A Comparison of Distributed Test Generation Techniques

Harry Gros-Desormeaux  Hacène Fouchal  Philippe Hunel

Abstract

Testing is an important step of the development of complex systems. It requires a lot of resources of memory and time. In addition, the complexity of such a step increases if the systems handle time constraints. In this paper, we present several solutions to generate test sequences in a distributed manner. Our methods are first explained and then compared from a theoretical point of view. Our solution leveraging DHT networks seems to be the most efficient to tackle test generation on very large-scale networks.

Keywords: Validation, Protocol Engineering, Conformance Testing, Timed Automata, P2P, DHT.

1. INTRODUCTION

Nowadays, modern systems tend to be more and more complex and need thorough validation phases before deployment. These steps are very time consuming and should be handled with care. This is the case for embedded and real-time systems from which successful deployment strongly depends on low development costs and reduced time to market. Actually, conformance testing is highly needed in order to avoid catastrophic errors and to tackle the industrial development of the product with confidence. Conformance testing is widely used in industrial validation steps and is viewed as a black box testing where input sequences of events are submitted to the implementation and output reactions from the implementation are compared to the expected ones of the specification. In this kind of testing, the tester usually does not know the internal behavior of the system and he can only interacts with it using external interfaces.

This study copes with the complexity problem arising when conformance testing is applied on huge systems which are represented by the Timed Labeled Transition System model. It is defined as an automaton where each transition can bear either an input action (an event from the environment) or output action (a reaction of the system) and sometimes timing constraints. It is widely used for the description of timed systems [2]. Our work enhances a test sequence generation technique, namely the UIO\(^1\) method [30], which derives a specific test sequence for each controllable state. Such situation represents a state where the system waits for stimuli from the environment, i.e. a state where its outcoming transitions contain only input actions. The purpose is to check if every controllable state (of the specification) will be correctly implemented on any implementation supposed to be conform to the system specification. Industrial systems are modeled by huge automata (containing millions of states) and exhaustive test sequence generation is not feasible in classical environments.

We suggest three solutions to deal with automata of millions states and to enable test generation without any reduction of the specification. A first solution distributes the specification on a peer layout obtained by graph partitioning in order to reduce communications. Our second solution allows partitions overlapping to reduce inter-peers communications. And our final scheme, based on DHT\(^2\) networks, eliminates the drawbacks of our previous solutions by equally distributing the specification on all peers of the distributed application.

Our paper is outlined as follows. First, some relevant works to the testing field are reviewed in section 2 and a test sequence generation method, namely UIO, is presented in section 3. Our three distributed test generation methods are then described. First, section 4 presents our first proposition leveraging graph partitioning. Then a more refined test generation is proposed in section 5. Another test generation using DHT is explained in section 6. Section 7 compares our three schemes and then section 8 concludes our work.

\(^1\)Unique Input Output \(^2\)Distributed Hash Table
2. RELATED WORK

There are many works dedicated to the verification of timed automaton [1, 9, 10]. Some tools [8, 3] have been developed for this purpose. Besides, some other studies proposed various testing techniques for timed systems. [23] deals with an adaptation of the canonical tester for timed testing and it has been extended in [24]. In [7], the authors derive test cases from specifications described in the form of a constraint graph. They only consider the minimum and the maximum allowable delays between input/output events. [6] presents a specific testing technique which suggests a practical algorithm for test generation. They have used a timed transition system model. The test selection is performed without considering time constraints. [29] gives a particular method for the derivation of the more relevant inputs of the systems. [27] suggests a technique for translating a region graph into a graph where timing constraints are expressed by specific labels using clock zones. [26] suggests a selection technique of timed tests from a restricted class of dense timed automaton specifications. It is based on the well known testing theory proposed by Hennessy [11]. [20] derives test cases from Timed Input Output automaton extended with data. Automata are transformed into a kind of Input Output Finite State Machine in order to apply classical test generation technique. [32] gives a general outline and a theoretical framework for timed testing. They proved that exhaustive testing of deterministic timed automaton with a dense interpretation is theoretically possible but is still difficult in practice. They suggested to perform a kind of discretization of the region graph model (which is an equivalent representation of the timed automaton model). Clock regions are only equivalence classes of clock valuations. Their discretization step size takes into account the number of clocks as well as the timing constraints. Then they derive test cases from the generated model. The second study [15] differs from the previous one by using discretization step size depending only on the number of clocks which reduces the timing precision of the action execution. The resulting model has to be translated into a kind of Input/Output Finite State Machine which could be done only under strong and unrealistic assumptions. Finally they extract test cases by using the Wp-method [17]. As we can notice, there are different ways to tackle the problem of timed testing. All of these studies focus on reducing the specification formalism in order to be able to derive test cases feasible in practice. In contrast to these studies, we use the timed automaton model without neither translation nor transformation of labels on transitions. In order to reduce the generation execution time, we suggest distributed techniques for test generation on timed automata.

3. TEST GENERATION

The validation testing process can be described with two phases. The first consists of deriving test sequences from the system specification. In the second step, a tester applies them to this system implementation to find behavior faults. This study only focuses on the first step. Here, we used the well known the UIO test generation adapted to the timed automaton model. In our case, the specification is a Timed Input Output Automaton.

**Definition 3.1.** A TIOA is an automaton with timed inputs and outputs.

\( A = \langle \sum_A, S_A, S_A^0, C_A, E_A \rangle \) where :

- \( \sum_A \) is a finite alphabet,
- \( S_A \) is a finite set of all states,
- \( S_A^0 \subseteq S_A \) is the set of the controllable states.
- \( C_A \) is a finite set of clocks,
- \( E_A \subseteq S_A \times S_A \times \sum_A \times 2^{C_A} \times \phi(C_A) \). An edge \( \langle s, s', a, \lambda, \delta \rangle \) represents a transition from the state \( s \) to \( s' \) over the symbol \( a \). \( \lambda \subseteq C_A \) is the set of the clocks which are re-initialized whenever a transition is drawn. \( \delta \) is a clock constraint over \( C_A \).

**Definition 3.2.** Let \( X \) be a set of clocks. The set of clock constrain ts \( \phi(X) \) is defined inductively by : 

\[ \delta := x \leq c | c \leq x | \neg \delta | \delta_1 \land \delta_2 \] 

with \( x \), a clock of \( X \) and \( c \in \mathbb{Q} \).

Less formally, \( A \) is the specification of the system. Transitions starting from \( S \setminus O \) represent user interactions with the system (input actions). Transitions starting from \( S \setminus I \) represent systems interactions with the user (output actions). The purpose of the UIO method is to find for any controllable state a sequence which identifies uniquely this state. For each controllable possible sequences, that is sequences of user and system actions that can be drawn from the controllable state, are extracted from the specification. Note that

\[ A \]
Algorithm 1: Test generation

Data: An automaton
Result: Test sequence each controllable state

\begin{algorithm}
\textbf{foreach} controllable state $e$ of the automaton \textbf{do}
\begin{algorithmic}
\State We set the structures which will receive the sequences
\State $\text{SeqLessRec}[e] = \emptyset$
\State $\text{Sequniq}[e] = \emptyset$
\end{algorithmic}
\end{algorithm}

threshold represents the maximum allowed depth sequence

\begin{algorithm}
\textbf{while} $d \neq \text{threshold}$ \textbf{do}
\begin{algorithmic}
\State \textbf{foreach} controllable state $e$ of the automaton \textbf{do}
\begin{algorithmic}
\If {$\text{Sequniq}[e] = \emptyset$} 
\State $\text{rss}$ = set of all sequences of depth $d$ starting from state $e$
\State \textbf{foreach} sequence $s \in \text{rss}$ \textbf{do}
\If {$s$ is accepted by some controllable states different from $e$} 
\If {$s$ is a better sequence than $\text{SeqLessRec}[e]$} 
\State $\text{SeqLessRec}[e] = s$
\Else 
\State $\text{Sequniq}[e] = s$
\EndIf
\EndIf
\EndFor
\State $d = d + 1$
\EndFor
\end{algorithmic}
\end{algorithm}

an possible sequence always starts with an input action and ends with an output one. An possible sequence identifies the controllable state $e$ if and only if this sequence cannot be drawn by all the states $S^0_A - \{e\}$. That is, only $e$ can draw this sequence.

Algorithm 1 describes the process used to find sequences which can identify each controllable state. We should notice that, sometimes, for some controllable states, we cannot find an UIO sequence.

Theorem 3.1. Let’s $n, d, t$ respectively be the number of states of an automaton $A$, the maximum depth of a test sequence and the maximum number of transitions per state in an automaton. Algorithm 1 finds all test sequences of depth $d$ in time $O(k \cdot n^2)$ with $k = dt^d$.

\begin{proof}
A possible sequence of depth $d$ is drawn with $O(d)$ steps. As a consequence, $O(nd)$ steps are needed to identify a state. Testing all possible sequences for a state takes $O(t^d nd)$ steps. Therefore, complexity for identifying all states with maximal depth $d$ is $O(dt^d \cdot n^2)$.
\end{proof}

Practically, the test generation complexity is $O(n^3)$ when the specification contains millions of states. This is nowadays infeasible at low cost, and actual solutions simplify or reduce the specification in order to achieve some practicability. These simplifications can lead to low quality test sequences depending on automata classes. We present in the following, a distributed solution, which uses algorithm 1 as a basis, to generate test sequences without simplifying the specification.

4. TEST GENERATION WITH DISTRIBUTED AUTOMATA

This section presents a decomposition method for dealing with large automata for test generation in distributed environments. We leverage graph partitioning to split the automaton in multiple partitions distributed on different computers in the network. In opposite to to the mainstream, the automaton is not reduced. Its components are just split through network nodes which communicate to ensure the automaton continuity. In this way, test generation with distributed automata can be decomposed in three steps:

- First, all automaton states are bi-partitioned recursively until each partition of states can hold in a network node memory.
- each partition, called sub-automaton and mapped to network nodes, is related to each other w.r.t external transitions to ensure automaton continuity.
- Finally, test generation starts from each sub-automaton which tries to identify its controllable states.
Before talking about the decomposition mechanics, let’s recall the graph bi-partition problem:

Let’s $G$ a graph and $S_1$ and $S_2$, two partitions of its vertices such as $|S_1| = |S_2|$. The problem of finding a minimum cutset between $S_1$ and $S_2$ — a minimum set of edges that connects $S_1$ and $S_2$ — is called graph bipartition problem.

In fact, graph partitioning [31, 13, 5, 4, 19] is commonly used in the process of load balancing in several distributed problems. This NP-complete problem [18], commonly used in the partitioning of communication graphs, has led to several heuristics whose goal were to minimize communication routes between processors or networks computers. Now, we derive this technique to make a distributed automaton in order to avoid too much communications exchange between sub-automata during test generation. Graph bi-partitioning will not be discussed here (it is beyond the scope of this paper). We use it as a black box and any bi-partitioning algorithm or $k$-partitioning method could be used in our scheme whenever it does not weigh the problem complexity. For sake of simplicity, let’s decide to use a simple and efficient heuristic, for example the Fiduccia-Matney variant [16] of the well known Kernighan and Ritchie’s algorithm [22] which emphasize the power of iterative methods for graph partitioning. Nevertheless, notice that is not such a limitation.

4.1. Test generation

Test generation does not really differ when using a distributed automaton. Test sequences are derived from each controllable states and are tested for all controllable states of the automaton. Whether no state accepts a sequence, it is singular and identifies the state it derives from. In that case, no other sequence is spawned for this state and the process stops when all states are identified or maximal sequence depth, set empirically, is reached.

Generating test sequence

As we said earlier, all sub-automata have to generate test sequences for their controllable states. So, two cases have to be considered:

- states for which jumping to other sub-automaton is not needed in the generation process,
- remaining states i.e states for which creating test sequence need other sub-automaton to spawn the entire sequence.

The first case is easy to handle since the sequence can be entirely built in the partition. For the latter, we cope with it as follows. The last state of the first sequence part is related to its following state in another sub-automaton. First sequence part is sent to this sub-automaton with the state number the sequence derived from. The remaining part sequence is built and is concatenated to the sequence message (first sequence part + initial state number) which jumps recursively from sub-automaton to sub-automaton until all the sequence is built. The last reached sub-automaton becomes responsible for generating sequences which can be built with the received sequence prefix from its predecessor that ends in its set.

More formally, $s(e_i, d)$ is a sequence derived from the state $e_i$ with length $d$. Whenever $s(e_i, d)$ overlaps automata, $s(e_i, d)$ become $\sum s_{P_k}^j (e_i)$ where $s_{P_k}^j (e_i)$ denotes the $j^{th}$ subsequence starting from the state $e_i$ in the partition $P_k$ with $i, l \in [1..n]$, $k \in [1..m]$ and $j \in [1, 2, \ldots, p]$. The last partition $P_k$ reached after the $(p - 1)^{th}$ jump try to generate newer possible sequences with prefix $\sum_{j=1}^{p-1} s_{P_k}^j (e_i)$ whenever the current processed sequence $s(e_i, d)$ is not singular.
Processing the sequence

A discriminatory phase is run into the sub-automaton to know whether a sequence is possible or not. Possible sequence is then wrapped in a message of the form \( [e, s(e, d), P_p] \) which is sent to all peers to continue sequence identification. Peers reply of existence of the sequence in their relevant partition to \( P_p \). When all controllable states replies are gathered, the state is identified or not. Whether the sequence is singular, the sub-automaton containing the state which the sequence derives from will receive a notification of singularity for the sequence.

4.2. Using Distributed Automata in P2P environments

Leveraging distributed automata in peer-to-peer environments seems to be an easy way now as we can map each partition of vertices to a peer. Peer-to-peer communication layers guarantee our message exchange in the network. This avoids us to ensure communication consistency and de facto, continuity in the automaton is guaranteed since vertices boundary between partitions only need peer name (peer address) to jump to other partition. Algorithm 2 described below derives test generation for peer-to-peer environments.

Algorithm 2: Partitioned Test Derivation

<table>
<thead>
<tr>
<th>Data: An automaton ( A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result: Singular sequences identifying controllable states</td>
</tr>
<tr>
<td>Automaton partitioning</td>
</tr>
<tr>
<td>Bi-partition recursively ( A ) in ( k ) sub-automata</td>
</tr>
<tr>
<td>Distribute the ( k ) sub-automata to peers in the network</td>
</tr>
<tr>
<td>Test Derivation</td>
</tr>
<tr>
<td>foreach parallel: peer ( p ) containing sub-automaton do</td>
</tr>
<tr>
<td>Run thread MessageListener</td>
</tr>
<tr>
<td>Run thread ProcessPool</td>
</tr>
<tr>
<td>foreach controllable state ( s_c ) in ( p ) do</td>
</tr>
<tr>
<td>Generate sequence ( seq ) from ( s_c ) with length ( d )</td>
</tr>
<tr>
<td>if ( seq ) size &lt; ( d ) then</td>
</tr>
<tr>
<td>Send ( {s_c, seq} ) to peer pointed by the last state of ( seq )</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>put ( seq ) in the pool ( P ) of possible sequences</td>
</tr>
<tr>
<td>Procedure MessageListener</td>
</tr>
<tr>
<td>Get message event</td>
</tr>
<tr>
<td>Pick ( seq ) from message</td>
</tr>
<tr>
<td>if ( event = message_composition ) then</td>
</tr>
<tr>
<td>Generate sequence of length ( d - (\text{length of } seq) ) from destination state of the last state of ( seq )</td>
</tr>
<tr>
<td>Append resulted sequence to ( seq )</td>
</tr>
<tr>
<td>if ( length ) of ( seq ) &lt; ( d ) then</td>
</tr>
<tr>
<td>Update message with ( seq )</td>
</tr>
<tr>
<td>Send message to the last peer pointed by the last state of ( seq )</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>Add ( seq ) to pool of possible sequences</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>if ( {seq } ) not marked) ( ) ) is recognized from destination state of the last state of ( seq )</td>
</tr>
<tr>
<td>then</td>
</tr>
<tr>
<td>Send notification of non-recognition for initial state of message to peer stored in message</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>Send notification of recognition for initial state of message to peer stored in message</td>
</tr>
</tbody>
</table>

Lemma 4.1. Let \( n, d, t \) be respectively the number of controllable states of an automaton, its depth, and its maximum number of transitions. The length of a singular sequence is \( d > \frac{\log n}{\log t} \).
Procedure *ProcessPool*

Get sequence $seq$ from pool $P$ if not empty

foreach controllable state $se$ in sub-automaton do
  if $se$ is not initial state of $seq$ AND $se$ accepts $seq$ then
    Try to generate new sequence from $seq$ in current sub-automaton
  else
    Send to peer containing initial state of $seq$ notification of singularity for this state

Proof. Let $A_n^p = \frac{n!}{(n-p)!}$ be a permutation for $p$ elements between $n$ and $P(U)$, the probability that a singular sequence exists at a controllable state of the automaton. We are searching for such a sequence, so $P(U) = 1$.

$$P(U) = \frac{A_n^p}{(t^d)^n} = 1 \quad \Rightarrow \quad A_n^p \approx (t^d)^n$$

$$\Rightarrow \quad t^d - n \approx t^d$$

Let $n = \alpha t^d$,

$$\Rightarrow \quad (1 - \alpha) \cdot t^d \approx t^d$$

$$\Rightarrow \quad \alpha \rightarrow 0$$

$$\Rightarrow \quad \alpha < 1 \quad \Rightarrow \quad d > \frac{\log n}{\log t}$$

The previous lemma determines the minimal length of our possible sequences. Indeed, the longer the sequence is, the more likely it is singular. By choosing longer sequences, we avoid useless communications exchange between peers.

Test derivation for large automata is now a possible matter since automata can be decomposed in multiple parts without reducing its specification. This decomposition exhibits some parallelism for test generation which allows the use of peer-to-peer environments as networks for distributed test generation.

5. TOWARDS A BETTER DISTRIBUTION OF THE SPECIFICATION

This section presents another method which distributes the specification, namely the automaton, on the network. Nodes collaborate in order to verify the sequence singularity. Such a technique gives some advantages. First, spatial complexity is low and then, load balancing is ensured due to the symmetric behavior of all nodes participating in the distributed computation.

To distribute the automaton over the network peers, it is split into consistent sub-automata.

**Definition 5.1.** A consistent sub-automaton $A_{E,d}$ of depth $d$ for the set of controllable states $E$ is a sub-automaton for which all sequences of depth $d$ can be drawn from $E$. It is said that $E$ is consistent for the sub-automaton $A_{E,d}$.

Initially, the list of all controllable states $E$ is divided into sub-lists of controllable states $E_i$ such that

$$\bigcup_{i \in \{1...k\}} A_{E_i,d} = A$$

Fig. 2 describes the test generation scheme for two consistent sub-automata. Note that these sub-automata overlap. A possible sequence is generated on the first peer, and then is tested on the other peers. The sequence is singular if it cannot be drawn on the other peers.

First, the whole automaton is divided into sub-automata which are sent on the network peers. Our test generation algorithm running at each network peer is a two-fold process. The active thread generates test
sequences with depth $d$ sent on all other peers. The passive thread get sequences from other peers and test them.

**Procedure active Thread**

```plaintext
foreach consistent controllable state $e$ get from sub-automaton $A$ do
  Generate an possible test sequence $S$ of depth $d > \frac{\log n}{\log t}$
  repeat
    Re-generate an possiblr test sequence $S$
    Send sequence at other peers
    Wait for all peer results
  until $S$ is singular
  $S$ is not singular
  Store $S$ as a singular Sequence for $e$
```

**Procedure Passive Thread**

Capture sequences sent on the network
Test each sequence $S$ and send them to the initiating node

This distributed test generation method gains advantage by generating test sequences locally. Unfortunately, the number of consistent sub-automata depends for the specification. Indeed, more memory is needed to store sub-automata whenever their intersection is not empty. The more sub-automata overlap, the more they consume memory. The next scheme reduces this drawback with an equal repartition of the specification on all peers participating in the test generation.

6. DHT FOR TEST GENERATION

DHTs [14, 33, 34, 25, 21] are a new generation of peer-to-peer networks which tries to reduce the drawbacks of the current peer-to-peer ones. They allow efficient routing ($O(\log n)$) but also provide exhaustive location of resources shared on the network. The previous distributed algorithm of test generation presented in section 3 is reconsidered here to be able to take advantage of DHTs features, in particular their principle of exhaustive localization of resources.

6.1. Distribution and localization of the specification

In a DHT network, all the peers have a virtual address (an identifier) defined using consistent hash functions (MD5, SHA-1, etc.). The routing scheme used on these networks derives from the approach due to Plaxton [28], which consists of using on each node a routing table containing the address of the neighbors. In this way, information is transmitted incrementally from node to node in order to reach its destination. The overlay network obtained, generally inspired from parallel architectures [12] (hypercube, ring, graph of DeBruijn, graph of Knodel, etc), allows efficient routing ($O(\log n)$ hops). Any distributed
object on a DHT has also an identifier obtained using the same hash functions used to identify the peers. The peer whose virtual address is closer to the object identifier is responsible for keeping its informations of localization (identifier of the peer which has the object).

Our new test generation scheme distributes the transitions of the automaton on the peers. These transitions are shared using their labels or their states. The figure 3 describes the distribution and the localization of a resource shared on the DHT. For example, some peers hold the labeled transitions “?a” of an automaton. Let us call \( f \), the hash function which associates a string to a numerical identifier. These peers share these transitions by contacting the node \( f(“?a”) \) and by storing their respective addresses in a resource localization table. To recover all the transitions labeled with “?a”, one has to contact the node \( f(“?a”) \) which returns identifiers of the peers which detaine these transitions.

Subsequently, the respective peers are contacted in order to get the aforementioned transitions.

6.2. Generation of possible sequences

Each peer of the network is responsible for the generation of the possible sequences derived from the controllable states it has. To generate such sequences of length \( d \), the consistent sub-automaton of depth \( d \) for all controllable states is built. The algorithm 7 describes the process of rebuilding this sub-automaton on a DHT. Then, the possible sequences are extracted from the sub-automaton and are tested for unicity on the network. The algorithm of test generation was rewritten to be used on the DHT. Rather than to test the unicity of a sequence on each controllable state of the automaton, the set of the states which accept the sequence is built. The cardinal of this set indicates whether the eligible sequence is a test sequence. The function \( \text{eligible.sequence} \) describes the recursive construction of the eligible sequences for a starting state \( e \). For example, the call \( \text{eligible.sequence}(A, \emptyset, e, 10) \) tries to find an eligible sequence of length 10 for the state \( e \).

6.3. Finding the test sequence

The checking of the unicity of a test sequence consists of showing that there exists only one state which accepts it. So, we build the set of states which accept such a sequence. The cardinality of this set indicates the unicity of the sequence.

Let \( E \) be a set of states, and \( l \) a label from a transition of the automaton. The function \( \text{recognized.previous.states}(E, l) \) returns the starting states of the transitions labeled with \( l \) and that have for destination states, the states in \( E \). Let \( l_1, l_2, \ldots, l_n \) the successive labels of one sequence \( S \). Let \( E_A \) be the whole set of controllable states of the automaton. We define in a recursive way the set

\[
U_i = \begin{cases} 
\text{recognized.previous.states}(E_A, l_n) & \text{if } i = n \\
\text{recognized.previous.states}(U_{i+1}, l_i) & \text{otherwise}
\end{cases}
\]
Algorithm 7: Construction of a consistent sub-automaton

Data: a list of controllable states \( E \)
Result: a consistent sub-automaton for \( E \)

Initialization of the variables constituting the sub-automaton
states_set ← \( E \)
transitions_list ← \( \emptyset \)

Initialization of the temporary variables for the construction of the sub-automaton
temporary_starting_states ← \( E \)
temporary_destination_states ← \( \emptyset \)

Construction of the sub-automaton

while depth > 0 do
    foreach state \( e \) of temporary_starting_states do
        Locate and recover the transitions which have as a starting state \( e \)
        Add these transitions in transitions_list
        Add the destination states of these transitions in temporary_destination_states
        Add temporary_destination_states in states_set
        temporary_starting_states ← temporary_destination_states
        temporary_destination_states ← \( \emptyset \)
    depth = depth - 1
return states_set and transitions_list

Function eligible_sequence(Automaton A, Sequence \( S \), State \( e \), depth \( d \))

if \( d > 0 \) then
    destination_states ← destination states having for starting state \( e \)
    foreach state \( f \) of the unit destination states do
        Add the label of the transition \( e \rightarrow f \) in \( S \)
        eligible_sequence(A, S, F, d-1)
    Discard the last label entered in \( S \)
else
    If \( S \) is unique return \( S \)

Theorem 6.1. A sequence \( S = \{l_1, l_2, \ldots, l_N \} \) is unique if and only if \( |U_1| = 1 \).

Proof. Let us suppose that \( U_1 \neq 1 \). If \( |U_1| = 0 \) then there does not exist a starting state which accepts a transition labeled with \( l_1 \) and consequently, the sequence is not possible, therefore non-unique. If \( |U_1| > 1 \) then there exists several starting states which accept transitions labeled with \( l_1 \). However, according to equation 1, \( U_1 \) is built using the starting states which accept transitions labeled with \( l_2 \). Consequently, there exists several states in the automaton which accept the sequence \( S \). It is not unique, which is absurd by assumption.

Now, let us assume that the sequence \( S \) is not unique. Either it is not possible, and in this case, there does not exist any state which accepts it and thus \( |U_1| = 0 \). Either several states accept this sequence and \( |U_1| > 1 \), which is absurd by assumption. Note that the same sequence cannot be accepted several times by the same state because the automaton specifying the system is deterministic.

Theorem 6.2. Let \( t \) be the average number of outgoing transitions per state for an automaton containing \( n \) states. The algorithm 9 finds if a sequence \( S \) of length \( d \) is unique with complexity \( O(dtn) \).

Proof. There exists, on average, \( t \) transitions per state. Therefore, there are \( tn \) transitions on average in the automaton. Without loss of generality, the number of transitions bearing a specific label is bound by \( tn \). Thus, the whole set of transitions with a specific label is found with \( O(tn) \) stages. Therefore, algorithm 9 determines the sequence unicity with \( O(dtn) \) stages since the sequence has for length \( d \).
Algorithm 9: Search for a test sequence

Data: a deterministic automaton and an possible sequence \( S = \{l_1, \ldots, l_n\} \)

Result: True if \( S \) is unique, False otherwise

\textit{Construction of the set of all states} \( U \) \textit{accepting the sequence} \( S \)

\( d \leftarrow \) length of the sequence \( S \)
\( U \leftarrow \) Set of the transitions labeled with \( l_d \)

\textbf{while} \( d > 1 \) \textbf{do}

\( V \leftarrow \) Set of the transitions labeled with \( l_{d-1} \)
\( X \leftarrow \text{starting\_states}(U) \cap \text{destination\_states}(V) \)
\( U \leftarrow \) transitions from \( V \) for which the destination states are \( X \)
\( d \leftarrow d - 1 \)

\textbf{if} \( |U| = 1 \) \textbf{then}
\( \text{return True} \)
\textbf{else}
\( \text{return False} \)

7. COMPARISON OF THE DISTRIBUTED TEST GENERATIONS

Generally, the performance of distributed applications rely upon the number of messages exchanged during the application. It is well known that optimizing this number often lead to a better makespan. Here, we assume that all exchanged messages have the same constant size. This latter assumption can be done here due to the small size of data exchanged between peers on all the three scheme described below.

Let’s compare the number of messages that can be exchanged for the three described previous schemes. First, let \( A \) be the whole automaton, \( S \) a sequence, \( d \) the depth of \( S \) and \( t \), the maximum number of transitions per state in the automaton. Without loss of generality, we can say that the set of all eligible sequences of depth \( d \) has for cardinality \( O(t^d) \) in the three schemes. Let’s call \( n \) the number of possible sequences to test.

For sake of equity, we assume now that \( k \) partitions are used for the first and the second scheme and that \( k \) nodes are used to store the automaton in a DHT network. We also assume that the localization process takes \( O(1) \) step for all schemes since all schemes are run on the same peer-to-peer network which is here a DHT.

For our scheme 1, in the worst case, each state of the sequence is on a peer, so \( O(d) \) hops are needed to generate the sequence and, because the sequence needs to be drawn from each controllable state in order to verify its singularity, \( O(k) \) peers are involved in the process. We said earlier that with high probability, the sequence is singular so we will not assume that hops are not needed for draws at other peers. \( O(k) \) messages are sent to the peer initiator to ensure the sequence singularity. Then, \( O(d + 2k) \) messages are exchanged for scheme 1 to verify a sequence singularity. For the whole process, \( O(n \cdot (d + 2k)) \) messages are exchanged.

For scheme 2, the generation step costs nothing since it is done on a single peer for an eligible sequence. \( O(k) \) messages are sent to verify the sequences at each peer and \( O(k) \) messages are fetched at the initiator to verify the sequence singularity. \( O(2k) \) messages are exchanged for scheme 2 to test an possible sequence. Assuming that the controllable states are equally distributed so \( O(\frac{n}{k} \cdot 2k) \) messages are exchanged totally. Test generation for scheme 2 takes \( O(2n) \) messages.

For scheme 3, let’s say that all the peers are needed to generate a sequence. So \( O(2d) \) messages are needed to find which peers have the suitable transitions. Then the automaton is fetched with \( O(k) \) messages and the sequence is generated. Then \( O(2d + k) \) messages are exchanged for a sequence generation and the same number of messages is needed to test a possible sequence. So \( O(4d + 2k) \) are used in scheme 3 to test a sequence singularity. As for scheme 2, assume that controllable states are equally distributed such that \( O(\frac{n}{k}) \) messages are exchanged to generate the eligible test sequences. Then \( O(\frac{n}{k} \cdot (4d + 2k)) \) messages are necessary to verify all sequences.
Without loss of generality, these distributed test generation are designed to be applied on very large specifications containing millions of states on large-scale networks. So, the \( d << k \) and therefore, the number of exchanged messages for the three schemes are the following:

- for the first presented scheme, there are \( O(kn) \) messages exchanged,
- for the second, there are \( O(n) \) messages exchanged,
- and for the last presented scheme, there are \( O(n) \) messages exchanged.

In fact, the second scheme consumes less messages than the others, unfortunately, the number of partitions strongly depends on the specification. As a consequence, it cannot be always applied practically. It turns out to be that our last scheme, the one leveraging the DHT networks, seems to be the most efficient. The last step to confirm this assumption is real-world experiments.

8. CONCLUSION

Generating test sequences for specifications with millions of states is nearly impracticable. Most of the solutions to deal with the explosive nature of the test generation simplify in some ways the automaton specifying the system. We propose several solutions which distribute the specification on a distributed systems and compare our schemes theoretically to foresee which scheme is better.

Our first scheme distributes the specification with graph partitioning to reduce inter-partition communications. The second scheme relaxes some constraints by allowing partition overlapping but consumes more memory. Our latter test generation method equally distributes the specification on all peers on the network and enables automaton reconstruction through the DHT localization process. Test generation on DHT network seems to be, as far we know, the first solution which does not simplify the specification and which shifts temporal complexity over the routing one. As a consequence, very large automata can be used on distributed environments like Internet where very low-grade peers can participate to the test generation. To fully validate our study, our distributed methods have to be implemented to ascertain our theoretical results.

Bibliography


2nd International Workshop on Verification and Evaluation of Computer and Communication Systems
A Comparison of Distributed Test Generation Techniques


