EPiC Series in Computing

Volume 45, 2017, Pages 42–48

SCSS 2017. The 8th International Symposium on Symbolic Computation in Software Science 2017

Asynchronous testing of real-time systems

Puneet Bhateja

DA-IICT Gandhinagar, India
puneet_bhateja@daiict.ac.in

Abstract

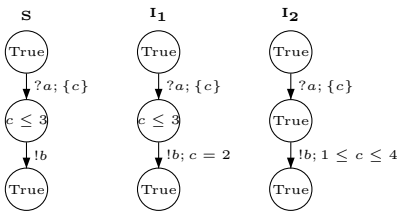
Conformance testing is an operational way of determining whether an implementation conforms to the specification or not. It has a rich underlying theory wherein the specification and the implementation under test (IUT) are each modeled by a timed automaton with inputs and outputs (TAIO), a variant of the classical timed automaton [1]. Test cases generated from the specification TAIO are symbolically executed against the implementation TAIO. Depending upon how test cases interact with the IUT, testing can be synchronous or asynchronous. In synchronous testing a test case interacts with the IUT directly, whereas in asynchronous testing a test case interacts with the IUT through a pair of first-in-first-out (FIFO) channels. Different approaches for synchronous testing of real-time systems have already been proposed [5],[7],[4],[8]. In this paper we propose an approach which is aimed at testing real-time systems asynchronously (i.e., remotely through some medium)

1 Introduction

We begin this section by defining TAIO, a state-based model that is central to our framework.

TAIO Definition: Formally, a TAIO is a tuple $A = (Q, q_0, C, \Sigma, I, E)$ where:

- Q is a finite set of locations.
- $q_0 \in Q$ is the initial location.
- C is a finite set of clocks.
- Σ is a set of actions (or the alphabet) which is further partitioned into set of input actions Σ_{in} and set of output actions Σ_{out}.
- I is a function which assigns an invariant to each location. Formally, $I : Q \rightarrow \Psi(C)$ where $\Psi(C)$ refers to the set of constraints over C. Each constraint is of the form $c\#x$, where $c \in C$, x is an integer constant, and $\# \in \{<, \leq, =, \geq, >\}$
- E is a finite set of edges $e = (q, q', \phi, r, a)$ where:
  - $q, q' \in Q$ are the source and destination locations, respectively.



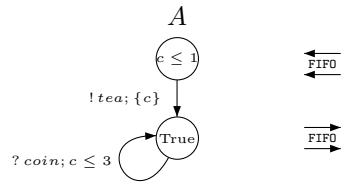
- $\phi \in \Psi(C)$ is a guard.
- $r \subseteq C$ is a set of clocks to be reset to zero.
- $a \in \Sigma$ is an action.

TAIO differs from the classical timed automaton [1] in two respects. One, the alphabet of TAIO is partitioned into input alphabet and output alphabet. The partitioning helps modeling of reactive systems, that is, systems which constantly interact with their environment. Secondly, there is an invariant associated with each location. A TAIO can remain in a location only as long as the invariant is true. Associating invariants with locations helps modeling of systems which are time-reactive.

Semantics of TAIO Suppose we are given a TAIO $A = (Q, q_0, C, \Sigma, I, E)$. Now this TAIO A implicitly defines an infinite labeled transition system $LTS(A) = (S, \rightarrow_A, s_0)$. Here S is a set of states, s_0 is the initial state, and $\rightarrow_A$ is a transition relation. Each state $s \in S$ is of the form (q, γ), where $q \in Q$ is a location of the underlying TAIO A and γ is a clock valuation. The initial state s_0 is equal to (q_0, γ_0), where γ_0 is the valuation assigning each clock zero. $LTS(A)$ admits both discrete and timed transitions. A discrete transition is of the form $(q, \gamma) \xrightarrow{a}_A (q', \gamma[r])$ where $a \in \Sigma$, and there is an edge $(q, q', \phi, r, a) \in E$ such that $\gamma \models \phi$ and $\gamma[r] \models I(q')$. Here $\gamma[r]$ is obtained from γ by resetting to zero all the clocks in the set r while leaving other clocks unchanged. A timed transition is of the form $(q, \gamma) \xrightarrow{t}_A (q, \gamma + t)$ where $t \in \mathbb{R}$ and it is the case that $\forall t' \leq t : \gamma + t' \models I(q)$. If $\exists s_1, s_2, \cdots s_n \in S$ such that $(s_0 \xrightarrow{a_1}_A s_1) \wedge (s_1 \xrightarrow{a_2}_A s_2) \wedge \cdots \wedge (s_{n-1} \xrightarrow{a_n}_A s_n)$, then we say that there is a path from the initial state s_0 to some state s_n on $a_1.a_2 \cdots a_n$. We also say that string $a_1.a_2.\cdots a_n$ is a trace of the TAIO A. The set of all traces of A is denoted by $Traces(A)$.

Synchronous testing Let us now explain the formal framework for testing real-time systems. The specification and the IUT are described by TAIOs **S** and **I**, respectively. The notion of conformance is described by the relation $\leq_{tr}$ defined over the set of all TAIOs. We say that $\mathbf{I} \leq_{tr} \mathbf{S}$ holds iff it is the case that $Traces(\mathbf{I}) \subseteq Traces(\mathbf{S})$. Informally, an IUT is said to be in conformance with the given specification if and only if it executes only those traces that are mentioned in the specification. Figure above shows three TAIOs. Here **S** is the specification, and $\mathbf{I_1}$ and $\mathbf{I_2}$ are two different IUTs. It should be noted that $\mathbf{I_1}$ conforms to the specification **S**. On the other hand $\mathbf{I_2}$ does not conform to **S** because $a4b \in Traces(\mathbf{I_2}) \setminus Traces(\mathbf{S})$. Note that in the figure we have not shown guard ϕ associated with an edge, provided its value is *True*. Likewise, we have not shown set of clocks r, provided it is an empty set.

Test generation means generating a test suite from the given specification TAIO. A test suite is essentially a set of test cases, each described as a labeled transition system. Some of the states of a test case are labeled *fail*. In synchronous testing the execution of a test case T against the IUT **I** is described by the process $T \| LTS(\mathbf{I})$, called synchronous product of T and $LTS(\mathbf{I})$. Note that here $LTS(\mathbf{I})$ refers to the labeled transition system defining the semantics of **I**. Processes T and $LTS(\mathbf{I})$ will perform common actions simultaneously. If T while executing



against the IUT **I** lands in state *fail*, we say that the **I** has failed the test case T. While we generate a test suite, we should ensure that the structure of each test case T should be such that it lands in a state *fail* only if the IUT is non conforming.

2 Asynchronous Behavior

We now explain, by an example, what do we mean by an asynchronous behavior of a system. The figure that follows shows a TAIO A interacting asynchronously with its environment, through a pair of FIFO queues (the input queue and the output queue). What is output queue for the system is input queue for the environment and vice versa. Asynchronous behavior of A means the behavior of A visible at the other end of the queues. Note that the TAIO A describes a tea-vending machine. The machine outputs *tea* spontaneously, and then goes on accepting *coin* repeatedly for three time units. Now the asynchronous behavior of A will be different from its actual behavior due to two reasons. (1) After A puts *tea* into its output queue, the *tea* will reach the user after a certain delay caused by the queue. (2) The queues may cause distortion also. For example, while the *tea* is still in the queue, the user may insert some *coin* into its output queue. This means that the actual behavior of A which is $tea!.(coin?)^\star$ could appear to the user as $coin?.tea!.(coin?)^\star$, as $coin?.coin?.tea!.(coin?)^\star$, and so on.

All this suggests that test cases generated for synchronous testing cannot be applied asynchronously. If we want to test a system asynchronously, we must first devise an approach for modeling the asynchronous behavior of the system from which test cases can be generated.

Now we show how to capture the asynchronous behavior of a given TAIO A with the help of another TAIO A'. The idea is to consider the given TAIO A and the pair of queues as one indivisible system A' so that the asynchronous behavior of A is equal to the synchronous behavior of A' [9]. To put forth our idea formally, we make following assumptions.

1. Each of the channels has a bounded capacity *cp*.

2. Once a message enters an input or output queue, it cannot stay there for more than *ub* (upper bound) units of time.

Given a TAIO $A = (Q, q_0, C, \Sigma_{in}, \Sigma_{out}, E, I)$, we define TAIO $A' = (Q', q'_0, C', \Sigma'_{in}, \Sigma'_{out}, E', I')$ where:

- $C' = C \cup C_{in} \cup C_{out}$

  Apart from the set of clocks C in the given TAIO A, we need $C_{in} = \{c^i_1, c^i_2, \cdots, c^i_{cp}\}$ and $C_{out} = \{c^o_1, c^o_2, \cdots, c^o_{cp}\}$ for our purpose. We will reset an input clock $c \in C_{in}$ when a message enters into an input queue, and likewise we will reset an output clock $c \in C_{out}$ when a message enters into an output queue.

- $\Sigma'_{in} = \Sigma_{in}$ and $\Sigma'_{out} = \Sigma_{out}$.

  The set of input actions and the set of output actions for A' are same as that of the given A.

- Q' comprises tuples of the form $(F_{in}, u, q, v, F_{out})$.

  Here $F_{in} \subseteq \{1, 2, \cdots, cp\}$ is a set of indexes of input clocks that are not in use (or are yet to be reset). Likewise $F_{out} \subseteq \{1, 2, \cdots, cp\}$ is a set of indexes of the output clocks that are not in use. Symbol $u \in (\Sigma_{in} \times \mathbb{N})^\star$ is a string of pairs that describes the contents of the input queue. Here each pair in the string is of the form (a, n) where $a \in \Sigma_{in}$ and n denotes the index of the input clock that was reset when a got into the input queue. For example, if $u = (a_1, n_1).(a_2, n_2).(a_3, n_3)$ then it means that the composite system is in a configuration wherein the contents of the input queue are $a_1.a_2.a_3$. It also means that when a_1 was entered into the input queue, clock $c^i_{n_1}$ was reset and likewise clocks $c^i_{n_2}$ and $c^i_{n_3}$ were, respectively, reset when a_2 and a_3 made there way into the input queue. Similarly $v \in (\Sigma_{out} \times \mathbb{N})^\star$ describes the contents of the output queue. Finally q denotes the location of the TAIO A.

- q'_0 is equal to the tuple $(F_{in}, \epsilon, q_0, \epsilon, F_{out})$

  Here $F_{in} = F_{out} = \{1, 2, \cdots, cp\}$ which means that initially all the clocks in the sets C_{in} and C_{out} are free. Also note that initially the TAIO A is in state q_0, and both the queues are empty.

- For each state $q' = (F_{in}, u, q, v, F_{out})$ in Q', it is the case that $I'(q') = I(q)$. In other words the invariant associated with the composite state is same as the invariant associated with its component q.

- The set of edges E' can be obtained, starting from the initial state q'_0, by the following rules:

R1 $\forall a \in \Sigma'_{in}$: If $\exists n \in F_{in}$ then
$(F_{in}, u, q, v, F_{out}) \overset{a;True;\{c^i_n\}}{\longrightarrow} (F'_{in}, (a, n).u, q, v, F_{out})$.

This rule says that it is always possible for an input symbol a to get into an input queue provided there is space for it. When there is no space in the input queue, the set variable F_{in} is empty. The moment a gets into the input queue the input clock c^i_n is reset. In the destination state the component u is changed to $(a, n).u$ indicating that a has entered into the input queue and when it entered input clock c^i_n was reset. In the destination state the set variable F_{in} is updated to $F'_{in} = F_{in} - \{n\}$ indicating that clock c^i_n is in use now.

R2 $\forall x \in \Sigma'_{out}$:
$(F_{in}, u, q, v.(x, n), F_{out}) \overset{x;c^o_n \leq ub;\{\,\}}{\longrightarrow} (F_{in}, u, q, v, F'_{out})$.

This rule says that any output symbol x present at the front end of the output queue can be removed provided it has not stayed in the queue for more than ub units of time. It also implies that when the symbol x entered into the queue, an output clock with index

n (i.e., c_n^o) was reset. Now x can be removed from the output queue subject to fulfillment of the constraint $c_n^o \leq ub$. After removing x the clock c_n^o becomes free and therefore $F'_{out} = F_{out} \cup \{n\}$. Finally no new clocks are reset during the transition.

R3 If $(q, q', a, \phi, r) \in E$, then
$(F_{in}, u.(a, n), q, v, F_{out}) \overset{\tau;(c_n^i \leq ub) \wedge \phi; r}{\longrightarrow} (F'_{in}, u, q', v, F_{out})$.

This rule says that if TAIO A has a transition from state q to q' on input symbol a, and the symbol a is available at the front end of the input queue then the TAIO quietly extracts symbol a form the queue. In TAIO A' this transition is labeled by symbol τ which means that the environment could not observe it. The transition takes place when $c_n^i \leq ub$ and ϕ are simultaneously true. Besides r, no additional clocks are reset during the transition. Since the input clock in no more in use, we have $F'_{in} = F_{in} \cup \{n\}$.

R4 If $(q, q', x, \phi, r) \in E$ and $\exists n \in F_{out}$ then
$(F_{in}, u, q, v, F_{out}) \overset{\tau;\phi;r \cup \{c_n^o\}}{\longrightarrow} (F_{in}, u, q', (x, n).v, F'_{out})$

This rule says that the TAIO A can quietly put symbol x into the output queue, without the external environment getting known. In TAIO A', this transition is labeled by action τ. It can take place if the TAIO A has a transition from state q to q' on output symbol x, and at the same time there is a space in the output queue. Besides r, output clock c_n^o is reset during the transition. After putting output clock with index n in use, we have $F'_{out} = F_{out} - \{n\}$.

3 Asynchronous Testing

Let us recall that **S** and **I** refer to the TAIOs describing the specification and the IUT, respectively. Now suppose that $\mathbf{S}'$ and $\mathbf{I}'$ are the TAIOs describing the asynchronous behaviors of the specification and the IUT, respectively. As part of asynchronous testing, we need to establish whether $\mathbf{I}' \leq_{tr} \mathbf{S}'$ holds or not. In this regard we are confronted with various questions.

Do we really need to test? Note that here $\mathbf{S}'$ and $\mathbf{I}'$ are both non deterministic finite state TAIOs. They would be non deterministic even if **S** and **I** (from which they are respectively obtained) are deterministic. Non determinism stems from the fact that in a given state rules **R3** and **R4** can be simultaneously applicable, and therefore there can be multiple transitions on symbol τ from that state. Now for non deterministic timed automatons, language inclusion is an undecidable problem [2]. This negative result rules out model checking, and leaves the designer with no option but the test the implementation vis-a-vis the specification.

How effective would asynchronous testing be? If we carefully look at each of the rules defining A' from A, we would notice that in every rule the premise is a linear-time condition. This means that the execution traces of A' are determined only by the execution traces of A. They are independent of the branching at each node of A. This proves that if $Traces(\mathbf{I}) \subseteq Traces(\mathbf{S})$ holds, then it is also the case that $Traces(\mathbf{I}') \subseteq Traces(\mathbf{S}')$ holds. Conversely if our test case predicts that $\mathbf{I}' \leq_{te} \mathbf{S}'$ does not hold, it will also mean that $\mathbf{I} \leq_{te} \mathbf{S}$ does not hold.

What is our testing approach? Our approach is based on simulating asynchronous testing by synchronous testing. Testing **I** asynchronously is equivalent to testing $\mathbf{I}'$ synchronously. As part of asynchronous testing, we will generate test cases from $\mathbf{S}'$ (i.e., the asynchronous reference behavior), and execute them against $\mathbf{I}'$, synchronously, to determine whether $\mathbf{I}' \leq_{tr} \mathbf{S}'$

holds or not. However we cannot use the same test generation algorithm that we used in synchronous testing, because here $\mathbf{S}'$ is non deterministic. Unfortunately we cannot convert $\mathbf{S}'$ into its deterministic equivalent as timed automatons are not closed under determinization. Even determining determinizability is an undecidable problem for timed automaton [2]. Also note that $\mathbf{S}'$ is partially observable, that is, it has transitions on un-observable action τ also.

What is the algorithm for test generation? Our test generation algorithm takes as an input a TAIO $A' = (Q', q'_0, C', \Sigma', E')$ which is assumed to be non deterministic and partially observable. Let $LTS(A') = (S', \rightarrow_{A'}, s'_0)$ be the labeled transition system defining the semantics of A'. Now let us define an auxiliary notation. Any subset $X \subseteq S'$ is considered $\tau-$closed when $(s \in X) \wedge (s \xrightarrow{\tau}_{A'} s') \text{ implies } s' \in X$. For any finite subset $X \subseteq S'$, its τ-closure will be denoted by $[X]_\tau$. A typical test case is a sub graph of the following labeled transition system $T' = (S_T, \rightarrow_T, s_0^T)$ where

- $S_T = \{[X]_\tau | X \subseteq S'\}$
- $s_0^T = [\{s'_0\}]_\tau$
- $X \xrightarrow{a}_T [Y]_\tau$ where $Y = \{s' | \exists s \in X \wedge s \xrightarrow{a}_{A'} s'\}$
- $X \xrightarrow{a}_T fail$ if $\nexists s \in X$ such that $s \xrightarrow{a}_{A'} s'$.

A test case is generated incrementally step by step, starting from the initial state. From a given state, whether a test case should be generated any further is decided in a non deterministic manner. Various non deterministic choices will give rise to different test cases.

We now explain the intuition behind the above test generation algorithm. Let us define a notation first. If σ is some execution trace of A', then $Ob(\sigma)$ corresponds to the observable trace which is obtained by removing τ actions. For example if $\sigma = a2b3.5\tau.2.5.a$, then $Ob(\sigma) = a2b6a$. T' is essentially that labeled transition system (1) which is in state $\{s_1, s_2, \cdots, s_n\}$ after processing $Obs(\sigma)$ iff $LTS(A')$ after processing σ is in state s_1 or s_2 or ...or s_n, and (2) which is in state *fail* after processing $Obs(\sigma)$ iff $\sigma \notin Traces(A')$.

Using this algorithm, we can generate a test suite from the reference asynchronous behavior $\mathbf{S}'$. It is not difficult to imagine that a test case while executing synchronously against the IUT $\mathbf{I}'$ will get into state *fail* only if $\mathbf{I}'$ does not conform to $\mathbf{S}'$.

4 Conclusion

In this paper we have proposed an approach for testing real-time systems asynchronously. While the existing asynchronous approaches were based on testing non real-time systems [6],[3],[9], ours is based on testing real-time systems.

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 1994.

[2] R. Alur and P. Madhusudan. Decision problems for timed automata: A survey. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004*, pages 1–24, 2004.

[3] P. Bhateja. A tagging protocol for asynchronous testing. In *5th IEEE Conference on theoretical aspects of software engineering, X'ian, China*, pages 11–18, 2011.

[4] L. B. Briones and M. Röhl. Test derivation from timed automata. In *Model-Based Testing of Reactive Systems*, pages 201–231, 2004.

[5] A. Hessel and K. Larsen. Time-optimal real-time test case generation using UPPAL. In *Formal Approaches to Software Testing, Montreal, Canada, 2003*, 2003.

[6] C. Jard, T. Jéron, L. Tanguy, and C. Viho. Remote testing can be as powerful as local testing. In *FORTE*, pages 25–40, 1999.

[7] A. Khoumsi, T. Jéron, and H. Marchand. Test cases generation for nondeterministic real-time systems. In *Formal Approaches to Software Testing, Montreal, Canada, 2003*.

[8] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In *11th International SPIN Workshop, Barcelona, Spain*, 2004.

[9] L. Verhaard, J. Tretmans, P. Kars, and E. Brinksma. On asynchronous testing. In *Protocol Test Systems*, pages 55–66, 1992.