Assume-Guarantee Based Compositional Reasoning for Synchronous Timing Diagrams

Nina Amla[1], E. Allen Emerson[1], Kedar Namjoshi[2], and Richard Trefler[3]

[1] Department of Computer Sciences, University of Texas at Austin †
{namla,emerson}@cs.utexas.edu
[2] Bell Laboratories, Lucent Technologies
kedar@research.bell-labs.com
[3] AT&T Research
trefler@research.att.com

Abstract. The explosion in the number of states due to several interacting components limits the application of model checking in practice. Compositional reasoning ameliorates this problem by reducing reasoning about the entire system to reasoning about individual components. Such reasoning is often carried out in the assume-guarantee paradigm: each component guarantees certain properties based on assumptions about the other components. Naïve applications of this reasoning can be circular and, therefore, unsound. We present a new rule for assume-guarantee reasoning, which is sound and complete. We show how to apply it, in a fully automated manner, to properties specified as synchronous timing diagrams. We show that timing diagram properties have a natural decomposition into assume-guarantee pairs, and liveness restrictions that result in simple subgoals which can be checked efficiently. We have implemented our method in a timing diagram analysis tool, which carries out the compositional proof in a fully automated manner. Initial applications of this method have yielded promising results, showing substantial reductions in the space requirements for model checking.

1 Introduction

Compositional reasoning [7] – reducing reasoning about a system to reasoning about its components – has been an active area of research for nearly three decades. Recently, it has gained further importance as a way of ameliorating the state explosion problem in model checking. For example, given programs P_1, P_2 and specification T, we would like to check whether the composed system satisfies T (written as $P_1//P_2 \models T$). Since reasoning about $P_1//P_2$ directly only exacerbates the state explosion problem, compositional reasoning techniques are designed to reason about P_1 in isolation from P_2 (and vice versa) to draw conclusions about $P_1//P_2$. There are, however, several difficulties which must be overcome, foremost among them are the task decomposition problem, the generation of auxiliary assertions and the general applicability of the compositional method to the task at hand.

† Partially supported by NSF 980-4736, TARP 003658-0650-1999 and SRC 98-DP-388.

Firstly, *task decomposition* is necessary since it is unlikely that P_1 by itself satisfies all of T: we would like to decompose T into T_1 and T_2 such that $T = T_1 \wedge T_2$ and then show that $P_1 \models T_1$ and $P_2 \models T_2$. Secondly, auxiliary assertions are usually necessary, since P_1 may satisfy T_1 only when its environment behaves like P_2. To solve this problem, *assume-guarantee* style reasoning adds auxiliary assertions, Q_2 (respectively Q_1) which represent assumptions about the behavior of P_2 (P_1) as an environment for P_1 (P_2). Such auxiliary assertions must often be generated by hand, however. Finally, naïve compositional rules based on this style of reasoning, for instance, $P_1//P_2 \models T$ holds if $P_1//Q_2 \models T_1$ and $P_2//Q_1 \models T_2$, are sound only for safety properties.

In this paper, we first present a new rule for assume-guarantee reasoning, which generalizes several earlier rules (cf. [15, 1, 3, 12, 13]), by removing the sources of incompleteness in some of these rules, by using processes, instead of temporal logic formulas, as specifications, and by allowing more general forms of process definition and composition. The new rule extends the naïve rule above with a check for soundness. As it deals uniformly with processes, it fits in well with a top-down refinement approach to designing systems. We show that this rule is also complete, in that if $P_1//P_2 \models T$, then it is possible to prove this fact with our rule.

Next, we explore the benefits of applying this rule in the case where T is specified as a timing diagram. Timing diagrams are visual descriptions of process behavior that are widely used in the hardware industry. We show that not only is task decomposition a relatively simple problem for timing diagrams, but also that it is possible to automatically generate auxiliary assertions directly from the specification. Furthermore, we identify a large class of timing diagrams for which the soundness check of the rule is always satisfied, and the auxiliary assertion generation and, therefore, the model checking process is efficient – linear in the size of the diagram and the structure. We have implemented our method in a timing diagram analysis tool, RTDT [4], which uses the tool COSPAN [8] to discharge model checking subgoals. We report here on its application to a memory controller and a PCI Interface Core; in both cases, we obtain substantial reduction in the space used for model checking.

The organization of the paper is as follows: we describe our new rule and prove its soundness and completeness in Section 2. The theory behind the application of this rule to timing diagrams is presented in Section 3. Our experiments with applying this rule are described in Section 4. We conclude the paper with a description of related work in Section 5.

2 Assume-Guarantee Based Compositional Reasoning

In this section, we first present the naïve compositional reasoning rule and explain why it is unsound. We then present our new rule, and show that it is both sound and complete. We begin by defining some basic concepts: processes, composition, and closure. Although the eventual application of our rule is to finite state processes, we develop it in a more general setting.

2.1 Preliminaries

For a non-empty set of typed variables V, an assignment of values to variables in V is called a *V-state*. A *V-sequence* $x = x_0, x_1, \ldots$ is a non-empty sequence (finite or infinite) of V-states. The length of x (number of states in x) is written as $|x|$. We write $x[i..j]$, for $j \geq i$, to denote the subsequence $x_i, \ldots, x_j$ and $x;y$ to denote concatenation of a finite sequence x to y. A *language* L over V is a set of finite or infinite sequences of V-states. A W-sequence x, where $V \subseteq W$, *satisfies* L iff x projected on to V belongs to L. The term $(\exists W : L)$ defines a language over $V \backslash W$. A $(V \backslash W)$-sequence x satisfies $(\exists W : L)$ iff there exists a sequence y, with the same length as x, such that y is in L and x and y differ only on the values of variables in W. For a language L over V, let $[L]$ mean that every V-sequence (finite or infinite) satisfies L. Thus, for L_1 and L_2 over V, $[L_1 \Rightarrow L_2]$ denotes $L_1 \subseteq L_2$.

A *process* P is specified by a tuple (V, I, R, F). V is a non-empty set of typed *variables*, partitioned into three sets: *private* variables V^p, *interface* variables V^i, and *external* variables V^e. The variables V', which are in 1-1 correspondence with V, represent values for V in the next state. The set of *modifiable* variables, V^m, is $V^p \cup V^i$. $I(V^m)$ is an *initial condition*, $R(V, (V^m)')$ is a *transition relation* and $F(V)$ is a *fairness* condition. A V-sequence x is an *execution* of P iff $I(x_0)$ and for all i such that $i + 1 < |x|$, $R(x_i, x_{i+1})$ holds. The set of finite executions is denoted by *finexec*(P). The *language* of P, $\mathcal{L}(P)$, is the set of finite executions of P together with those infinite executions of P that satisfy F. The *observable language* of P, denoted by $\mathcal{L}^{\mathcal{O}}(P)$, is the projection of its language on $V^i \cup V^e$. In the rest of the paper, we assume that private variables of a process are distinct from the variables of all other processes, since this does not affect the observable language.

For processes P and A, the relationship "P *implements* A", denoted by $P \models A$, is defined only if $V^i(A) \subseteq V^i(P)$, and is defined as $[\mathcal{L}^{\mathcal{O}}(P) \Rightarrow \mathcal{L}^{\mathcal{O}}(A)]$, which can be written as $[\mathcal{L}(P) \Rightarrow (\exists V^p(A) : \mathcal{L}(A))]$. This matches the usual definition when A is an automaton, since a sequence over $V^p(A)$ is a run of the automaton.

For a language L on variables V, the *closure* of L, denoted by $cl(L)$, is a language consisting of V-sequences x where, for every $i < |x|$, there exists a sequence y such that $x[0..i]; y \in L$. For any process P, there is a process $CL(P)$ with the property $[\mathcal{L}^{\mathcal{O}}(CL(P)) \equiv cl(\mathcal{L}^{\mathcal{O}}(P))]$. If P is finite-state, $CL(P)$ is formed from P by changing the fairness condition of P to *true*.

A process Q *does not block* process P iff (i) any initial state of P can be extended to an initial state of $P//Q$, and (ii) for any reachable state of $P//Q$, any transition of P from that state can be extended to a joint transition of $P//Q$. A process is *machine closed* iff every finite execution can be extended to an infinite fair execution.

The *composition* of the processes $P_1 = (V_1, I_1, R_1, F_1)$ and $P_2 = (V_2, I_2, R_2, F_2)$, denoted by $P_1//P_2$, is the process $P = (V, I, R, F)$, where $V = V_1 \cup V_2$, $V^p = V_1^p \cup V_2^p$, $V^i = V_1^i \cup V_2^i$, $I = I_1 \wedge I_2$, $R = R_1 \wedge R_2$, and $F = F_1 \wedge F_2$. The *disjunction* of the processes P_1 and P_2, denoted by $P_1 + P_2$, is defined as the process $P = (V, I, R, F)$, where $V = V_1 \cup V_2 \cup \{c\}$, $V^p = V_1^p \cup V_2^p \cup \{c\}$,

$V^i = V_1^i \cup V_2^i$, $I = (c \wedge I_1) \vee (\neg c \wedge I_2)$, $R = (c' = c) \wedge ((c \wedge R_1) \vee (\neg c \wedge R_2))$, and $F = (\mathsf{FG}(c) \wedge F_1) \vee (\mathsf{FG}(\neg c) \wedge F_2)$. The private variable c serves to choose initially between the two processes. The following proposition summarizes the properties of these constructions needed for the later proofs.

Proposition 0. For processes P_1, P_2, P,
(a) $[\mathit{finexec}(P_1//P_2) \equiv \mathit{finexec}(P_1) \wedge \mathit{finexec}(P_2)]$,
$[\mathcal{L}(P_1//P_2) \equiv \mathcal{L}(P_1) \wedge \mathcal{L}(P_2)]$, and $[\mathcal{L}^{\mathcal{O}}(P_1//P_2) \equiv \mathcal{L}^{\mathcal{O}}(P_1) \wedge \mathcal{L}^{\mathcal{O}}(P_2)]$
(b) $[(\exists\{c\} : \mathcal{L}(P_1 + P_2)) \equiv \mathcal{L}(P_1) \vee \mathcal{L}(P_2)]$
(c) $[\mathcal{L}^{\mathcal{O}}(CL(P)) \equiv cl(\mathcal{L}^{\mathcal{O}}(P))]$

This definition of processes and of composition is quite general: it includes Moore and Mealy styles of definition as special cases, and processes in a composition can modify shared variables. Interleaving composition can be defined by adding a shared "turn" variable.

2.2 Compositional Reasoning Rules

To show that $P_1//P_2 \models T_1//T_2$ holds, one may attempt to show that $P_1 \models T_1$ and $P_2 \models T_2$. This "non-circular" proof often does not work if the components are tightly coupled, since P_1 may satisfy T_1 only in the presence of P_2. Hence, several so-called "circular" proof rules have been proposed, of which this is an example: to show $P_1//P_2 \models T_1//T_2$, show that (i) $P_1//T_2 \models T_1$, and (ii) $P_2//T_1 \models T_2$. This rule can be shown to be sound for non-blocking safety properties (i.e., for finite computations). It is, however, *unsound* for liveness properties. To see this, consider the following instantiation.

```
process P1: var x: boolean; initially x=true or x=false; transition x'=y
process P2: var y: boolean; initially y=true or y=false; transition y'=x
property T1: eventually(x) , property T2: eventually(y)
```

Although both hypotheses hold, it is not true that $P_1//P_2 \models T_1//T_2$, as the computation where x and y are always *false* is a valid computation of $P_1//P_2$. In an attempt to fix this problem, several proposed rules (cf. [1,3]) replace hypothesis (ii) with, say, $P_2//CL(T_1) \models T_2$. Using the safety closure of T_1 prevents any possibility of circular reasoning amongst liveness properties. On the other hand, this makes it difficult to apply the rule when liveness properties are needed as assumptions. We adopt a different strategy to fixing the problem: we use an additional hypothesis that checks if the circular reasoning is sound. For simplicity, we present this rule for the composition of two processes; it can be easily extended to apply to any finite composition.
Rule: To show that $P_1//P_2 \models T$, find Q_1 and Q_2 such that the following conditions are satisfied.

C0 $V^i(Q_1) \subseteq V^i(P_1)$, Q_1 does not block P_2, and symmetrically for Q_2.
C1 $P_1//Q_2 \models Q_1$, and $P_2//Q_1 \models Q_2$
C2 $Q_1//Q_2 \models T$
C3 Either $P_1//CL(T) \models (T + Q_1 + Q_2)$, or $P_2//CL(T) \models (T + Q_1 + Q_2)$

Note: Notice that hypothesis C3 need not be checked when T is a safety property, as $[\mathcal{L}^{\mathcal{O}}(CL(T)) \Rightarrow \mathcal{L}^{\mathcal{O}}(T)]$ holds in this case.

Theorem 0 (Soundness). The rule is sound for arbitrary P_1, P_2 and T.

Proof. We have to show that $P_1//P_2 \models T$ follows from the conditions C0-C3. This, by definition, is equivalent to showing that $[\mathcal{L}(P_1//P_2) \Rightarrow \mathcal{L}^{\mathcal{O}}(T)]$. By the results in [2], any language L can be can be written as a conjunction of the safety property $cl(L)$ and the liveness property $(cl(L) \Rightarrow L)$. Based on this characterization, we break up the proof into the following two parts.

Safety $[\mathcal{L}(P_1//P_2) \Rightarrow cl(\mathcal{L}^{\mathcal{O}}(T))]$, and
Liveness $[\mathcal{L}(P_1//P_2) \wedge cl(\mathcal{L}^{\mathcal{O}}(T)) \Rightarrow \mathcal{L}^{\mathcal{O}}(T)]$

In the following, let W be the private variables of $Q_1//Q_2$.

Lemma 0. $[\mathit{finexec}(P_1//P_2) \Rightarrow (\exists W : \mathit{finexec}(Q_1//Q_2))]$

Proof Sketch. This follows from conditions C0 and C1 by induction on the length of executions. □

First, we show the safety part by proving the equivalent (as $cl(\mathcal{L}(P))$ is the set of executions of P) statement $[\mathit{finexec}(P_1//P_2) \Rightarrow cl(\mathcal{L}^{\mathcal{O}}(T))]$. Let U be the private variables of T.

$$
\begin{array}{ll}
 & \mathit{finexec}(P_1//P_2) \\
\Rightarrow & \quad \text{(by Lemma 0)} \\
 & (\exists W : \mathit{finexec}(Q_1//Q_2)) \\
\Rightarrow & \quad \text{(as } cl(\mathcal{L}(P)) \text{ includes } \mathit{finexec}(P) \text{)} \\
 & (\exists W : cl(\mathcal{L}(Q_1//Q_2))) \\
\Rightarrow & \quad \text{(by C2; monotonicity of } cl \text{)} \\
 & (\exists W : cl(\mathcal{L}^{\mathcal{O}}(T))) \\
\Rightarrow & \quad \text{(} W \text{ contains private variables not occurring in } T \text{)} \\
 & cl(\mathcal{L}^{\mathcal{O}}(T))
\end{array}
$$

Next, we show the liveness part.

$$
\begin{array}{ll}
 & \mathcal{L}(P_1) \wedge \mathcal{L}(P_2) \wedge cl(\mathcal{L}^{\mathcal{O}}(T)) \\
\Rightarrow & \quad \text{(by Proposition 0(c))} \\
 & \mathcal{L}(P_1) \wedge \mathcal{L}(P_2) \wedge \mathcal{L}^{\mathcal{O}}(CL(T)) \\
\Rightarrow & \quad \text{(by condition C3)} \\
 & \mathcal{L}(P_1) \wedge \mathcal{L}(P_2) \wedge \mathcal{L}^{\mathcal{O}}(T + Q_1 + Q_2) \\
\Rightarrow & \quad \text{(by Proposition 0(b); } W \cup U \cup \{c\} \text{ consists of private variables)} \\
 & (\exists W \cup U \cup \{c\} : \mathcal{L}(P_1) \wedge \mathcal{L}(P_2) \wedge (\mathcal{L}(T) \vee \mathcal{L}(Q_1) \vee \mathcal{L}(Q_2))) \\
\Rightarrow & \quad \text{(distributing } \wedge \text{ over } \vee \text{; Proposition 0(a) and condition C1)} \\
 & (\exists W \cup U \cup \{c\} : \mathcal{L}(T) \vee \mathcal{L}^{\mathcal{O}}(Q_1//Q_2)) \\
\Rightarrow & \quad \text{(distributing } \exists \text{ over } \vee \text{; condition C2)} \\
 & (\exists W \cup U \cup \{c\} : \mathcal{L}(T)) \vee (\exists W \cup U \cup \{c\} : \mathcal{L}^{\mathcal{O}}(T)) \\
\Rightarrow & \quad \text{(} W \cup \{c\} \text{ consists of private variables not in } T \text{)} \\
 & \mathcal{L}^{\mathcal{O}}(T)
\end{array}
$$

□

Theorem 1 (Completeness-1). The rule is complete for non-blocking processes P_1, P_2 that have disjoint interface variables.

Proof. Suppose that $P_1//P_2 \models T$ holds. Let $Q_1 = P_1$ and $Q_2 = P_2$. As Q_1 is non-blocking and has disjoint interface variables from P_2, it satisfies the condition C0; similarly for the symmetric case. Condition C1 is satisfied as $P_1//P_2 \models P_1$ and $P_1//P_2 \models P_2$ holds trivially. Condition C2 is $P_1//P_2 \models T$, which is true by assumption. Condition C3 holds as $P_1 \models (T + P_1 + P_2)$ by weakening. □

Theorem 2 (Completeness-2). The rule is complete for arbitrary processes.

Proof. Suppose that P_1, P_2, T are processes such that $P_1//P_2 \models T$. Each P_i can be made non-blocking by adding a transition for each blocking condition to a special state that has a self-loop. If P_1, P_2 have shared interface variables V, then rename the variables V to W_1 and W_2 in the processes P_1 and P_2 respectively, and modify T to T', which also accepts computations that diverge from T by differing on the values of W_1 and W_2 or by entering a blocking state. The result of the $\models$ check is unchanged with the new processes. From the previous theorem, therefore, there is a proof of $P_1//P_2 \models T$. □

3 Compositional reasoning with Timing Diagrams

In the previous section, we gave a sound and complete rule for assume-guarantee based compositional reasoning. In this section we show how to apply that rule to specifications in the form of timing diagrams. By focusing on timing diagrams, which are a highly regular specification formalism, we obtain several benefits. Firstly, for a large class of timing diagrams the soundness check C3 in the rule follows directly as a consequence of the expressiveness of the formalism and so can be dispensed with. Secondly, we take advantage of the fact that many timing diagrams have efficient model checking procedures. Finally, we also show that the generation of helper assertions is not only automatic but efficient for a large class of timing diagrams.

Timing diagrams are a graphical notation commonly used to specify timing behavior of hardware systems. Synchronous Regular Timing Diagrams (SRTD's) [4] are a class of timing diagrams that correspond to a subset of the ω-regular languages. SRTD's have a formal syntax and semantics and there are efficient, polynomial time algorithms for model checking SRTD's (see [4] for details). These facts make SRTD's an effective formal specification notation.

An SRTD is specified by describing a number of waveforms with respect to a given clock. The clock waveform is a sequence of boolean values ($\{0, 1\}$), where the value toggles at consecutive points. A change in the clock value from 0 to 1 is called a *rising* edge, while a change from 1 to 0 is called a *falling* edge. The waveforms are sequences of values over $\{0, 1, X, D\}$, where X indicates a don't-care value, and D a don't-care transition. A change in value of a waveform (e.g., $0 \to 1$) must occur at rising or falling edges of the clock. The waveforms of an SRTD are partitioned into an initial *precondition* part that does not contain any don't-care transitions and the following *postcondition* part. In turn, the post-condition may be partitioned using *pause* markers. For example, in the SRTD of Figure 1, there are three signals, $A.p$, $B.q$ and $A.r$, the clock, a precondition marker, etc.

A don't care value (X) is used to specify that the value at a point is unknown, unspecified or unimportant. A maximal sequence of don't-care transition values (D) on a waveform must be preceded by a definite boolean value b, and followed by the value $\neg b$. The sequence of D values indicates that the transition from b to $\neg b$ occurs exactly once in the specified interval. A pause specifies that there is a break in explicit timing at that point, i.e. the value of the signals, except the clock, remains unchanged for an arbitrary but finite period of time. At each pause point, there must be at least one signal whose waveform has a definite change of value relative to the following point. This signal indicates the end of the pause. One such signal is designated as the "owner" of the pause.

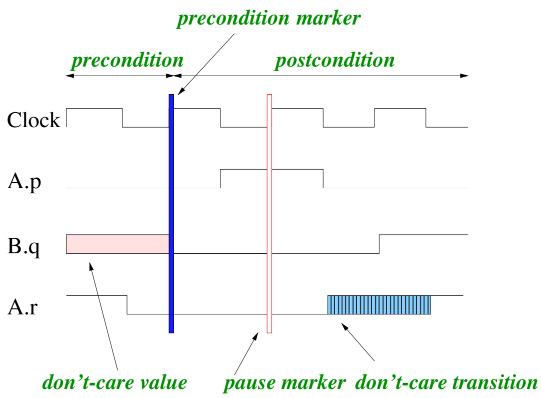

Fig. 1. Annotated Synchronous Regular Timing Diagram

An SRTD defines an ω-regular language. In [4], it is shown that we can construct regular expressions for the precondition T_{pre} and the postcondition T_{post} of an SRTD T. In the remainder of the paper, we use T_{pre} (T_{post}) to denote both the syntactical definition of precondition (postcondition) and its associated regular expression.

An infinite computation σ satisfies an SRTD T (written $\sigma \models T$) if and only if every finite segment of σ that satisfies the precondition is immediately followed by a segment that satisfies the postcondition of the diagram. The precondition, however, may be satisfied in an overlapping manner, which leads to two distinct notions of satisfaction, overlapping and non-overlapping semantics.

Definition 0 (Overlapping Semantics). An infinite computation σ satisfies an SRTD T under the overlapping semantics ($\sigma \models_o T$) iff every occurrence of T_{pre} in σ is followed by an occurrence of T_{post}. Formally, this is true iff $\sigma \notin (\Sigma^*; T_{pre}; \neg T_{post})$, where Σ is the set of valuations to the signals in T.

To define non-overlapping semantics, it is convenient to assume that there is an auxiliary proposition p such that for all sequences σ, p is true at the ith point iff T_{pre} is satisfied by a prefix of the suffix sequence starting at point i.

Definition 1 (Non-overlapping Semantics). An infinite computation σ satisfies an SRTD T under the non-overlapping semantics ($\sigma \models_n T$) iff every occurrence of T_{pre} that does not overlap an occurrence of T_{pre} or T_{post} is immediately followed by an occurrence of T_{post}. This is true iff $\sigma \in ((\neg p)^*; T_{pre}; T_{post})^\omega + ((\neg p)^*; T_{pre}; T_{post})^*; (\neg p)^\omega$.

Proposition 1. For any SRTD T, $\sigma \models_o T$ implies $\sigma \models_n T$.

3.1 Translation Algorithms

In order to use SRTD's as a specification language in a compositional model checking paradigm we need to augment the above definitions of SRTD's with some information about the modularity of the design being verified. This is achieved by defining an ownership function $O : S \rightarrow N$ that maps each signal to the implementation module that controls it, where S is the set of signals and N is a set of module names. The ownership function O can be used to partition the SRTD T into *fragments*, $T_1, \ldots, T_n$. The fragment T_i consists of T_{pre}, and only those waveforms in T_{post} that are owned by module i. An SRTD fragment may not be a well-formed SRTD since a fragment may contain a pause whose pause owner is in another fragment. In Figure 1, the ownership function O maps signals $A.p$ and $A.r$ to module A and $B.q$ to module B, and we have one fragment consisting of waveforms p and r and another with waveform q.

We present an algorithm that translates an SRTD into a non-deterministic ω-automaton (ω-NFA) for the *complement* of the SRTD property under the non-overlapping semantics – the construction for the overlapping case is similar and is described in [4]. Then, we give an algorithm that constructs a process that generates the non-overlapping language of the SRTD fragments.

To construct an ω-NFA $A_{\bar{T}}$ for the complement of the timing diagram language of T, we proceed as follows. First, we construct a deterministic automaton A_{pre} from T_{pre} that accepts at the first point on a string where the precondition holds. We do so by creating a non-deterministic automaton that accepts the language $\Sigma^*; T_{pre}$ and determinizing it, so that it enters an accepting state at every point on an input string where T_{pre} holds. We then eliminate outgoing edges from accepting states of this automaton. The number of reachable states in the resulting DFA can be exponential in the length of the precondition if the precondition has don't-care values. Otherwise, there are only linearly many reachable states, as the reachable part of the DFA is just the automaton for the string matching problem, which can be constructed efficiently (cf. [6]).

Next, for each signal i, we construct an ω-DFA $A_{\overline{post(i)}}$ that tracks the waveform for signal i over the length of the postcondition. This automaton checks at each clock point 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, where it checks that the value of the signal remains unchanged, and which it leaves when the pause owner signal changes value. The automaton for signal i accepts a computation iff either the waveform pattern is incorrect at some point, or if signal i is the owner of the kth pause in T and the automaton stays in the waiting state for pause k forever.

The automaton $A_{\bar{T}}$ works in the following manner: from the initial state, it runs A_{pre} on the input until this accepts; then it guesses a failing postcondition signal i and runs $A_{\overline{post(i)}}$, accepting if this accepts. If $A_{\overline{post(i)}}$ terminates (so the postcondition holds for signal i), $A_{\overline{T}}$ returns to its initial state.

Theorem 3. (Correctness) For any SRTD T and infinite sequence σ, $\sigma \models_n T$ iff $\sigma \notin L(\mathcal{A}_{\bar{T}})$.

The *size* of an SRTD T is product of the number of signals and the number of clock points.

Theorem 4. (Model Checking Complexity) For a process M and an SRTD T, under the non-overlapping semantics, the time complexity of model checking is linear in the size of M and T_{post}, and exponential in the size of T_{pre}.

Theorem 5. For a process M and an SRTD T such that T_{pre} does not contain don't-care values the time complexity of model checking under the non-overlapping semantics is linear in the size of M and T.

Theorem 6. For a process M and an SRTD T, the time complexity of model checking under the overlapping semantics is linear in the size of M and T.

These constructions can be modified easily to construct similar automata for SRTD fragments; the modification consists of choosing the failing postcondition signal only amongst the postcondition signals of the fragment.

3.2 Automatic Construction of Helper Processes

We now present an algorithm that constructs a helper processes Q_j that generates the non-overlapping language corresponding to the fragment T_j of the diagram. The process Q_j works as follows. It sets each signal i in T_j nondeterministically until the precondition holds, then it generates values for the signals of T_j as specified in the postcondition. For a don't-care value, the output is chosen nondeterministically. For a don't-care transition, the point at which the transition occurs is chosen nondeterministically as well. If the process is the owner of a pause, it non-deterministically decides when to generate this event and maintains the current value till that point. The process has a fairness constraint that forces this event to occur within a finite period. Otherwise, it maintains its value until the event that signals the end of the pause occurs, without any requirement for termination.

Proposition 2. **(Correctness)** For any SRTD fragment T_j, the corresponding helper process Q_j is non-blocking, and σ is a computation of $(//j : Q_j)$ iff $\sigma \models_n T$.

The key feature of this construction is that, for every pause k, only the process that includes the signal owning the pause has a fairness constraint enforcing the occurrence of the pause breaking event. This ensures non-interference between the fairness conditions, which is the essence of the soundness check in our compositional rule.

Theorem 7. **(Non-interference)** For SRTD T under the non-overlapping semantics, the corresponding processes $Q_1, \ldots, Q_n$, where $n > 1$, and computation σ, $\sigma \in cl(\mathcal{L}^{\mathcal{O}}(Q_1// \ldots //Q_n))$ implies $\sigma \in \mathcal{L}^{\mathcal{O}}(Q_1 + \ldots + Q_n)$.

Proof Idea. If σ is in $cl(\mathcal{L}^{\mathcal{O}}(Q_1// \ldots //Q_n))$, it must satisfy the waveform pattern at each point. If it is not in $\mathcal{L}^{\mathcal{O}}(Q_1 + \ldots + Q_n)$, this can only be because σ never produces the pause breaking event of a pending pause. But such a pause is owned by a particular Q_i; hence, σ is a computation of the Q_j's, $j \neq i$. $\square$

Theorem 8. For SRTD T with corresponding processes $Q_1, \ldots, Q_n$, the number of states of $Q_1// \ldots //Q_n$ can be exponential in the size of T.

For linear timing diagrams, those with no overlapping don't-care transitions, no don't-care values at any pause and no don't-care values in the precondition, we have the following theorem.

Theorem 9. For linear SRTD T and the corresponding processes $Q_1, \ldots, Q_n$, the number of states of $Q_1// \ldots //Q_n$ is bounded by $\mathcal{O}(|T|)$.

3.3 Compositional Model Checking of SRTD's

In this section, we will describe a proof methodology that uses SRTD's as the property T in the proof rule in Section 2. We would like to show that $P_1//P_2 \models_n T$, where T is an SRTD (respectively, $P_1//P_2 \models_o T$). By our construction in Section 3.2, we know that any SRTD T can be automatically decomposed into helper processes Q_1 and Q_2 relative to an ownership function. In order to apply the compositional rule with these choices for the Q_i's, we need only check condition C1 and C3, as conditions C0 and C2 are true by construction. In the non-overlapping case, condition C3 need not be checked, as it follows from Theorem 7. Thus, the only condition to be checked is C1. The details of this check are described in the following section.

4 Applications

We have incorporated the algorithms described in the previous sections into the RTDT tool [4]. RTDT has a user-friendly editor that allows a designer to create and edit SRTD's and a translator that complies the SRTD's into ω-automata. RTDT forms an easy to use interface to the verification tool COSPAN [8]. COSPAN is based on the automata-theoretic, language containment approach to model checking, where both the implementation and the specification are specified as ω-automata.

COSPAN checks $A \models B$ by considering only the infinite fair executions. In order to check inclusion for the finite executions as well, we utilize machine closure. If A is machine closed, any finite execution x of A can be extended to an infinite fair execution; thus, if the COSPAN check is successful, x matches some finite computation of B. The alternative is to use COSPAN's facilities for checking finite computations, but this requires the product of A and B to be constructed twice – once for each check. The machine closure method turns out to be better, as in some of our examples, processes are trivially machine closed. We added the ability to check machine closure to COSPAN.

In our current implementation, we use the non-overlapping semantics since it requires that we only check condition C1. We would like to take advantage of the linear-time (Theorems 5,6) model checking algorithms to discharge the obligation $P_1//Q_2 \models Q_1$ (similarly for the other obligation) in C1. We use Proposition 1 to replace the more expensive check $P_1//P_2 \models_n T$ by the computationally cheaper check $P_1//P_2 \models_o T$.

We used RTDT in conjunction with COSPAN to verify two systems. The first is a synchronous memory access controller and the second is Lucent's Synthesizable PCI Interface Core.

4.1 Memory Access Controller

The memory access controller system has an arbiter that provides arbitration between two user processes and a memory controller that controls three target processes. The user processes may non-deterministically request a transaction and the arbiter grants one user permission to initiate the transaction. That user process may then issue a memory instruction by asserting either the read or write line and setting the address bus. The target whose tag matches the address awakens, services the request, then asserts the *ack* line on completion.

We verified that this system satisfied both read and write memory transactions formulated as SRTD's. Table 1 presents the verification statistics of both the compositional and non-compositional approaches. In Table 1, Arb and Mem refer to the arbiter and memory controller implementation processes and Arb' and Mem' are the automatically generated helper processes. $mc(Arb/Mem')$ and $mc(Arb'//Mem)$ refer to the machine closure check performed by COSPAN. T_a (T_m) is the SRTD fragment that corresponds to process Arb (Mem). Table 1 indicates that the compositional checks are more efficient than model checking $Arb//Mem \models T$ directly. The cost of checking $Arb//Mem' \models T_a$ is more than checking $Arb'//Mem \models T_m$ and this is because most of the signals in the SRTD's for both the read and write transactions belonged to the arbiter.

Model Checking Task	Number of Variables	Number of Reachable States	Bdd Size	Space (MBytes)	Time (seconds)
SRTD for the read transaction					
Arb//Mem $\models$ T	260	2.5e+06	50084	22	73
mc(Arb//Mem')	114	1.9e+06	14772	0	2
mc(Arb'//Mem)	86	1.9e+04	14793	0	3
Arb'//Mem $\models$ Tm	129	1.1e+05	17993	6	23
Arb//Mem' $\models$ Ta	201	1.1e+06	34861	14	46
SRTD for the write transaction					
Arb//Mem $\models$ T	258	2.6e+06	54834	22	77
mc(Arb//Mem')	112	1.0e+06	14551	0	2
mc(Arb'//Mem)	99	3.8e+04	15432	0	4
Arb'//Mem $\models$ Tm	106	1.1e+05	16854	2	11
Arb//Mem' $\models$ Ta	220	7.3e+05	42844	17	67

Table 1. Verification Statistics for Memory Access Controller Design

4.2 Lucent's PCI Synthesizable Core

The second example is the Lucent Technologies PCI Interface Core, which is a set of building blocks that bridges an industry standard PCI Bus interface to a high performance F-Bus. The F-Bus supports multiple masters and slaves and there are separate master and slave interfaces to the PCI Bus. The PCI Interface Core is designed to be fully compatible with the PCI Local Bus specification [14].

In previous work [4], we used Lucent's PCI Bus Functional Model [5], which is a sophisticated environment that was developed to test the PCI Interface Core for functionality and compliance with the PCI specification. The Functional Model consists of the PCI Core blocks and abstract models for both the PCI Bus and the F-Bus. This model has about 1500 bounded state variables and was too large for model checking directly. We, therefore, restricted our verification efforts to a part of this design called *pcim-core* that deals with basic PCI functionality. The *pcim-core* process consists of a master controller *mcntrl*, a slave controller *scntrl*, a configuration process *config* and an address multiplexer *admux*. In addition there is an environment process *pcim-ENV* that contains all the inputs to the *pcim-core* process. We added a number of constraints on *pcim-ENV* to reduce the size of the state space. These constraints were property specific and were different for each property we checked.

Model Checking Task	Number of Variables	Number of Reachable States	Bdd Size	Space (MBytes)	Time (seconds)
SRTD Burst Property 1					
MC'//SC//Env $\models$ Ts	293	5.2e+05	158490	14	302
MC//SC'//Env $\models$ Tm	79	1.2e+07	44066	3	40
MC//SC//Env $\models$ T	335	4.4e+08	273140	20	511
SRTD Burst Property 2					
MC'//SC//Env $\models$ Ts	291	3.8e+05	115488	9	124
MC//SC'//Env $\models$ Tm	74	9.9e+06	42436	3	40
MC//SC//Env $\models$ T	331	1.8e+08	241792	18	430
SRTD Non Burst Property 1					
MC'//SC//Env $\models$ Ts	127	2.5e+28	587771	93	5281
MC//SC'//Env $\models$ Tm	58	1.4e+09	77411	3	74
MC//SC//Env $\models$ T *	–	–	6725219	342	138110

* did not complete due to shortage of space

Table 2. Verification Statistics for PCI Synthesizable Core Design

We formulated a number of properties as SRTD's by looking at the timing diagrams found in the PCI specification [14] and the PCI Core User's manual [5]. These SRTD's were defined over signals controlled by *mcntrl* and *scntrl*. We used RTDT to automatically construct the helper processes MC' and SC' and the property automata T_m and T_s. In Table 2, ENV refers to the composition of *pcim-ENV*, *config* and *admux*, while MC and SC refer to *mcntrl* and *scntrl* respectively. Machine closure was trivially satisfied since the *pcim-core* process did not contain any fairness.

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 start and end of a transaction. *PciIrdy* is asserted by the master to indicate that it is ready to transfer data. Similarly the slave uses *PciTrdy* to signal that it is ready for data transfer. Data is transferred between master and slave when both *PciIrdy* and *PciTrdy* are asserted on a rising clock edge. The *PciStop* signal is used by the slave to indicate termination of the transaction and the *PciDevsel* signal is used to indicate the chosen device. The first property in Table 2 stated that "in an on-going transaction, once the *PciStop* signal is asserted, the *PciTrdy* and *PciDevsel* signals remain constant until the data phase completes (*PciIrdy* is deasserted)". The second property specified that "if *PciFrame* is deasserted when both *PciIrdy* and *PciTrdy* are asserted then the data phase completes successfully ". The final property specified the non-burst mode, "if *PciFrame* is asserted for exactly one clock cycle and *PciIrdy*, *PciDevsel* and *PciTrdy* are eventually asserted then in the next clock cycle the transaction ends". Table 2 indicates that the compositional checks are far more efficient than the corresponding non-compositional checks. The non-compositional check for the non-burst property ran out of memory, the numbers shown in Table 2 are the BDD size, space and time just before memory exhaustion. The slave controller *scntrl* has a lot of interaction with both *config* and *admux* processes and this resulted in these processes being pulled into the cone of influence. This is reflected in the significant disparity in the numbers for the two compositional checks.

5 Related Work and Conclusions

As mentioned in the introduction, compositional reasoning for concurrently active processes has been the subject of much work over the past three decades. Our first contribution in this paper is the development of a sound and complete rule for reasoning about arbitrary processes, including those with fairness constraints. Earlier work (cf. [15, 1, 3, 12, 13]) either applies only to restricted kinds of processes or temporal logic formulas, or proposes incomplete rules. Our rule extends a simple reasoning rule that is known to be sound for safety properties with an additional soundness check for liveness properties. Thus, in a sense, the rule isolates the difficulties with reasoning about liveness in the soundness check.

The possibility of using timing diagrams for compositional verification appears to have been first recognized in a paper by Josko [10] on modular reasoning. This paper, however, uses timing diagrams only for illustrative purposes. In later work (cf. [9]), a compositional verification methodology proposed in [11] is used to verify timing diagrams. This work uses timing diagrams as a convenient notation for expressing temporal properties – the assume-guarantee reasoning is left to the verifier. In contrast, our work shows how assume-guarantee pairs can be generated mechanically from timing diagram specifications, resulting in a completely automated compositional verification method.

In our work, we show that timing diagram specifications in the form of SRTD's are naturally decomposable into assume-guarantee properties about the components of the system. We also show that, although timing diagrams can express liveness properties, the naïve compositional reasoning rule can be applied safely, as the additional soundness check always succeeds for the non-overlapping semantics. We show how to apply the compositional rule in a fully automated manner. Our experiments with the memory controller and the PCI interface core show that compositional reasoning can indeed be done successfully in this way, producing substantial savings in the time and space required for the verification. Although, in these examples, the natural decomposition of the timing diagram property suffices for generating the helper process, it is possible that this will not true in some cases. Thus, heuristics for automatically generating helper processes may be needed – which we leave for future work.

References

1. M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, May 1995.
2. B. Alpern and F. Schneider. Defining liveness. *Information Processing Letters*, 21(4), 1985.
3. R. Alur and T. Henzinger. Reactive modules. In *IEEE LICS*, 1996.
4. N. Amla, E.A. Emerson, R.P. Kurshan, and K.S. Namjoshi. Model checking synchronous timing diagrams. In *FMCAD*, volume 1954 of *LNCS*, 2000.
5. Bell Laboratories, Lucent Technologies. PCI Core User's Manual (Version 1.0). Technical report, July 1996.
6. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*, chapter 34. MIT Press and McGraw-Hill, 1990.
7. W.P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Non-compositional Proof Methods*. 1999. Draft book.
8. R.H. Hardin, Z. Har'el, and R.P. Kurshan. COSPAN. In *CAV*, volume 1102 of *LNCS*, 1996.
9. J. Helbig, R. Schlor, W. Damm, G. Dohmen, and P. Kelb. VHDL/S - integrating statecharts, timing diagrams, and VHDL. *Microprocessing and Microprogramming*, 38, 1993.
10. B. Josko. Model checking of CTL formulae under liveness assumptions. In *ICALP*, volume 267 of *LNCS*, 1987.
11. B. Josko. *Modular Specification and Verification of Reactive Systems.* Universität Oldenburg, 1993.
12. K.L. McMillan. Circular compositional reasoning about liveness. In *CHARME*, volume 1703 of *LNCS*, 1999.
13. K.S. Namjoshi and R.J. Trefler. On the completeness of compositional reasoning. In *CAV*, volume 1855 of *LNCS*. Springer-Verlag, 2000.
14. PCI Special Interest Group. PCI Local Bus Specification Rev 2.1. Technical report, June 1995.
15. A. Pnueli. In transition from global to modular reasoning about programs. In *Logics and Models of Concurrent Systems*, NATO ASI Series, 1985.