired that our approach returned an automaton that is sound and precise.

This paper is the first step towards a complete theory of formal passive testing of systems with asynchronous communication. A first line to continue our work consists in drawing stronger conclusions from our observations. As we previously said, we are working on two lines. The first one considers different classes of properties. We have started to define simple properties that refer only to sequences of inputs or only to sequences of outputs and ignore the other observations. For example, "if we see $!o$ then the next output must be $!o'$." Other types of properties, inspired by classical work on temporal logics, are "a sequence eventually happens", "a sequence never happens" and "if a sequence happens, then we shouldn't observe a certain action".

A second line of work considers the inclusion of time information taking as initial step our work on testing in the distributed architecture with time information [17]. As discussed above, in general we cannot be certain that a particular sequence σ was produced by the SUT even if we observe a trace in $\mathcal{L}(\sigma)$. However, suppose that we know that there can be at most time t_m between an output being produced and it being observed and between an input being observed and it being received by the SUT. Further, suppose that if the

SUT is in a state where it can produce output (possibly after a sequence of internal transitions) and it does so then this is within time t_o. Then, if time $2t_m + t_o$ passes without any observations being made then we know that the SUT must have reached a *quiescent* state: one from which it cannot produce output without first receiving input. If quiescence is observable then sometimes we can know that a property (σ, O_σ) has not been satisfied. There is additional scope to strengthen the conclusions that can be made. For example, if we have property $P = (\sigma, O_\sigma)$ where σ is an input sequence and we observe $\sigma\sigma'!o$ for input sequence σ' and output $!o \notin O_\sigma$ then we can conclude that the SUT failed P if there is a sufficient gap between the sending of the last input in σ and the output $!o$ being observed.

A third line of work considers more sophisticated mechanisms to improve the framework. First, we could have probabilities associated with *swapping* the order of actions and reasoning about how likely it is that an observation resulted from a property failing. Another improvement would be to consider stochastic information regarding delays. That is, probability distributions, rather than fix bounds, would be used to decide when we should expect that an action will not be observed in the future.

Finally, we would like to create a tool to support our theory and analyse realistic use cases to assess the applicability and usefulness of our framework.

Acknowledgements

We would like to thank the reviewers of the paper for the careful reading.

References

1. C. Andrés, M. G. Merayo, and M. Núñez. Formal passive testing of timed systems: Theory and tools. *Software Testing, Verification and Reliability*, 22(6):365–405, 2012.
2. E. Bayse, A. Cavalli, M. Núñez, and F. Zaïdi. A passive testing approach based on invariants: Application to the WAP. *Computer Networks*, 48(2):247–266, 2005.
3. G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48(3):117–126, 1986.
4. A. Cavalli, C. Gervy, and S. Prokopenko. New approaches for passive testing using an extended finite state machine specification. *Information and Software Technology*, 45(12):837–852, 2003.
5. C. Colombo, G. J. Pace, and P. Abela. Safer asynchronous runtime monitoring using compensations. *Formal Methods in System Design*, 41(3):269–294, 2012.
6. R. P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51(1):161166, 1950.
7. M.-C. Gaudel. Testing can be formal, too! In *6th Int. Joint Conf. CAAP/FASE, Theory and Practice of Software Development, TAPSOFT'95, LNCS 915*, pages 82–96. Springer, 1995.
8. W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman. Model-based quality assurance of protocol documentation: tools and methodology. *Software Testing, Verification and Reliability*, 21(1):55–71, 2011.

9. C. Hagenah and A. Muscholl. Computing epsilon-free NFA from regular expressions in $O(n{\cdot}\log(n)^2)$ time. *Informatique Théorique et Applications*, 34(4):257–278, 2000.
10. F. C. Hennie. Fault-detecting experiments for sequential circuits. In *5th Annual Symposium on Switching Circuit Theory and Logical Design*, pages 95–110. IEEE Computer Society, 1964.
11. O. Henniger. On test case generation from asynchronously communicating state machines. In *10th Int. Workshop on Testing of Communicating Systems, IWTCS'97*, pages 255–271. Chapman & Hall, 1997.
12. R. M. Hierons. The complexity of asynchronous model based testing. *Theoretical Computer Science*, 451:70–82, 2012.
13. R. M. Hierons, K. Bogdanov, J.P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luettgen, A.J.H Simons, S. Vilkomir, M.R. Woodward, and H. Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2), 2009.
14. R. M. Hierons, J.P. Bowen, and M. Harman, editors. *Formal Methods and Testing, LNCS 4949*. Springer, 2008.
15. R. M. Hierons, M. G. Merayo, and M. Núñez. Scenarios-based testing of systems with distributed ports. *Software - Practice and Experience*, 41(10):999–1026, 2011.
16. R. M. Hierons, M. G. Merayo, and M. Núñez. Implementation relations and test generation for systems with distributed interfaces. *Distributed Computing*, 25(1):35–62, 2012.
17. R. M. Hierons, M. G. Merayo, and M. Núñez. Using time to add order to distributed testing. In *18th Symposium on Formal Methods, FM'12, LNCS 7436*, pages 232–246. Springer, 2012.
18. J. Hromkovic, S. Seibert, and T. Wilke. Translating regular expressions into small ϵ-free nondeterministic finite automata. *Journal of Computer Systems and Science*, 62(4):565–588, 2001.
19. J. Huo and A. Petrenko. On testing partially specified IOTS through lossless queues. In *16th Int. Conf. on Testing Communicating Systems, TestCom'04, LNCS 2978*, pages 76–94. Springer, 2004.
20. J. Huo and A. Petrenko. Transition covering tests for systems with queues. *Software Testing, Verification and Reliability*, 19(1):55–83, 2009.
21. D. Lee, A.N. Netravali, K.K. Sabnani, B. Sugla, and A. John. Passive testing and applications to network management. In *5th IEEE Int. Conf. on Network Protocols, ICNP'97*, pages 113–122. IEEE Computer Society, 1997.
22. A. Mammar, A. R. Cavalli, W. Jimenez, W. Mallouli, and E. Montes de Oca. Using testing techniques for vulnerability detection in C programs. In *23rd Int. Conf. on Testing Software and Systems, ICTSS'11, LNCS 7019*, pages 80–96. Springer, 2011.
23. G.H. Mealy. A method for synthesizing sequential circuits. *Bell System Techical Journal*, 34:1045–1079, 1955.
24. G. Morales, S. Maag, A. R. Cavalli, W. Mallouli, E. Montes de Oca, and B. Wehbi. Timed extended invariants for the passive testing of web services. In *8th IEEE Int. Conf. on Web Services, ICWS'10*, pages 592–599. IEEE Computer Society, 2010.
25. A. Simão and A. Petrenko. Generating asynchronous test cases from test purposes. *Information and Software Technology*, 53(11):1252–1262, 2011.