Journal of Computing and Information Technology - CIT 19, 2011, 1, 11–23
doi:10.2498/cit.1001762

RESETting Timed Machines

Ariel Stulman
Computer Science Department, Jerusalem College of Technology, Jerusalem, Israel

Many real-time applications enable RESET to account for all kinds of unexpected problems, or to accommodate for a users' want of restarting. Additionally, some software testing techniques must allow for resetting timed-Implementations Under Test (t-IUT). Dedicated internal logic is probably the most common of solutions for accomplishing such tasks. There are situations, however, where such a privilege doesn't exist; thus, it cannot be built upon. Testing pre-engineered timed-IUTs is one such case. In this paper we wish to present an algorithm for the direct generation of timed RESET sequences from the timed-IUT specification, such that it should be optimal w.r.t. to execution time.

Keywords: D.2.4 [Software Engineering]: Software/-Program Verification – Formal Methods; F.1.1 [Theory of Computation]: Models of Computations – Automata; D.2.m [Software Engineering]: Miscellaneous – Real-time systems

1. Introduction and Related Work

Many algorithms were devised for specific tasks within the conformance testing envelope (status messages [10], separating family of sequences, distinguishing sequences [13], UIO sequences [1],[28], characterizing sequences [11],[19],[25], and identifying sequences [19]; a survey of the main methods can be found in [24]). The ability to reset the implementation under test (IUT) to its original state, however, must also be part of the algorithmic suite when testing a system or some communication protocol. This is true, whether we are interested in testing a specific state transfer or when verifying complete conformance to specification. Not always can we rely on the existence of some magical RESET button for this purpose. Therefore, a special type of sequence, a RESET sequence (a.k.a. synchronizing sequence) was devised. It is a sequence that uses inherent transfers within the IUT to achieve its purpose. See Kohavi's book [19] for basic algorithm and results.

Due to the wide use of finite state machines (FSM) as the modeling technique for the IUTs, its inherent flaw was automatically projected into system testing. Standard FSMs do not take into account temporal constraints that may influence the IUT; and as such, most methods developed for system testing were not applicable to real-time systems. With the proposal of Alur and Dill's 'Timed automata' [3] in 1994, an entirely new field of system testing for real-time (reactive) systems emerged. It soon became the prevalent model for real-time systems, and the underlying theory behind many model checking tools; e.g., UPPAAL [23] and KRONOS [8]. The emerging field quickly became a thriving research area with many diverse sub-topics (see [5],[6],[7],[9],[21],[22],[26]],[27],[28], and many others).

Papers dealing specifically with state identification sequences, however, were not concerned with actually producing algorithms for usage on an IUT with timing constraints (t-IUT). Their main objective was in demonstrating how a t-IUT can be transformed into a regular, un-timed IUT, for which all necessary algorithms pre-exist (see [7],[18],[20],[29] and others). A few papers did provide algorithms for direct generation of such sequences ([30],[31]). There, however, the notion of defining optimality as a function of the sequence length (as was done with un-timed machines) is retained. For real-time systems, however, defining optimality as minimal execution time would gain many advantages [14].

In this paper we describe an algorithm for generation of a synchronizing sequence while conforming to execution time optimality. We do so while retaining the direct generation algorithm

given in [30]. This algorithm will be extended as to provide the ability of RESETting a timed IUT without insertion of explicit reset logic as would be required otherwise. Thus, the contribution of this paper is twofold: development of timed synchronizing sequence optimal with respect to execution time, and extension of the algorithm as to provide for an internal RESET logic where possible.

2. Preliminaries

2.1. Modeling time

The notion of time can be described as *discrete* or *dense* [3]. In a discrete timing model, time increases monotonically by some constant decided upon *a priori* (usually 1) every cycle of the system. When such is the case, there is no need for a real clock. A variable that represents the current "time" is sufficient. Using such a model limits the accuracy with which physical systems can be modeled.

A more natural model for physical processes operating over continuous time is the dense-time model. When we talk about "real" timing of systems, we must have a clock that keeps the time. The advancement of the clock is irrespective of the cycle time of the system, and can increase monotonically without bound.

Assumption 1: We use a discrete-time model in the rest of this discussion. This will allow us to use numerical values in the examples and discussions. This assumption will be lifted later, in Section 4.8. [*Justification*: Physical hardware of computer systems functions in discrete cycles, so the use of a discrete-time model is more in-tune with hardware. In addition, by choosing a very small granulation value, we can approximate a physical system to within any accuracy level.]

2.2. I/O Automata

There are many variants of FSM used to model systems. The majority of the state identification techniques discussed in the literature are based on the *Mealy machine* [16]. This fact is based upon the ability of the Mealy machine to exchange messages (input and output) with its environment. A *deterministic* transition is stimulated by input from the environment, and as a consequence, the machine can return an output message back to the environment (a *reactive* machine). Since a *black box* testing model [17] – where one cannot see the internal structure of the IUT, but has access to its input and output ports – is the basic premise of checking experiments, the Mealy machine [19] is a suitable candidate for representing the internal logic of an IUT.

Definition 1 (Mealy Machine). A *Mealy machine*, M is a 6-tuple $\langle S, s_0, I, O, \delta, \lambda \rangle$, where:

- S is a finite set of states.
- $s_0 \in S$ is the initial state of the system.
- I is a finite set of input events ($I = \{\imath_1, \imath_2, \ldots, \imath_p\}$).
- O is a finite set of output events ($O = \{o_1, o_2, \ldots, o_r\}$).
- δ is the state-transfer function ($\delta : S \times I \rightarrow S$).
- λ is the output function ($\lambda : S \times I \rightarrow O$).

For simplicity, we extend the state transfer function, δ, from single input symbols to input strings as follows: for some initial state s_1, let the input sequence $\sigma = \alpha_1 \cdot \alpha_2 \cdot \ldots \cdot \alpha_k$ take the machine successively through the states $s_{j+1} = \delta(s_j, \alpha_j)$, where $j = 1, 2, \ldots, k$, such that the final state of $\delta(s_1, \sigma) = s_{k+1}$. In the same manner we extend the output function, λ, from a single output to an output string such that: $\lambda(s_1, \sigma) = \beta_1 \cdot \beta_2 \cdot \ldots \cdot \beta_k$, where $\beta_j = \lambda(s_j, \alpha_j)$ with $j = 1, 2, \ldots, k$. Obviously, $s_j \in S$, $\alpha_j \in I$ and $\beta_j \in O$.

2.3. State uncertainty

It may be the case that some state information of an IUT is missing. The *state uncertainty* of the IUT is defined as the set of states that may adequately complete the missing state information [19]. If the initial state of the IUT is unknown, we speak of an *initial state uncertainty*; a set containing all possible states that may constitute the IUTs initial state. For example, consider the automata in Figure 1 representing a simplified internal logic of a sender using the alternating bit protocol (ABP) for data-link layer network communication [4,15]. It shows the four states

of the machine, the possible transfers between these states, and the output ($o \in O$) generated by a transfer triggered by input ($\iota \in I$) in the form of ι/o on its label. If the machine can begin in any of the four states, we say that the *initial state uncertainty* is {ABCD}.

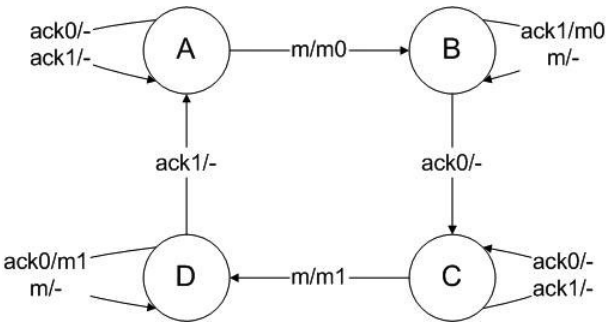

Figure 1. Automata representation of alternating bit protocol.

When the current state of the IUT is unknown, we speak of a *current state uncertainty* set. Suppose, for example, that σ =ack1. After inputting of σ into the IUT in Figure 1, {ABC} will be the *current state uncertainty*.

2.4. Successor tree

A *successor tree* is a tree representing the states that can be reached by the IUT based on all possible input combinations. The purpose of the tree is to graphically display the nth successors of the root; constituting an aid in the selection of the most suitable input sequence to meet required goals.

The root of the tree contains the initial state of the IUT. When it is not known, we associate with the root an *initial state uncertainty* vector. Tree edges represent input to the IUT. Every node is associated with a *current state uncertainty* vector representing accumulated state uncertainty knowledge until that point, with σ being the concatenation of labels on the edges that form the path from the root to the node. For example, a partially extended successor tree for the automata in Figure 1 with an initial state uncertainty of {ABCD} is shown in Figure 2[1].

Since the *degree* of the tree is $|I|$, the number of acceptable inputs in the language of the automata, at level j ($0 \leq j \leq \infty$) we may have $|I|^j$ nodes. It is quite obvious that in order to reduce the size of the successor tree, some restrictions must be placed (avoid redundancy, etc.).

2.5. Timed I/O Automata

In this work we define a *timed automaton* as a Mealy machine with the addition of temporal constraints on the transition function δ. A transition cannot fire, and hence no output or internal state change can occur if the clock guard

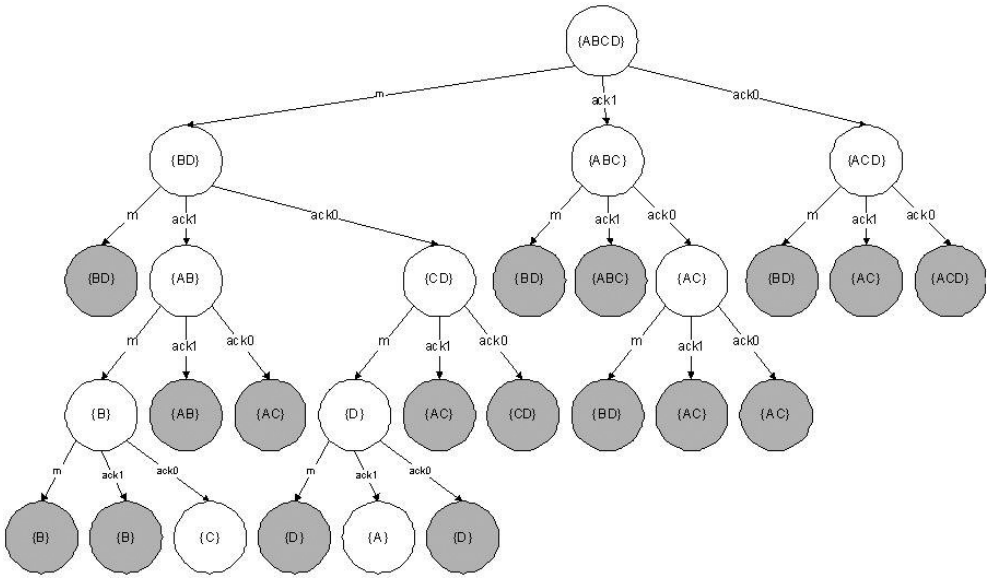

Figure 2. Partially extended successor tree for alternating bit protocol.

[1] Nodes in the tree were grayed when redundancy was found. We didn't extend those nodes as one can tell how these sub-trees will develop from similar nodes elsewhere in the tree.

on the transition is not satisfied. Intuitively, a clock or set of clocks must be included within the system to allow for the definition of time.

Definition 2 (Clock Constraint). Let a *clock constraint*, Δ, over an internal set of clocks, C, be defined as a Boolean expression of the form c op z, where $c \in C$, op is a classical relational operator $(=, \leq, \geq, >, <, \neq)$, and $z \in \mathbf{Z}_{\geq 0}$.

Definiton 3 (Clock Guard). Let a *clock guard*, ψ, over C be a conjunction of clock constraints over C: $(\psi = \Delta_1 \wedge \Delta_2 \wedge \ldots \wedge \Delta_m)$. Let Ψ be the set of possible clock guards pertaining to the system, and $\Psi_k \subseteq \Psi$ be the set of all clock guards that contain at most k clock constraints.

Definition 4 (Clock Assignment). Let a clock assignment, φ, over C be a conjunction of assignments of values to some (or all) of the clocks in C, and let Φ be the set of all such conjunctions: $\Phi = \{c_1 = r_1 \wedge \ldots \wedge c_m = r_m \,|\, c_i \in C' \subseteq C \wedge r_i \in \mathbf{Z}_{\geq 0} \wedge 1 \leq m \leq |C| \wedge \forall c_i, c_j \, i \neq j\}$.

Definiton 5 (Timed I/O Automata). A *timed i/o automata*, TA, is a tuple $\langle S, s_0, I, O, C, \Psi, \delta, \lambda \rangle$, where:

- S is a finite set of states.
- $s_0 \in S$ is the initial state of the system.
- I is a finite set of input events $(I = \{\iota_1, \iota_2, \ldots, \iota_p\})$.
- O is a finite set of output events $(O = \{o_1, o_2, \ldots, o_r\})$.
- C is a finite set of clocks.
- Ψ is the set of clock guards pertaining to the system (see definition 3).
- δ is the state transfer relation $(\delta : S \times I \times \Psi \rightarrow S)$.
- λ is the output relation $(\lambda : S \times I \times \Psi \rightarrow O \times \Phi)$.

Assumption 2. For simplicity, we assume that if TA receives $\iota_k \in I$ at instant t, it will also output the corresponding $o_b \in O$ at that exact instant. [*Justification*: In essence, output is produced on a transition generated by the input. Reducing transition time to be infinitely small, we may consider the input and output as occuring at the same instant.]

Assumption 3. For simplicity, we assume that $|C| = 1$, that $\delta : S \times I \times \Psi_1 \rightarrow S$ and that $\lambda : S \times I \times \Psi_1 \rightarrow O \times \Phi$[2]. The single clock limitation will be lifted in Section 4.7.

2.6. Sequence generation environment

There are many possible environments within which we can develop a timed-RESET algorithm. The current work, however, is concerned with the *black box model* in which we will not have access to the internal logic of some IUT. We only have its specification model in terms of TA, so we must deduce the required information from it so it can be applied to an implementation. Use of this environment is chosen partially because the sequence we are interested in also pertains to conformance testing, for which use of this environment is quite common.

In addition, given the following definitions:

Definition 6 (Fully specified). There exists a definition for each state, $s_j \in S$, and every input, $\alpha_k \in I$ [and for TA for all $\psi_h \in \Psi$]; i.e.: $\lambda(s_j, \alpha_k)$ and $\delta(s_j, \alpha_k)$ [$\lambda(s_j, \alpha_k, \psi_h)$ and $\delta(s_j, \alpha_k, \psi_h)$ – resp.] are defined $\forall(s_j, \iota_k) \in S \times I$ [$\forall(s_j, \iota_k, \psi_h) \in S \times I \times \Psi$ – resp.].

Definition 7 (Strongly connected). For every pair of states, $s_i, s_j \in S$, there exists an input sequence [timed input sequence], σ_{ij} [t-σ_{ij} – resp.], which takes the automata [TA – resp.] from s_i to s_j; i.e. $\delta(s_i, \sigma_{ij}) = s_j$ [$\delta(s_i, \alpha_k, \psi_h) = s_j$ – resp.].

Definition 8 (Reduced). For every pair of states, $s_i, s_j \in S$, there exists an input sequence, σ, which distinguishes them; i.e.: $\lambda(s_i, \sigma) \neq \lambda(s_j, \sigma)$ or $\delta(s_i, \sigma) \neq \delta(s_j, \sigma)$ [for TA: $\lambda(s_i, \sigma, \psi_h) \neq \lambda(s_j, \sigma, \psi_h)$ and the state elements of $\delta(s_i, \sigma, \psi_h)$ and $\delta(s_j, \sigma, \psi_h)$ are not equal] for some σ.

Definition 9 (Deterministic). Within Mealy machines determinism is inherent. To define TA as deterministic it must hold that $\forall(s_j, \iota_k, \psi_h)$ $(s_j, \iota_k, \psi_g) \in S \times I \times \Psi$, ψ_h and ψ_g are mutually exclusive; i.e., $\psi_h \wedge \psi_g$ is unsatisfiable.

Assumption 4. We assume that in the current context sequence generation and possible testing is performed on a *fully specified*, *strongly*

[2] This says that we will talk about a TA that has a single clock and that each clock guard on its transfers contains at most one clock constraint.

connected, reduced and *deterministic* IUT or t-IUT. [Justification: This is very common assumptions made in literature dealing with the basic problem (see [25], [29], [5] and others). Obviously, there are works that deal with lifting any or all of these assumptions.]

3. RESET Sequence – Problem and Solution

3.1. Problem description

During the lifetime of most machines or software, the need for (self) RESETing is widely acceptable. Within the context of protocol or system testing this feature must also be available. One cannot begin execution of any test sequence without first bringing the IUT to some identified state from which we know how to proceed. The state must be unequivocally identified, regardless of the current state of the IUT.

This RESET option can be realized using a specific transfer of the machine from every possible state to a predefined state; thus, achieving RESET. This solution will add an additional transfer in our automaton from every state, complicating circuit or programming logic as a consequence. We wish to construct a RESET sequence that will utilize existing transfers within the machine while achieving an identical solution.

Definition 10 (RESET Sequence). An input sequence, σ_{RS_k}, is said to be a RESET *sequence* (RS) of machine M, if the final state of M after insertion of σ_{RS_k} is s_0 (or some other state designated as a RESET state) regardless of the initial state prior to initiation of σ_{RS_k}. Let σ_{RS_k} be the set of all RSS accepted by M as a solution to the RESET problem; thus $\sigma_{RS_k} \in \sigma_{RS}$.

Definition 11 (Optimal Reset Sequence). We classify a RS for machine M as *optimal*, $\sigma_{RS_{op}}$, if for all RSS, σ_{RS}, accepted by M, $\sigma_{RS_{op}}$ is shortest. i.e. $\{|\sigma_{RS_{op}}| \leq |\sigma_{RS_k}| \, \forall \sigma_{RS_k} \in \sigma RS\}$.

3.2. Sequence generation

The RESET sequence can be easily generated by extending the well known synchronizing sequence (SS) (which brings the machine to some known state) [19] concatenated with an additional sequence to compensate for the transfer to the RESET state. This method, however, doesn't necessarily foster the development of the optimal solution, $\sigma_{RS_{op}}$. We could, however, utilize and extend the algorithm described in [19] for generating a SS using a truncated successor tree to reach an optimal solution as well:

Definition 12 (Synchronizing Tree). A *synchronizing tree* (ST) is a successor tree in which we deem a node as terminal (leaf) when one of the following occurs:

1. **The loop rule:** The *current uncertainty* associated with the node was already associated with a node in a preceding level.
2. **The redundancy rule:** The *current uncertainty* associated with a node is also associated with other nodes on the same level. One is chosen as a non-terminal candidate for future expansion of the tree; the others are deemed terminal.

Using a breadth first search (BFS) algorithm, we look for a non-terminal node (leaf) that contains a singleton. The SS is constructed by concatenating the labels on the edges of the ST leading from the initial uncertainty (root) to the *first* node found satisfying the search criterion.

Clearly the *length* of the SS is equal to the depth of the solution node within the ST. Since the first node that can guarantee a solution was used, by the definition of BFS no other node exists at a higher level of the tree that can also satisfy our requirements. This implies that there is no SS with a shorter path; thus, the SS found is an optimal synchronizing sequence.

Continuing the BFS as the method for tree expansion – until we encounter a node containing the singleton considered as the RESET state (s_0 in Definition 5) - will also solve for the optimal RESET sequence for programs or circuits.

[19 §13.1] showed that the length of the synchronizing sequence, assuming it exists, is no longer than $\frac{1}{2}(n-1)^2 n$. Since we can augment the SS with a reset directing sequence of no longer than $(n-1)$ to reach the RESET state, we are guaranteed that the sequence is no longer than $\frac{1}{2}(n-1)^2 n + (n-1)$. Thus, our algorithm is bound (in the worst case), by this value[3].

[3] Actually, this bound isn't tight. [19 §Appendix 13.1] showed that the tight bound is $\frac{1}{6}n(n-1)(n+1)$, which is better than the above by a constant factor.

3.3. ST Example

To demonstrate the usage of this method, we will use the alternating bit protocol of Figure 1. It was suggested that "the protocol may be initialized by sending bogus messages and acks with sequence number 1. The first message with sequence number 0 is a real message". [2] Using the tree in Figure 2, it is easy to deduce that the sender can be initialized (brought to state A) using input of 'm.ack0.m.ack1' (with 'm.ack0.m' also being a synchronizing sequence – to state D).

4. RESET Sequence for Timed Machines – Problem and Solution

The problem of RESETting a TA is essentially the same; mainly, to optimally bring the TA to some pre-defined RESET state (normally s_0) without the use of a specific RESET logic. In contrast to regular automata, we must take into account the timing constraints that restrict transitions. Some transitions may be applicable only at specific times and inapplicable at others. The RS must contain the timing of the inputs as well, so we can coordinate the input to meet the constraints on transitions.

4.1. Context and contribution

As mentioned earlier, previous related work dealt mainly with the transformation of TAs into regular automata for which solutions pre-exist ([7],[18],[29] and others). In [20], for example, the transformation is done in a three step process: (1) compute the product of the automaton with a Tick automata which models the tester's timing. The result of this combination (2) is used to generate a time-abstracting bisimulation quotient graph, which is similar to the well known region graph presented in [3]. Lastly, this graph is transformed into a non-deterministic Mealy machine for which all algorithms already exist. Only [30] provided an algorithm for the direct generation of synchronizing sequences for TAs. The idea was to adapt STs for the extraction of optimal timed-synchronizing sequences by introducing the timing parameters into the tree itself.

The main drawback with the above algorithm, however, was the use of the old definition for sequence optimality; mainly, minimal sequence length. Optimality for TAs state identification experiment sequences should be defined in terms of execution time [14]. When a time constraint is attached to every input event, equal length sequences do not necessarily complete within the same amount of time. It is quite possible that longer sequences complete faster than shorter ones. Does it really matter if we have fewer inputs if the total time for execution is elongated?

In addition, especially when RESETting systems is in question, the basic premise that is to be done during its' execution. Generating a sequence that is optimal on paper, is by no means optimal for the system users.

4.2. Optimal t-RS w.r.t. execution time

Definition 13 (Timed- RESET Sequence). A timed- RESET sequence (t-RS), σ_{t-RS_m}, is a RS that includes timing of input. σ_{t-RS_m} is of the basic form $\alpha_1[\psi_1] \cdot \alpha_2[\psi_2] \ldots \alpha_k[\psi_k]$; where, $\alpha_p \in I$ and $\psi_q \in \Psi$. Let σ_{t-RS} be the set of all timed RSs accepted by a TA as a solution to the RESET problem; thus, $\sigma_{t-RS_m} \in \sigma_{t-RS}$.

Definition 14 (Sequence Execution time). The execution time of a sequence, π_σ, is determined by the minimum[4] time that must elapse before a sequence can be inputted in its entirety.

Definition 15 (Optimal timed- RESET sequence). A timed RESET sequence, $\sigma_{t-RS_{op}}$, is considered optimal w.r.t. execution time if $\{\pi_{\sigma_{t-RS_{op}}} \leq \pi_{\sigma_{t-RS_m}} | \forall \sigma_{t-RS_m} \in \sigma_{t-RS}\}$.

Consider, for example, a RESET sequence $\sigma_{t-RS_1} = \alpha_1[c < 2] \cdot \alpha_2[c \geq 7]$. This sequence *must* allow for clock $c \in C$ to reach 7 before it can terminate ($\pi_{\sigma_{t-RS_1}} = 7$). A second solution sequence, $\sigma_{t-RS_2} = \alpha_1[c = 1] \cdot \alpha_2[c < 3] \cdot \alpha_3[c \geq 3] \cdot \alpha_4[c = 6]$, *must* terminate when $c \in C$ reaches 6 ($\pi_{\sigma_{t-rs_2}} = 6$). By definition 15, σ_{t-RS_2} is optimal even though $|\sigma_{t-RS_2}| > |\sigma_{t-RS_1}|$.

[4] We used the term *minimum* because constraints can be composed of inequalities that allow for flexibility of input synchronization. When calculating the execution time, we attempt to enter input events as quickly as possible; hence, *minimum* time.

4.3. Points to notice

The nature of applying the RESET sequence to a timed machine would stipulate that it is insufficient to bring the machine solely to its original state (s_0, for example). Rather, the clocks of the machine must also be brought to their original values as well.

Our first attempt at a possible solution is composed of some RESET sequence that brings the machine to its initialized state (S_{RS}) and has a special input (α_{RS}) that will give the clocks their initial value as well (φ_{RS}). This will allow for that addition of a single edge to the machine (see Figure 3), as opposed to numerous edges proportional to the number of states.

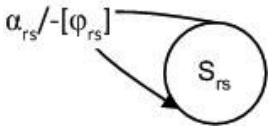

Figure 3. Additional clock reset edge.

If possible, we would like to avoid even such a minute addition to the machine. This can be accomplished *if and only if* an edge exists from some state s_k to s_{RS} such that it consists of some arbitrary output and the clock assignment φ_{RS}. We can search for a timed synchronizing sequence to state s_k, and then concatenate the last transfer to s_{RS}[5]. If a number of such edges exist, we must find the optimal one so that the entire sequence can be declared optimal.

4.4. Generating $\sigma_{t-RS_{op}}$

In order to generate a timed RESET sequence, we can adapt the synchronization algorithm for extraction of sequences. We are required, however, to do a major overhaul if it's to be used for optimality w.r.t. execution time. Even though the first node found during a BFS represents the shortest sequence (see 3.2 and [30]), it does not necessarily represent the fastest executing sequence. Consequently, a different tree expansion algorithm must be used. In addition, the rules governing efficiency vis-à-vis the tree pruning mechanism must also change to tolerate future possible run-time developments:

Definition 16 (Timed Successor Tree). A timed successor tree is a successor tree with the addition of time constraints on its edges [30].

These time constraints are inequalities that group similarly behaving time values. The use of inequalities allows for the avoidance of node/state explosion (see [27] and [29] pertaining this issue). These inequalities must be extracted from the constraints on the TA's edges. Technically, we must have a coupling of possible input with each possible constraint in order to allow for all possible input/time combinations. Practically, however, we can group constraints based on input, which further reduces the possible combinations needed.

For example, in the TA of Figure 4, the inequalities that "cover" the different possible groupings are $c \leq 2$, $2 < c < 3$, $c = 3$, $3 < c < 4$ *and*, $c \geq 4$. All other groupings might introduce non-determinism into our model. If we group the constraints based on inputs, however, we can minimize the possible combinations without introducing non-determinism. Thus, we get $c < 3$, $c = 3$, $3 < c < 4$ *and*, $c \geq 4$ for the input of 0, and $c \leq 2$ *and* $c > 2$ for the input of 1 – these values are used in Figure 5.

Similarly to their task in regular successor trees, nodes in a timed successor tree represent the state of the TA. Edges, representing the stimulation of the TA through input, are marked with an input literal and paired with an associated time constraint. To achieve TA stimulation, literals must be inserted in compliance with the associated time constraint. For example, an edge labeled $0[c = 3]$ would represent the input of 0 when clock $c = 3$, while a label of $1[c < 17]$ would allow for the input of 1 anytime before $c = 17$.

In order to allow for the comparison for optimality between sequences represented by two paths in the tree vis-à-vis their execution time, we must allow them knowledge of their cumulative run-time information.

Definition 17 (Traversal Latency). We define the *traversal latency* from one node in the timed successor tree to its immediate descendant (son), as the difference between the clock value when reaching the node and the value of the constraint that must be satisfied on the edge to its son. Thus, this value quantifies the time

[5] For this work, the transfer from s_k to s_{RS} must be available (as far as the clock guards that constrain the transfer) after the machine arrives at s_k. Otherwise, this transfer will not be applicable.

we must wait before we can traverse towards the next node in the tree. Let us denote this value as $\tau_{n_i \Longrightarrow n_k}$, where n_i is the father of n_k.

Assumption 5. We assume a minimal latency of 1 *within* a node, before the next tree traversal can be executed. Thus, to each latency value we must add 1 to accommodate for this additional time [Justification: Although transitions within the TA were assumed instantaneous (see Assumption 2), latency within states was not. If we assume that at every state the minimal possible work is done, we must allow for a minimal traversal latency, a latency of 1].

Definition 18 (Elapsed Time): We can generalize Definition 17 for an arbitrary descendant k of node j, by accumulating the latency times on the path from j to k. This value will constitute the total time that must elapse during the descent of a timed successor tree from node j to node k while conforming to the timing constraints on the tree's edges. Let $\tau_{j \Longrightarrow k} = \sum(\tau_{j_i \Longrightarrow j_{i+1}} + 1)$[6], where the path from j to k is composed of nodes $j = j_1, j_2, \ldots j_n = k$. Thus, the time to reach node k from the root of the tree is $\tau_{root \Longrightarrow k}$.

For example, in the marked node of the tree in Figure 5, the elapsed time is the summation of delays collected on its path from the root. Traversing from the root, the node marked as 1 can be executed immediately, as it's assumed the clocks are initialized. Assumption 5, however, requires a minimal delay before the next traversal can be executed; hence, the node is marked with a latency of 1. Next we need to traverse to the node marked by 2. Assuming (for the sake of this example) that the clock was not (re)set on the last transfer, the total latency time for the node marked 2 is: $\tau_{root \Longrightarrow 1} + 1 + \tau_{1 \Longrightarrow 2} + 1 = 0 + 1 + 3 + 1 = 5$.

Now we can develop a timed synchronizing tree suited for our needs:

Definition 19 (Timed Synchronizing Tree). A timed synchronizing tree (t-ST), is a timed successor tree pruned for efficiency by one of two rules:

1. **The loop rule:** the *current state uncertainty* associated with a node k is also associated with another node, j, and $\tau_{root \Longrightarrow j} < \tau_{root \Longrightarrow k}$.[7] k is pruned.
2. **The redundancy rule:** the *current state uncertainty* associated with a node k is also associated with other non-terminal nodes as well (regardless of their level in the tree), and $\tau_{root \Longrightarrow j} = \tau_{root \Longrightarrow k}$. Arbitrarily[8], one is selected for future tree extension; the others terminated.

For expansion of the t-ST until a solution is found, we examine the non-terminal with the lowest elapsed time for an uncertainty vector that contains a singleton; the concatenation of the labels on the path from the root to the aforementioned node is a timed synchronizing sequence, $\sigma_{t-ss_{op}}$, optimal with respect to execution time. With each input literal in $\sigma_{t-ss_{op}}$ we associate the timing constraint with which it was coupled on the edge in the t-ST. Using these constraints, we can synchronize the input of sequence literals into the TA (and when testing is concerned, into the t-IUT) to reach a viable synchronizing solution.

If, however, the node with the lowest elapsed time value does not constitute a viable solution, we stem a single level sub-tree with the aforementioned node as its local root, and associate with each new node its respective uncertainty vector and $\tau_{root \Longrightarrow node}$. We repeat the process with a new non-terminal leaf node containing the lowest elapsed time value until a solution is found.

In theory, extending this algorithm to be a RESET sequence for a program or circuit can be accomplished by terminating only when a node with the singleton S_{RS} is found. It does not suffice, however, to find S_{RS} in order to achieve optimality. As long as the clocks are not reset to their initial values (φ_{RS}), we haven't fully RESETted the system. Assuming the existence of a RESET transfer (see Figure 4 – transfer marked $\alpha_{RS}/-[\varphi_{RS}]$ – grayed to portray that it might not actually exist), it does not suffice to concatenate $\sigma_{t-ss(op)} \cup \alpha_{RS}$, as there might be a better solution using a different path (possibly through the transfer marked $1/\beta[\varphi_{RS}]$ from C to A). To overcome these shortcomings, we must continue the

[6] The addition of 1 at every level is in order to accommodate for Assumption 5.

[7] It is possible that a node will become terminal even though when created it was not deemed so. This can occur if a node is created on another sub-branch of the tree with the same *current state uncertainty* yet having a lower elapsed time value.

[8] It is plausible that the wisest selection would be the node located highest in the tree. This would allow for the solution, given that it passes through one of these nodes, to be optimal with respect to length of sequence as well.

search using non-terminal leaf nodes until we find a solution node (containing S_{RS} as singleton) that the edge in the tree leading into the node was associated with the clock RESET output. This will guarantee (assuming it exists) that the solution is actually optimal.

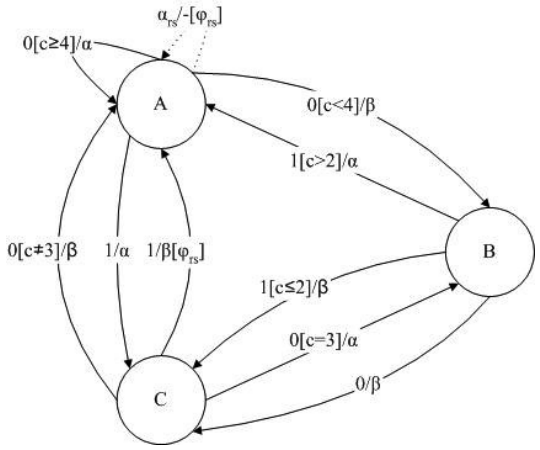

Figure 4. Description of a single clock TA.

4.5. Example for t-ST

Figure 5[9] shows the t-ST for the machine of Figure 4. A solution sequence of $\sigma_{t-ss(op)} = 1[c \leq 2] \cdot 0[c < 3] \cdot 1[c \leq 2]$ can be determined by the tree.

Assuming state A in the RESET state (S_{RS}), the tree must be further extended so as to find a node with the singleton A. One is found on the third level of the tree, $0[c = 3] \cdot 1[c > 2]$. RESET of clocks, however, is not necessarily achieved[10]. If the reset transfer ($\alpha_{RS}/ - [\varphi_{RS}]$) does exist, one possible solution would be its concatenation to the mentioned sequence. This, however, doesn't unequivocally constitute the optimal solution. As a matter of fact, adding the untimed input of 1 to $\sigma_{t-ss(op)}$ will achieve the optimal solution;

$$\pi_{1[c\leq 2]\cdot 0[c<3]\cdot 1[c\leq 2]\cdot 1} = 4 < \pi_{0[c=3]\cdot 1[c>2].\alpha_{RS}} = 6.$$

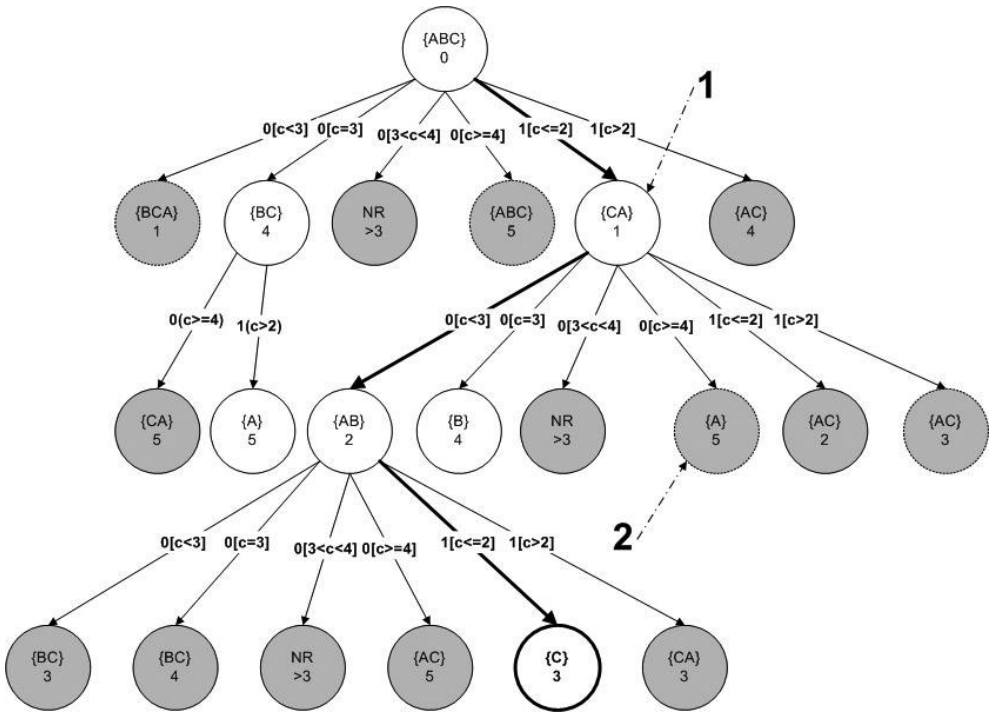

Figure 5. Timed synchronization tree for Figure 4.

[9] There are three points that need to be clarified about this figure:
a) The labeling of each node contains the state uncertainty and the minimal latency accumulated down the path from the root to the node.
b) As defined in Definition 18 and based on Assumption 5, the minimal accumulated latency times include an additional 1 per level traversed.
c) The nodes labeled NR (not relevant) are, as their name suggests, irrelevant at this point in the discussion. They were included for the sake of showing nodes if a dense time model were to be used – see Section 4.8.

[10] If we actually came from node C, the clocks were RESET. If, however, we really transferred from node B, they were not.

4.6. Proof of optimality

In order to prove optimality of the timed synchronizing sequence, hence, the timed RESET sequence extracted using the above algorithm, we must first prove two sub-theorems:

Proposition I:

A descendant node necessarily has a higher execution time than all of *its* ancestors ($\pi_{des} > \pi_{anc}$).

Proof: Inherent from their name, the path to node N_{des} (descendant node) passes through node N_{anc} (ancestor node). Time *must* elapse until we can reach N_{des} from N_{anc}; namely, $\tau_{ans \Longrightarrow dec} > 0$[11]. If the time that must elapse in order to reach N_{anc} is $\tau_{root \Longrightarrow anc} > 0$, the total time needed to reach node N_{des}, $\tau_{root \Longrightarrow des}$, is $\tau_{root \Longrightarrow anc} + \tau_{ans \Longrightarrow des}$; a value obviously greater than $\tau_{root \Longrightarrow anc}$. □

Proposition II:

The length of time that must elapse before we can reach a node j in the t-ST, $\tau_{root \Longrightarrow j}$, is equal to the execution time of the [sub-]sequence the node represents, π_{σ_j}.

Proof: This theorem evolves directly from the method used for timed synchronizing sequence extraction and construction. □

Now the proof of execution time optimality is direct:

Proposition III: A timed synchronizing sequence [timed RESET sequence, resp.] constructed from a non-terminal leaf node in a t-ST with the lowest eplased time is optimal, $\sigma_{t-ss_{op}}$ [$\sigma_{t-RS_{op}}$, resp.], as defined in definition 15.

Proof (by negation): Let us assume the existence of a timed synchronizing sequence, $\sigma_{t-ss_{possible}}$, that executes faster than the sequence found, $\sigma_{t-ss_{found}}$; specifically, $\pi_{\sigma_{t-ss_{possible}}} < \pi_{\sigma_{t-ss_{found}}}$. By Proposition II, $\tau_{root \Longrightarrow possible} < \tau_{root \Longrightarrow found}$ for nodes $N_{possible}$ and N_{found} representing $\sigma_{t-ss_{possible}}$ and $\sigma_{t-ss_{found}}$, respectively.

Since until the first acceptable solution N_{found} was found, only non-terminal nodes with the lowest elapsed time were selected for inspection, $N_{possible}$ must either be another non-terminal leaf node such that $\tau_{root \Longrightarrow possible} < \tau_{root \Longrightarrow found}$ or one of their future descendants satisfying the same inequality.

By method of node selection and Proposition II, at any point in time the non-terminal leaf node with the lowest elapsed time represents a [sub-]sequence that executes at least as fast as [sub-]sequences represented by the other non-terminal leaf nodes and all of their future descendants; thus, $N_{possible}$ cannot represent a faster executing timed synchronizing sequence than N_{found} ($\pi_{\sigma_{t-ss_{possible}}} \geq \pi_{\sigma_{t-ss_{found}}}$). □

4.7. Multi-clock t-IUT

Based on Assumption 3, the discussion so far only contained constraints based on a single clock. We can, however, relax that assumption and still use the same algorithm to find an optimal t-SS, $\sigma_{t-ss_{op}}$ [t-RS, $\sigma_{t-RS_{op}}$]. As with a single clock, we must find time boundaries that define time regions (for multiple clocks regions are polyhedrons of order N =number of clocks) and insert them into the tree[12]. Once we have the time constraints, the remainder of the algorithm functions in the similar manner to what is described above.

4.8. Laxing Assumption 1

Although we believe that the use of a discrete time model is well justified for real machines, for presentation completeness we wish to show that this assumption isn't crucial within the current work.

The discrete time model allowed us to limit the discussion to integer value clock constraints. Use of inequalities as boundaries of ranges, however, expands the range of allowed timing to real values as well. By grouping similarly behaving time values into a single edge on the automaton and matching timed successor tree, we can easily apply the algorithm to real (dense) time.

[11] It is inconsequential if N_{des} is a direct descendant of N_{anc} or an indirect one. In either case we denoted the time that elapses to reach N_{des} from N_{anc} by $\tau_{N_{anc} \Longrightarrow N_{des}}$.

[12] See [27] and [29] for the method used for finding time regions and the criteria for including or excluding regions.

We do need, however, to change some of the definitions presented in this paper. Every definition that uses the set of non-negative integers, $\mathbf{Z}_{\geq 0}$, should be replaced with the set of non-negative reals, $\mathbf{R}_{\geq 0}$ (see for example Definition 2 and Definition 4). This would allow the clocks to accommodate for all possible real values; a simple consequence of a dense time model. As far as the time regions are concerned, we must take into account regions that are not limited to integral values. The region limits can be any real value of clocks. This does not impact our method, as the mathematical use of inequalities is not constrained to integers alone. There will be, however, regions that were excluded as non-relevant[13] when a discrete time model was used, that must be included when a dense model is used.

4.9. Discussion

The method presented uses a search method based on minimum elapsed time to find a solution. This might lead to the notion that termination isn't guaranteed. To show termination is indeed achieved, we distinguish between two cases: when a solution exists and when one does not.

We begin with the case where a solution is lacking. It is obvious that an exhaustive tree spread must occur before we can quit with a negative result. As we are talking about finite automata with a finite set of states, input alphabet and regions in region graph, there are only a finite set of combinations we can go through before we encounter combinations already previously expanded in the tree. The pruning mechanism of Definition 19 would eliminate any node in the t-TS that was examined and extended previously. Thus, the lack of a solution would be discovered when there are no more non-terminal nodes to expand. Given the termination is guaranteed by exhaustive search and combination options ruled out, it follows that if a solution does indeed exist, it will be discovered before all options are depleted.

The last issue that needs to be addressed is the complexity of the generation algorithm; specifically, on the bounds of the size of t-ST.

As mentioned in Section 2.4, the size of ST would be $|I|^j$ at level j. In a worst case scenario, the depth of TS is given by the longest synchronizing sequence possible, is $\frac{1}{2}(n-1)^2 n$ (see section 3.2); thus, the tree size is highly exponential relative to number of nodes ($O(|I|^{n^3})$). Our algorithm would be exponential as well, having $(|I| * |rg|)^j$; where $|rg|$ is the number of regions in the region graph eluded to in section 4.7. This values negatively impacts the feasibility of this class of algorithms (both original and proposed), limiting its practical use to very small problem sets.

5. Conclusion and Future Work

There are many usages for RESET sequences, and most devices as we know them actually provide such a capability. Albeit, it is accomplished in a crude manner: shutting down and rebooting or providing internal dedicated logic. RESET sequences were developed to provide an elegant, controlled solution to the problem. Our method gracefully extends this algorithm to timed devices (circuits), systems, or development software (such as testing software); albeit, to only a small problem set due to space complexity issues.

There are a number of points, however, that deserve further exploration, including: the introduction of delays into transfers – laxing Assumption 2, adding non-determinism, partially specified machines, or laxing any other of the assumptions in Assumption 3. In addition, the algorithm presented in this paper assumed that the delay, or work, performed within all states is minimized to the absolute minimum; hence a maximum common latency of 1 for all nodes (see Assumption 5). In a real system some nodes require more execution times than others. Further work must be done to the order of constructing an optimal RESET sequence that also takes the varying delay within states into account. Finally, a direct sequence generation method for large systems that can be executed within acceptable time and space costs would be desirable. We leave these to further research.

[13] See, for example, the nodes marked as NR (or not-relevant) in Figure 5 – as it is impossible in a discrete model for the clock to be in the range $3 < c < 4$

References

[1] A. V. AHO, A. T. DAHBURA, D. LEE, M. U. UYAR, An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours. *IEEE Trans. Commun.*, 39 (1991), pp. 1604–1615.

[2] ALTERNATING BIT PROTOCOL. (2008, March 28). In *Wikipedia, The Free Encyclopedia*. Retrieved from `http://en.wikipedia.org/w/index.php?title=Alternating_bit_protocol&oldid=201625945`

[3] R. ALUR, D. DILL, A Theory of Timed Automata. it Theoretical Computer Science, 126 (1994), pp. 183–235.

[4] K. A. BARTLETT, R. A. SCANTLEBURY, P. T. WILKINSON, A note on reliable full-duplex transmission over half-duplex links. *Commun. ACM*, 12, 5 (May 1969), pp. 260–261. DOI=10.1145/362946.362970.

[5] G. BEHRMANN, P. BOUYER, K. G. LARSEN, R. PELÁNEK, Lower and upper bounds in zone-based abstractions of timed automata. Int. J. Softw. Tools Technol. Transf. (STTT), 8 (2006), pp. 204–215.

[6] G. BEHRMANN, K. G. LARSEN, J. I. RASMUSSEN, Optimal scheduling using priced timed automata. *SIGMETRICS Perform. Eval. Rev.*, 32 (2005), pp. 34–40.

[7] S. BLOCH, H. FOUCHAL, E. PETITJEAN, S. SALVA, Some Issues on Testing Real-time Systems. *Int. J. of Comp. and Info. Science*, 2 (2001), pp. 230–239.

[8] M. BOZGA, C. DAWS, O. MALER, A. OLIVERO, S. TRIPAKIS, S. YOVINE, Kronos: A Model-Checking Tool for Real-Time Systems. In *Proceedings of the 10th international Conference on Computer Aided Verification* (June 28–July 02, 1998). A. J. Hu and M. Y. Vardi, Eds. Lecture Notes In Computer Science, vol. 1427. Springer-Verlag, London, pp. 46–550.

[9] R. CARDELL-OLIVER, T. GLOVER, A Practical and Complete Algorithm for Testing Real-Time Systems. *Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lyngby, Denmark, (September 14–18, 1998), pp. 251–261. Lecture Notes in Computer Science, 1486, Springer-Verlag, London.

[10] W. Y. CHAN, C. T. VUONG, M. R. OTP, An improved protocol test generation procedure based on UIOs. *SIGCOMM Comput. Commun. Rev.*, 19 (1989), pp. 283–294.

[11] T. S. CHOW, Testing Software Design Modeled by Finite State Machines. *IEEE Trans. Software Eng.*, SE-4 (1978), pp. 178–187.

[12] D. EPPSTEIN, Reset Sequences for Monotonic Automata. *SIAM J. on Computing*, 19(3) (1990), pp. 500–510.

[13] G. GONEC, A Method for the Design of Fault Detection Experiment. *IEEE Trans. on Comp.*, C-19 (1980), pp. 551–558.

[14] A. HESSEL, K. G. LARSEN, B. NIELSON, P. PETTERSSON, A. SKOU, Time-Optimal Test Cases for Real-Time Systems. *Formal Modeling and Analysis of Timed Systems: First International Workshop*, FORMATS 2003, Marseille, France, (September 6–7, 2003), pp. 234–245.

[15] G. J. HOLZMANN, *Design and Validation of Computer Protocols*. Prentice-Hall, 1990.

[16] J. E. HOPCROFT, J. D. ULLMANN, *Intoduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, USA 1979.

[17] E. P. HSIEH, Optimal Checking Experiments for Sequential Machines. *IEEE Trans. on Computers*, C-20 (1971), pp. 1152–1166.

[18] A. KHOUMSI, L. OUEDRAOGO, A new method for transforming timed automata. *Proceedings of the Brazilian Symposium on Formal Methods (SBMF)*, Recife, Brazil, (Nov. 29–Dec. 1, 2004), pp. 101–128. Elsevier.

[19] Z. KOHAVI, *Switching and Finite Automata Theory*. 2^{nd} ed, McGraw-Hill Higher Education (1978).

[20] M. KRICHEN, S. TRIPAKIS, State identification problems for timed automata. *Proceedings of the 17th IFIP Intl. Conf. on Testing of Communicating Systems*, Montreal, Canada, (May 31–June 2, 2005), pp. 175–191. LNCS, 3502, Springer, Berlin.

[21] K. G. LARSEN, F. LARSSON, P. PETTERSSON, W. YI, Compact Data Structures and State-Space Reduction for Model-Checking Real-Time Systems. *Real-Time Systems*, 25 (2003), pp. 255–275.

[22] K. G. LARSEN, M. MIKUCIONIS, B. NIELSEN, A. SKOU, Testing real-time embedded software using UPPAAL-TRON: an industrial case study. *Proceedings of the 5th ACM international Conference on Embedded Software*, Jersey City, NJ, USA, (September 18–22, 2005), pp. 299-306. ACM, New York.

[23] K. LARSEN, P. PETTERSSON, W. YI, Uppaal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1) (1997), pp. 134–152.

[24] D. LEE, M. YANNAKAKIS, Principles and Methods of Testing Finite State Machines – a Survey. *Proc. of the IEEE*, 84 (1996), pp. 1090–1123.

[25] G. LUO, G. VON BOCHMANN, A. F. PETRENKO, Test Selection Based on Communicating Nondeterministic Finite State Machines Using a Generalized Wp-method. *IEEE Trans. Software. Eng.*, 20 (1994), pp. 149–162.

[26] K. NAIK, B. SARIKAYA, Protocol Conformance Test Case Verification using Timed – Transition. *Proceedings of the 14^{th} International Symposium on Protocol Specification, Testing and Verification*, Vancouver, Canada, (1995) pp. 103–118. Chapman & Hall, Ltd. London, UK

[27] E. PETITJEAN, H. FOUCHAL, From Timed Automata to Testable Untimed Automata. *24th IFAC/IFIP International Workshop on Real-Time Programming*, Schloss Dagstuhl, Germany, (May 30–June 2, 1999).

[28] K. K. SABNANI, A. T. DAHBURA, A Protocol Test Generation Procedure. *Comp. Networks ISDN Sys.*, 15 (1998), pp. 285–297.

[29] S. SALVA, H. FOUCHAL, S. BLOCH, Metrics for Timed Systems Testing. *4th OPODIS International Conference on Distributed Systems*, Paris, France, (December 20–22, 2000), pp. 177–200. Suger, Saint-Denis, rue Catulienne, France.

[30] A. STULMAN, S. BLOCH, H. G. MENDELBAUM, Optimal Homing Sequences for Machines with Timing Constraints. *WSEAS Transactions on Systems*, 9 (2004), pp. 2793–2801.

[31] A. STULMAN, S. BLOCH, H. G. MENDELBAUM, Optimal Synchronizing Sequences for Machines with Timing Constraints. *ESA'05 International Conference on Embedded Systems and Applications*, Las Vegas, NV, USA, (June 27–30, 2005), pp. 45–52.

Received: November, 2009
Revised: December, 2010
Accepted: March, 2011

Contact address:
Ariel Stulman
CS Department
Jerusalem College of Technology
Jerusalem, Israel
e-mail: stulman@jct.ac.il

ARIEL STULMAN received his bachelor's degree in technology and applied sciences from the Jerusalem College of Technology, Jerusalem, Israel, in 1998. He then went on to get his masters from Bar-Ilan University, Ramat-Gan, Israel, in 2002. In 2005 he achieved a Ph.D. from the University of Reims Chanpagne-Ardenne, Reims, France. As of 2006 he holds a position at Computer Department of the Jerusalem College of Technology. His research interests are in the fields of software testing, formal methods, real-time systems, and web technologies and testing.

Dr. Stulman is a member of the ACM.