Efficient Decompositional Model-Checking for Regular Timing Diagrams*

Nina Amla†
Dept. of Computer Sciences
University of Texas at Austin

E. Allen Emerson‡
Dept. of Computer Sciences
University of Texas at Austin

Kedar S. Namjoshi§
Bell Laboratories
Lucent Technologies

Abstract

There is a growing need to make verification tools easier to use. A solution that does not require redesigning the tool is to construct front-ends providing specification notations that are close to those used in practice. Timing diagrams are such a widely used graphical notation, one that is often more appealing than a "linear" textual notation. This paper introduces a class of timing diagrams called *Regular Timing Diagrams (RTDs)*. RTDs have a precise syntax and a formal semantics that is simple and corresponds to common usage. In addition, RTDs have an inherent compositional structure, which is exploited to provide an efficient algorithm for model-checking an RTD with respect to a system description. The algorithm has time complexity that is a small polynomial in the size of the diagram and linear in the size of the structure. We demonstrate the applicability of our algorithms by verifying that a master-slave system satisfies its specification RTDs.

1 Introduction

The design of hardware systems includes the specification of timing behavior for circuit components. In industrial practice, this behavior is most often described graphically as timing diagrams. Timing diagrams are, however, often used informally and ambiguously – making it difficult to use them for specification and verification of correct behavior. This paper addresses the growing interest in formalizing timing diagrams to permit their incorporation into automated verification by providing a precise syntax and a formal semantics that is simple and corresponds to common usage.

We introduce a class of timing diagrams called *Regular Timing Diagrams* (RTDs). These diagrams describe, over a finite time period, changes of signal values, and precedence and timing dependencies between such events, such as "signal a rises within 5 time units of signal b falling" and "signal b is low when signal a rises". The time intervals are specified by constants, ensuring that the diagram defines a *regular* language. A RTD, like the circuit it describes, may be either *asynchronous* or *synchronous*. A *synchronous* diagram includes one or more "clocks" with fixed periods and ensures that the time interval between any pair of events is determined up to the clock period. The ordering between events is partial in general; such RTDs are called *ambiguous*. An unambiguous RTD has a total ordering on events. Since an RTD is defined for a finite time period, an important question that arises in defining the semantics is "how does an infinite computation

*This work was supported in part by NSF grants CCR 941-5496, CCR 980-4736 and SRC Contract 97-DP-388.

†E-mail: namla@cs.utexas.edu URL: http://www.cs.utexas.edu/users/namla
‡E-mail: emerson@cs.utexas.edu URL: http://www.cs.utexas.edu/users/emerson
§E-mail: kedar@research.bell-labs.com URL: http://cm.bell-labs.com/cm/cs/who/kedar

satisfy a timing diagram ?" Fisler [13] defines two kinds of semantics: in the *invariant* semantics, the timing diagram must be satisfied at *every* state of a computation, while in the *basic iterative* semantics, the diagram must be satisfied iteratively. We introduce a generalization of the iterative semantics, in which a timing diagram must be satisfied at only those points of the computation that satisfy a precondition of the diagram. This permits a system to satisfy diagrams that express the correctness of different aspects of its operation. For ambiguous diagrams, we further classify these semantics into a *weak* aspect, where a fresh linear ordering of the events is chosen for each satisfaction of the diagram, and a *strong* aspect, where a single linear order is chosen that applies to each satisfaction of the diagram.

The key observation that leads to efficient Model Checking [5, 18, 6] is that timing diagrams are compositional (conjunctive) in nature. This can be visualized informally as the waveforms acting independently and only interacting with other waveforms through a dependency. Rather than building a *monolithic NFA* or a temporal logic formula corresponding to a timing diagram, we demonstrate that it is possible to decompose a timing diagram into properties of isolated waveforms and their interactions. This results in a conjunction of simpler properties that can be conveniently represented by a succinct $\forall$-automaton. Furthermore, the conjunctivity can be exploited to verify smaller components of the timing diagram in isolation, avoiding the construction of the entire $\forall$-automaton. A $\forall FA$ [17, 24] is a finite state automaton that accepts an input iff *every* run of the automaton along the input meets the acceptance criterion. $\forall FA$'s can be exponentially more succinct than *NFA*'s (non-deterministic finite state automata) and naturally express properties that are conjunctive in nature. We present efficient algorithms that convert RTDs under the various semantics into $\forall FA$ that are in the worst case of size cubic in the size of the diagram.

The use of $\forall FA$ permits the efficient use of the automata-theoretic language containment paradigm [25] to model checking. For the system M and RTD T, the verification check can be cast as $\mathcal{L}(M) \subseteq \mathcal{L}(\mathcal{A}_T)$, where $\mathcal{A}_T$ is the (small, polynomial size) $\forall FA$ for the diagram T. This is equivalent to $L(M) \cap \neg\mathcal{L}(\mathcal{A}_T) = \emptyset$. The complement language of a $\forall FA$ is accepted by an *NFA* with identical structure but complemented acceptance condition. Hence, complementation (the $\neg\mathcal{L}(\mathcal{A}_T)$ term) is trivial, and the complexity of the model checking procedure is linear in the size of the structure and the size of the $\forall FA$ $\mathcal{A}_T$. In addition, it is often possible to decompose $\mathcal{A}_T$ itself into a conjunction of smaller $\forall$ automata, which may be checked independently with M. It is also easy to produce a "symbolic" description of $\neg\mathcal{L}(\mathcal{A}_T)$ which may be input to a symbolic model-checker. To demonstrate the applicability of this method, we verified that a master-slave memory system satisfies its RTD behavior using the symbolic-model checker VIS [3]. In studying the timing diagrams used in industry, we are led to believe that RTDs are sufficiently expressive for most industrial verification needs.

The rest of the paper proceeds as follows. In Section 2, we give a precise syntax and semantics for Regular Timing Diagrams. Section 3 outlines the algorithms that convert RTDs into $\forall FA$ and the model-checking procedure. Section 4 describes how the algorithms are used with VIS for the verification of a master-slave system. Finally, we state our conclusions and discuss related work in Section 5.

2 Regular Timing Diagrams - Syntax and Semantics

A Regular Timing Diagram (RTD) is specified by a number of waveforms over a set of "symbolic" values SV and timed dependencies between points on the waveforms. The set of symbolic values includes 1 (High), 0 (Low) and X (unspecified). The set SV is ordered by $\sqsubseteq$, where $a \sqsubseteq b$ iff $a = b$ or $a = X$.

2.1 Regular Timing Diagrams: Syntax

Definition 1 (RTD) *A* RTD *is a tuple* (WF, SD, CD)*, where*

- *WF is a finite set of waveforms over SV. A waveform A of length n is a function $A : [0, n) \to SV$. The ith point on A is denoted by (A, i). Informally, $A(i)$ represents the "signal level" in the time interval between points (A, i) and $(A, i + 1)$.*
- *SD is the set of* sequential *dependencies on the points of WF. Each dependency is specified as $(A, i) \stackrel{[a,b\rangle}{\to} (B, j)$, where $a \in \mathbf{N}, b \in \mathbf{N} \cup \{\infty\}$, $1 \leq a$ and $a \leq b$. (The $\rangle$ bracket indicates either a closed or an open interval)*
- *CD is a collection of mutually disjoint sets of points, called* concurrent dependencies. *The set of initial and final points of the diagram form predefined concurrent dependencies.*

The syntax given above allows several definitions that run counter to intuition. For instance, dependencies may be cyclically related, or it may be possible that the precise location of a dependency is ambiguous due to the presence of X (undetermined) parts of a waveform. These are ruled out by giving a notion of "well-formed" RTDs, which is defined below.

Definition 2 (Event) *The* events *of an RTD (WF, SD, CD) are defined inductively as follows, where the alternatives are applied in the order shown.*

1. *For every waveform A in WF, $(A, 0)$ is an event.*
2. *For an event (A, i) with non-X value, a change along waveform A to a non-X successor value $A(j)$ defines (A, j) as an event.*
3. *If (A, i) is a member of a concurrent dependency that contains an event, then (A, i) is an event.*
4. *If (A, i) is an event and $(A, i) \stackrel{=k}{\to} (B, j)$, then (B, j) is an event.*

Notice that for any input string of vectors of signal values, every event has at most one position on the string. This "precise location" property of events is the key to our efficient model checking algorithm. For every event e, it is possible to construct a small *DFA* we call $locator(e)$ that accepts at the position on an input string where the event holds. This *DFA* essentially encodes the sequence of applications of the rules above that define the point e as an event.

A *symbolic point* of an RTD is either a concurrent dependency or a singleton containing a point that is not in any concurrent dependency. Informally, events in a symbolic point should occur simultaneously. The sequential dependencies of an RTD induce the following ordering relation $\prec$ on symbolic points : $p \prec q$ iff

- $(A, i) \in p$ and $(A, i + 1) \in q$, for points $i, i + 1$ of some waveform A in WF, or
- There exist $e \in p$ and $f \in q$ such that $e \stackrel{\alpha}{\to} f$ is a sequential dependency.

Definition 3 (Well-formed RTD) *A RTD is* well-formed *iff the transitive closure of $\prec$, $\prec^+$, is not reflexive and every point of the RTD is an event.*

2.2 Regular Timing Diagrams: Semantics

The semantics of an RTD is a set of infinite computations over *words*; each word is a vector with a value for each waveform of the timing diagram. The set of words is denoted by Σ. The operator $\sqsubseteq$ defined earlier is extended to words as follows : $u \sqsubseteq w$ iff for each i, $u(i) \sqsubseteq w(i)$. A computation of the system to be verified consists of an infinite sequence of words from Σ. Since the syntax of a RTD describes only finite sequences of such event-sets, a key question is the appropriate extension to infinite computations.

The predefined initial and final concurrent dependencies may be thought of as "begin" and "end" markers of the finite sequence of events described by the RTD syntax (for instance, a "memory-read" transaction). One may thus consider an infinite sequence to satisfy a timing diagram iff whenever the initial condition holds, the dependencies of the diagram are satisfied before the final condition. This statement, though, is still open to many interpretations, some of which are defined below. We first define what it means for a timing diagram to satisfy a finite sequence of words.

Definition 4 (Assignment) *An* assignment π *is a function* $\pi : SP \rightarrow [0, n)$*, for some* n*, that is monotonic w.r.t.* $\prec$ *(*$p \prec q$ *implies* $\pi(p) < \pi(q)$*) and maps the initial point of* SP *to* 0*.*

Two assignments $\pi : SP \rightarrow [0, n)$ and $\xi : SP \rightarrow [0, m)$ are *equivalent* iff they order symbolic points identically w.r.t. $<$ and $=$. Any assignment π induces the function $\hat{\pi}$ which maps a point (A, i) to k iff the (unique, by definition) symbolic point that includes (A, i) is mapped to k by π. From the definition, it follows that all points in a concurrent dependency are assigned a common position.

Definition 5 (RTD satisfaction) *A RTD* $T = (WF, SD, CD)$ *is satisfied by a finite sequence* y *over* Σ^+ *w.r.t. an assignment* $\pi : SP \rightarrow [0, |y|)$ *(written as* $y \models_\pi T$*) iff the following conditions hold.*

- *Point consistency: For every point* (A, i)*, if* $\hat{\pi}((A, i)) = k$*, then* $A(i) \sqsubseteq y_k(A)$*.*
- *Waveform consistency: Let* $\hat{\pi}((A, i)) = k$ *and* $\hat{\pi}((A, i + 1)) = l$*. For every* $j \in [k, l)$*,* $A(i) \sqsubseteq y_j(A)$*.*
- *Dependency consistency: For every sequential dependency* $e \overset{[a,b\rangle}{\rightarrow} f$*,* $(\hat{\pi}(f) - \hat{\pi}(e)) \in [a, b\rangle$*.*

For many systems, it is the case that the initial condition for the timing diagram does not recur before the final condition holds. For such systems, we may consider the following semantics. System computations may be described by the expression $(\Delta^+ \vee (\#\Delta^+\$))^\omega$, where # and \$ are special vectors of Σ representing the begin- and end- markers respectively, and $\Delta = \Sigma \backslash \{\#, \$\}$. The sequence of the form $\#\Delta^+\$$ is called a *transaction*.

Definition 6 (Weak Iterative Semantics) *An infinite sequence* z *satisfies a RTD* T *under the* weak iterative semantics *(written as* $z \models_w T$*) iff for every transaction* $\#y\$$ *on* z*, there exists an assignment* π *such that* $\#y\$ \models_\pi T$*.*

Definition 7 (Strong Iterative Semantics) *An infinite sequence* z *satisfies a RTD* T *under the* strong iterative semantics *(written as* $z \models_s T$*) iff there exists an assignment* ξ *such that for every transaction* $\#y\$$ *of* z*, there is an equivalent assignment* π *such that* $\#y\$ \models_\pi T$*.*

A notable class of systems where the assumption of non-overlapping transactions does not hold is those that involve some measure of pipelining. We may then consider the following generalization of the weak iterative semantics.

Definition 8 (Overlapping Semantics) *An infinite sequence z satisfies a RTD T under the* overlapping semantics *(written as $z \models_o T$) iff whereever # holds along z, there exists y such that $\#y\$$ is a prefix of the suffix computation from that point and for some assignment π, $\#y\$ \models_\pi T$.*

For the rest of the paper, we consider only the weak and strong iterative semantics in detail; the algorithm for the overlapping semantics is a slight modification of that for the weak iterative semantics and has the same complexity. We consider now an alternative formulation of Definition 5, which forms the basis for the "decompositional" algorithms for model checking. If $\#y\$$ satisfies the timing diagram, each event, by Definition 2 may be located precisely on the sequence. The key observation is that, since each dependency consists of precisely located events, it can be checked independently of the others. Let pt be the partial function that defines the location of events on a finite sequence.

Theorem 1 *For a RTD $T = (WF, SD, CD)$, and any finite transaction $z = \#y\$$, there exists an assignment π such that $z \models_\pi T$ iff each of the following conditions holds:*

- *Every event of T can be located on z and has a value consistent with that in T; i.e., pt is total, and if $pt(z, (A, i)) = k$ then $A(i) \sqsubseteq z_k(A)$.*
- *Let $pt(z, (A, i)) = k$ and $pt(z, (A, i+1)) = l$. For every j in $[k, l)$, $A(i) \sqsubseteq z_j(A)$.*
- *For each sequential dependency $e \overset{range}{\rightarrow} f$, $(pt(z, f) - pt(z, e)) \in range$.*
- *For each pair of events e, f in a concurrent dependency, $pt(z, e) = pt(z, f)$.*

Notice that the theorem essentially transforms the existential ($\exists$) condition of Definition 5 into a universal ($\forall$) condition; this forms the basis for the "decompositional" check.

3 Decompositional Algorithms

Theorem 1 is fundamental to decomposing RTDs into a conjunction of properties of individual waveforms and ordering/timing restrictions on their interactions, which is the key to efficient model-checking. In this section, we provide algorithms that translate RTDs under both strong and weak iterative semantics into $\forall FA$. The basic iterative and overlapping semantics can be handled similarly. For clarity, we often describe the *NFA* for the complement language instead of the $\forall FA$.

Definition 9 ($\forall FA$ (Dual Run Automata)) *A $\forall FA$ on infinite strings $\mathcal{A} = (\Sigma, Q, \delta, q_0, \Phi)$ is comprised of a finite input alphabet Σ, a finite state set Q, a transition relation $\delta \subseteq Q \times \Sigma \times Q$, a start state q_0 and an acceptance condition Φ.*

A *run* r of $\mathcal{A}$ on input x in Σ^ω is an infinite sequence of states of $\mathcal{A}$, where r_0 is an initial state, and for each i, $(r_i, x_i, r_{i+1}) \in \delta$. $\mathcal{A}$ accepts x by "dual-run" acceptance according to Φ iff *every* run r on x satisfies Φ. For any $\forall FA$ $\mathcal{A}$, let $\overline{\mathcal{A}}$ be the *NFA* with the same transition relation but complemented acceptance condition $\neg\Phi$.

Theorem 2 *([17, 24]) For any $\forall FA$ $\mathcal{A}$, $\neg\mathcal{L}_{\forall FA}(\mathcal{A}) = \mathcal{L}_{NFA}(\overline{\mathcal{A}})$.*

3.1 RTDs under the weak iterative semantics

We describe here the *NFA* that accepts the complement of the weak-iterative language of an RTD $T = (WF, SD, CD)$. First, generate finite string automata for each waveform and dependency as follows:

- Waveform : If $(A, i+1)$ is defined in terms of (A, i), then $locator((A, i))$ is extended to ensure that the signal values up to the change of value that defines $(A, i+1)$ are above $A(i)$ in $\sqsubseteq$ order. Otherwise, $locator((A, i))$ is used to determine that the value at the position where (A, i) holds is above $A(i)$ in $\sqsubseteq$ order.
- Sequential dependency : For a sequential dependency $e \xrightarrow{\alpha} f$, the automaton is a parallel composition of $locator(e)$ and $locator(f)$ that accepts iff the time between the acceptance of these *DFA*'s is within α.
- Concurrent dependency : The $\forall FA$ for a concurrent dependency C checks that for a fixed event e in C and every other event f in C, $locator(e)$ and $locator(f)$ accept at the same position on the input sequence.

The ω-*NFA* for the complement language operates as follows on an infinite input sequence : it nondeterministically "chooses" a transaction $\#y\$$ on the input, "chooses" which waveform or dependency fails to hold of the transaction, and accepts if the automaton for that entity (defined as given above) rejects. Notice that the automata defined above are either *DFA* or $\forall FA$, both of which can be trivially complemented. The $\forall FA$ obtained from this *NFA* by complementing the acceptance condition defines the language of the RTD under the weak iterative semantics. Denote this $\forall FA$ by $\mathcal{A}_T$. For the diagram $T = (WF, SD, CD)$, let L be the size in unary of the largest constant in SD. Define $|T| = \#points + |SD| + |CD|$. The size of $\mathcal{A}_T$ is cubic in $|T|$ and L.

Theorem 3 (Correctness) *For any RTD T, and $x \in \Sigma^\omega$, $x \models_w T$ iff $x \in L(\mathcal{A}_T)$. The size of $\mathcal{A}_T$ is polynomial in $|T|$ and the unary length of the largest constant in T.*

3.2 RTDs under the strong iterative semantics

Under the strong iterative semantics, every transaction on an input computation has to satisfy the RTD w.r.t. a single event ordering. The *NFA* for the complemented language accepts a computation iff

- Some transaction violates a waveform or dependency constraint. This is checked by the automaton defined for the weak-iterative semantics. Or,
- There is a transaction and a pair of events that occur in a different order from that in the first transaction. This is done by an automaton that "chooses" a pair of events unordered by $\prec^+$, executes the locator dfa's for these events in parallel on the first transaction to determine their order, then "chooses" a subsequent transaction and executes the locator dfa's for the same events on that transaction to determine the new order, and accepts if the orders differ.

Let $\mathcal{A}_T$ denote the $\forall FA$ obtained from this *NFA* by complementing the acceptance condition. The size of $\mathcal{A}_T$ is cubic in $|T|$ and L for the first case; for the second, it is quadratic in $|T|$ and L with a multiplicative factor of the number of concurrent event pairs.

Theorem 4 (Correctness) *For any RTD T and $x \in \Sigma^\omega$, $x \models_s T$ iff $x \in L(\mathcal{A}_T)$. The size of the $\forall FA$ $\mathcal{A}_T$ is polynomial in $|T|$ and L.*

3.3 Model Checking

The translation of an RTD to a small $\forall FA$ implies that the language containment approach to model checking based on [25] gives an efficient algorithm. We need to check that $\mathcal{L}(S) \subseteq \mathcal{L}(\mathcal{A}_T)$, where S is the system to be verified and $\mathcal{A}_T$ is the $\forall FA$ for the RTD T. This is equivalent to $\mathcal{L}(S) \cap \neg\mathcal{L}(\mathcal{A}_T) = \emptyset$. Complementation (the $\neg\mathcal{L}(\mathcal{A}_T)$ term) is trivial for $\forall FA$; the complemented automaton (an *NFA*) has the same structure but complemented acceptance condition. Hence, the emptiness check can be done in time linear in the size of the structure and a small polynomial in the size of T. The space complexity, by the results of [21], is logarithmic in the sizes of both S and T.

Theorem 5 *For a transition system S and a RTD T, the time complexity of model checking is linear in the size of S and a small polynomial in the size of T and the unary size of the largest constant in T.*

An alternative way of utilizing the $\forall FA$ construction is to note that, for the weak iterative semantics, the automaton essentially defines a language $(\Delta^+ \vee \# \bigwedge_i (L_i)\$)^\omega$, where the L_i's represent the languages of the dependencies. The lemma below shows that the ω-repetition distributes over the $\bigwedge_i$ in the following sense.

Lemma 1 *For finite-string languages L_i $(i \in [0, n))$ which are subsets of Δ^+,*
$(\Delta^+ \vee \# \bigwedge_i (L_i)\$)^\omega = \bigwedge_i (\Delta^+ \vee \# L_i\$)^\omega$.

By this lemma, one can construct smaller ω-automata, one for each dependency, and check that the language of each has an empty intersection with $\mathcal{L}(S)$. This is often more efficient than the combined check, and may lead to quicker detection of any errors. We refer to this as the "decompositional" approach.

4 Applications

We demonstrate the use of these algorithms in the verification of a master-slave memory system using the model-checker VIS, which is based on the automata-theoretic language containment approach to model checking. The decompositional approach and weak iterative semantics was used.

In the master-slave system (Figure 1), the master issues a read or a write instruction by asserting the corresponding line, and the slaves respond by accessing memory and performing the operation. The master chooses the instruction, puts the address on the address bus and then 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.

The master-slave system was simplified by abstracting away some inessential details. First, the address bus was simplified to the tag of the slaves. Since VIS does not allow variables to be both input and output, the bidirectional data bus is represented as two 1 bit boolean variables that denote the input and output data buses. The simplified master-slave system was designed in Verilog. For both the read (Figure 2(a)) and write (Figure 2(b)) cycle RTDs, we created (as Verilog modules) both the complement of the $\forall FA$ and the complement *NFA* for each dependency and waveform.

The language emptiness check passed for both the ambiguous read and write RTD translations. The master-slave system has 61 BDD variables and 109 reachable states represented with 275 BDD nodes. The product of the master-slave system all the modules for the read RTD has 169 BDD variables and a reachable state space of 17796 states represented with 2019 BDD nodes. The product of the master-slave system with the module for a single waveform has 70 BDD variables and a reachable state space of 110 states represented

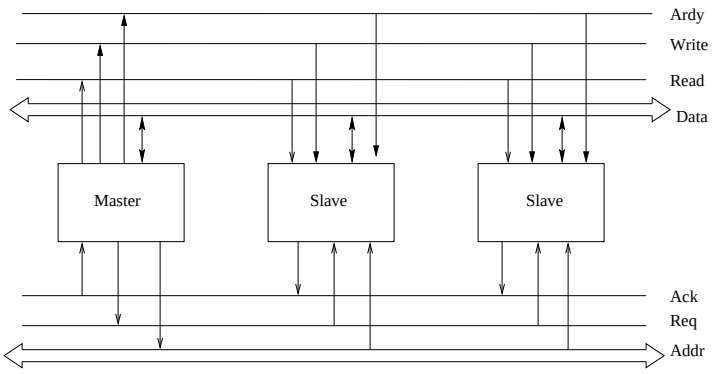

Figure 1: Master-Slave Architecture

with 330 BDD nodes. The product of the master-slave system with the module for a single dependency has 70 BDD variables and a reachable state space of 390 states represented with 532 BDD nodes. These results show that the decompositional procedure is indeed feasible and that the size of the system to be verified together with a single dependency automaton may not be significantly larger, in terms of BDD variables, than the system itself. Since the system under verification was small, the multiplicative factor of the RTD automata is more visible.

5 Conclusions and Related Work

Several researchers have investigated timing diagrams and their use in automated verification. Boriello [2] proposes an approach to formalizing timing diagrams. Timing diagrams are described informally as regular expressions but no specific details or translation algorithms are given. Many other researchers [1, 22, 19, 4] have formalized timing diagrams and translated them to other formalisms (interval logics, trigger graphs etc). Formal notions of timing diagrams are also proving to be useful in test generation and logic synthesis (cf. [23, 15, 12]).

Fisler [13, 14] proposes a timing diagram syntax and semantics that allows non-regular languages, and finds that these languages occur at all levels of the Chomsky hierarchy. The paper [14] provides a decision procedure that determines whether a regular language is contained in an unambiguous timing diagram language, and [13] provides an algorithm that translates a certain class of timing diagrams into CTL [5]. A key difference with our work is that while this algorithm is restricted to a subset of unambiguous timing diagrams under the basic invariant semantics, our algorithms are defined for all types of diagrams under both iterative and invariant semantics. The regular containment procedure [14] has a high complexity (in PSPACE), while our algorithms have *polynomial* time complexity in the diagram size.

A signal contribution in this area is the work done by Damm and his colleagues at the University of Oldenburg on Symbolic Timing Diagrams (STD) [9, 20, 8, 16, 7]. STDs may be compiled into first-order temporal logic formulae which are then used for model checking. STDs are extended in [11, 10] to RTSTD's (Real-time STDs), where a translation into a timed propositional temporal logic TPTL is provided. Both these research efforts consider infinite languages and ambiguity. The main difference with our work lies in the fact that their translation is monolithic, in the sense that all dependencies are considered together; there may be an exponential blowup in the size of the resulting formulae when the diagram is highly ambiguous.

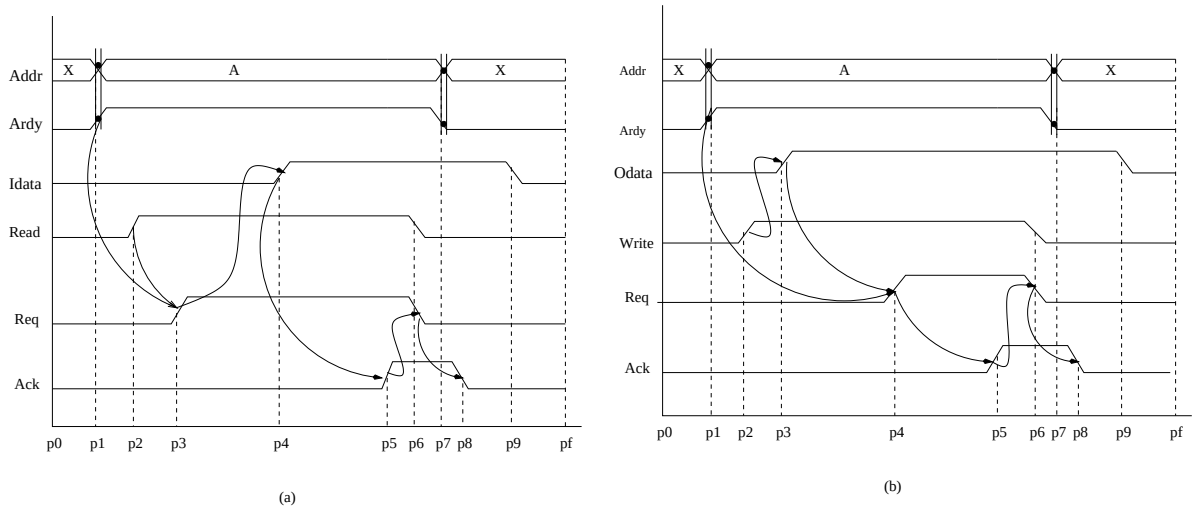

Figure 2: Ambiguous RTDs for (a) Read Cycle (b) Write Cycle

While it is possible to model-check the first order temporal logic presented in [9, 10], the procedure is not very efficient.

This paper presents "regular" timing diagrams (RTDs), which have a simple syntax and precise, simple semantics that closely correspond with common usage. The key contribution is polynomial time, decompositional algorithms for model checking such timing diagram specifications, which are based on a decomposition of the RTD semantics into properties of each waveform and the way they interact. This decomposition may also provide a way of composing RTDs and thereby building new RTDs hierarchically. Our algorithms generate $\forall FA$ (NFA) corresponding to the RTD (the negation of the RTD). We can choose to use either the $\forall FA$ (by splitting it into smaller $\forall FA$'s) or its complement NFA in verifying that a system satisfies an RTD.

We have shown how our algorithms may be used in conjunction with a model-checker such as VIS, to verify systems with specifications formulated as RTDs. We are currently working on a tool that implements these translation and verification algorithms. As mentioned earlier, the algorithms proposed in this paper can also be used with synchronous RTDs. In general, synchronous RTDs have less ambiguity and more structure; thus, it is possible that more efficient algorithms can be devised for them. Other plans for future work include possible extensions to timing diagrams to allow for finite repetition and ambiguity across clock cycles, overlapping semantics for pipeline systems, and other features sometimes needed in practice.

Acknowledgements We would like to thank Bob Kurshan and Steven Keckler for helpful discussions and insightful comments.

References

[1] C. Antoine and B. Le Goff. Timing Diagrams for Writing and Checking Logical and Behavioral Properties of Integrated Systems. In *Correct Hardware Design Methodologies*. Elsevier Sciences Publishers, 1992.

[2] G. Borriello. Formalized Timing Diagrams. In *EDAC92*. IEEE Comput. Soc. Press, 1992.

[3] 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.

[4] V. Cingel. A Graph-based Method for Timing Diagrams Representation and Verification. In *Correct Hardware Design and Verification Methods*. Springer Verlag, 1993.

[5] 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.

[6] 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.

[7] W. Damm and J. Helbig. Linking Visual Formalisms: A Compositional proof System for Statecharts cased on Symbolic Timing Diagrams. In E. R. Olderog, editor, *Programming Concepts, Methods and Calculi*. Elsevier Science B.V. (North Holland), 1994.

[8] W. Damm, H. Hunger, P. Kelb, and R. Schlör. Using Graphical Specification Languages and Symbolic Model Checking in the Verification of a Production Cell. In C. Lewerentz and T. Lindner, editors, *Formal Development of Reactive Systems. Case Study Production Cell, LNCS 891*. Springer Verlag, 1994.

[9] 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.

[10] K. Feyerabend. Real-time Symbolic Timing Diagrams. Technical report, Department of Computer Science, Oldenburg University, September 1994.

[11] K. Feyerabend and B. Josko. A Visual Formalism for Real Time Requirement Specifications. In *AMAST Workshop on Real-time systems and Concurrent and Distributed Software*. Springer Verlag, 1997.

[12] K. Feyerabend and R. Schlör. Hardware synthesis from requirement specifications. In *EURO-DAC'96 with EURO-VHDL'96*. IEEE Computer Society Press, September 1996.

[13] K. Fisler. *A Unified Approach to Hardware Verification Through a Heterogeneous Logic of Design Diagrams*. PhD thesis, Computer Science Department, Indiana University, August 1996.

[14] K. Fisler. Containment of Regular Languages in Non-Regular Timing Diagrams Languages is Decidable. In *CAV*. Springer Verlag, 1997.

[15] 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.

[16] J. Helbig, R. Schlör, W. Damm, G. Doehmen, and P. Kelb. VHDL/S - Integrating Statecharts, Timing diagrams, and VHDL. *Microprocessing and Microprogramming*, 1996.

[17] Z. Manna and A. Pnueli. Specification and Verification of Concurrent Programs by $\forall$ Automata. In *POPL*, 1987.

[18] 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.

[19] Y.S.. Ramakrishna, P.M. Melliar-Smith, L.E. Moser, L.K. Dillon, and G. Kutty. Really Visual Temporal Reasoning. In *Real-Time Systems Symposium*. IEEE Publishers, 1993.

[20] R. Schlör. A Prover for VHDL-based Hardware Design. In *Conference on Hardware Description Languages*, 1995.

[21] A.P. Sistla, M. Vardi, and P. Wolper. The Complementation Problem for Büchi Automata with Applications to Temporal Logic. *TCS*, 49, 1987.

[22] E.M. Thurner. Proving System Properties by means of Trigger-Graph and Petri Nets. In *EUROCAST*. Springer Verlag, 1996.

[23] W.D. Tiedemann. Bus Protocol Conversion: from Timing Diagrams to State Machines. In *EUROCAST*. Springer Verlag, 1992.

[24] M. Vardi. Verification of Concurrent Programs. In *POPL*, 1987.

[25] M. Vardi and P.Wolper. An Automata-Theoretic approach to automatic program verification. In *LICS*, 1986.