To appear at ASYNC'05

Modeling and Verifying Circuits Using Generalized Relative Timing

Sanjit A. Seshia   Randal E. Bryant
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
{Sanjit.Seshia, Randy.Bryant}@cs.cmu.edu

Kenneth S. Stevens
Intel Strategic CAD Labs
Hillsboro, OR 97124, USA
kstevens@ichips.intel.com

Abstract

We propose a novel technique for modeling and verifying timed circuits based on the notion of generalized relative timing. Generalized relative timing constraints can express not just a relative ordering between events, but also some forms of metric timing constraints. Circuits modeled using generalized relative timing constraints are formally encoded as timed automata. Novel fully symbolic verification algorithms for timed automata are then used to either verify a temporal logic property or to check conformance against an untimed specification. The combination of our new modeling technique with fully symbolic verification methods enables us to verify larger circuits than has been possible with other approaches. We present case studies to demonstrate our approach, including a self-timed circuit used in the integer unit of the Intel® Pentium® 4 processor.

1. Introduction

Timing assumptions are commonly used in the design of both asynchronous and synchronous circuits in order to improve performance. Examples include the GasP circuits [30], the Global STP circuit in the Intel® Pentium® 4 processor [12], and the RAPPID instruction decoder [29]. However, the use of timing assumptions comes at an added verification cost: The circuit behavior must be verified under these constraints, and furthermore, the constraints must themselves be verified pre- and post-layout.

A promising recent approach to this verification problem is to use a design methodology based on *relative timing* [28]. In the relative timing (RT) paradigm, timing assumptions are made *explicit*, by adding to an untimed design constraints on the relative ordering of signal transitions. In contrast, other methods use *implicit* timing assumptions, where the timing assumptions are either implicit in a design style (such as Burst-Mode techniques, e.g. [22]) or imposed at the gate-level in the circuit model (such as metric timed circuit design [19]). Using the RT paradigm, verification proceeds in two steps:

1. *Checking correctness under timing constraints:* RT constraints are identified and the correct operation of the circuit is verified under those constraints. Typically, one either checks that the implemented circuit $\mathcal{I}$ only exhibits behaviors of a specification $\mathcal{S}$, or that it satisfies a specific property φ formulated in a suitable temporal logic.
2. *Verifying that the circuit obeys timing constraints:* The identified RT constraints are themselves verified using standard simulation or static timing analysis techniques. The constraints can be verified pre-layout to ensure that they have sufficient margin based on expected design parameters. The constraints also must be validated post-layout with extracted data to ensure that place and route, sizing, and buffer insertion have not skewed the delays beyond acceptable values.

The RT approach of explicitly stating timing constraints has the advantage that it applies to many asynchronous design styles [28]. It supports a design philosophy of adding timing constraints incrementally and of giving the designer flexibility in using timing constraints. Also, unlike gate-level metric timing, it does not rely on conservatively set min-max bounds on gate delays.

However, current RT-based verification techniques (e.g., [24, 13]) fall short in three respects. First, not all timing constraints can be expressed as the relative ordering of signal transitions. Secondly, current verification tools are yet to scale up to relatively large circuits and achieve the success obtained by symbolic methods for untimed systems (e.g., [5]). Finally, previous work on relative timing-based verification [24, 13] does not satisfactorily address the problem of verifying that the circuit obeys the constraints.

In this paper, we address these shortcomings by making the following novel contributions:

- *A generalized notion of relative timing:* We introduce the concept of a *generalized relative timing* (GRT) constraint, one that specifies a relative ordering not just between events, but between the time intervals between pairs of events. This generalization adds the capability to model some metric timing information which is formally modeled using real-valued *clock* variables. The resulting circuit model, in general, is a *timed automaton*. However, since metric timing constraints are typically far fewer than non-metric GRT constraints, we employ relatively few clock variables.
- *Application of fully symbolic verification methods:* We use new fully symbolic verification techniques based

on Boolean encoding methods [26] to verify the resulting timed automaton. The term *fully symbolic* means that the verifier represents both timed and untimed parts of the state space in a unified, symbolic representation. Along with the modeling methodology described above, we can verify circuits that are significantly larger than those verifiable with other methods. As an example we have efficiently analyzed the Global STP circuit [12], finding an error in the published circuit, and then successfully verifying a fixed version.

Related work. Several techniques have been proposed in the past 15 years to model timing constraints in circuit design. A common approach is to specify upper and lower bounds on the delay between when a transition is enabled and when it fires. Formalisms such as timed transition systems [10], timed Petri nets [25] and timed event and event/level structures [19, 4, 17] are used for this purpose, and the constraints are referred to as *gate-level metric timing* constraints. This is an intuitive model, but since the timing information is provided at the gate-level, verification tools based on this model are restricted to relatively small circuits. Even with the use of partial order reduction methods (e.g., [4, 17]), the size of the untimed state space still presents a performance bottleneck. Furthermore, designers must be relatively conservative on how they set the bounds, since these can rely on post-layout information.

Another formalism for modeling timed systems is that of timed automata [1], which is more expressive than timed transition systems [2], in that it can model "more global" timing constraints. Maler and Pnueli [14] model asynchronous circuits using timed automata, but their model is also at the gate-level, requiring one clock variable per gate. Thus, it suffers from the same scaling problems as the afore-mentioned metric timing methods. Our work also uses timed automata as the modeling formalism, but in an entirely different way: We model timing constraints at a higher level of abstraction, and introduce clock variables only where necessary.

The observation that enables us to selectively use clock variables is that most timing constraints are on pairs of events that have a common start event, i.e., a "point-of-divergence". A similar observation was made by Negulescu and Peeters [20, 21], who present the notion of a *chain constraint*, which specifies that one sequence of transitions must occur before another with both sequences sharing a common prefix. A "point-of-divergence" constraint is more restrictive than a chain constraint in a logical sense (it specifies a relative ordering for *all* intermediate sequences of transitions between the start and end events), but for the same reason, it is more compact to specify. Moreover, we can model more general kinds of constraints, as we describe in Section 2.

There has been prior work on RT-based verification, with a focus on automatically generating constraints. Peña *et al.* [24] present an approach based on the notion of *lazy transition systems*. Their approach automatically and iteratively generates RT constraints to rule out spurious counterexamples; however, the process of adding RT constraints relies on knowing min-max bounds on gate delays. Kim *et al.* [13] present a verification methodology based on a different technique of automatically generating RT constraints, but do not address the problem of verifying that the circuit obeys the constraints. While we do not automatically generate timing constraints, our work targets a more general class of timing constraints, and provides ways of verifying that the constraints hold for the circuit.

Clarisó and Cortadella [6] present a gate-level modeling approach that represents gate delays by symbols, rather than by constant bounds. Thus, this model is more expressive than metric timing. However, the verification problem is even harder than for timed transition systems, and the approach is restricted to small circuits.

Another contribution of our paper is in the application of novel fully symbolic verification techniques to timed circuits. These techniques are based on our earlier paper on using Boolean methods in quantifier elimination in quantified separation logic (QSL) [26]; we refer the reader to that paper for a detailed comparison of model checking methods for timed automata. In the context of asynchronous circuits, there has been significant work on model checking algorithms; see, for example, the work by Myers, Yoneda, *et al.*(e.g., [19, 4, 34, 17]). The main difference with our work is that these methods are symbolic in the real-valued part, but explicit-state in the Boolean part; hence, in spite of incorporating partial-order reduction, large circuits are often outside their capacity. We also extend the ideas of our previous paper to perform fully symbolic *simulation checking*. Simulation checking has been explored earlier in the context of timed systems, for example, by Tasiran *et al.* [32].

There has been significant work on methods that use compositional reasoning or abstraction to achieve better scalability (e.g., [35]). Our focus, in this paper, is on demonstrating scalability without using compositional reasoning or abstraction; however, nothing precludes using the techniques presented herein along with such methods.

Paper outline. We introduce the idea of generalized relative timing in Section 2. In Section 3, we describe how timed circuits are formalized as timed automata, and in Section 4, we outline fully symbolic verification algorithms for timed automata. Case studies are presented in Section 5.

2. Modeling Timed Circuits

A timed circuit is a triple $(\mathcal{V}, \mathcal{R}, \mathcal{T})$, where $\mathcal{V}$ is a set $\{v_1, v_2, \ldots, v_n\}$ of circuit *signals*, $\mathcal{R}$ is a set $\{r_1, r_2, \ldots, r_m\}$ of *rules*, and $\mathcal{T}$ is a set $\{\tau_1, \tau_2, \ldots, \tau_p\}$ of *timing constraints*. The set of initial values of signals in $\mathcal{V}$ is specified as a Boolean formula $I_{\mathcal{V}}$.

The circuit signals, which are the *state variables* of the system, are comprised of inputs, outputs, and intermediate signals. A *transition* (also referred to as *event*) is a change in logic level of a signal. Transition $v_i\!\uparrow$ corresponds to the transition of v_i from 0 to 1, and $v_i\!\downarrow$ to the transition from 1 to 0. We will use the symbol e_i to refer to either transition for signal v_i.

The untimed circuit behavior is defined by the set of rules $\mathcal{R}$, which comprises $m = 2n$ rules, one for each signal transition.[1] The 2 rules for the ith signal v_i are written as

$$\mathcal{E}_{v_i\uparrow} \mapsto v_i\!\uparrow \quad \text{and} \quad \mathcal{E}_{v_i\downarrow} \mapsto v_i\!\downarrow$$

where $\mathcal{E}_{e_i}$ is a Boolean formula over $\mathcal{V}$ indicating the enabling condition for transition e_i to fire.

Although we have only introduced two events per signal (corresponding to up and down transitions), it would be straightforward to add finitely-many instances of each event. That is, for a given event e_i, we can keep track of not only each instance of e_i, but also every second, third, ..., k^{th} instances of e_i for a constant k, with the use of additional state bits to keep track of a "count." However, we have rarely needed to track more than one instance of each event.

We will assume an *inertial* gate model (but without bounds on gate delays). Thus, it is allowed for a transition that was enabled to become disabled without having fired, as long as the circuit satisfies its specification. In the absence of an explicit timing constraint involving transition e_i, the time taken for e_i to fire after being enabled can be any value in $[0, \infty)$; i.e., rules, by themselves, are purely untimed.

2.1. Generalized Relative Timing

The novel aspect of how we model circuits is in the formulation of *generalized relative timing* constraints, which combine relative timing with a capability to incorporate some metric timing information.

Let $\Delta(e_i, e_j)$ denote the time interval between an occurrence of e_j and *the* occurrence of e_i immediately preceding it.

The following definition formalizes the notion of generalized relative timing (GRT):

Definition 1 *Let e_i, e'_i, e_j, e_k be four transitions such that $e_j \neq e_k$. Then, a generalized relative timing constraint on e_i, e'_i, e_j, e_k is of the form:*
For all occurrences of transitions e_j and e_k,

$$\Delta(e_i, e_j) < \Delta(e'_i, e_k) + d$$

where d is a rational constant.

It is sometimes useful to use a non-strict inequality ($\leq$) instead of the strict inequality used above, or to drop one of the $\Delta(\cdot, \cdot)$ terms in the inequality so as to impose an upper or lower bound on the time interval between events.

Point-of-divergence constraint. An extremely common sub-class of GRT constraints are those such that $e_i = e'_i$, $d = 0$, and the *same* occurrence of e_i immediately precedes all occurrences of both e_j and e_k. In this case, the timing constraint specifies that measuring time from the point e_i occurs, e_j must always occur before e_k. We will refer to this special case as a *point-of-divergence* (POD) constraint. (The name comes from the divergence in two paths starting from transition e_i.) We write a POD constraint as $e_i \rightarrow e_j \prec e_k$. Typically, e_j and e_k causally depend on e_i. However, note that this need not be the case! By the definition of $\Delta(e_i, e_j)$, the point-of-divergence in the constraint is simply the occurrence of e_i that is *closest in time* to e_j and e_k, which need not have caused either of them.

Note also that the concept of a POD constraints is essentially the same as that of the original RT constraint, since, in order to implement a relative ordering between events, one would have to trace them back to a point-of-divergence; hence the name *generalized* relative timing.

Metric timing constraints. The presence of d in the definition allows us to express a limited form of metric timing constraints. In particular, we can express constraints of the form $d_1 \leq \Delta(e_i, e_j) \leq d_2$. Note, however, that we cannot directly specify the min-max timing assumptions used in timed transition systems [10] and related formalisms, since that would require constraining the delay between when a transition is *enabled* and when it fires.[2]

Compound timing constraints. In some cases, such as the Global STP circuit that is our primary case study, we have observed the need for *compound timing constraints* formed as an XOR of two (simple) timing constraints. Such a constraint is written as τ_i XOR τ_j. We have needed such compound constraints to reason about relative ordering between instances of events from different cycles of circuit operation. Further discussion of such constraints is deferred to the case study in Section 5.1.

In all our case studies to date, we have found the class of generalized relative timing constraints to be sufficient. In fact, most constraints tend to be simple (i.e., not compound) POD constraints. Metric timing constraints are used only when there is explicit use of delay values in the design.

2.1.1. Examples. We present two examples to illustrate our methodology for modeling timing constraints.

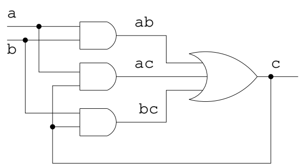


Figure 1. Implementation of a C-element

First, consider the implementation of a C-element using three AND gates and an OR gate, as shown in Figure 1. a and b denote the input signals, and c is the output. It is easy to see that in order to work correctly, it is sufficient for the

1 Notice that this is similar to the language of production rules [15].

2 However, note that the formalism that we use, viz. timed automata, is general enough to express such constraints [2].

circuit in Figure 1 to respect the following two fundamental mode constraints, formulated here as POD constraints: $\texttt{c}\uparrow \rightarrow \texttt{ac}\uparrow \prec \texttt{b}\downarrow$ and $\texttt{c}\uparrow \rightarrow \texttt{bc}\uparrow \prec \texttt{a}\downarrow$.

While POD constraints suffice for the preceding example, in general, we might need a more expressive timing constraint. Figure 2 depicts a simple buffer stage element generated from the CASH compiler that compiles ANSI-C programs into asynchronous circuits [33]. For correct operation, this circuit relies on two timing assumptions: data transfers between stages use a bundled data protocol, and a stage incorporates a matched delay element.

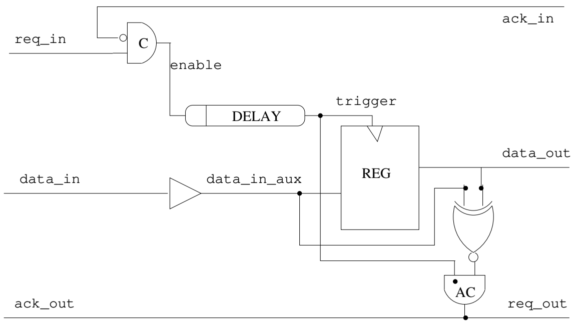


Figure 2. Buffer stage from CASH compiler

The matched delay can be formalized with the following two timing assumptions τ_1^{CASH} and τ_2^{CASH}:

$$\Delta(\texttt{data\_in}\uparrow, \texttt{data\_in\_aux}\uparrow) < \Delta(\texttt{enable}\uparrow, \texttt{trigger}\uparrow) \quad (\tau_1^{\text{CASH}})$$
$$\Delta(\texttt{data\_in}\downarrow, \texttt{data\_in\_aux}\downarrow) < \Delta(\texttt{enable}\uparrow, \texttt{trigger}\uparrow) \quad (\tau_2^{\text{CASH}})$$

To ensure that the stage respects the bundled data protocol, we additionally need to impose two POD constraints:
$\texttt{enable}\uparrow \rightarrow \texttt{data\_out}\uparrow \prec \texttt{req\_out}\uparrow$, and
$\texttt{enable}\uparrow \rightarrow \texttt{data\_out}\downarrow \prec \texttt{req\_out}\uparrow$.

Note that the matched delay assumptions τ_1^{CASH} and τ_2^{CASH} of a stage can be reformulated as POD constraints by tracing back to the `enable` signal of the previous stage. However, this breaks modularity, since the timing constraints involving signals of a module reference internal signals of another module. In general, we have found that while it is often possible to reformulate metric timing constraints as POD constraints, it is at the cost of modularity.

2.2. Verifying Timing Constraints

The verification methods presented in this paper prove that the timed circuit design is correct given the set of timing constraints $\mathcal{T}$. However, it does not prove that the constraints actually hold given the true delays in the design. Timing constraints can be constructed that do not hold in a design, as will be shown later in Section 5.1. Therefore, these must be proven outside the symbolic verification environment. We briefly describe this process to show a consistent design flow exists based on this tool.

Given a POD constraint $e_i \rightarrow e_j \prec e_k$ we must prove that any sequence of events from e_i to e_j always occurs before the events from e_i to e_k. This is accomplished by tracing and timing the maximum and minimum delay paths from the POD to the end points, and comparing the results. We compute the maximum delay of the left path ($e_i \rightsquigarrow e_j$) and the minimum delay for the right path ($e_i \rightsquigarrow e_k$). This ensures that no combination of delays will cause e_k to occur before e_j. The same conditions exist for the general form of constraints $\Delta(e_i, e_j) \leq \Delta(e'_i, e_k) + d$ where the tracing may occur to different starting points, and a constant delay is added when the path delays are compared.

We illustrate static timing validation using the circuit in Figure 1. There are two POD constraints, the first of which is $\texttt{c}\uparrow \rightarrow \texttt{ac}\uparrow \prec \texttt{b}\downarrow$. Validating this constraint requires evaluation of the max-delay path from $\texttt{c}\uparrow$ to $\texttt{ac}\uparrow$. This is simply the maximum rise delay through the gate corresponding to `ac` since signal `a` is already asserted. Similarly, the minimum delay path from $\texttt{c}\uparrow$ to $\texttt{b}\downarrow$, which depends on how the gate is connected to its environment, is calculated and compared with the maximum rising delay of the gate `ac` to validate this constraint. The second constraint $\texttt{c}\uparrow \rightarrow \texttt{bc}\uparrow \prec \texttt{a}\downarrow$ is similarly validated.

The capability of automatically tracing and timing maximum and minimum delay paths, and comparing the results is supported in most commercial timing tools such as PrimeTime [31]. Therefore, it is possible to automatically validate all the constraints in $\mathcal{T}$. However, some complications arise in automatically tracing signals through sequential elements (such as the C-element of Figure 1), since static tools may not correctly cut feedbacks that exist solely to retain state. Fully automatic translation and validation of GRT constraints using static timing tools is left to future work.

The timing constraints used in this paper were identified manually, many with the assistance of a relative-timing enhanced verification engine [27]. Automatic generation of GRT constraints is left to future work.

3. From Circuits to Timed Automata

We now describe how we translate a timed circuit $(\mathcal{V}, \mathcal{R}, \mathcal{T})$ into a timed automaton. The technical details in Section 4 will be based on the timed automaton model.

3.1. Preliminaries

A *timed automaton* [1] is a generalization of a finite automaton with a set of real-valued clock variables. A state of a timed automaton is a concatenation of a vector of Boolean values, corresponding to finite-state variables, and a vector of real values, corresponding to real-valued clock variables.

A set of states of a timed automaton can be represented symbolically as a formula in a quantifier-free fragment of first-order logic called *separation logic* (SL), also known as *difference logic*. This formula is the *characteristic function* of the set of states. A formula ϕ in separation logic is a Boolean combination of Boolean variables and *separation predicates* (also known as *difference-bound constraints*) involving real-valued variables, as given by the following grammar:

$$\phi ::= \mathbf{true} \mid \mathbf{false} \mid v \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid x_1 \geq x_2 + d \mid x_1 > x_2 + d$$

We use a special variable x_0 to denote the constant 0; this allows us to express bounds of the form $x \geq d$. We denote Boolean variables by $v, v_1, v_2, \ldots$, real variables by $x, x_1, x_2, \ldots$, and SL formulas by $\phi, \phi_1, \phi_2, \ldots$. Note that Boolean operators like $\vee$ and $\Rightarrow$ can be constructed from $\neg$ and $\wedge$. Similarly, the relations $>$ and $\geq$ suffice to represent equalities and other inequalities. Deciding the satisfiability of a SL formula is NP-complete [11].

For ease of presentation, we will use a definition of timed automata that is intuitive in our context. This definition is equivalent to the guarded-command real-time program notation used by Henzinger *et al.* [11].[3]

Definition 2 *A* timed automaton *is a quintuple* $\mathcal{A} = (\mathcal{V}, \mathcal{X}, \mathcal{G}, \mathcal{I}_\mathcal{A}, \phi_0)$ *where:*

- $\mathcal{V}$ *is a set of Boolean state variables;*
- $\mathcal{X}$ *is a set of clock variables taking values in* $\mathbb{R}^{\geq 0}$*;*
- $\mathcal{G}$ *is a set of guarded commands of the form* $\psi \Longrightarrow A$*, where* ψ *is a guard condition (a SL formula over* $\mathcal{V}$ *and* $\mathcal{X}$*), and* A *is a set of assignments, i.e., a set of transitions of Boolean state variables and resets of clock variables to* 0*;*
- $\mathcal{I}_\mathcal{A}$ *is a SL formula over* $\mathcal{V}$ *and* $\mathcal{X}$ *expressing an invariant condition on all states of* $\mathcal{A}$*; and*
- ϕ_0 *is a SL formula over* $\mathcal{V}$ *and* $\mathcal{X}$ *characterizing the set of initial states.*

The set of guarded commands $\mathcal{G}$ represents the transition relation of the automaton. The semantics of a guarded command $\psi \Longrightarrow A$ is as follows. If ψ is true in a state σ, then the guarded command is enabled in that state. Any guarded command that is enabled in a state σ can execute in σ. The time a system can spend in a state can be any non-negative amount allowed by the invariant $\mathcal{I}_\mathcal{A}$.

Note that timed automata differ from models commonly used in the asynchronous circuits literature, such as timed transition systems [10] or timed Petri nets [25], in that the time interval between an arbitrary pair of events can be directly expressed. This expressiveness is key for modeling the class of timing constraints described in Section 2.1.

3.2. Translation

The translation of a timed circuit $(\mathcal{V}, \mathcal{R}, \mathcal{T})$ to a timed automaton $\mathcal{A}$ is performed in three steps.

Initialization. The set of Boolean state variables of $\mathcal{A}$ is initialized to be the set of signals $\mathcal{V}$, while the set of clock variables $\mathcal{X}$ is initialized to $\emptyset$.

3 The only difference with the standard definition given by Alur and Dill [1] is that our definition has no notion of *accepting states*.

Each rule of the timed circuit gets translated to a corresponding guarded command of the timed automaton; thus, there is exactly one guarded command for each transition e. For transition e with corresponding rule $\mathcal{E}_e \mapsto e$, we initialize its guarded command to be $\mathcal{E}_e \Longrightarrow e$.

The invariant $\mathcal{I}_\mathcal{A}$ is initialized to be $\mathbf{true}$, and ϕ_0 is set to be $I_\mathcal{V}$ (the set of initial signal values).

Adding auxiliary variables. For each timing constraint, we add an additional Boolean or clock variable to store timing information.

Let τ_i be the i^{th} timing constraint.

If τ_i is a POD constraint, we only introduce a fresh Boolean state variable b_i into $\mathcal{V}$.

Suppose τ_i is not a POD constraint, and is of the form $\Delta(e_i, e_j) \leq \Delta(e'_i, e_k) + d$. Then we not only introduce a fresh Boolean state variable b_i into $\mathcal{V}$, but also add two clock variables x_{e_i,e_j} and $x_{e'_i,e_k}$ to $\mathcal{X}$.

Encoding timing constraints. We encode timing constraints in sequence, running through the set $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_p\}$. As we encounter timing constraints containing a transition e, we update the guarded command corresponding to it.

Suppose we are encoding timing constraint τ_t, which mentions transition e. Let the current form of the guarded command γ for e be $\psi \Longrightarrow A$.

How we modify γ depends on whether the timing constraint is a POD constraint or not, and on the role of e in the constraint, as elaborated below:

- *POD constraint:* Suppose the constraint is of the form $e_i \rightarrow e_j \prec e_k$. There are three cases, with γ being modified differently in each case:

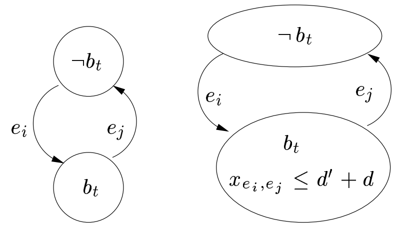
  Case $e = e_i$: $\gamma := \psi \Longrightarrow A'$,
  where $A' = A \cup \{b_t\!\uparrow\}$.

  Case $e = e_j$: $\gamma := \psi \Longrightarrow A'$,
  where $A' = A \cup \{b_t\!\downarrow\}$.

  Case $e = e_k$: $\gamma := \psi' \Longrightarrow A$,
  where $\psi' = \psi \wedge \neg b_t$.

  The intuition is that we take the product of the timed automaton (constructed so far) with a two-state monitor automaton as shown in Figure 3(a) to enforce the ordering specified by the POD constraint. The variable b_t encodes the states of this automaton. Transition e_k can only occur in the state labeled $\neg b_t$; i.e., the state in which b_t is $\mathbf{false}$.
- *Non-POD constraint:* Suppose the constraint is of the form $\Delta(e_i, e_j) \leq \Delta(e'_i, e_k) + d$. To encode this constraint, we introduce a non-negative constant d' such that $\Delta(e_i, e_j) \leq d' + d$ and $d' \leq \Delta(e'_i, e_k)$. The value of d' is usually known at design time since a non-POD constraint arises only in design styles that make use of some form of metric timing, such the matched delay assumption used in the circuit in Figure 2.

  We have four cases to consider:



(a) POD (b) Non-POD

Figure 3. Monitor automata for timing

Case $e = e_i$: $\gamma := \psi \Longrightarrow A'$,
where $A' = A \cup \{b_t\uparrow, x_{e_i,e_j} := 0\}$.
Case $e = e_i'$: $\gamma := \psi \Longrightarrow A'$,
where $A' = A \cup \{x_{e_i',e_k} := 0\}$.
Case $e = e_j$: $\gamma := \psi \Longrightarrow A'$,
where $A' = A \cup \{b_t\downarrow\}$.
Case $e = e_k$: $\gamma := \psi' \Longrightarrow A$,
where $\psi' = \psi \wedge x_{e_i',e_k} \geq d'$.

In addition, we update the invariant $\mathcal{I}_\mathcal{A}$ of the timed automaton by conjoining the current invariant with the SL formula $b_t \Rightarrow x_{e_i,e_j} \leq d + d'$.

The intuition behind this translation is as follows. First, notice that the Boolean variable b_t encodes, as before, the state of a monitor automaton, depicted in Figure 3(b). However, in this case, when b_t is $\mathbf{true}$, x_{e_i,e_j} cannot progress beyond $d + d'$, as enforced by the invariant $\mathcal{I}_\mathcal{A}$. Since the clock variable x_{e_i,e_j} is reset when e_i fires, this forces e_j to occur within $d + d'$ time units of e_i. Secondly, clock variable x_{e_i',e_k} is reset when e_i' fires, and the augmented guard for e_k ensures that e_k can only fire d' time units after e_i'. The above two mechanisms, in conjunction, ensure that the timing constraint τ_t is enforced.

The extension of the translation to handle compound timing constraints is straightforward; a XOR of two constraints can be encoded by making a non-deterministic choice to either monitor one constraint or the other. We omit the details.

3.3. Example

Consider the circuit in Figure 2. The rule corresponding to the transition $\texttt{trigger}\uparrow$ is

$$\neg\texttt{trigger} \wedge \texttt{enable} \mapsto \texttt{trigger}\uparrow$$

Timing constraints τ_1^{CASH} and τ_2^{CASH} both mention the transition $\texttt{trigger}\uparrow$.

Following the translation scheme described in this section, we introduce 3 clock variables $x_{\texttt{enable}\uparrow,\texttt{trigger}\uparrow}$, $x_{\texttt{data\_in}\uparrow,\texttt{data\_in\_aux}\uparrow}$, and $x_{\texttt{data\_in}\downarrow,\texttt{data\_in\_aux}\downarrow}$. The final guarded command for $\texttt{trigger}\uparrow$ is

$$\neg\texttt{trigger}\wedge\texttt{enable}\wedge(x_{\texttt{enable}\uparrow,\texttt{trigger}\uparrow} \geq d') \Longrightarrow \texttt{trigger}\uparrow$$

where d' is the delay corresponding to the delay element in the figure.

The invariant $\mathcal{I}_\mathcal{A}$ is the Boolean formula

$$(b_1 \Rightarrow x_{\texttt{data\_in}\uparrow,\texttt{data\_in\_aux}\uparrow} < d')\wedge(b_2 \Rightarrow x_{\texttt{data\_in}\downarrow,\texttt{data\_in\_aux}\downarrow} < d')$$

b_1 and b_2 are set by $\texttt{data\_in}\uparrow$ and $\texttt{data\_in}\downarrow$ respectively, and are reset by $\texttt{data\_in\_aux}\uparrow$ and $\texttt{data\_in\_aux}\downarrow$ respectively. Thus, our encoding simply formalizes the constraint that the delay through the buffer is less than that of the delay element.

4. Fully Symbolic Verification Techniques

We now outline the algorithms for fully symbolic verification of timed automata. Since the algorithms in themselves are not the main contribution of this paper, we omit background material and details of the algorithms; these can be found in [7, 26].

4.1. Quantified Separation Logic

There are two key operations in fully symbolic verification of timed automata, both of which are expressible in *quantified separation logic* (QSL), the extension of separation logic with quantifiers. The first operation is to decide the satisfiability of SL formulas. The second (and harder) operation is to eliminate quantifiers from a QSL formula.

Formally, a QSL formula ω is generated by the following grammar: $\omega ::= \phi \mid \neg\omega \mid \omega_1 \wedge \omega_2 \mid \exists x.\omega \mid \exists v.\omega$

In previous work [26], we show how to perform both operations using Boolean methods. In particular, we transform the problem of eliminating quantifiers on real-valued clock variables to one of eliminating quantifiers on Boolean variables. Given a QSL formula ω with quantifiers over real-valued variables, we transform it to an *equivalent* QSL formula ω_{bool} that has quantifiers only over Boolean variables. These quantifiers can then be eliminated using standard Boolean techniques (e.g., [5, 16]) that are based on Binary Decision Diagrams (BDDs) or Boolean satisfiability (SAT) solvers. Moreover, for a special class of QSL formulas occurring in model checking of timed automata, the transformation can be greatly optimized. The resulting quantifier elimination technique can yield improvements over other methods [26].

For brevity, we omit the details of how we eliminate quantifiers over clock variables. We illustrate the main ideas using the following example.

Let $\omega_a = \exists x_a.\phi$ where $\phi = x_a \leq x_0 \wedge x_1 \geq x_a \wedge x_2 \leq x_a \wedge \beta$, and β is a complicated Boolean formula representing many circuit states (evaluations of circuit signals). Our goal is to eliminate variable x_a to obtain a SL formula that is logically equivalent to ω_a.

The first step is to introduce Boolean variables for each separation predicate in ω_a that involves x_a. Let $b_{i,j}^{\geq,c}$ represent the predicate $x_i \geq x_j + c$. Replacing predicates with their corresponding Boolean variables, we obtain the Boolean encoding $\phi^a_{bool} = b^{\geq,0}_{0,a} \wedge b^{\geq,0}_{1,a} \wedge b^{\geq,0}_{a,2} \wedge \beta$.

The second step is to construct a SL formula that represents the arithmetic information lost in the above Boolean encoding. This formula, denoted by ϕ^a_{cons}, is the conjunction of the following two SL formulas:

1. $b^{\geq,0}_{0,a} \wedge b^{\geq,0}_{a,2} \implies x_0 \geq x_2$
2. $b^{\geq,0}_{1,a} \wedge b^{\geq,0}_{a,2} \implies x_1 \geq x_2$

Finally, we construct the SL formula $\omega_{bool} = \exists b^{\geq,0}_{0,a}, b^{\geq,0}_{1,a}, b^{\geq,0}_{a,2}.[\phi^a_{cons} \wedge \phi^a_{bool}]$. The only quantified variables in this formula are Boolean. The remaining separation predicates can now be replaced with dummy Boolean variables (retaining a mapping of predicates to dummy variables), and the resulting quantified Boolean formula (QBF) can be simplified using Boolean techniques (e.g., BDDs) to a Boolean formula. Replacing the dummy variables, we obtain the final result: $x_0 \geq x_2 \wedge x_1 \geq x_2 \wedge \beta$.

The advantage of this approach is apparent when β corresponds to a huge number of circuit states. An explicit-state technique would need to enumerate every such circuit state and perform the quantifier elimination for each.

4.2. Fully Symbolic Model Checking

The afore-mentioned Boolean methods for quantifier elimination can be used with a model checking algorithm given by Henzinger *et al.* [11] for checking if a timed automaton satisfies a property specified in the timed μ-calculus (in which any property in Timed Computation Tree Logic, TCTL, can be expressed). Here we only describe, in brief, the algorithm for a simple but very useful case, that of computing the set of reachable states of the timed automaton (checking safety properties). The general algorithm can be found in [26].

Consider a timed automaton $\mathcal{A} = (\mathcal{V}, \mathcal{X}, \mathcal{G}, \mathcal{I}_\mathcal{A}, \phi_0)$. The following algorithm computes a SL formula ϕ_{reach} representing the set of reachable states of $\mathcal{A}$.

1. $\phi_{new} := \phi_0$.
2. **Do**
   (a) $\phi_{old} := \phi_{new}$
   (b) $\phi' := \textbf{post}_{\textbf{time}}(\phi_{old})$ {Let time elapse}
   (c) $\phi'' := \textbf{post}_\mathcal{G}(\phi')$ {Fire a transition}
   (d) $\phi_{new} := \phi_{old} \vee \phi''$ {Union of sets}
   While $(\phi_{old} \neq \phi_{new})$ {Check termination}
3. $\phi_{\text{reach}} := \phi_{new}$.

The symbolic "next-state" operators $\textbf{post}_{\textbf{time}}$ and $\textbf{post}_\mathcal{G}$ are defined as follows:

$$\textbf{post}_{\textbf{time}}(\phi) \doteq \exists\delta\{\delta \geq 0 \wedge \phi - \delta \wedge \forall\epsilon[0 \leq \epsilon \leq \delta \Rightarrow \mathcal{I}_\mathcal{A} - \epsilon]\} \tag{1}$$

where $\phi - \delta$ denotes the formula obtained by subtracting δ from all clock variables occurring in ϕ, computed as $\phi[x_i - \delta/x_i, 1 \leq i \leq n]$, where $x_1, x_2, \ldots, x_n$ are the clock variables in ϕ_i (and similarly for $\mathcal{I}_\mathcal{A} - \epsilon$).

Intuitively, δ is the time elapsed since the last transition fired. The inner quantified formula in (1) above ensures that while allowing time to elapse, the values of clock variables must always respect the invariant $\mathcal{I}_\mathcal{A}$. The formula obtained after eliminating quantifiers from $\textbf{post}_{\textbf{time}}(\phi)$ represents all states reachable from ϕ by allowing some duration of time to elapse within the constraints imposed by $\mathcal{I}_\mathcal{A}$.

The operation $\textbf{post}_\mathcal{G}$, when applied to a set of states ϕ, returns the set of states reached from ϕ by firing some transition. Formally,

$$\textbf{post}_\mathcal{G}(\phi) \doteq \bigvee_{(\psi \Longrightarrow A) \in \mathcal{G}} (\phi \wedge \psi)[A] \tag{2}$$

where $\phi[A]$ denotes the set of states reached from ϕ after performing the signal transitions and clock resets in A. For instance, if $A = v_i\uparrow$, then $\phi[A] = (\exists v_i.\phi) \wedge v_i$. We omit the details for brevity.

Quantifier elimination is required in computing $\textbf{post}_{\textbf{time}}$ and $\textbf{post}_\mathcal{G}$. Checking satisfiability of SL formulas is required to check the termination condition. It is also needed for checking if a error state is reachable: If ϕ_{bad} characterizes the set of error states, then an error state is reachable iff $\phi_{\text{reach}} \wedge \phi_{bad}$ is satisfiable.

4.3. Fully Symbolic Simulation Checking

Suppose we want to check if a circuit implementation $\mathcal{I}$, modeled as a timed automaton, conforms to a specification $\mathcal{S}$, also given as a timed automaton. Since language containment is undecidable for timed automata in general [1], we use the formal notion of *simulation* [18] as our notion of conformance. Furthermore, for ease of presentation, we will restrict our specifications to be untimed finite automata. For all the case studies presented in this paper, considering untimed finite-state specifications has sufficed.

Let $\Sigma_\mathcal{I}$ and $\Sigma_\mathcal{S}$ be the sets of states of $\mathcal{I}$ and $\mathcal{S}$ respectively. Let $\mathcal{F}(\sigma_\mathcal{I}, \sigma_\mathcal{S})$ be $\mathtt{true}$ iff states $\sigma_\mathcal{I} \in \Sigma_\mathcal{I}$ and $\sigma_\mathcal{S} \in \Sigma_\mathcal{S}$ agree on the values of variables common to both $\mathcal{I}$ and $\mathcal{S}$. A binary relation $R \subseteq \Sigma_\mathcal{I} \times \Sigma_\mathcal{S}$ is a *simulation relation* iff for all $\sigma_\mathcal{I}$ and $\sigma_\mathcal{S}$ such that $R(\sigma_\mathcal{I}, \sigma_\mathcal{S})$ holds, both the following conditions hold:

1. $\mathcal{F}(\sigma_\mathcal{I}, \sigma_\mathcal{S})$.
2. For every successor state $\sigma'_\mathcal{I}$ of $\sigma_\mathcal{I}$, either there exists some successor $\sigma'_\mathcal{S}$ of $\sigma_\mathcal{S}$ such that $R(\sigma'_\mathcal{I}, \sigma'_\mathcal{S})$ or $R(\sigma'_\mathcal{I}, \sigma_\mathcal{S})$.

This is a standard definition of simulation that allows the specification to "stutter".

If for every initial state $\sigma_{\mathcal{I},0}$ of $\mathcal{I}$ there is a corresponding initial state $\sigma_{\mathcal{S},0}$ of $\mathcal{S}$ such that $R(\sigma_{\mathcal{I},0}, \sigma_{\mathcal{S},0})$, then we say that $\mathcal{I}$ *is simulated by* $\mathcal{S}$ and write $\mathcal{I} \preceq \mathcal{S}$.

Fully symbolic simulation checking is done in two steps:

1. *Compute simulation relation:* We compute a symbolic representation of the simulation relation, starting with

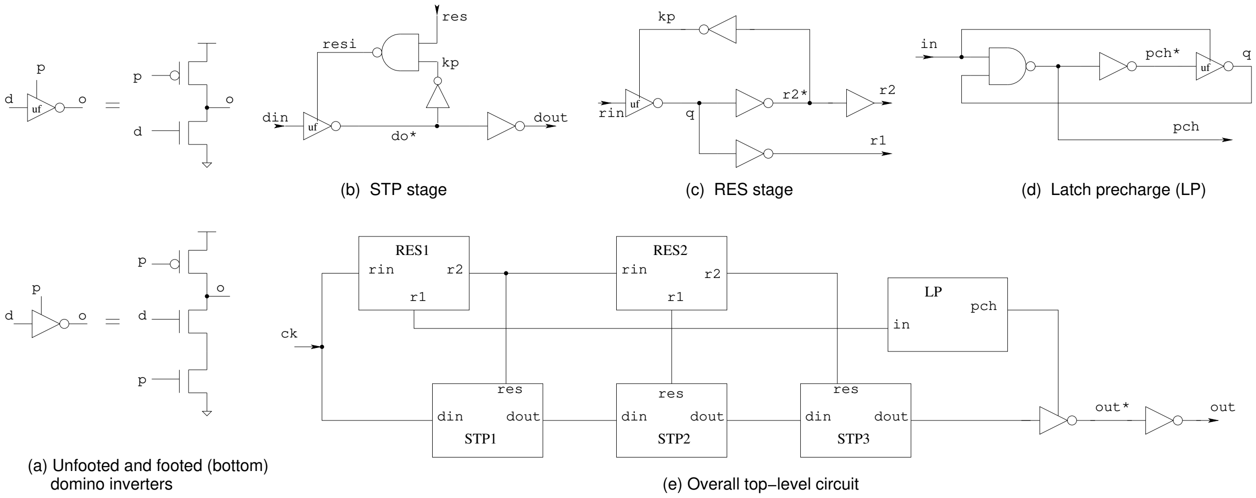


Figure 4. Global STP circuit

an over-approximation and refining it iteratively. The symbolic representation is a SL formula ϕ_R over the Boolean variables $\mathcal{V}_\mathcal{I}$ and clock variables $\mathcal{X}_\mathcal{I}$ of $\mathcal{I}$ and the Boolean variables $\mathcal{V}_\mathcal{S}$ of the specification $\mathcal{S}$. We will abuse notation a little here to use $(\mathcal{V}_\mathcal{I}, \mathcal{X}_\mathcal{I})$ and $\mathcal{V}_\mathcal{S}$ interchangeably with $\sigma_\mathcal{I}$ and $\sigma_\mathcal{S}$ respectively. Thus, we will write the symbolic representation as $\phi_R(\sigma_\mathcal{I}, \sigma_\mathcal{S})$.

We set our initial approximation ϕ_R^0 to be the formula $\mathcal{F}(\sigma_\mathcal{I}, \sigma_\mathcal{S})$ that equates the common state variables of $\mathcal{I}$ and $\mathcal{S}$ for the reachable states.

Then, we compute the i^{th} approximation ϕ_R^i, $i \geq 1$, as follows:

$$\phi_R^i := \phi_R^{i-1}(\sigma_\mathcal{I}, \sigma_\mathcal{S}) \ \wedge \ [\ \forall \sigma'_\mathcal{I} \,.\, T_\mathcal{I}(\sigma_\mathcal{I}, \sigma'_\mathcal{I}) \Rightarrow \\ (\ \phi_R^{i-1}(\sigma'_\mathcal{I}, \sigma_\mathcal{S}) \vee \exists \sigma'_\mathcal{S}.T_\mathcal{S}(\sigma_\mathcal{S}, \sigma'_\mathcal{S}) \wedge \phi_R^{i-1}(\sigma'_\mathcal{I}, \sigma'_\mathcal{S})\ )\ ]$$

Here $T_\mathcal{I}$ and $T_\mathcal{S}$ denote the transition relations of $\mathcal{I}$ and $\mathcal{S}$ respectively; the transition relation of the timed automaton $\mathcal{I}$ can be easily derived from the definitions of $\mathbf{post}_{\mathbf{time}}$ and $\mathbf{post}_\mathcal{G}$ given in Section 4.2.

This fixpoint computation is guaranteed to terminate, say in N steps, with the N^{th} approximation being $\phi_R(\sigma_\mathcal{I}, \sigma_\mathcal{S})$.

2. *Check initial states:* Let $\phi_{\mathcal{I},0}$ and $\phi_{\mathcal{S},0}$ denote the set of initial states of $\mathcal{I}$ and $\mathcal{S}$ respectively. By definition, we can conclude that $\mathcal{I} \preceq \mathcal{S}$ iff the following formula is valid:

$$\phi_{\mathcal{I},0}(\sigma_\mathcal{I}) \Rightarrow [\exists \sigma_\mathcal{S}.\phi_{\mathcal{S},0}(\sigma_\mathcal{S}) \wedge \ \phi_R(\sigma_\mathcal{I}, \sigma_\mathcal{S})]$$

Both steps have the same key operations as for model checking, viz., quantifier elimination in QSL and satisfiability (validity) checking of SL. We perform these using the Boolean methods as described in Section 4.1.

5. Case Studies

Our main case study is the Global STP circuit, a self-timed circuit used in the integer unit in the Intel® Pentium® 4 processor [12]. Other case studies include the GasP FIFO control circuit [30], STAPL circuits [23], and the STARI circuit [9].

Experiments reported on here were run on a Linux workstation with a 2 GHz Pentium® 4 processor and 1 GB of memory. Our fully symbolic verification tool, called TMV, is BDD-based. TMV is written in O'Caml except for the BDD engine, for which we use the CUDD package [8].

5.1. Global STP

Figure 4 is a hierarchical depiction of the Globally Reset Domino with Self-Terminating Precharge (Global STP) circuit. The circuit we discuss here is a gate-level depiction of the simplest form of the published circuit [12], with N-logic blocks replaced by wires, and static blocks replaced by inverters; our verification methods apply to the more general circuits as well. The top-level circuit is shown in Figure 4(e), with the input `ck` being a 4-GHz clock and the output being a delayed version of the same clock. In the beginning of the clock cycle, the last footed domino gate is being reset, while the first three STP stages go through an evaluation. After the precharge of the last domino gate has been turned off, the evaluate signal propagates to the output, where it is held until the next cycle. Interestingly, note that the three STP stages are reset in the same cycle in which they evaluate. The specification of this circuit, given purely in terms of its input signal `ck` and output signal `out`, is depicted as a state graph in Figure 5.

This circuit relies on a number of timing constraints to ensure correct operation. We were able to formulate all these timing constraints either as POD constraints or as a XOR of 2 POD constraints. We discuss some of the more interesting timing constraints here.

Consider the i^{th} STP stage, for all $i \in \{1, 2, 3\}$ (refer to Figure 4(b)). Short circuit current in the domino inverter must

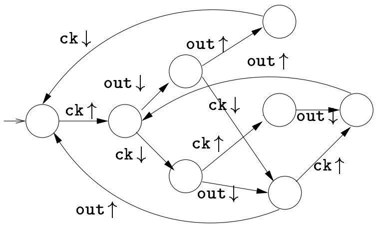


Figure 5. Global STP specification

be avoided by ensuring that the pullup and pulldown transistors are not both conducting. This is avoided with the following POD constraint that does not allow the pullup to assert until after the pulldown has been turned off. This constraint states that for stage STP1, the delay of a clock phase must be shorter than the delay through the RES1 block:

$$\mathtt{ck}\uparrow\ \rightarrow\ \mathtt{STP}i\mathtt{.din}\downarrow\ \prec\ \mathtt{STP}i\mathtt{.resi}\downarrow \qquad (\tau_{1,i}^{\mathrm{GSTP}})$$

The pulse width of the outputs in the RES stage of Figure 4(c) are determined by the delay through the output buffers and the self-resetting loop. The following constrains the minimum pulse width on RES2.r2:

$$\mathtt{RES2.r2}{*}\uparrow\ \rightarrow\ \mathtt{RES2.r2}\uparrow\ \prec\ \mathtt{RES2.r2}{*}\downarrow \qquad (\tau_{2}^{\mathrm{GSTP}})$$

Next, consider the footed domino inverter in Figure 4(e). The reset phase must terminate before the data is removed to guarantee the domino gate correctly latches data. Tracing the paths from the clock, we can express this in terms of the following ordering between two sequences of transitions: $\mathtt{ck}\uparrow\mathtt{RES1.r1}\uparrow\mathtt{LP.pch}\downarrow\mathtt{LP.pch}{*}\uparrow\mathtt{LP.q}\downarrow\mathtt{LP.pch}\uparrow\ \prec$ $\mathtt{ck}\uparrow\mathtt{STP1.dout}\uparrow\mathtt{STP2.dout}\uparrow\mathtt{STP3.do}{*}\downarrow\mathtt{STP3.kp}\uparrow\mathtt{STP3.resi}\downarrow$ $\mathtt{STP3.do}{*}\uparrow\mathtt{STP3.dout}\downarrow$. This is enforced with the following constraint:

$$\mathtt{ck}\uparrow\ \rightarrow\ \mathtt{LP.pch}\uparrow\ \prec\ \mathtt{STP3.dout}\downarrow \qquad (\tau_{3}^{\mathrm{GSTP}})$$

To prevent incorrect overlap of the reset of the domino gate in each STP stage we need a constraint stating that STPi.res↓ triggered by the previous rising edge of ck must occur before STPi.kp↑ triggered by the current rising edge of ck. This is a multi-cycle constraint, which when written in terms of a sequence of transitions, is $\mathtt{ck}\uparrow\mathtt{STP}i\mathtt{.res}\uparrow\mathtt{STP}i\mathtt{.res}\downarrow\ \prec$ $\mathtt{ck}\uparrow\mathtt{ck}\downarrow\mathtt{ck}\uparrow\mathtt{STP}i\mathtt{.din}\uparrow\mathtt{STP}i\mathtt{.do}{*}\downarrow\mathtt{STP}i\mathtt{.kp}\uparrow$. We can rephrase this multi-cycle constraint as a compound timing constraint $\tau_{4,i}^{\mathrm{GSTP}}$ XOR $\tau_{5,i}^{\mathrm{GSTP}}$, where $\tau_{4,i}^{\mathrm{GSTP}}$ and $\tau_{5,i}^{\mathrm{GSTP}}$ are two POD constraints given below:

$$\mathtt{ck}\uparrow\ \rightarrow\ \mathtt{STP}i\mathtt{.res}\downarrow\ \prec\ \mathtt{STP}i\mathtt{.kp}\uparrow \qquad (\tau_{4,i}^{\mathrm{GSTP}})$$

$$\mathtt{ck}\uparrow\ \rightarrow\ \mathtt{STP}i\mathtt{.res}\downarrow\ \prec\ \mathtt{ck}\uparrow \qquad (\tau_{5,i}^{\mathrm{GSTP}})$$

To see why this is so, let us perform a case analysis. The first case is when the second instance of transition ck↑ occurs before STPi.res↓. In this case, the same instance of ck↑ precedes both STPi.kp↑ and STPi.res↓, and hence we can simply write it as the POD constraint $\tau_{4,i}^{\mathrm{GSTP}}$. However, if the second instance of ck↑ does not precede STPi.res↓, it simply means that STPi.res↓ occurs before ck↑ fires again; i.e., $\tau_{5,i}^{\mathrm{GSTP}}$ holds, and so does the multi-cycle constraint.

Finally, consider the domino inverter in the LP stage, depicted in Figure 4(d). To avoid a short-circuit in this inverter, the following constraint is *necessary*:

$$\mathtt{ck}\uparrow\ \rightarrow\ \mathtt{LP.pch}{*}\downarrow\ \prec\ \mathtt{RES1.r1}\downarrow \qquad (\tau_{6}^{\mathrm{GSTP}})$$

In all, we needed 33 timing constraints, as shown in Table 1 (we count a compound timing constraint as a single constraint). We were able to prove that under these timing constraints, the circuit conforms to its specification. We also model checked the circuit to verify the absence of short-circuits in all the domino inverters. Run-times were within a few minutes (see Table 1) and memory consumption was less than 150 MB.

Next, we turned to verifying all the timing constraints, successfully verifying all but one: τ_{6}^{GSTP}. Consider this constraint. It takes only 5 gate delays going from ck↑ to RES1.r1↓, while it takes 7 going from ck↑ to LP.pch∗↓. This means that the circuit, as described in the paper [12], has a short-circuit error.

To eliminate this error, we replaced the unfooted domino inverter in the LP stage by a footed domino inverter. With this replacement, constraint τ_{6}^{GSTP} becomes unnecessary. Correctness of the modified circuit was verified *without* using this constraint in about 4 minutes.

5.2. Other Circuits

Among the other circuits we verified, we briefly report here on the modeling of two: the GasP control circuit [30] and the STAPL left-right buffer circuit [23]. A single stage of

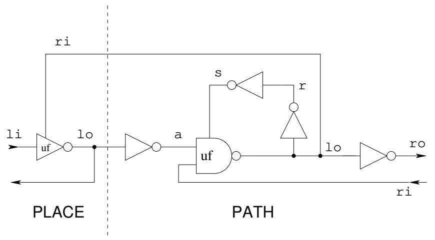


Figure 6. GasP stage

the GasP control circuit is depicted at the gate-level in Figure 6 with normally distributed pullup and pulldown collapsed into the unfooted domino inverter. To ensure correct operation of this circuit, we needed to specify 4 POD constraints for each stage. A sample constraint is

$$\mathtt{PATH.lo}\downarrow\ \rightarrow\ \mathtt{PATH.ri}\downarrow\ \prec\ \mathtt{PATH.s}\downarrow \qquad (\tau_{1}^{\mathrm{GASP}})$$

We checked conformance of the circuit in Figure 6 to its specification. We also connected 10 stages together in a ring with exactly one full stage, and model checked it for absence of short circuits and to verify that exactly one stage was full at any given point of time. Both verifications completed within a minute, as shown in Table 1.

The STAPL left-right buffer (see Figure 4.7 in [23]) is different from the other two circuits in that it uses metric timing constraints. For correct operation, the circuit employs two pulse generators with pulse-lengths less than constants $\sigma_{\mathbf{true}}$ and $\sigma_{\mathbf{false}}$ respectively, along with two corresponding paths in the circuit that are respectively required to take longer than constants $\xi_{\mathbf{true}}$ and $\xi_{\mathbf{false}}$, and an additional constraint that $\xi_{\mathbf{true}} \geq \sigma_{\mathbf{true}}$ and $\xi_{\mathbf{false}} \geq \sigma_{\mathbf{false}}$. These timing constraints naturally lend themselves to being modeled as metric constraints with clock variables, with 2 constraints (4 clock variables) per buffer stage. In addition to these constraints, each stage also requires 6 POD constraints. We checked conformance of this circuit with the specification given by Nyström and Martin [23] and also model checked a ring of 3 buffers (for same properties as the GasP circuit); both verifications completed successfully within a few minutes.

5.3. Comparison with Other Tools

Table 1 summarizes our experimental results on the 3 circuits discussed so far.

Circuit	$\|\mathcal{V}\|$	$\|\mathcal{T}\|$			Verif. Time (sec.)	
		POD	XOR	Metric	Conf.	MC
Global STP	28	27	6	0	215.14	66.32
GasP-10	60	40	0	0	0.02	26.10
STAPL-3	30	18	0	6	235.61	278.05

Table 1. Summary of experimental results with TMV. $|\mathcal{V}|$ is the number of signals, $|\mathcal{T}|$ is the number of timing constraints with associated break-up into categories, "Conf." is the time for conformance (simulation) checking, and "MC" is the time for model checking.

We compared the performance of TMV to ATACS [3], which is based on metric timing. ATACS uses model checking algorithms that are explicit-state in the Boolean component and prune the search space using partial-order reduction methods.[4] In modeling the Global STP (the corrected version) and STAPL circuits, we assigned min-max delay ranges to all gates so that timing is analogous to counting transitions, but for the GasP circuit we had to assign ranges more carefully so that all POD constraints were satisfied. For all three circuits, ATACS did not finish within an hour, running out of memory for the STAPL and Global STP circuits.

Our tool also outperforms the conformance checking tool ANALYZE [27] that was enhanced with the capability to model relative-timing constraints (but not metric timing). For example, for the Global STP circuit, ANALYZE was able to check conformance for individual modules (e.g., a single STP stage) in just a few seconds, but did not finish within 25 days for the flat circuit. This illustrates the need for combining the GRT modeling methodology with a fully symbolic verification tool.

For the circuits discussed so far, most timing constraints are simple POD constraints, and very few constraints are metric. Hence, we only needed to introduce few clock variables, if any. This enabled TMV to scale well on these circuits.

4 The results reported for ATACS are for the partial-order reduction option that yielded best results.

As mentioned in Section 2, metric constraints can usually be reformulated as POD constraints, but at the cost of modularity. Using the STARI circuit [9], we studied the relative performance of TMV for two different ways of modeling constraints. (The reader is referred to Greenstreet's thesis [9] for a description of the circuit.) All timing constraints for this circuit can be modeled as POD constraints, where the POD is the clock that is distributed to both sender and receiver modules. This breaks modularity, since timing constraints for each buffer stage between the sender and the receiver require tracing back to the global clock. One can also formulate these constraints as metric timing constraints specifying that, for each buffer stage, an output data bit and `ack` must follow an input within a clock phase. In our circuit model, we abstracted the data-path to only one bit, and modeled only one of the two bits making up the dual rail encoding. Thus, each stage contributes two Boolean state variables. The resulting timed automaton has 4 clock variables (one per metric constraint) for every two stages.

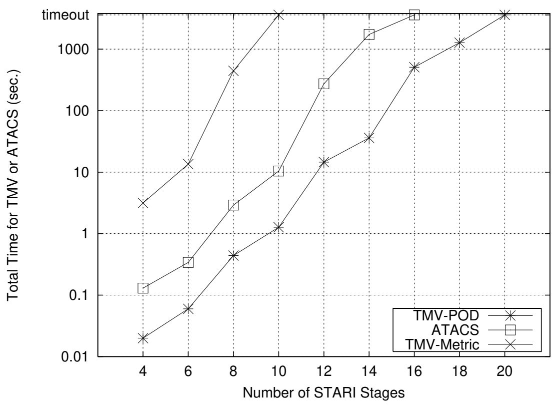


Figure 7. Results for STARI circuit. Note that the Y-axis is on a log scale. A timeout of 3600 seconds was imposed on all runs.

We computed the set of reachable states for STARI circuits (initialized to be half-full) for increasing numbers of buffer stages and in three different ways: (1) using ATACS, (2) using TMV with purely POD constraints, and (3) using TMV with modularly specified metric constraints. The results are displayed in Figure 7. Using TMV with purely POD constraints is the most scalable approach, followed by ATACS. When used on a model with metric constraints, TMV scales very poorly. The reason for this appears to be that each clock zone has few corresponding Boolean states, thus reducing the benefits of using fully symbolic Boolean methods of quantifier elimination. On the model based purely on POD constraints, TMV runs an order of magnitude faster than ATACS.

6. Conclusions and Future Work

We have proposed a novel approach to modeling timing constraints in digital circuits, based on the notion of generalized relative timing. Circuits with such timing constraints are formally encoded as timed automata, to which we apply fully symbolic verification techniques. Our approach is illustrated on real circuits, such as the Global STP circuit.

Possibilities for future work include automatically generating timing constraints, and applying compositional reasoning and abstraction methods to further scale up verification to larger circuits.

Acknowledgments

We thank Tiberiu Chelcea, Seth Goldstein, and Mika Nyström for supplying us with examples. We also thank Chris Myers for help with ATACS. This research was supported in part by ARO grant DAAD19-01-1-0485.

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[2] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In *Real Time: Theory in Practice*, LNCS 600, pages 74–106. Springer-Verlag, July 1991.

[3] ATACS verification tool. http://www.async.ece.utah.edu/tools/.

[4] W. Belluomini. *Algorithms for Synthesis and Verification of Timed Circuits and Systems*. PhD thesis, Univ. of Utah, 1999.

[5] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In 5^{th} *IEEE Symposium on Logic in Computer Science (LICS)*, pages 428–439, 1990.

[6] R. Clarisó and J. Cortadella. Verification of timed circuits with symbolic delays. In *Proc. Conference on Asia South Pacific Design Automation (ASP-DAC)*, pages 628–633, 2004.

[7] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[8] Colorado University decision diagrams package (CUDD). http://vlsi.colorado.edu/˜fabio/CUDD.

[9] M. R. Greenstreet. *STARI: A Technique for High-Bandwidth Communication*. PhD thesis, Princeton University, 1993.

[10] T. A. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for real-time systems. In *18th Annual ACM Symposium on Principles of Programming Languages*, pages 353–366. ACM press, 1991.

[11] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[12] G. Hinton, M. Upton, D. Sager, *et al.* A 0.18 CMOS IA-32 processor with a 4-GHz integer execution unit. *IEEE Journal of Solid-State Circuits*, 36(11):1617–1627, November 2001.

[13] H. Kim, P. A. Beerel, and K. S. Stevens. Relative timing based verification of timed circuits and systems. In *8th International Symposium on Asynchronous Circuits and Systems*, pages 115–126. IEEE Press, Apr. 2002.

[14] O. Maler and A. Pnueli. Timing analysis of asynchronous circuits using timed automata. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 189–205, 1995.

[15] A. J. Martin. Synthesis of asynchronous VLSI circuits. Technical Report CS-TR-93-28, Computer Science Department, California Institute of Technology, 1993.

[16] K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *14th International Conference on Computer-Aided Verification (CAV)*, LNCS 2404, pages 250–264. Springer-Verlag, July 2002.

[17] E. Mercer. *Correctness and Reduction in Timed Circuit Analysis*. PhD thesis, University of Utah, 2002.

[18] R. Milner. An algebraic definition of simulation between programs. In *2nd International Joint Conference on Artifical Intelligence (IJCAI)*, pages 481–489, Sept. 1971.

[19] C. J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Stanford Univ., 1995.

[20] R. Negulescu. A technique for finding and verifying speed-dependences in gate circuits. In *Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, pages 189–198, 1997.

[21] R. Negulescu and A. Peeters. Verification of speed-dependences in single-rail handshake circuits. In *4th International Symposium on Asynchronous Circuits and Systems (ASYNC'98)*, pages 159–171. IEEE Press, 1998.

[22] S. M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. PhD thesis, Stanford Univ., 1993.

[23] M. Nyström and A. Martin. *Asynchronous Pulse Logic*. Kluwer Academic Publishers, 2002.

[24] M. A. Peña, J. Cortadella, A. Kondratyev, and E. Pastor. Formal verification of safety properties in timed circuits. In *6th International Symposium on Asynchronous Circuits and Systems (ASYNC'00)*, pages 2–11. IEEE Press, 2000.

[25] C. Ramchandani. Analysis of asynchronous concurrent systems by timed Petri nets. Technical Report Project MAC Tech. Rep. 120, Mass. Inst. of Technology, Feb. 1974.

[26] S. A. Seshia and R. E. Bryant. Unbounded, fully symbolic model checking of timed automata using Boolean methods. In *Intl. Conference on Computer-Aided Verification*, LNCS volume 2725, pages 154–166, July 2003.

[27] K. S. Stevens. *Practical Verification and Synthesis of Low Latency Asynchronous Systems*. PhD thesis, University of Calgary, Calgary, Alberta, September 1994.

[28] K. S. Stevens, R. Ginosar, and S. Rotem. Relative timing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(1):129–140, February 2003.

[29] K. S. Stevens, S. Rotem, R. Ginosar, P. Beerel, C. J. Myers, K. Y. Yun, R. Koi, C. Dike, and M. Roncken. An asynchronous instruction length decoder. *IEEE Journal of Solid-State Circuits*, 36(2):217–228, 2001.

[30] I. Sutherland and S. Fairbanks. GasP: A minimal FIFO control. In *7th International Symposium on Asynchronous Circuits and Systems*, pages 46–53. IEEE Press, March 2001.

[31] Synopsys. *PrimeTime User Guide: Advanced Timing Analysis*, Version T-2002.09.

[32] S. Tasiran, R. Alur, R. P. Kurshan, and R. K. Brayton. Verifying abstractions of timed systems. In *7th International Conference on Concurrency Theory (CONCUR)*, LNCS volume 1119, pages 546–562, August 1996.

[33] G. Venkataramani, M. Budiu, T. Chelcea, and S. C. Goldstein. C to asynchronous dataflow circuits: An end-to-end toolflow. In *Intl. Workshop on Logic Synthesis (IWLS)*, pages 501–508, June 2004.

[34] T. Yoneda and H. Ryu. Timed trace theoretic verification using partial order reduction. In *5th Intl. Symposium on Asynchronous Circuits and Systems*, pages 108–124, 1999.

[35] H. Zheng, C. J. Myers, D. Walter, S. Little, and T. Yoneda. Verification of timed circuits with failure directed abstractions. In *21st International Conference on Computer Design (ICCD)*, pages 28–35. IEEE Press, 2003.