Model Checking Synchronous Timing Diagrams

Nina Amla[1], E. Allen Emerson[1], Robert P. Kurshan[2], and Kedar S. Namjoshi[2]

[1] Department of Computer Sciences, University of Texas at Austin ***
{namla,emerson}@cs.utexas.edu
http://www.cs.utexas.edu/users/{namla,emerson}
[2] Bell Laboratories, Lucent Technologies
{k,kedar}@research.bell-labs.com
http://cm.bell-labs.com/cm/cs/who/{k,kedar}

Abstract. Model checking is an automated approach to the formal verification of hardware and software. To allow model checking tools to be used by the hardware or software designers themselves, instead of by verification experts, the tools should support specification methods that correspond closely to the common usage. For hardware systems, timing diagrams form such a commonly used and visually appealing specification method. In this paper, we introduce a class of synchronous timing diagrams with a syntax and a formal semantics that is close to the informal usage. We present an efficient, decompositional algorithm for model checking such timing diagrams. This algorithm has been implemented in a user-friendly tool called RTDT (the Regular Timing Diagram Translator). We have applied this tool to verify several properties of Lucent's PCI synthesizable core.

1 Introduction

Model checking [8, 24, 9] is a fully automated method for determining whether a hardware or software design, represented as a finite state program, satisfies a temporal correctness property. Currently, many model checking tools are used most effectively by verification experts. In order to make these tools accessible to the hardware or software designers themselves, the tools should support specification methods that correspond closely to common usage. For hardware systems, *timing diagrams* form such a commonly used and visually intuitive specification method. Timing diagrams are, however, often used informally, without a well-defined semantics, which makes it difficult, if not impossible, to use them as specifications for formal verification. In this paper, therefore, we precisely define a class of timing diagrams called *Synchronous Regular Timing Diagrams* (SRTD's) and provide a formal semantics that corresponds closely to the informal usage.

A key issue in using timing diagrams for model checking is whether the algorithms that translate timing diagrams into more basic specification formalisms

*** This work was supported in part by NSF grant 980-4736 and TARP 003658-0650-1999.

such as temporal logic or ω-automata yield formulas or automata that are of small size. Previous work on model checking for timing diagrams, e.g., with Symbolic Timing Diagrams [10, 5, 7], with non-regular timing diagrams [12] and with Presburger arithmetic [3] provides algorithms that are, in the worst-case, of exponential or higher complexity in the size of the diagram. Our timing diagram syntax facilitates a decompositional, *polynomial-time* algorithm for model checking. Our experience with verifying Lucent's PCI synthesizable core and other protocols indicates that the SRTD syntax can express common timing properties and is expressive enough for industrial verification needs.

In previous work [1, 2], we proposed a class of timing diagrams called RTD's (for Regular Timing Diagrams) that are particularly well-suited for describing *asynchronous* timing, such as that arising, for instance, in asynchronous read/write bus transactions. It is also quite common to have a *synchronous* timing specification, where the changes in values along a signal waveform are tied to the rising or falling edges of a clock waveform. While these specifications can be encoded as RTD's, the encoding introduces a large number of dependency edges between each transition of the clock and each waveform, which results in RTD's that are visually cluttered and have (unnecessarily) increased complexity for model checking. The SRTD notation proposed in this paper is, therefore, tailored towards describing synchronous timing specifications in a visually clean manner. More importantly, we exploit the structure of SRTD's to provide a model checking algorithm that is more efficient than that for RTD's. We present a *decompositional* model checking algorithm that constructs an ω-automaton of size quadratic in the timing diagram size (compared with a cubic size complexity in [2] for RTD's). This automaton, which represents all system computations that *falsify* the diagram specification, is composed with the system model and it is checked if the resulting automaton has an empty language using standard algorithms (cf. [26]). If the language is not empty, there is a system computation that falsifies the specification; otherwise, the system satisfies the specification.

This algorithm is implemented in a tool - the *Regular Timing Diagram Translator* (RTDT). RTDT provides a user-friendly graphical editor for creating and editing SRTD's and a translator that compiles SRTD's to the input language of the formal verification tool COSPAN/FormalCheck [14]. The output of the tool can be easily re-targeted to other verification tools such as SMV [21] and VIS [6]. We used RTDT to verify that Lucent's synthesizable PCI Core satisfies several properties encoded as SRTD's; the SRTDs were formulated by looking at the actual timing diagrams in the PCI Bus specification [23] and the PCI Core User's manual [4].

The rest of the paper is organized as follows. Section 2 presents the syntax and semantics of SRTD's. In Section 3, we describe the decompositional translation algorithm that converts SRTD's into ω-automata. The features of the RTDT tool are described in Section 4. Section 5 illustrates some applications of the RTDT tool, including a Master-Slave memory access protocol and the synthesizable PCI Core of Lucent's F-Bus. We conclude with a discussion of related work in Section 6.

2 Synchronous Regular Timing Diagrams

A Synchronous Regular Timing Diagram (henceforth referred to as an SRTD or diagram), in its simplest form, is specified by describing a number of waveforms with respect to the clock. A *clock point* is defined as a change in the value of the clock signal. The clock is depicted as waveform defined over $\mathcal{B} = \{0, 1\}$ where the value toggles at consecutive clock points. A *clock cycle* is the period between any two successive rising or falling edges of the clock waveform.

In SRTD's, an *event*, which is a change in the signal value, must occur at either the rising edge of the clock (rising edge triggered) or at the falling edge (falling edge triggered). In the SRTD in Figure 1, signals p and r are falling edge triggered while q is rising edge triggered. Timing diagrams may either be *unambiguous*, where the events are linearly ordered, or *ambiguous*, where the events are partially ordered with respect to time [11]. Synchronous timing diagrams are generally unambiguous but the don't-care transitions do introduce some degree of ambiguity in SRTD's.

2.1 Syntax

In most applications of timing diagrams, the waveform behavior specified by the diagram must hold of a system only after a certain *precondition* holds. This condition may be a boolean condition on the values of one or more signals (a *state* condition), or a condition on the signal values over a finite period of time (a *path* condition). To accommodate this type of reasoning, we permit the more general form of path preconditions to be specified in an SRTD. Preconditions are specified graphically by a solid vertical marker that partitions the SRTD into two parts, a precondition part that includes all the events at and to the left of the marker and a postcondition part that contains all the events to the right of the marker. The precondition of the diagram in Figure 1 is a path precondition, given by the path $\langle\bar{p}(q+\bar{q})r\rangle;\langle\bar{p}(q+\bar{q})\bar{r}\rangle;\langle\bar{p}\bar{q}\bar{r}\rangle$ (the angle brackets indicate the constraints on signal values at a clock edge, while ";" indicates succession in time, measured by clock edges).

A common feature in synchronous timing diagrams is a way to express that the value of a signal during a certain period is not important. We use *don't-care values* to specify that the value at a point is unknown, unspecified or unimportant. In Figure 1, the don't-care values on waveform q are used to state that q should not be considered in the precondition. With the addition of preconditions, one can express properties of the form "if B rises then A rises in exactly 5 time units". In order to specify richer properties such as "if B rises then A rises within 5 time units", we need a way of stating that the exact occurrence of the rising transition of A is not important as long as it is within the specified time bound. In SRTD's, we use a *don't-care transition* to graphically represent this temporal ambiguity. The don't-care transition is defined for a particular waveform over one or more clock cycles; its semantics specifies that the signal may change its value at any time during the specified interval and that, once it changes, it remains stable for the remainder of the interval. This stability requirement is the

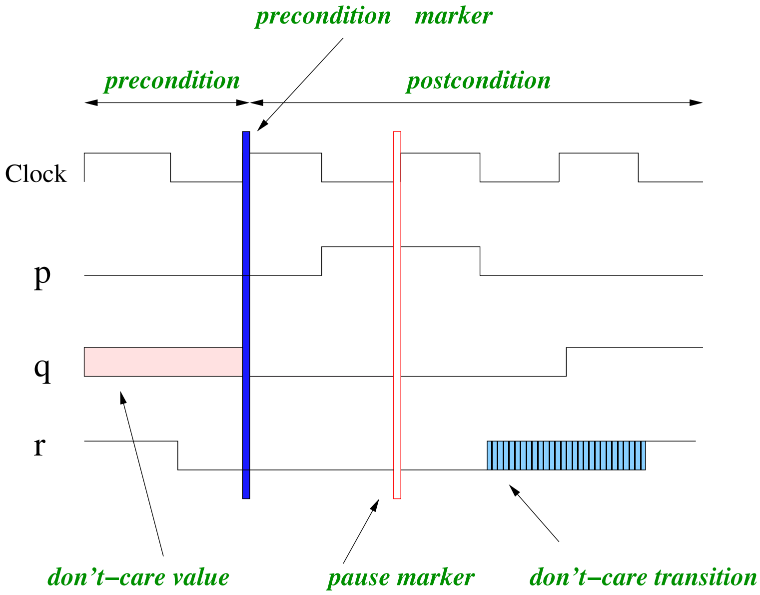

Fig. 1. Annotated Synchronous Regular Timing Diagram

only difference between don't-care transitions and don't-care values. In Figure 1, the don't-care transition allows signal r to rise in either the third or fourth clock cycle.

In addition, in loosely coupled systems, it may not be always necessary to explicitly tie every event to the clock. This is useful in stating eventuality properties like "every memory request is eventually followed by a grant", and is represented diagrammatically by a pause marker. A *pause* specifies that there is a break in explicit timing at that point, i.e. the state of the signals (except the clock) remains unchanged (stutters) for an arbitrary finite period of time before changing. In Figure 1 the pause at the end of the second clock cycle indicates that the state $\langle p\bar{q}\bar{r}\rangle$ stutters for a finite period (until p changes at a falling edge). The pauses allow us to express richer properties like "if A rises then eventually B rises".

We have observed that, in practice, both pauses and don't-care objects occur in timing diagrams, and that preconditions are often implicit in the assumptions that are made with respect to when a diagram must be satisfied. In reviewing many specifications and our discussion with engineers, we are led to believe that SRTD's correspond closely to informal usage and are expressive enough for industrial verification needs.

We now define SRTD's formally. An SRTD is defined over a set of "symbolic values" $\mathcal{SV}=\mathcal{B}\cup\{X,D\}$, where X is a don't-care value and D indicates a don't-care transition. The set $\mathcal{SV}$ is ordered by $\sqsubseteq$, where $a \sqsubseteq b$ iff either $a=b$

or $a \in \{X, D\}$ and $b \in \mathcal{B}$. The alphabet of an SRTD defined over n signals is $\mathcal{SV}_n = \{(a_1 a_2 ... a_n) | a_1 \in \mathcal{SV} \wedge ... \wedge a_n \in \mathcal{SV}\}$. Here, we have taken the set of defined values to be the boolean set $\mathcal{B}$ but our algorithms and results also apply when this is any fixed finite set, such as an enumeration of the possible values of a multi-valued signal.

Definition 1 (SRTD) *An SRTD T is a tuple (c,S,WF,M) where*

- $c > 1$ *is an integer that denotes the number of clock points.*
- S *is a non-empty set of signal names (excluding the clock).*
- WF *is a collection of waveforms; for each signal $A \in S$, its associated waveform is a function $WF_A : [0, c) \rightarrow \mathcal{SV}$, while the associated waveform for the clock is $WF_{clk} : [0, c) \rightarrow \mathcal{B}$.*
- M *is a finite (non-empty) ascending sequence $0 \leq M_0 < M_1 < ... < M_{k-1} < c-1$ of position markers. M_0 is the precondition marker, while for each $i > 0$, M_i is the i-th pause marker.*

To facilitate defining the semantics as well as the algorithms it is also helpful to view an SRTD as a collection of "segments", where each segment is essentially a vertical "slice" of the timing diagram, encompassing all waveforms, between two successive markers or a marker and the start/end of the diagram. The k markers in M partition the interval $[0, c)$ in an SRTD T into $k + 1$ disjoint subintervals $I_0 = [0, M_0]$, $I_1 = (M_0, M_1]$,...,$I_{k-1} = (M_{k-2}, M_{k-1}]$, $I_k = (M_{k-1}, c-1]$. The length m_0 of the interval I_0 is M_0+1, while for intervals I_i, with $i \in [1, k)$, the length m_i of I_i is $M_i - M_{i-1}$, and the length of the last interval I_k is $c-1-M_{k-1}$. The k markers, therefore, partition an SRTD into $k + 1$ segments.

Definition 2 (Segment) *The segment $Post_i$ ($i \in [0, k]$) that corresponds to the interval I_i of length m_i is defined to be a function $Post_i : S \times [0, m_i) \rightarrow SV$, where for each $j \in [0, m_i)$ and $A \in S$, $Post_i(A)(j) = WF_A(M_{i-1} + 1 + j)$. We refer to $Post_0$ as Pre.*

Any SRTD $T = (c, S, WF, M)$ can be represented as the tuple of segments $(Pre, Post_1, ..., Post_k)$ as defined above. Segment Pre represents the precondition, while segments $Post_i$, for $i > 0$, represent successive post-condition segments. For instance, the SRTD in Figure 1 has three segments, one precondition segment and two postcondition segments. For each signal A, $Post_i(A)$ is a function from $[0, m_i) \rightarrow \mathcal{SV}$ which describes the waveform for signal A in the ith segment. This representation of an SRTD is useful in the sequel.

We impose certain well-formedness criteria on SRTD's. In prepartion, we define an event to be *precisely locatable* if it occurs at a clock point where the signal value changes from 0 to 1 or vice versa. In Figure 1, the falling edge of waveform p in the third clock cycle is precisely locatable while the don't care transition in waveform r is not a precisely locatable event.

Definition 3 (Well-formed SRTD) *An SRTD $T = (Pre, Post_1, ..., Post_k)$ is well-formed iff*

- *The precondition segment Pre does not have any don't-care transitions*[1]*. Note that the precondition can have don't-care values.*
- *There is at least one* precisely locatable *event in the clock cycle immediately following each pause.*
- *Any non-empty sequence of D's (don't-care transitions) must be immediately preceded by a boolean value and must be followed by the negation of this value.*

2.2 Semantics

An SRTD defines properties of *computations*, which are sequences of *states*, where a state is an assignment of values from $\mathcal{B}$ to each of the n waveform signals. A computation is defined over the alphabet $\mathcal{B}_n = \{(a_1 a_2 ... a_n) | a_1 \in \mathcal{B} \wedge ... \wedge a_n \in \mathcal{B}\}$. For any $y \in \mathcal{B}_n^+$, we use y_A to denote the projection of y on to the coordinate for signal A.

Definition 4 ($\dot\sqsubseteq$) *For a finite waveform segment $Post_i(A) : [0, m_i) \rightarrow \mathcal{SV}$ and a projection y_A of length m_i of computation y ($y_A \in \mathcal{B}^{m_i}$), $Post_i(A) \dot\sqsubseteq y_A$ iff*

- *For every $p \in [0, m_i)$, $Post_i(A)(p) \sqsubseteq y_A(p)$.*
- *For every p, q, if $Post_i(A)[p..q]$ has the form $(a; D^+; \bar{a})$ then $y_A[p..q]$ has the form $(a^+; \bar{a}^+)$, where $a, \bar{a} \in \mathcal{B}$ and $\bar{a} \neq a$.*

Definition 5 (Segment Consistency) *A segment $Post_i$ of length m_i is satisfied by a sequence $y \in \mathcal{B}_n^{m_i}$ iff for each signal A, $Post_i(A) \dot\sqsubseteq y_A$ holds.*

We will now construct regular expressions for the precondition Pre_T and the postcondition $Post_T$ of a SRTD T. By the definition above of segment consistency, any Pre or $Post_i$ segment can be represented as an extended regular expression of the form $\bigwedge_{s \in S} r_s$, where r_s encodes the constraints for the waveform for signal s in the segment. The regular expression for $Post_T$ is the concatenation of sub-expressions that correspond to each $Post_i$ segment separated by an expression for each pause. Thus, $Post_T = (seg_1; val_1^*; seg_2; val_2^*; ...; seg_{k-1})$, where seg_i is the regular expression for segment $Post_i$ and val_i is the vector of values at the last position $(m_i - 1)$ in $Post_i$, which is just before the pause marker separating it from $Post_{i+1}$.

Definition 6 (Always followed-by) $\mathbf{G}(p \hookrightarrow q)$ *holds of a computation σ iff, for all i, for all j, $j \geq i$, if sub-computation $\sigma[i \ldots j] \models p$, then there exists k such that*
$\sigma[j+1 \ldots k] \models q$.

[1] We can relax this requirement and our translation algorithm is still applicable. In that case, however, we cannot guarantee an efficient translation.

In the definition above, p and q are arbitrary path properties; however, when p is a state property, $\mathbf{G}(p \hookrightarrow q)$ is equivalent to $\mathbf{G}(p \Rightarrow \mathbf{X}q)$, where $\mathbf{X}$ is the next time operator. An infinite computation σ satisfies an SRTD T (written $\sigma \models T$) if and only if every finite segment that satisfies the precondition is immediately followed by a segment that satisfies the postcondition of the diagram. This is formalized in Definition 7.

Definition 7 (SRTD Semantics) *An infinite computation σ satisfies an SRTD T ($\sigma \models T$) iff $\sigma \models \mathbf{G}(Pre_T \hookrightarrow Post_T)$.*

3 Model Checking SRTD's

We first present an algorithm that translates an SRTD into an ω-automaton for the *negation* of the SRTD property. We then present the model checking algorithm that makes use of this automaton.

3.1 Translation Algorithm

The algorithm translates SRTD's into ω-automata, which are finite state automata accepting infinite computations as input (cf. [17]). It proceeds by decomposing T into waveforms and producing sub-automata that track portions of each waveform. It consists of the following four steps.

1. Partition the diagram into the precondition part and the postcondition part.
2. Construct a single deterministic automaton $\mathcal{A}_P$ for the precondition. This automaton tracks the values of all signals simultaneously over the number of clock cycles of the precondition. Since the precondition cannot contain don't-care transitions, this automaton has linearly many states in the length of the precondition.
3. Construct a deterministic automaton S_i for each signal i of the postcondition. The automaton S_i tracks the waveform for signal i over all the postcondition segments. The automaton checks at each clock cycle that the waveform has the specified value. For a don't-care transition, the automaton maintains an extra bit that records whether the transition has occurred. For a pause, the automaton goes into a "waiting" state, which it leaves when the precisely locatable (non-don't-care) event signaling the end of the pause occurs. The number of states of this automaton is thus linear in the length of the postcondition. In our model, the pause condition is required to hold for only a finite (but unbounded) number of cycles. Thus, S_i has a fairness condition which ensures that the automaton does not stay in a waiting state forever.
4. Construct an *NFA* $\mathcal{A}_{\bar{T}}$ for the negation of the SRTD property of T that operates as follows on an infinite input sequence: it nondeterministically "chooses" a point where the precondition holds, runs the *DFA* $\mathcal{A}_P$ at this point and if $\mathcal{A}_P$ accepts it then "chooses" a postcondition *DFA* S_i and runs this automaton at the point where $\mathcal{A}_P$ accepted and accepts if this automaton rejects.

A $\forall FA$ [20, 25] is a finite state automaton that accepts an input iff *every* run of the automaton along the input meets the acceptance criterion. An SRTD T can be represented succinctly by a $\forall FA$ $\mathcal{A}_T$ that has the identical structure as the *NFA* $\mathcal{A}_{\bar{T}}$ but complemented acceptance condition.

The *size* of an SRTD is the product of the number of signals and the number of clock cycles. The number of clock cycles does not include the indeterminate amount of time represented by a pause; it refers only to the explicitly indicated clock cycles in the diagram. The automata produced by the translation algorithm all have linearly many states, in terms of the size of the SRTD.

Theorem 1 (Correctness) *For any SRTD T and $x \in \mathcal{B}_n^\omega$, $x \models T$ iff $x \in L(\mathcal{A}_T)$.*

Theorem 2 (Complexity) *For any SRTD T and the equivalent $\forall FA$ $\mathcal{A}_T$, the size of $\mathcal{A}_T$ is quadratic in $|T|$.*

Proof. The size of an SRTD $T{=}(Pre, Post_1, ..., Post_k)$ is $n * c$, where n is the number of waveforms and c is the number of clock points. We assume that the transitions in $\mathcal{A}_T$ are labeled with boolean formulas over the n signals. The size of the transitions in $\mathcal{A}_T$ is the sum of the length of the formulas labeling the transitions. The size of $\mathcal{A}_T$ is $s + t$, where s is the number of states and t is the transition size.

The number of states s in the monolithic automaton for the precondition $\mathcal{A}_P$, is bounded by p, therefore $s = O(c) = O(|T|)$. Since each transition encodes the values of the signals at each point, the size of each transition is $O(n)$, while there are p transitions. Thus, the transition size is linear in $|T|$.

The number of states s in S_i is bounded by the number of clock points c, therefore $s = O(c) = O(|T|)$. Except for the pause transition, the transitions are labeled with constant size formulae. The pause transition may be dependent on a number of (simultaneous) signal value changes, so it can have size at most n. Thus, the overall transition size for S_i is of order $|T|$; hence, S_i has size linear in the size of T.

The size of the $\forall FA$ $\mathcal{A}_T$ is the sum of the sizes of the precondition and the n postcondition automata and is thus $(n + 1).|T|$.
□

3.2 Model Checking

Theorem 2 shows that the the SRTD property can be represented succinctly by a $\forall FA$. Representing the property by an *NFA* requires a postcondition automaton that is essentially the product (intersection) of all the S_i automata – we call this a *monolithic* translation. This monolithic NFA can be of size exponential in the size of the SRTD, because it needs to take into account all possible interleavings of the don't-care transitions of the postcondition.

Recall that the property represented by the SRTD T is $\mathbf{G}(Pre_T \hookrightarrow Post_T)$. Since $Post_T = \bigwedge_i S_i$, this property can be decomposed into the conjunction of individual checks $\mathbf{G}(Pre_T \hookrightarrow S_i)$. In a typical model checker, this check is performed by determining if there is a computation of the system that satisfies the *negation* of the property. The check can be done by determining if there is a path to a point where A_P accepts, followed by a computation where S_i rejects. Since S_i is a DFA, it can be complemented to form an automaton of the same size. Hence, model-checking can be done efficiently with this decomposed representation of the postcondition. A similar observation was made for the analysis of asynchronous timing diagrams in [2].

Theorem 3 *For a transition system M and an SRTD T, the time complexity of model checking is linear in the size of M and quadratic in the size of T.*

Proof. The $\forall FA$ $\mathcal{A}_T$, corresponding to T, is the automaton for $\mathbf{G}(Pre_T \hookrightarrow \bigwedge_i S_i)$, where Pre_T is the automaton for Pre and each S_i is the automaton for the postcondition segment of waveform i. The problem of checking $M \models A_T$ can be decomposed into $\bigwedge_i M \models A_i$, where $\mathcal{A}_i$ is the automaton for $\mathbf{G}(Pre_T \hookrightarrow S_i)$. We can check $M \models A_i$ in time linear in the size of M and A_i, which by Theorem 2 is $O(|M|.|T|)$. But we have $|S|$ such verification tasks, thus the time complexity of checking $M \models A_T$ is $O(|M|.|T|^2)$.
$\square$

4 The RTDT Tool

RTDT is a tool that translates SRTD's into ω-automata definitions which are input to the verification tool COSPAN [14]. The tool has an editor to create SRTD's and a translator that outputs the corresponding descriptions in COSPAN's input language.

The RTDT editor is a graphical environment, written in Java, that enables a user to create and edit SRTD's. The editor is almost entirely mouse driven. There are options to open, save and print existing SRTD descriptions. A user may easily add/delete waveforms, clock cycles and pauses. The precondition defaults to the initial clock point but the user can set the precondition to a path condition. Editing a waveform is done by positioning the mouse on the waveform and clicking either the left or right button. The editor is designed to ensure that the diagrams created are well-formed SRTD's by construction. The tool provides a user-friendly interface for specifying SRTD's. Figure 2 is a screen shot of the interface.

The output of the tool is currently targeted to the formal verification tool COSPAN/FormalCheck [14]. We use a macro in COSPAN/FormalCheck called a *strobe* that recognizes a specified waveform. The RTDT translator automatically generates strobe definitions for the diagram using the algorithm outlined in Section 3. The resulting strobe definition can be viewed through the editor as in Figure 2 or saved in a file to be used as the specification in model checking the system under verification.

COSPAN/FormalCheck generates a failure trace when a property fails to hold of a system. This trace corresponds to a bad path through the system and

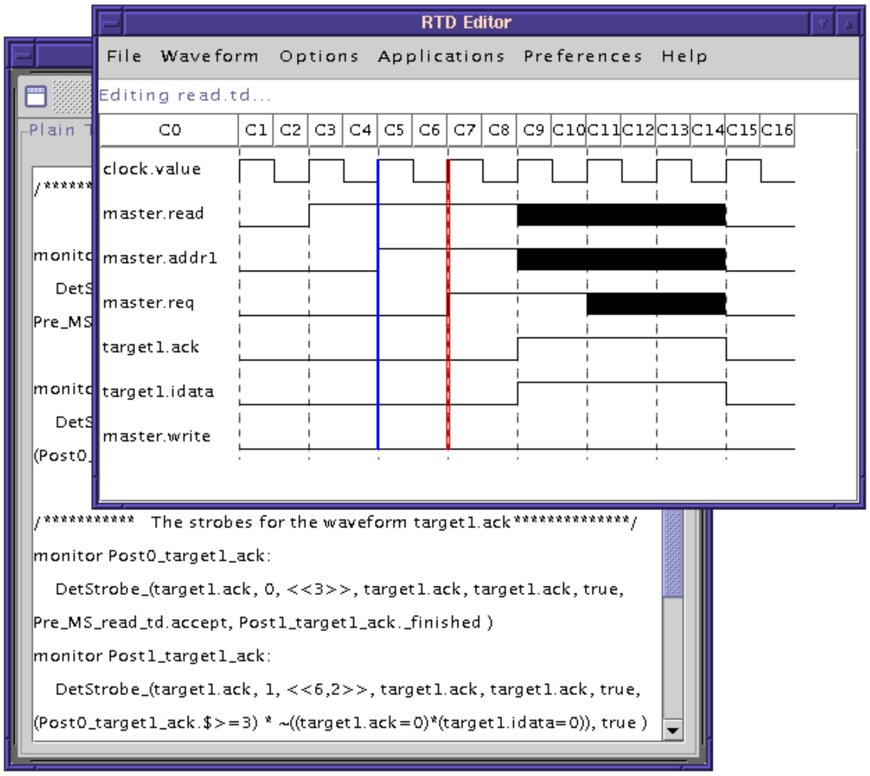

Fig. 2. Editing and Viewing Screens of RTDT

is usually long and very hard to read. Currently many model checkers display these traces graphically as synchronous timing diagrams. RTDT offers the added advantage of allowing the user to edit these SRTD traces. One can remove irrelevant signals and clock cycles. Furthermore the precondition may be used to direct attention to erroneous regions of the design. This feature allows the use of a relevant portion of the trace as a new property; this is useful in debugging the system.

Although the output of the tool is currently targeted to COSPAN/FormalCheck, with very little effort, the tool can be re-targeted to generate output suitable for other formal verification tools such as SMV [21] or VIS [6].

5 Applications

The true test of the efficiency of our algorithms is how they fare in practice on industrial examples of all sizes. Towards this end, we used RTDT with

COSPAN/FormalCheck to verify two systems. The first is a synchronous master-slave memory system and the second is the synthesizable Core of Lucent's F-Bus [4].

5.1 Master-slave Memory System

The Master-Slave memory system consists of one master module and three slave modules. In the master-slave system, the master issues a memory instruction and the slaves respond by accessing memory and performing the operation. The master initiates the start of a transaction by asserting either the read or write line. Next the master puts the address on the address bus and asserts the *req* signal. The slave whose tag matches the address awakens, services the request, then asserts the *ack* line on completion. Upon receiving the *ack* signal the master resets the *req* signal, causing the slave to reset the *ack* signal. Finally, the master resets the address and data buses.

Design	Number of BDD variables	Average BDD size	Average Space (MBytes)	Average Time (seconds)
master–slave	67	11405	0.85	–
read (C)	95	13433	0.86	0.32
read (M)	205	22079	1.46	3.19
write (C)	95	11542	0.86	0.31
write (M)	205	21915	1.45	2.51

Table 1. Verification Statistics for Master-Slave Design

We verified that this system satisfied both read (see Figure 2) and write memory transactions formulated as SRTD's. The SRTD's were created with the RTDT editor and the translator was used to generate the corresponding COSPAN/FormalCheck descriptions. We used COSPAN/FormalCheck to model check the system with respect to these descriptions.

In order to compare our decompositional algorithms with monolithic algorithms, we did the verification checks both decompositionally and monolithically. In Table 1, *read(M)* corresponds to the verification check on the master-slave design and the monolithic automaton for the read SRTD while *read(C)* corresponds to the verification check done on the master-slave design and automata for a single waveform. The numbers in Table 1 for BDD size, space and time for the decompositional check is the average over the individual verification checks for each waveform. For example, the total amount of time taken to verify the *read* SRTD decompositionally was 3.23 seconds and this is a little more than the time taken for the single monolithic verification. Our verification numbers show that the decompositional checks consistently use less space while generally taking more time. Notwithstanding the Lichtenstein-Pnueli thesis [18], in practice, as one reaches the space limitations of symbolic model checking tools, efficiency with respect to space is of more importance. We observe that the decompositional check, with respect to BDD size and space, is not much larger than the size of the system itself. The monolithic verification is, however, significantly more expensive.

5.2 Lucent's PCI Synthesizable Core

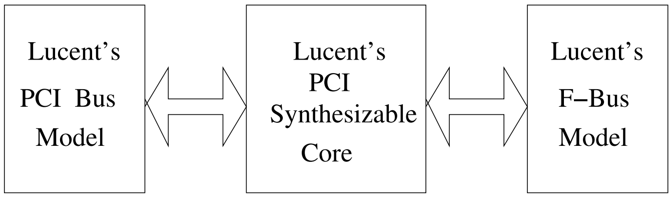

Fig. 3. Block Diagram of Lucent's F-Bus with PCI Core

The PCI Local Bus is a high performance, 32-bit or 64-bit bus with multiplexed data and address lines, which is now an industry standard. The PCI bus is used as an interconnect mechanism between processor/memory systems and peripheral controller components. Lucent Technologies' PCI Interface Synthesizable Core is a set of synthesizable building blocks that designers can use to to implement a complete PCI interface. The PCI Interface Synthesizable Core is designed to be fully compatible with the PCI Local Bus specification [23]. The Synthesizable Core bridges an industrial standard PCI bus to an F-Bus, which is 32-bit internal buffered FIFO bus that supports a Master-slave architecture with multiple masters and slaves.

We used Lucent's PCI Bus Functional Model shown in Figure 3, which is a sophisticated simulation environment that was developed to test the Synthesizable Core for functionality and compliance with the PCI specification [23]. The Functional Model consists of the PCI Core blocks and abstract models for both the PCI Bus and the F-Bus. The PCI Bus and F-Bus models were designed to fully exercise the PCI Synthesizable Core in both the slave and master modes. This model has about 1500 bounded state variables and was too large for model checking. We had to perform some abstractions, like freeing variables and removing variables from consideration for "cone of influence" reductions. These abstractions were property-specific and had to be modified for each property checked.

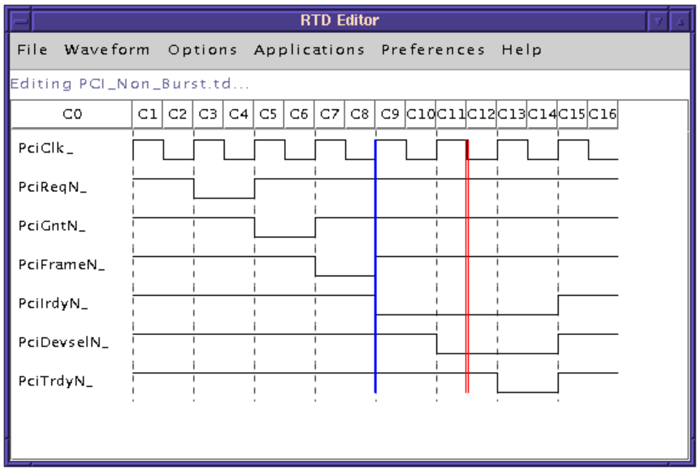

Fig. 4. SRTD for the Non Burst Transaction of the PCI Bus

The Synthesizable Core design is synchronous to the PCI clock. The basic bus transfer on the PCI is a burst, which is composed of an address phase followed by one or more data phases. In the non-burst mode, each address phase is followed by exactly one data phase. The data transfers in the PCI protocol are controlled by three signals *PciFrame*, *PciIrdy* and *PciTrdy*. The master of the bus drives the signal *PciFrame* to indicate the beginning and end of a transaction. *PciIrdy* is asserted by the master to indicate that it is ready to transfer data. Similarly the target uses *PciTrdy* to signal that it is ready for data transfer. Data is transferred between master and target on each clock edge for which both *PciIrdy* and *PciTrdy* are asserted. We verified that the PCI Core is PCI compliant by verifying several timing diagram properties for both the burst and non-burst modes. We formulated the properties as SRTD's by looking at the actual timing diagrams that occurred in the PCI specification [23] and the PCI Core User's Manual [4]. Figure 4 is one of the properties that we verified for the non burst mode.

The verification was done both monolithically and decompositionally and Table 2 presents the verifications statistics. In Table 2, the size, space and time numbers for properties with the suffix *(M)* correspond to the verification check on the abstracted PCI Core and the monolithic automaton for the property. The suffix *(C)* refers to the average over the individual decompositional verification checks on the abstracted system and the automata for each waveform. Table 2 shows a savings of up to 30% in BDD size and corresponding savings

Design	Number BDD variables	Average BDD size	Average Space (MBytes)	Average Time (seconds)
PCI Prop1 (M)	740	715157	36.2	411
PCI Prop1 (C)	664	417816	22.1	279
PCI Prop2 (M)	1036	688424	23.9	209
PCI Prop2 (C)	996	554866	19.1	182
PCI Prop3 (M)	749	3742074	198.6	16793
PCI Prop3 (C)	699	2680421	171.7	5677

Table 2. Verification Statistics for Lucent's Synthesizable PCI Core

in space. In practice, as one reaches the space bounds of a model checking tool, it may be beneficial to trade time for space. Our results demonstrate that our decompositional approach is more space efficient than a monolithic one.

6 Related Work and Conclusions

Various researchers have investigated the formal use of timing diagrams. *SACRES* [5, 7] is a verification environment for embedded systems that allows users to graphically specify properties as Symbolic Timing Diagrams (STDs) [10]. STDs cannot, however, express real-time properties or don't-cares; both are important in representing synchronous timing diagrams. Moreover, the translation algorithm is monolithic, and in general results in an exponential translation. Fisler [12] provides a procedure to decide regular language containment of non-regular timing diagrams, but the model checking algorithms have a high complexity (PSPACE). Cerny et al. present a procedure [16] for verifying whether the finite behavior of a set of action diagrams (timing diagrams) is consistent. Amon [3] uses Presburger formulas to determine whether the delays and guarantees of an implementation satisfy constraints specified as a timing diagram. This work uses Timing Designer[2] to specify the constraints and delays. They have developed tools that generate Presburger formulas corresponding to the timing diagrams and manipulate them. This model cannot, however, handle synchronous signals, and the algorithm for verifying Presburger formulas is multi-exponential in the worst case.

In contrast, for SRTD's, we have presented a decompositional, efficient algorithm for model checking, which has time complexity that is linear in the

[2] Timing Designer is a commercial timing diagram editor from Chronology Corporation.

size of the system model and quadratic in the size of SRTD. Our experience with verifying the PCI core and other protocols indicates that the syntax of SRTD's suffices to express common timing properties, and is expressive enough for industrial verification needs.

We have also developed a tool, RTDT, which implements the translation from SRTD's to ω-automata and have used it to verify several timing specifications from the PCI specification for Lucent's Synthesizable Core. The output of RTDT is currently targeted to the COSPAN/FormalCheck tool, but it can be easily re-targeted to produce output suitable for other model checkers (e.g. SMV, VIS). In addition to editing and translating SRTD's, the tool allows the user to view error traces and convert these into timing properties. There are other timing diagram editors [19, 13, 15, 22] which employ the timing specifications for test generation, simulation or synthesis but they do not, to the best of our knowledge, have a formal verification capability.

Acknowledgments: We would like to thank Vanya Amla, Pankaj Chauhan and Phoebe Weidmann for helpful discussions.

References

1. N. Amla and E.A. Emerson. Regular Timing Diagrams. In *LICS Workshop on Logic and Diagrammatic Information*, June 1998.
2. N. Amla, E.A. Emerson, and K.S. Namjoshi. Efficient Decompositional Model Checking for Regular Timing Diagrams. In *CHARME*. Springer-Verlag, September 1999.
3. T. Amon, G. Borriello, T. Hu, and J. Liu. Symbolic Timing Verification of Timing Diagrams Using Presburger Formulas. In *DAC*, 1997.
4. Bell Laboratories, Lucent Technologies. PCI Core User's Manual (Version 1.0). Technical report, July 1996.
5. A. Benveniste. Safety Critical Embedded Systems Design: the SACRES approach. Technical report, INRIA, May 1998. URL:http://www.tni.fr/sacres/index.html.
6. R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Shiple, G. Swamy, and T. Villa. VIS. In *FMCAD*, 1996.
7. U. Brockmeyer and G. Wittich. Tamagotchis need not die-Verification of STATE-MATE Designs. In *TACAS*. Springer-Verlag, March 1998.
8. E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Workshop on Logics of Programs*, volume 131. Springer Verlag, 1981.
9. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic. *ACM Transactions on Programming Languages and Systems*, 8(2), 1986.
10. W. Damm, B. Josko, and Rainer Schlör. Specification and Verification of VHDL-based System-level Hardware Designs. In Egon Borger, editor, *Specification and Validation Methods*. Oxford University Press, 1994.
11. K. Fisler. *A Unified Approach to Hardware Verification Through a Heterogeneous Logic of Design Diagrams*. PhD thesis, Computer Science Department, Indiana University, August 1996.

12. K. Fisler. Containment of Regular Languages in Non-Regular Timing Diagrams Languages is Decidable. In *CAV*. Springer Verlag, 1997.
13. W. Grass, C. Grobe, S. Lenk, W. Tiedemann, C.D. Kloos, A. Marin, and T. Robles. Transformation of Timing Diagram Specifications into VHDL Code. In *Conference on Hardware Description Languages*, 1995.
14. R.H. Hardin, Z. Har'El, and R.P. Kurshan. COSPAN. In *CAV*, volume 1102, 1996.
15. K. Kastein and M. McClure. Timing Designer use for Interface Verification at Symbios Logic. *Integrated System Design*, May 1997. URL:http://-www.chronology.com.
16. K. Khordoc and E. Cerny. Semantics and Verification of Timing Diagrams with Linear Timing Constraints. *ACM Transactions on Design Automation of Electronic Systems*, 3(1), 1998.
17. R.P. Kurshan. *Computer-aided verification of coordinating processes: the Automata-theoretic approach.* Princeton University Press, 1994.
18. O. Lichtenstein and A. Pnueli. Checking that Finite State Concurrent Programs satisfy their Linear Specifications. In *POPL*, 1985.
19. K. Luth. The ICOS Synthesis Environment. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, 1998.
20. Z. Manna and A. Pnueli. Specification and Verification of Concurrent Programs by ∀-Automata. In *POPL*, 1987.
21. K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Press, 1993.
22. D. Mitchell. Test Bench Generation from Timing Diagrams. In David Pellerin, editor, *VHDL Made Easy.* 1996. URL:http://www.syncad.com.
23. PCI Special Interest Group. PCI Local Bus Specification Rev 2.1. Technical report, June 1995.
24. J.P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proc. of the 5th International Symposium on Programming*, volume 137 of *LNCS*, 1982.
25. M. Vardi. Verification of Concurrent Programs. In *POPL*, 1987.
26. M. Vardi and P. Wolper. An Automata-theoretic Approach to Automatic Program Verification. In *LICS*, 1986.