A Distributed Cache Management for Test Derivation

Harry Gros-Desormeaux Hacène Fouchal
Philippe Hunel

GRIMAAG
Université des Antilles et de la Guyane, France
{harry.gros-desormeaux, hfouchal, phunel}@univ-ag.fr

Abstract: Complex systems need to be validated before industrial development. The last step in the validation process is testing. This part has to be considered with care in order to avoid troubles. This step takes a long time and requires a lot of resources.

In this paper, a complex system is described a Timed Labeled Transition System (TLTS). In such description, we focus on the specification of the event ordering respecting time constraints.

Since the TLTS may be very large (million of states for industrial systems), we present a solution to reduce the test derivation complexity. We aim to decompose the derivation process among some hosts participating to the generation algorithm. Each host will deal with a part of the system independently ; each host will derive test sequences for some fixed states from the system.

Some computations are redundant. In order to reduce them, on each host we use the *Bloom filters* concept used to manage a local cache containing computed sequences.

Then, we show how to compute the results given by all hosts in order give a set of test sequences for the whole system.

We suggest an implementation of this technique on the JXTA environment deployed on some hosts. We analyze a large number of experiments on different TLTS. Finally, we have shown that the use of Bloom filters make increases the test derivation performances.

Key-words : Distributed Environment, P2P Computing, Conformance Testing, Protocol verification.

1 Introduction

Complex systems are being used everywhere. However, the success of their deployment depends on low development cost and reduced time to market. In order to achieve this goal, validation steps should be handled with care. Among these steps, conformance testing is highly needed to avoid dramatic errors and to tackle the industrial development of the product with a high confidence.

Peer-to-peer frameworks revealed great potential for scientific applications which require a lot of resources. Recently, projects like SETI@home[1] has started to bring attention to massive parallel computing and now, it is common to tackle long running time computations with peer-to-peer environments. The term peer-to-peer refers to machines which interact in a distributed fashion to reach the same goal.

In this study, we model a complex system by the Timed Labeled Transition System (TLTS) model. It is defined as an automaton where each transition can bear either an input action (an event from the environment) or an output action (a reaction of the system). On each transition, we may have timing constraints expressed as equations on defined clocks. Finally, on transitions, we may have clock resets. Each state represents a stable state of the system. It is widely used for the description of timed systems [AD94]. Then, we present a test sequence generation technique which derives a specific test sequence -from the specification -for each controllable state (represents situations where the system waits for stimuli from the environment) of a specification. 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.

Some industrial systems are modeled by huge automata (containing millions of states). An exhaustive test sequence generation is not possible in a sequential environment. For this reason, we present a distributed algorithm which considers the test derivation issue. Each host involved in the computation will receive the whole automaton (representing the system) as well as the list of states for which it has to generate test sequences. Network diversity leads us to design two algorithms which generate tests sequence : one which hold for homogeneous networks and another version which takes account of network heterogeneity. Each host will handle a local cache to keep the recent computed test sequences. The management of such caches is done by using the Bloom filters concept. Algorithms have been implemented on the JXTA environment with a PC cluster.

This paper is structured as follows: Section 2 gives an overview on studies done on testing of timed systems and peer-to-peer computing. Section 3 describes the test sequence generation algorithm. Section 4 details the decomposition issue and gives an insight to our distributed algorithm. Section 5 is devoted to the improvement of the testing process by using local caches based on Bloom filters. Section 6 gives a conclusion and some ideas about future work in particular on different evolutions of the distributed algorithm.

2 Related work

In this section, we present briefly the main related work in testing real-time systems as well as the most important approaches in peer-to-peer computing.

2.1 Timed system testing

[SVD01] 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 since the number of test sequences is very high.

[ENDKE98] 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, finally they extract test cases by using the Wp-method [FBK+91].

[NS01] 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 in [DNH84].

In [CL97], 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.

[CO02] presents a method for networks of deterministic timed automata extended with integer data variables where only a part of the system can be visible.

[RNHW98] gives a particular method for the derivation of the more relevant inputs of the systems.

[PF99] suggests a technique for translating a region graph into a graph where timing constraints are expressed by specific labels using clock zones.

[HNTC01] 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.

In [KT04], suggests a framework for a black-box conformance testing of real-time systems, where specifications are modeled as nondeterministic and partially-observable timed automata. This work is an extension of a previous work on the same issue [AT02].

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 a distributed technique to perform the test generation.

2.2 Peer-to-Peer Computing

Peer-to-peer frameworks revealed great potential for scientific applications which require a lot of resources. Recently, projects like SETI@home has started to bring attention to massive parallel computing and now, it is common to tackle long running time computations with peer-to-peer environments (Distributed.net [2], Distributed Folding Project[3], Grub[4], Evolution@Home[5], etc). The term peer-to-peer refers to machines which interact in a distributed fashion to reach the same purpose. These systems can be :

- *centralized :* A single machine, the server, manages the peer-to-peer layer. This configuration exhibits an important drawback : if the server disappears, the peer-to-peer environment collapses.
- *decentralized :* Each machine can play the same role in the network. They can manage the peer-to-peer layer as they can share their resources as workstations.

- *mixed :* The peer-to-peer network can be divided in sub-networks where some peers are servers and the others, workstations. This configuration tends to provide more tolerance to faults when a server leaves since peer-to-peer systems are usually build over volatile nodes which can appear or disappear at "will".

Generally, we tend to find two types of peer-to-peer applications even though they exhibit only a fragment of the peer-to-peer philosophy. On one hand, we find file-share based applications like Napster[6], Overnet[7], Kazaa[8], Bittorent[9] which are well-known today due to the several Majors' lawsuits brought recently. On the other hand, CPU cycle stealing projects like the ones cited above as peer-to-peer computing are very popular due to their altruistic interests. Nevertheless, other projects exist, like OceanStore which offers global persistent data store designed to scale to billions of users. Even VoIP applications (cf. Skype[10], Gizmo Project[11],...) leverage peer-to-peer concepts for constructing their "small world". Peer-to-peer philosophy is so attractive that it spawned a revolution in the manner to tackle some type of problems.

By gaining more and more attention, peer-to-peer model drove Sun Microsystems to design the JXTA technology which provides interoperability, independent platform and ubiquity. These advantages offer simplicity to anyone who wants to make a portable peer-to-peer application by hiding the complexity of the physical network as well as the knowledge of all operating systems core of the different hosts participating in the network. Thereby JXTA appeared as a suitable platform to implement our application.

JXTA provides several predefined protocols for developing large-scale peer-to-peer applications. In the JXTA environment, nodes are groups that propose services, and synergize in order to reach a common goal. Each node which connects to the peer-to-peer layer, a loosely-consistent DHT* network, joins the principal peer group before belonging to one or more dedicated groups. In fact, groups advertise services and can be only reached according to their policies. Another strength of JXTA is to provide *relay points* in order to bypass firewalls. So, nodes can fetch data to these relays when they are forbidden to receive network data packets. The *pipes* feature, a JXTA communication concept, abstract routes between two endpoints in the peer-to-peer network. *Pipes* are published through *advertisements* on the network and allow nodes with similar pipe advertisements to communicate between them. So we can bypass the physical details of the network and focus only on communications. For example, from the node point of view, we do not need to know which node will receive our packet. We only send data through a pipe if binded, JXTA services will ensure consistency and routing in the peer-to-peer layer.

JXTA provides simplicity and easiness for who wants to implement a peer-to-peer application today. Although it has shown some drawbacks in the past as a peer-to-peer computing platform — due to bad bandwidth management — it seems that it has been improved and now is able to reach optimal efficiency for WANs [AHJN05]. We use JXTA as our peer-to-peer platform to develop our distributed application.

*Distributed Hash Table

3 Test sequence generation algorithm

The purpose is to find for any controllable state a sequence in order to identify the state.A controllable state is a state where outgoing transitions are labeled by input actions. That means when the system reaches such states, it can only wait for external events. We will extracts for each controllable state a sequence of action starting with an input action and ends with an output one. If this sequence is not recognized by any other state, then this state is considered as identified. We will operate in a similar way for all other states with sequences containing an arbitrary number of input actions. This number – that we will call from now *depth sequence* — is fixed in advance.

In fact, the algorithm is a little bit similar to the UIO technique [SD88] used for untimed system testing described as IOFSM (Input Output Finite State Machine).

Algorithm 1: Test generation

Data: An automaton
Result: Derived sequence for each controllable state

```
We initialize depth sequence to 1
d=1
foreach controllable state e of the automaton do
    We initialize the data structure which will store the sequences
    SeqUniq[e] = ∅
threshold is the maximum depth sequence
while d ≠ threshold do
    foreach controllable state e of the automaton do
        if SeqUniq[e] = ∅ then
            RSS= set of all sequences of depth d starting from state e
            foreach Sequence s ∈ RSS do
                if s is not accepted by other controllable states than e then
                    SeqUniq[e] = s
    d = d + 1
```

Comments

The aim of this algorithm is to derive for each controllable state a test sequence. A test sequence starts with a input event (to ensure controllability from the user during the test execution) and ends with a output event. Each derived sequence for a state should be checked on all other states and should not be applicable on these states. It should be unique.

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 found all test sequences of depth d in time $\mathcal{O}(k \cdot n^2)$ with $k = dt^d$.*

Proof. A state accept a test sequence of depth d in $\mathcal{O}(d)$ steps. So, we know if a sequence is unique in $\mathcal{O}(nd)$ and testing all sequences of a rss set is done in $\mathcal{O}(t^d nd)$, then finding a unique sequence of depth d for each state costs $\mathcal{O}(dt^d n^2)$ steps. □

Example:

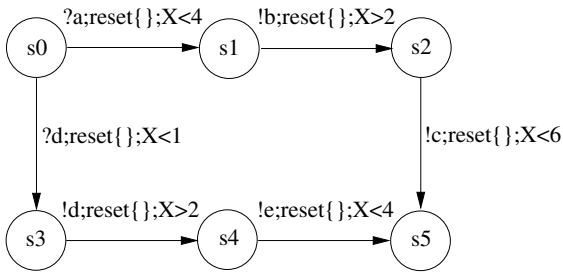

Figure 1: An example of specification

On Figure 1, state s_0 is controllable. Then, the following sequences are extracted:

- $(?a, X < 4), (!b, X > 2)$ is recognized by s_0,
- $(?d, X < 1), (!d, X > 2), (!e, X < 4)$ is recognized by s_0,
- $(?b, X > 1), (!c, X > 6)$ is not recognized by s_0.

The next section presents a peer-to-peer algorithm which computes the algorithm described below.

4 A distributed approach

We design a MPMD[†] algorithm based on the *master-slave* paradigm which implements the test generation algorithm. As we said in section 3, we can find almost all test sequences with a suitable depth. For large automata (with millions states), this depth turn out to be too high for sequential applications which take too much time to complete. Our distributed algorithm takes as input an automaton and outputs all minimal test sequences for each state.

[†] Multiple Program Multiple Data

4.1 A general framework

We describe briefly our distributed algorithm composed of the master and the slaves:

- A master gives jobs (computation of some controllable states) to slaves and then get the results (derived sequences) when jobs end.
- Slaves fetch jobs from the Master, compute sequences and send them to the Master.

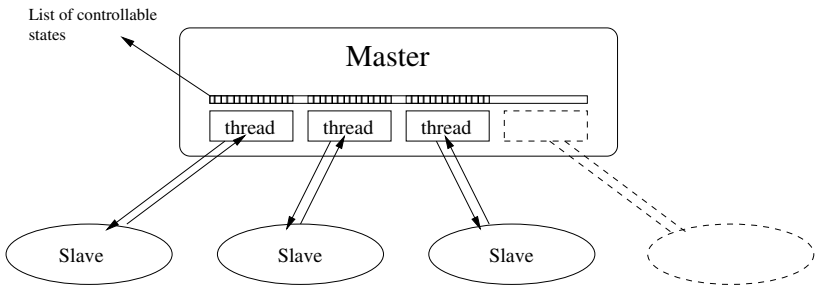

Figure 2: The Master-Slave approach

4.1.1 The master

The role of the master is to coordinate the global work. It distribute *jobs* to the working slaves and gives jobs to additional nodes which join the peer-to-peer network. Our algorithm takes as input an automaton and gives a singular test sequence for each controllable state. The set of all controllable states of the automaton is divided in sublists. A job is defined as a computation of test sequences for controllable states contained in a sublist. That means that each job will handle a sublist of states. Whenever a slave joins the peer-to-peer network, it fetches a job from the master. Test sequence results are saved for subsequent return to the master.

From an implementation point of view, we start a thread process for each slave which connects to the master and send to it a job taken from the *job list* as well as the automaton. We send the automaton only once, the slave stores it for subsequent work if any.

4.1.2 The slaves

Slaves are nodes which help to the global work by running the generation algorithm on a subset of controllable states. Each slave starts a *Message Event Listener* thread to capture messages sent by the master and asks for jobs. When receiving their first job, they get in the same time the automaton and store it for later use. Each job is recognized by a list of integers which represent the states that it has to compute. Then the generation algorithm is run on each of these states and result sequences are saved in a *results* list. Subsequently

Algorithm 2: Master (sublist fixed size version)

```
Master()
begin
    Results R = ∅
    We generate the automaton
    A = Automaton(nbStates,minNbTrans,maxNbTrans,nbClocks)
    Compute JobList J = {J_1, ..., J_n}
    while J ! = ∅ do
        We wait that a slave connect
        Wait for connection
        We send job and automaton
        Start Thread SendAutomatonAndJob()
    Wait for all Threads
    Exit(0)
end

SendAutomatonAndJobs()
begin
    if Slave has not received the automaton then
        Send(A)
    Send(J_i)
    We wait that the slave finished and send its results
    Wait for Results
    Add Results to R
    Close connection
end
```

the *results* are sent to the Master. Slaves may be added freely to our platform even though it is not always an efficient way to solve the problem.

4.2 Job list decomposition

We have first designed an algorithm which partitions the list of controllable states in equal parts before scatters them between slaves. Suitable for homogeneous networks, this scheme has exhibited some insufficiency whenever machines and communications links were manifold. Consequently, we develop a scheme which copes with network heterogeneity. Indeed, we can choose between two strategies when dealing with jobs size. Dynamic approaches gracefully adapt jobs size whenever new nodes join the network whereas static ones fix *a priori* this size once for all. In these two cases, job size limit is an important parameter. If a job size is too high, we may loose performance and scalability since some joining machines could not find any job. A contrario, if jobs are too small, we communicate too often and performances collapse. Furthermore, too many processors imply too much communications. Indeed, performance degrades whenever communication

Algorithm 3: Slave

```
Slave()
Automaton A = ∅
begin
    Connect to the Master
    Start Thread MessageListenerEvent()
end

MessageSlaveListenerEvent()
begin
    if A = ∅ then
        Ask for Automaton to Master
        Store automaton in A
    Ask for Job
    Compute step one for each state of the received job
    Send Results to Master
end
```

time overlaps processing time.

In the next subsections, we present two schemes for distributing jobs to slave nodes. On one hand, a simple static strategy which partitions the sublist size in fixed parts and on the other hand, a simple dynamic job distribution which varies the received job size according to its *resource capabilities*.

4.2.1 The fixed approach

Here, we decompose the list of all controllable states in subsets of fixed size which are distributed to nodes joining the network. These jobs are processed and results are stored in the master. This scheme, though efficient for homogeneous networks, does not really suit for their heterogeneous counterparts.

4.2.2 A simple dynamic distribution

Our following technique is more adapted to cope more gracefully with heterogeneous networks. The major drawback one can notice when dealing with heterogeneous networks is the inadequate size of the job provided to a node which is not tuned w.r.t its power and its communication link bandwidth to the master. In our adaptive algorithms, a node supplies its *characteristics* to the master in order to get a *fitting job* when it joins the peer-to-peer network.

A straightforward scheme

The simplest manner to distribute jobs to processors is to give *fitting job* according to the node characteristics. Indeed, from now on, a joining node gives its characteristics when

joining the network. In our dynamic scheme, we give to it a job having adequate size to its capability. Although this *modus operandi* seems simple, it is likely to be efficient.

5 Speeding up the test generation process

The test generation process seen in the previous section can introduce *redundant computations* which slow down our application. Indeed, several controllable states can share one or more sequences since they are spawn from all possible sequences found along a "trail of consecutive states". Since our process does not have any storage mechanism, each non-singular processed sequence is lost whenever a new state has to be identified. Thus, keeping in memory sequences already processed by the use of a cache seems to be a natural way to avoid this "amnesia". We present in this section some cache mechanisms based on *Bloom Filters* — an elegant alternative to lookup tables — which helps our algorithm to avoid the same computation.

Bloom Filters are generally used when the domain size or the set which needs to check membership is too large to be kept in memory. Bloom filters were used in early UNIX spell-checker [McI82] as well as in database computations for speeding up semi-joins operations [LR95]. In [Goh03], Goh proposes a bloom filter based scheme to search keywords in encrypted documents in constant time. Recently, Broder [BM02] has given a very comprehensive and historical survey of the use of Bloom filters for network applications. Indeed, Bloom Filters are extensively used in some Web cache sharing mechanisms [RW98], [FCAB00] to reduce the shared cache size. In fact, this probabilistic structure can be used whenever space complexity has to be considered with allowable membership error. In the early seventies, Bloom [Blo70] has shown that great storage reduction can be gained at the expense of introduced false positives when multiple hash functions are used to determine fewer element membership among lots of one. We leverage this mechanism to store the non-singular processed sequence in a Bloom filter which help us to avoid to re-process sequences already "seen". Unfortunately, the algorithm can "miss" possible singular sequence due to false sequences spawn by this technique. Therefore we can still find longer singular sequences since minimal singular sequences prefix longer ones. So, we are allowed to miss some singular sequences at the expense of their minimality.

5.1 Bloom Filters

Let's $S = s = s_1, s_2, \ldots, s_n$, a set of n elements which has to be to stored and A, an array of m bits initially set to 0 which will store the set S. Bloom Filters are d independent hash functions h_i which maps pseudo-randomly each element of the universe in a series of random numbers over the range $\{0, \ldots, m-1\}$ such as $A[h_i(s_k)] = 1$. An element membership is tested as follows : the element is hashed through the d hash functions and each resulted location in the array is tested. If one hash function maps to a bit at 0 in the

array, the element does not belong to the set S. This technique can drastically save storage space whenever the number of elements to store is low compared to the universe space. The drawback with this method is that it introduces some false positives which probability to occur is

$$(1 - (1 - \frac{1}{m})^{dn})^d \simeq (1 - e^{-dn/m})^d \quad (1)$$

There exists a trade-off to find between the size of the array of bits, the number of hash

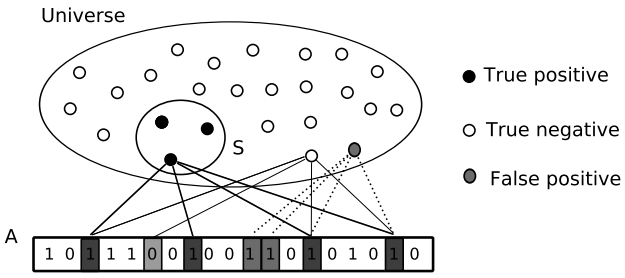

Figure 3: Bloom Filters with 4 hash functions

functions and the probability to find false positives. Nevertheless, it becomes relatively easy to find an optimal parameter by fixing the two others.

5.2 Experimental Results

Using Bloom Filters seems to speedup a lot the process whenever initial sequence depth does not suffice to find all possible unique sequences as shown on Figure 4. This figure details the time execution and the gain of the algorithm over a set of automata of 1000, 2000,..., 9000 states having the same number of labels(16). The average transitions per state is 5 and the average number of clocks is 5. Four types of experiments have been done : a sequential algorithm without Bloom (line : seq wo Bloom Filters), a sequential algorithm with Bloom (line : seq w Bloom Filters), a distributed algorithm without Bloom (line : 2 proc wo Bloom Filters) and a distributed algorithm with Bloom (line : 2 proc w Bloom Filters). In fact, it is shown that our algorithm becomes more efficient whenever the number of states rises. This can be explained by the fact that the more "the population" grows, the less you can find different "individuals" spawned from the same "strain" : the set of the all possible sequences does not vary from a certain number of states.

However, some cares must be taken with the choice of the depth. For very large deterministic automata with many transitions at each state, the non-singular processed sequence set can be large too and so must be the cache size. Fortunately, the larger the depth sequence is, the more the probability to find a unique sequence is if it exists. This explains why Bloom Filters does not really help our process from certain depth threshold. Indeed, our first possible sequence spawn at this depth threshold is unique with very high probability and cache mechanism does not trigger.

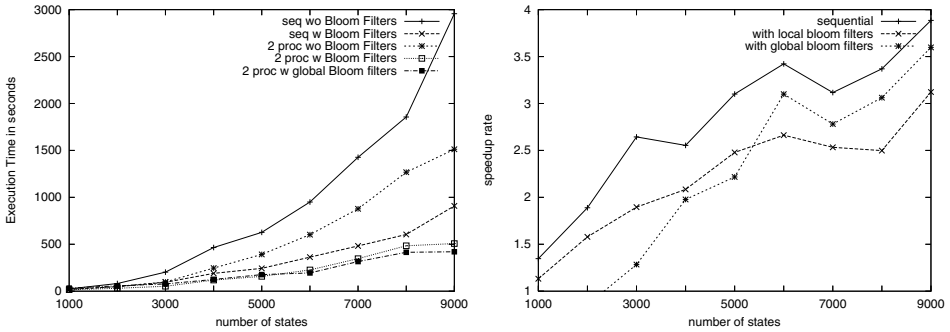

Figure 4: Time execution and Gain algorithm

The use of Bloom Filters were also conducted on a straightforward parallel version of our algorithm where Bloom Filters were used locally at each process running on each slave. As we could expect, results are less efficient whenever the number of slaves participating in the computation grows. To a great extent, if the master distributes *jobs* of one state to slaves as numerous as jobs, the Bloom filter is totally inefficient since it cannot help the others slaves. Indeed, slaves do not know their counterparts. As in some uses of Bloom Filters in network, they have to be shared using gossiping strategy for example.

A shared Bloom filter version has also been implemented. Its behavior is not so different from the ones used in shared web cache mechanisms. Each slave hold a local bloom filter which updates the master's one if need be. In this case, the filter is sent with the sequence results and is merged with the master's filter. The master then updates other local bloom filters when slaves fetch new jobs. So the master is responsible for sharing the updated global filter in our distributed environment.

As we could expect, results with global bloom filter have shown some improvements for large automata. However, we can notice that before some threshold (here, the number of states of the automaton), the bloom filter is inefficient. This is due insofar to costly communications which are used to share the filter between the slaves. Fortunately, they become negligible as the computation time for identifying a controllable state rises.

6 Conclusion and future work

We have suggested in this paper a distributed algorithm which is able to find test sequences for large timed systems (a suitable model to describe real-time protocols, embedded systems, web services). Large-scale parallelism opens the way to test sequence generation. As far as we know, this study is one of the first attempt to merge large-scale peer-to-peer parallelism and test generation. In order to have better results, we suggest to use a local cache on each slave which is able to keep some calculated sequences which may be needed later. This cache uses the Bloom filters concepts (an elegant technique to lookup tables). The use of these filters have shown better results (for the distributed algorithm) even if

Bloom filters offer a better gain in a sequential algorithm.

This study needs to be extended in order to handle fault tolerance. In the present solution, each slave has to reply to the master with its set of test sequences. In case of troubles, the algorithm is not able to take care of them.

The use of this technique on industrial system (multimedia protocol, real-time system, ..) will be undertaken very soon in order to show the efficiency of the technique.

References

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AHJN05] Gabriel Antoniu, Phil Hatcher, Mathieu Jan, and David A. Noblet. Performance Evaluation of JXTA Communication Layers (extended version). Technical Report PI-1686, IRISA, January 2005.

[AT02] Karine Altisen and Stavros Tripakis. Tools for Controller Synthesis of Timed Systems. July 03 2002.

[Blo70] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[BM02] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey, 2002.

[CL97] Duncan Clarke and Insup Lee. Automatic Generation of Tests for Timing Constraints from Requirements. In *Proceedings of the Third International Workshop on Object-Oriented Real-Time Dependable Systems,* Newport Beach, California, February 1997.

[CO02] Rachel Cardell-Oliver. Conformance Test Experiments for Distributed Real-Time Systems. In *International Symposium on Software Testing and Analysis (ISSTA?02), ACM Press, July 2002*, July 2002.

[DNH84] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[ENDKE98] A. En-Nouaary, R. Dssouli, F. Khendek, and A. Elqortobi. Timed Test Cases Generation Based On State Characterization Technique. In *19th IEEE Real Time Systems Symposium (RTSS'98)* Madrid, Spain, 1998.

[FBK+91] S. Fujiwara, G. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test Selection Based on Finite-State Models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.

[FCAB00] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[Goh03] E. Goh. Secure Indexes, March 16 2003.

[HNTC01] Teruo Hogashino, Akio Nakata, Kenichi Taniguchi, and Ana R. Cavalli. Generating Test Cases for a Timed I/O Automaton Model. October 2001.

[KT04] Krichen and Tripakis. Real-Time Testing with Timed Automata Testers and Coverage Criteria. 2004.

[LR95] Zhe Li and Kenneth A. Ross. PERF Join: An Alternative to Two-way Semijoin and Bloomjoin. In *CIKM*, pages 137–144, 1995.

[McI82] M. Douglas McIlroy. Development of a spelling list. *IEEE Trans. Communications*, COM-30:91–99, 1982.

[NS01] Brian Nielsen and Arne Skou. Automated Test Generation from Timed Automata. In T. Margaria and W. Yi, editors, *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems,* Genova, Italy, volume 2031 of *Lecture Notes in Computer Science*, pages 343–357. Springer-Verlag, April 2001.

[PF99] E. Petitjean and H. Fouchal. From Timed Automata to Testable Untimeed Automata. In *24th IFAC/IFIP International Workshop on Real-Time Programming, Schloss Dagstuhl, Germany*, 1999.

[RNHW98] P. Raymond, X. Nicollin, N. Halbwatchs, and D. Waber. Automatic testing of reactive systems, Madrid, Spain. In *Proceedings of the 1998 IEEE Real-Time Systems Symposium, RTSS'98*, pages 200–209. IEEE Computer Society Press, December 1998.

[RW98] Alex Rousskov and Duane Wessels. Cache digests. *Computer Networks and ISDN Systems*, 30(22–23):2155–2168, 1998.

[SD88] K. Sabnani and A. Dahbura. A Protocol Test Generation Procedure. *Computer Networks and ISDN Systems*, 15:285–297, 1988.

[SVD01] J. Springintveld, F.W. Vaandrager, and P. R. D'Argenio. Timed Testing Automata. *Theoretical Computer Science*, 254(254):225–257, 2001.