Verifying Timed Behavior Automata with Input/Output Critical Races*

David K. Probst and Hon F. Li
Department of Computer Science
Concordia University
1455 de Maisonneuve Blvd. West
Montreal, Quebec H3G 1M8

Abstract. Timed behavior automata are finite-state generators of *timed behaviors*, which are infinite timing-constrained pomsets of system events. Automatic verification is not showing inclusion of infinitary timed string languages. Rather, model checking starts by linking specification mirror and implementation network; verification is showing satisfaction of an infinite timing-constraint graph with recurrence structure. Branching controlled by input/output races models more interesting timing properties such as timeout and exception handling. We show how to mirror timed behavior automata with input/output races, and show that constraint-graph satisfaction can still be computed efficiently by linear-time shortest-path algorithms. The advantage of TBA over timed ω-automata is ease of mirroring. Untimed behavior automata [14] are modified in two ways: (i) output actions are performed inside a timing window relative to their enabling, and (ii) it is assumed that input actions are performed inside a timing window relative to their enabling. Failure and timeout semantics define the process response to violations of this assumption protocol.

1 Introduction

Many researchers are developing formal methods for the specification, verification and systematic design of reliable *reactive* systems, such as control and communication systems and embedded real-time systems. Reactive systems are *open* systems that maintain an ongoing interaction with their environment. Interest in verifying timing properties has focused attention on proposals for modelling, specifying and reasoning about real-time systems [1–10]. Most work on timed reactive systems starts from a semantic model of *timed behavior* as a sequence of timed states. In this model, behaviors are infinite sequences of states—where states are stamped by *time*—that may occur in a possible run of the timed system. The computational model, i.e., the effective representation of realizable timed systems, is typically a timed transition system or a timed ω-automaton

* This research was supported in part by NSERC grants A3363, A0921 and MEF0040121, and by Micronet, A Network of Centres of Excellence. E-mail: probst@crim.ca and probst@vlsi.concordia.ca.

[9]. In this paper, we develop an alternative to the standard approach, viz., the *timed behavior automata* introduced in [16], which are effective representations of realizable timed systems based on a model of partially-ordered, dense, branching time. The main advantage of our computational model, TBA, is an easily-applied, simple theory without state explosion.

In contrast to the standard approach, TBA starts from a semantic model of *timed behavior* as a timing-constrained pomset of system events [11]. In this model, behaviors are infinite pomsets of system events—where successor arrows are stamped by *time bounds*—that may occur in a possible run of the timed system. Time bounds are necessary timing constraints among (possibly nonlocal) system events. They refine two aspects of untimed pomsets and pomtrees, viz., (i) necessary temporal precedences, and (ii) the requirement of justice (weak fairness). The computational model is timed behavior automata, which are finite-state generators of sets of infinite timing-constrained pomsets (equivalently, of an infinite timing-constrained pomtree). Race-controlled branching is new in this paper.

In TBA, there are minimum and maximum delays in the scheduling of enabled actions, and special "concentrators" to allow nonbinary delay constraints to be verified efficiently by shortest-path algorithms in graphs. Using restricted TBA, which disallows mixed-type critical races, we were able to prove timing properties such as timing-window bounds on system responses, and show that system inputs do not arrive too fast [16]. We achieved tractable (polynomial-time) model checking of finite-state real-time systems. The question left posed in [16] was whether extension of the branching structure of restricted TBA to include more interesting timing properties such as timeout and exception handling—by allowing mixed-type critical races to control TBA branching—would substantially complicate the specification and verification picture. Introducing timing windows for input actions (for timeout) and allowing temporary disruption of output actions (for exception handling) does not significantly increase the difficulty of forming the mirror of a timed behavior automaton used as a requirements specification. As a result, the additional complexity and state explosion of timed ω-automata are avoided.

The semantic model (notion of timed behavior) and the computational model (class of automata used to generate or recognize timed behaviors) have a major impact on the practicality of real-time model checking. The timed ω-automata school defines verification as showing language inclusion, and tests the emptiness of the intersection of the implementation automaton and the complement of the specification automaton. The problem is that nontrivial specification automata do not complement well (problems range from exponential blowup to nonexistence). With a weaker computational model, one can prove bounded-response and bounded-invariance properties of timed transition systems (for example, using deductive bounded-operator reasoning). More practical but often deductive, this approach has not been extended to more complicated real-time properties. The primary advantage of TBA is the ease of forming the complement of the specification. Verification is showing satisfaction of an infinite timing-constraint graph with recurrence structure [8]. By incorporating timeout and exception handling into TBA, we broaden the class of real-time properties that can be specified and verified (the broadened class is surprisingly wide), and preserve tractable (polynomial-time) automatic verification. The satisfaction of finite timing-constraint graphs is checked efficiently by shortest-path algorithms in graphs. The recurrence structure provides finite checkability of the infinite constraint graph.

In general, timing constraints in *closed* timed systems are enforced by some mixture of bounded invariance (which refines "push-away" causality) and bounded response (which refines "pull-back" justice). Refinement has a distinctive flavor in partial-order representations, and is only partly from the qualitative to the quantitative level of modelling. When timing assumptions are added to a precedence automaton with justice, zero-valued push-away values become strictly-positive real numbers, while "finite but unbounded" pull-back values (written $-\infty$) become strictly-negative real numbers. The branching structure in timed behavior automata allows input/output races to control the resolution of *some* nondeterministic choice. Timing constraints are independently specified for each branch. Race-controlled branching (daemon choice) is in addition to any branching due to input or output choice. That is, TBA branching models both same-type nondeterministic choice and mixed-type critical races.

As in [15–16], we assume some familiarity with [14], primarily for termination of unfolding of networks of behavior automata. The following features of untimed behavior automata are retained in timed versions: (i) a finite partial-order representation that explicitly distinguishes concurrency, branching and recurrence, and (ii) a state encoding that is both constraint comprehensive (includes all constraints) and state minimal (has fewest states). There is no time component in state encodings, and there are no timed states.

2 Abstract Specification of Timed Reactive Systems

Abstract specifications refer to externally-visible computational behaviors. An untimed behavior automaton generates sets of partially-ordered computations with precedence constraints and justice (weak fairness) [12–14]. In the untimed case, (i) no action may occur until it is enabled, and (ii) once enabled, an output action must occur eventually. A timed behavior automaton generates sets of partially-ordered computations with timing constraints among (possibly non-local) events. There are minimum-delay and maximum-delay constraints, called push-away and pull-back arrows, respectively. In the timed case, (i) no action may occur until it has been continuously enabled for a minimum delay, and (ii) once an action has been continuously enabled for a maximum delay, it must occur immediately. In the assumption/guarantee style of specification, violations of timing constraints on input events are handled by failure and timeout semantics.

In restricted TBA, an output action remains scheduled during a timing window relative to its enabling, while an input action remains scheduled until it occurs; in full TBA, input actions also have timing windows and output actions may be briefly disrupted inside their timing windows. For a timed system to be correct, the timing constraint graph produced by coupling specification mirror mP to implementation network *Net* must be feasible.

A process P has disjoint sets of input and output ports. Ports may be the loci of generalized control elements such as tests and conditions. Process behaviors result from use of process P by P's environment. P's input actions are under the control of P's environment, while P's output actions are under the control of P. An input action may be process-scheduled testing of an environment-controlled state predicate.

Safety properties constrain both the process and its environment. A safety violation is the performance of an action that is not scheduled. A process receiving unsafe input obeys its *failure* semantics. Liveness properties also constrain both the process and its environment. A liveness violation is the nonperformance of an enabled action before the expiration of its timing window. A process failing to receive timely input obeys its *timeout* semantics.

2.1 Formal Construction of Behavior Automata

Timed behavior automata are constructed in four phases. Untimed versions are succinct encodings of sets of infinite pomsets [12,13]; timed versions add timing constraints. First, there is a "small" deterministic finite-state machine D that expresses the branching and recurrence structure of the process. In full TBA, branching is caused by any of the following: (i) input choice, (ii) output choice and (iii) input/output critical race. Second, there is an expansion of each transition of dfsm D into an finite poset, with sockets to define poset concatenation. Third, there is a labelling of successor arrows to define the state encoding. Fourth, each successor arrow is replaced by a matched pair consisting of: (i) a push-away arrow labelled with a strictly-positive real number, and (ii) a pull-back arrow labelled with a strictly-negative real number. Maxima and minima of presets are related to members of presets by special timing-constraint arrows labelled with $+0$.

We sketch the formal definition of behavior automaton. Given disjoint alphabets Act (process actions), Arr (successor arrow labels), Com (dfsm D transitions) and Soc (sockets), define Pos as the set of finite labelled posets over Act $\cup$ Soc. Each member of Pos is a labelled poset (B,Γ,ν), where (i) Γ is a partial order over B $\subseteq$ Act $\cup$ Soc, and (ii) $\nu : \Omega \rightarrow$ Arr assigns a label to each element in the successor relation Ω (the transitive reduction of Γ). A behavior automaton is a 3-tuple (D,ξ,ψ), where (i) D is a dfsm over Com, (ii) ξ: Com $\rightarrow$ Pos maps dfsm transitions to labelled posets, and (iii) ψ: Soc $\rightarrow$ powerset(Act) maps sockets to sets of process actions. ψ defines which process actions may fill a socket when a command is concatenated to a sequence of earlier commands. There is an imaginary reset action $\bullet$. In timed behavior automata, successor arrows in posets are replaced by matched pairs of push-away and pull-back arrows. When multiple actions enable an action a, a's pull-back arrow is incident to the pseudoaction that is the maximum of a's preset. Pseudoactions trivially extend the alphabet Act, but may be removed by transitive closure and disjunction.

3 Timed Behavior Automata

Figure 1 shows an untimed behavior automaton for a C-element, where the labels on successor arrows provide a "state" encoding whose only function is to terminate the unfolding of *closed* systems of coupled behavior automata during verification. Separate specification of timing properties in TBA causes this function—of labels—to be preserved unchanged. The untimed specification is written in the standard assume/guarantee style, where the dashed arrows are the assumption protocol and the solid arrows are the guarantee protocol. As long as the environment satisfies the assumption protocol, the process will satisfy the guarantee protocol. The failure semantics defines the process response to the first occurrence of bad input. Two possibilities are: (i) the process becomes undefined, and (ii) the bad input is ignored. The brackets are a *justice* requirement on the process; they assert that an enabled process output action must occur eventually. There is no justice requirement on the environment; a process input action may be enabled yet never occur. When timed behavior automata are constructed by adjoining timing assumptions to untimed behavior automata, justice requirements are replaced by quantitative pull-backs to pseudoactions (maxima of enabling presets).

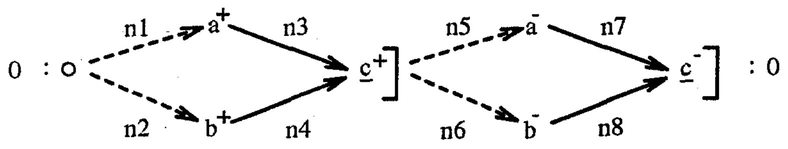

Fig. 1. Untimed behaviour automaton for a C-element

Figure 2 shows a restricted timed behavior automaton for a timed wire (minus the state-encoding labels) with push-away and pull-back arrows to specify timing windows. The wire remains excited after input a for at least 2 units, but not more than 5 units, before responding with output $\underline{b}$. We use matched pairs of arrows: a push-away (bounded-invariance) arrow $a \longrightarrow \underline{b}$ labelled with +2, and a pull-back (bounded-response) arrow $a \longleftarrow \underline{b}$ labelled with −5 (upper bounds are expressed as negative lower bounds in the opposite direction) [3]. Both push-away and pull-back arrows in constraint graphs are minimum-delay constraints. The triangle inequality in constraint graphs is: to derive a timing label t on an arbitrary $a \longrightarrow b$, find all directed paths from a to b, and compute the sum of minimum delays along each path; if there are distinct sums, then take the maximum.

Figure 3(b) shows the sensible way to quantify a justice requirement in a partial-order representation with timing constraints. A pull-back constraint is specified relative to an enabling condition (pseudoaction ξ is the last of d and e). If β is the last of a and b, then β is a or b, and a pull-back to β is a pull-back to a or b. In Figure 3(b), there is no minimum delay from $\underline{f}$ to d or e. Rather,

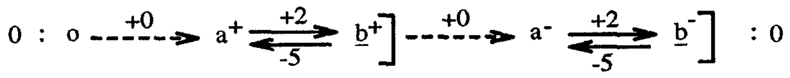

Fig. 2. Timed behavior automaton for a wire

either $d - \underline{f} \geq -6$ or $e - \underline{f} \geq -6$. We say that pull-back arrows $d \leftarrow \underline{f}$ and $e \leftarrow \underline{f}$ form a complete disjunctive set.

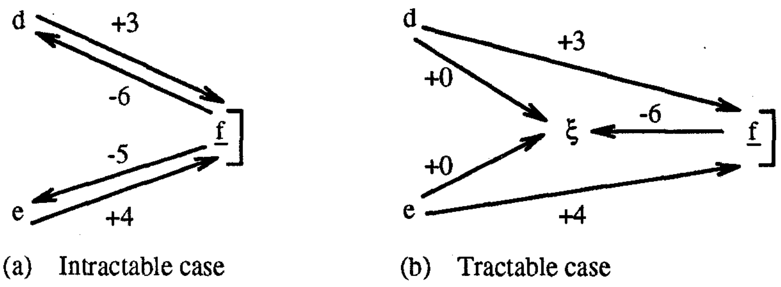

Fig. 3. Quantifying justice in a partial-order representation

Figure 4 shows a restricted timed behavior automaton for a timed C-element. $\beta = \max\{a, b\}$ and $\alpha = \min\{a, b\}$. Action $\underline{c}$ remains scheduled during a timing window relative to β, while α remains scheduled until it occurs. $\psi(\circ) = \{\bullet, \alpha\}$. In the appropriate context, the special push-away arrows with timing label +0 may be read as zero-valued pull-back or push-away arrows in the opposite direction—for the purposes of applying the triangle inequality—provided their assertions are interpreted disjunctively. To derive a timing label t on a pull-back arrow $a \leftarrow b$, recursively find all complete disjunctive sets of directed paths from b to a, and compute the sum of minimum delays along each path; if there are distinct sums in a given disjunctive set, then take the minimum.

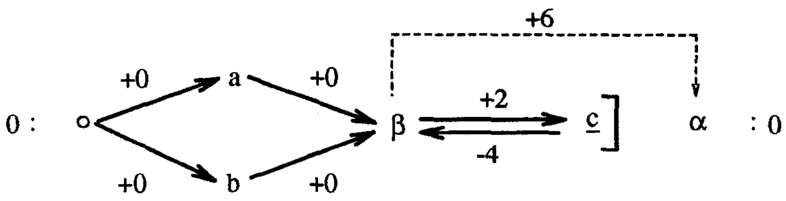

Fig. 4. Timed behavior automaton for a timed C-element

3.1 Branching Controlled by Input/Output Critical Races

Figure 5 shows a timed behavior automaton for a wire with timeout. The deterministic finite-state machine D underlying this automaton has a start node (reached by reset action •) and two self-loops: a transition n for normal operation, and a transition t for timeout. In Figure 5, both n and t have been expanded into finite posets with timing constraints. Sockets define concatenation of posets; here, both sockets satisfy $\psi(\circ) = \{\bullet, \underline{b}\}$. In t, socket $\circ$ appears in the middle of the poset. As long as the environment satisfies the assumption protocol (given by the dashed arrows), the process will satisfy the guarantee protocol (given by the solid arrows). Since the assumption protocol contains (i) push-away arrows, (ii) pull-back arrows and (iii) race-controlled branching, we will provide the semantics in stages.

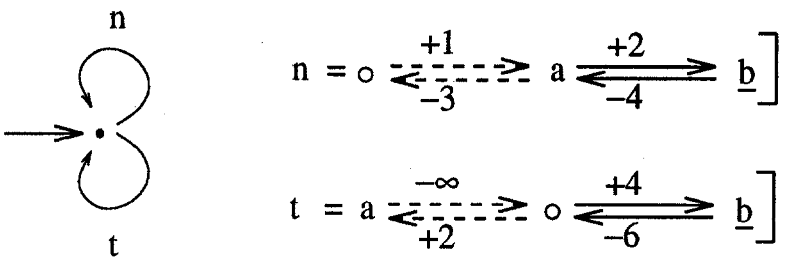

Fig. 5. Timed behaviour automaton for a timeout wire

To deal with inputs that might arrive too fast, normal transition n has a dashed push-away arrow from socket $\circ$ to input action a labelled with +1. The failure semantics is that earlier input may be *ignored.* To deal with inputs that might arrive too slowly, there are two items: (i) timeout transition t has a dashed push-away arrow from $\circ$ to a labelled with +2, and (ii) normal transition n has a dashed pull-back arrow from a to $\circ$ labelled with −3. The timeout semantics is that the process *may* time out as soon as 2 time units without input have elapsed—relative to the most recent • or $\underline{b}$—but *must* time out as soon as 3 time units have elapsed. Any later input will be ignored until input is re-enabled. Between 2 and 3 time units, the decision to time out is taken nondeterministically.

Mirror construction becomes clear as soon as we see what is required by the requirements specification. The process may ignore input a in $[0, 1)$. The process must select n for input in $[1, 2)$. The process may select either n or t for input in $[2, 3)$, but must select t after 3 time units. If either n or t is selected, then the process must produce output $\underline{b}$ in the specified timing window. We use mirror mP of specification P as a conceptual implementation tester. The mirror provides or withholds input to exercise each requirement of the assume/guarantee protocol. This means in particular that mP determines both (i) whether 1 is a lower bound on ignoring input a, and (ii) whether 2 and 3 are lower and upper bounds on noticing the absence of input a. It checks whether these constraints are supported by chains of implementation constraints. Correctness of race-controlled branching is satisfaction of a timing-constraint graph, nothing more.

Correctness is a preorder in the untimed case [12–14]. This remains true in the timed case. We show this for the specification in Figure 5. The implementation lower bound for necessarily accepting input must be less than or equal to the specification lower bound for necessarily accepting input. The implementation timeout *decision* window—that is, the interval between lower and upper bounds for timing out—must be contained within the specification timeout decision window. The implementation *output* window—that is, the interval between lower and upper bounds for performing an output action—must be contained within the corresponding specification output window, for either timeout or normal operation.

4 Composition of Timed Behavior Automata

We construct an extremely simple implementation network to study both composition and verification. The network fails in several ways to implement the specification in Figure 5.

Figure 6 shows a timed behavior automaton for a fork with timeout. There are two self-loops in dfsm D: a normal transition n and a timeout transition t. Both sockets satisfy $\psi(\circ) = \{\bullet, \beta\}$. Again, socket $\circ$ is in the middle of t. The fork may ignore input in the interval $[0, 1)$. The fork may time out if 6 time units without input have elapsed, but must time out if 7 time units have elapsed. All times are relative to the most recent $\bullet$ or β.

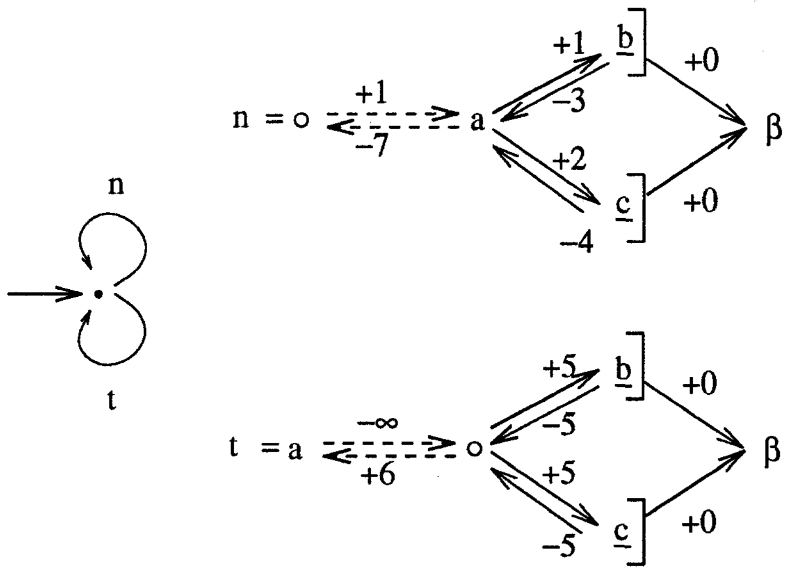

Fig. 6. Timed behaviour automaton for a timeout fork

Figure 7 shows a timed behavior automaton for a C-element with an acknowledgment protocol. There is recurrence but no branching in dfsm D. The socket satisfies $\psi(\circ) = \{\bullet, \underline{c}\}$. The C-element may ignore input in the interval $[0, 1)$ relative to the most recent $\bullet$ or $\underline{c}$.

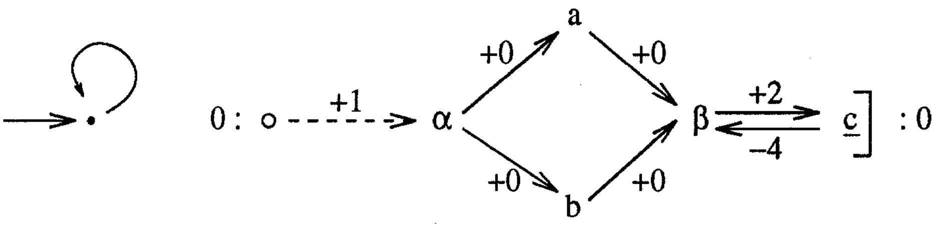

Fig. 7. Timed behaviour automaton for a C-element with an acknowledgment protocol

Figure 8 shows a fragment of the timing-constraint graph produced when the outputs of the timed fork become the inputs of the timed C-element. This creates a timeout wire. Figure 8 contains constraint information from two branches of the closed-system pomtree. The central horizontal line is normal operation; the upper and lower lines are timeout. The pair of dashed arrows shows the timeout decision window of the fork relative to its output β. When is the earliest that the network could time out relative to $\underline{c}$? Since $-4+6 = 2$, the network cannot time out before 2 time units. When is the latest it could time out? Since $-7+2 = -5$, the network must time out after 5 time units. That is, the network timeout *decision* window is $[2, 5]$. In both cases, we have been following chains of arrows regardless of type between upper a and central $\underline{c}$. When the network does time out, since $-4+5+2 = 3$ and $-4+-5+2 = -7$, we deduce the timing bounds on network timeout output $\underline{c}$. The network timeout *output* window is $[3, 7]$. Here, we have been following chains of arrows regardless of type between central $\underline{c}$ and lower $\underline{c}$. The fork may ignore input in the interval $[0, 1)$ relative to β, so the network may ignore input earlier than output $\underline{c}$. Here, we have observed that $\underline{c}$ is at least 2 time units after β. We do not know the dashed push-away from $\underline{c}$ to a that is necessary to prevent all component failure.

5 Correctness and Race-Controlled Branching

We define correctness of a timed system by using mirror mP of specification P as a conceptual implementation tester. We form closed real-time system S by linking mirror mP to implementation *Net*. Unfolding the resulting finite-state generator of closed system S produces a timing-constrained pomtree with constraints from all assumption and guarantee protocols. The timing-constrained pomtree is in fact a timing-constraint graph. Correctness is defined as satisfaction of this graph taking into account the semantics of race-controlled branching.

In restricted TBA, mirror mP is formed by inverting the type of P's actions and the assumption/guarantee interpretation of P's arrows, turning P's

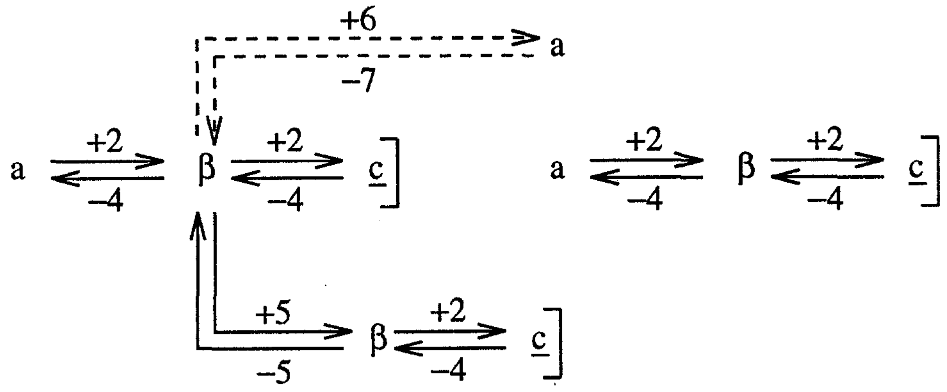

Fig. 8. Pomtree fragment: timeout fork composed with timed C-element

dashed arrows into solid arrows and vice versa. Brackets on mP actions and special arrows are preserved unchanged. Since each instance of branching in closed real-time system S is under the control of either mirror mP or some component of implementation *Net*, we can check whether intraprocess guarantee protocols support interprocess assumption protocols in pomtree S. Dashed arrows coming from implementation components are proof obligations to show that these components do not receive unsafe input. Dashed arrows coming from mirror mP are proof obligations to show that the implementation does not violate the guarantee protocol of the specification.

In full TBA, mirror formation is only slightly more elaborate. We perform type inversion as before, but put a different interpretation on *some* solid arrows in mirror mP. Consider mP solid arrows incident to or from mirrored input actions in normal and timeout transitions at daemon choice points. The push-away solid arrow in normal transition n of mP should guarantee that there is no input safety violation in any implementation component. The pull-back solid arrow in normal transition n of mP should be an upper bound on when implementation *Net* must time out. The push-away solid arrow in timeout transition t of mP should be a lower bound on when implementation *Net* may time out.

5.1 Safety Correctness

Consider a dashed push-away arrow representing the assumption that inputs do not arrive too fast to (i) an implementation component (input safety violation), or (ii) mirror mP (output lower-bound violation). For each dashed push-away arrow $a \rightarrow b$ with timing label t, there must be a solid directed path from a to b whose weight is at least t. Portions of the solid path may consist of chains of solid pull-back arrows. One solid path from a to b with appropriate weight is sufficient to verify the dashed push-away constraint.

5.2 Liveness Correctness

Consider a set of dashed pull-back arrows representing the assumption by mP that the implementation does not produce an output too slowly (output upper-bound violation). This is a complete disjunctive set of dashed pull-back arrows from a bracketed system action c to each member of its noncausal preset $A = \text{pre}(c)$. Each dashed pull-back arrow $a \leftarrow c$, $a \in A$, must be supported by the weakest solid pull-back chain $a \leftarrow c$ in the complete disjunctive set of solid directed pull-back paths from c to a. By well-behavedness, there is only one such disjunctive set for each action $a \in A$. All the solid directed paths in a complete disjunctive set are necessary to verify the dashed pull-back constraint. This condition presupposes the liveness correctness of the untimed system, viz., the causal preset of c as determined by solid push-away chains must not be a proper superset of the noncausal preset of c as determined by dashed push-away arrows [14].

5.3 Race-Controlled Branching Correctness

In mirror mP, consider the (newly) solid arrows that define the timeout decision window and the (newly) dashed arrows that define the timeout output window, viz., the solid push-away arrow in t, the solid pull-back arrow in n, and the matched pair of dashed arrows in t. Branching correctness means that both (i) the implementation decision window is contained within the specification decision window, and (ii) the implementation output window is contained within the specification output window.

6 Verification Example

The characteristic feature of the model-checking algorithms in [13–16] is that, in the recursive generation of system actions in closed system S, all branching is caused by the *output choice* of some process, possibly mP. In race-controlled branching, this is no longer the case. Specification P may contain a branch point whose outgoing transitions are selected by (i) whether timeout does occur, or (ii) whether an exception is raised. A daemon in mP systematically trys both possibilities at each race-controlled branch point. Suppose timeout does not occur. Then, there is an mP-assumed upper bound on the interval without input; is it guaranteed? Suppose timeout does not occur. Then, there is an mP-assumed lower bound on the interval without input; is it guaranteed? The daemon generates mirrored input actions, and then asserts timing-constraint assumptions. In daemon choice, mP assumptions must be supported by *Net* guarantees.

Rather than systematically describing the modifications to the restricted-TBA verification algorithm [16] when race-controlled branching is incorporated, we work the following verification example. Figure 5 (a timeout wire) is the specification, and Figures 6 and 7 (i.e., a timeout fork linked to a timed C-element) is the implementation network. The first task is to determine whether input-failure lower bounds of implementation components are guaranteed when implementation *Net* is driven by specification mirror mP (i.e., when there is a solid push-away arrow from $\underline{c}$ to a labelled with +1). Adding this arrow to Figure 8, we easily verify the (fork) assumed β to a separation of +1 ($2+1 \geq 1$), and the (C-element) assumed $\underline{c}$ to α separation of +1 ($1+1 \geq 1$). Although α is not shown in Figure 8, there is a push-away arrow from a to α labelled with +1. However, the network output window $[4,8]$ is not contained within the specification output window $[2,4]$. Only one of the two chains of *Net* solid arrows supports the corresponding mP dashed arrow [16]. Implementation *Net* is incorrect.

Because this is only an example, we proceed to determine whether race-controlled branching is correct. From Section 4, the network timeout decision window is $[2,5]$. This is not contained in the specification timeout decision window $[2,3]$. mP says the lower bound on noticing the absence of input is 2; *Net* agrees ($-4+6 \geq 2$). mP says the upper bound is 3; *Net* disagrees ($-7+2 \not\geq -3$). The network does not time out too soon, but may time out too late. Is the network timeout output window correct?

We determine whether input lower bounds of implementation components are guaranteed when the fork times out. The first fork timeout leads to network timeout output, since the C-element does not fail for this input ($-4+5 \geq 1$). From Section 4, which *assumed* no failure, the network timeout output window is $[3,7]$. This is not contained in the specification timeout output window $[4,6]$. mP says the lower bound on producing output is +4; *Net* disagrees ($-4+5+2 \not\geq 4$). mP says the upper bound is -6; *Net* disagrees ($-4+-5+2 \not\geq -6$). The network satisfies neither bound. Changing the input-failure lower bound of the C-element from +1 to +2 shows that this example is less trivial than it appears, and might be developed into a benchmark. The interesting part is repetitive timeout by the fork until its input is no longer ignored by the C-element. With a lower bound of +2, the network output window changes from $[3,7]$ to $[8,12]$. The possibility of internal failure during timeout leads to moderately interesting algorithm design.

If the implementation had been correct, then the termination table would have contained two states of closed system S [14]. These two states differ only in whether mP and fork are both at the end of transition n or both at the end of transition t.

7 Exception Handling

Timing windows for input actions give rise to timeout, while (temporary) disruption of output actions gives rise to exception handling. Both are mixed-type (input/output) critical races. The notion behind timeout is *urgency* (certain actions must be performed by certain times), while the notion behind exception handling is *importance* (at any moment, a process must be performing its most important task). In some sense, exception handling is the dual of timeout. Timeout means that after output $\underline{b}$ is produced, either new input is received in timing window 1, or else new output is produced in timing window 2. Exception handling means that after input a is received, either more important input c is received before $\underline{b}$ is produced, or the original timing-window guarantee for $\underline{b}$ is met. We change this to make exception handling the *strict* dual of timeout. After input a is received, either exceptional input c is received in timing window 1, or else normal output $\underline{b}$ is produced in timing window 2.

This is not the standard intuition, viz., that an executing task Σ defines a floating window during which a priority interrupt is enabled (i.e., from task initiation to normal termination of Σ). Instead, we use a fixed upper bound on the interval, starting at input a, during which the production of $\underline{b}$ may be disrupted by the arrival of c; this models the commit time to $\underline{b}$. To avoid paradox, the commit time should be less than or equal to the lower bound for the production of $\underline{b}$.

8 Conclusion

The new idea in this paper is the fusion of (i) timing windows for input actions, and (ii) timing-constraint graphs with recurrence structure. Since early and late inputs are discarded, an input action occurs inside its timing window or not at all. Actions—including those involved in race-controlled branching—are generated without reference to the passage of time. Constraint checking is independent of generation. Timing constraints do not make model checking harder. During verification, all branching in closed system S is caused by either (i) process output choice, or (ii) mP daëmon choice. In both instances, "causally enabled" system actions are recursively generated *before* timing constraints are asserted.

In the interleaving approach, there are two ways to add timing constraints to untimed behaviors (i.e., sequences of untimed states): either (i) add constraints *locally* between adjacent states (as in timed transition systems), or (ii) add constraints *globally* between arbitrary states (as in timed ω-automata) [7]. Adding timing constraints globally is necessary to model *real* real-time systems. For example, the login procedure of Nicollin and Sifakis needs input choice, as well as both local and global timeout [10]. Adding global constraints to ω-automata leads to complex theory and difficult implementation. Automatic verification with timed automata is currently impractical because partition of the uncountable state space into finitely many regions produces unmanageably large regions. Timed behavior automata combine (i) the flexibility of global timing constraints, (ii) a simple theory, and (iii) minimal state explosion. Even with race-controlled branching, time advancement is not used to generate timed behaviors during verification. This avoids the additional state explosion due to time in other approaches. We have no interest in which sequences of timed states actually occur; we only verify timing constraints.

We expect the addition of race-controlled branching to the current implementation of restricted TBA—in the POM verification system—to present no major programming difficulties [16].

References

1. M. Abadi and L. Lamport, *An old-fashioned recipe for real time*, in W.-P. de Roever, (Ed.), Real-Time: Theory in Practice, REX Workshop on Real-Time, Proceedings, Lecture Notes in Computer Science 600, Springer–Verlag, 1992, pp. 1–27.
2. R. Alur, C. Courcoubetis and D. Dill, *Model checking for real-time systems*, in Proceedings of the Fifth Annual Symposium on Logic in Computer Science, IEEE Computer Society Press, 1990, pp. 414–425.
3. R. Alur, *Techniques for automatic verification of real-time systems*, Ph.D. Thesis, Stanford University, Report STAN-CS-91-1378, August 1991.
4. R. Casley, R.F. Crew, J. Meseguer and V.R. Pratt, *Temporal structures*, Math. Structures in Computer Science, **1:2**, July 1991, pp. 179–213.
5. W.-P. de Roever, (Ed.), Real-Time: Theory in Practice, REX Workshop on Real-Time, Proceedings, Lecture Notes in Computer Science 600, Springer–Verlag, 1992.
6. D.L. Dill, *Timing assumptions and verification of finite-state concurrent systems*, in J. Sifakis, (Ed.), Automatic Verification Methods for Finite State Systems, Proceedings, First Workshop on Computer-Aided Verification, Lecture Notes in Computer Science 407, Springer–Verlag, 1990, pp. 197–212.
7. T.A. Henzinger, *The temporal specification and verification of real-time systems*, Ph.D. Thesis, Stanford University, Report STAN-CS-91-1380, August 1991.
8. F. Jahanian and A.K.-L. Mok, *A graph-theoretic approach for timing analysis and its implementation*, IEEE Trans. on Computers, **C-36:8**, August 1987, pp. 961–975.
9. O. Maler, Z. Manna and A. Pnueli, *From timed to hybrid systems*, in W.-P. de Roever, (Ed.), Real-Time: Theory in Practice, op. cit., pp. 447–484.
10. X. Nicollin, J. Sifakis and S. Yovine, *From ATP to timed graphs and hybrid systems*, in W.-P. de Roever, (Ed.), Real-Time: Theory in Practice, op. cit., pp. 549–572.
11. V.R. Pratt, *Modelling concurrency with partial orders*, Int. Journal of Parallel Prog., **15:1**, February 1986, pp. 33–71.
12. D.K. Probst and H.F. Li, *Abstract specification, composition and proof of correctness of delay-insensitive circuits and systems*, Concordia University, Report CONC-CS-VLSI-88-2, April 1988 (Revised March 1989).
13. D.K. Probst and H.F. Li, *Using partial-order semantics to avoid the state explosion problem in asynchronous systems*, in E.M. Clarke and R.P. Kurshan, (Eds.), Second Workshop on Computer-Aided Verification, June 1990, DIMACS Series, Vol. 3, 1991, pp. 15–24. Also Lecture Notes in Computer Science 531, Springer–Verlag, 1991, pp. 146-155.
14. D.K. Probst and H.F. Li, *Partial-order model checking: A guide for the perplexed*, in K.G. Larsen and A. Skou, (Eds.), Third Workshop on Computer-Aided Verification, Proceedings, Department of Mathematics and Computer Science, Aalborg University, Report IR-91-5, July 1991, pp. 405–416. Also Lecture Notes in Computer Science 575, Springer–Verlag, 1992, pp. 322–331.
15. D.K. Probst and L.C. Jensen, *Controlling state explosion during automatic verification of delay-insensitive and delay-constrained VLSI systems using the POM verifier*, in S. Whitaker, (Ed.), Proceedings of the 3rd NASA Symposium on VLSI Design, Moscow, ID, October 1991, pp. 8.2.1–8.2.8.
16. D.K. Probst and H.F. Li, *Verifying timed behavior automata with nonbinary delay constraints*, in G.v. Bochmann and D.K. Probst, (Eds.), Fourth Workshop on Computer-Aided Verification, Participants' Proceedings, July 1992, pp. 121–134. Also Lecture Notes in Computer Science 663, Springer–Verlag, 1993, pp. 123–136.