Efficient Model Checking for Timing Diagrams

by

Nina Amla, B.E, M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2001

Efficient Model Checking for Timing Diagrams

Approved by
Dissertation Committee:

To my parents

Acknowledgments

The time I spent at the University of Texas and Austin has been influenced by many. I am indebted to my advisor Allen Emerson. Allen provided a challenging and rewarding milieu; he has taught me that research is about clarity of thought leading to precision in exposition.

The two summers I spent at Bell Labs influenced this thesis immeasurably. Bob Kurshan made this possible. Bob has had a profound impact on my research perspective. I owe much to Jayadev Misra for his constant encouragement, technical insights and for giving crucial advice when I needed it most. I am grateful to Adnan Aziz for the many productive discussions and for being so generous with his time. I thank Mohamed Gouda and Aloysius Mok for serving on my committee and for the useful feedback.

Kedar Namjoshi has been a superb mentor, both officially and unofficially, and a good friend. Our association goes back to the first projects we collaborated on in Allen's class and continues to this day. Thanks to Richard Trefler, first a wonderful and supportive friend, and now a close collaborator.

To Phoebe Weidmann, my closest friend, thank you for everything. I value the friendship of John Havlicek, Pete Manolios and Natasha Sharygina and have enjoyed our many discussions on technical and non-technical matters.

I thank Emery Berger, Emilio Camahort, Esra Erdem, Sergei Gorinsky, John Gunnels, Sam Guyer, Mike Hewett, Subramanian Iyer, Rajeev Joshi, Vineet Kalhon, Lyn Pierce, Jack Sarvela, Jun Sawada, Vasilis Samoladas, Yannis Smaragdakis and Thomas Wahl for the great company and memories from graduate school. Brian Victor, whom I have known the longest, thanks for for being such a dear friend.

None of this would have been possible without the love and encouragement of my family. My mother, a teacher all her life, provided me with a ready role model. Thanks to my father, my guide at all times. I thank my sister Anita for her unflagging support and her company these last years in Austin. Vanya I thank for being my anchor and best friend. Mukund, thanks for having enough ambition, patience and drive for the two of us. Lovable Laila and Bert, thanks for always reminding me that happiness is as accessible as a walk around town lake.

Acknowledgments have to stop somewhere: 'What do I have that I have not received?'

NINA AMLA

The University of Texas at Austin
May 2001

Efficient Model Checking for Timing Diagrams

Publication No. ____________________

Nina Amla, Ph.D.
The University of Texas at Austin, 2001

Supervisor: E. Allen Emerson

Non-terminating systems that continually interact with their environment are called *reactive*. These types of systems are commonplace and are largely acknowledged to be hard to validate using conventional techniques. In a landmark paper, Pnueli argued that temporal logics are an effective way to reason about the correctness of reactive systems. *Model checking* is a formal technique that efficiently determines if a reactive system satisfies a temporal logic specification. In the last decade, model checking has been used extensively to verify complex hardware and software systems. However, in practice, model checking suffers from a phenomenon called *state explosion*, where the global state transition graph may be exponential in the number of subcomponents in the system. The state explosion problem severely limits the size of the systems that one can model check automatically. Another obstacle is that formal specification methods, based on temporal logic or automata, are largely unknown in the design community. This dissertation addresses both these issues by introducing a visual notation that is already used in the informal specification of hardware systems and by providing efficient model checking algorithms for these specifications.

The first part of the dissertation presents, *Regular Timing Diagrams* (RTDs), an expressive notation for specifying the temporal behavior of asynchronous systems. RTDs have a formal syntax and a simple and precise semantics that correspond to informal usage. We have developed efficient algorithms to translate RTDs into automata on infinite strings (ω-automata). We present *decompositional* model checking algorithms, that exploit the fact that RTDs can be cleanly decomposed into their constituent parts. These polynomial-time algorithms are a significant improvement over previous monolithic algorithms that are exponential in the worst case.

The second part of the dissertation introduces Synchronous Regular Timing Diagrams (SRTDs) that are used to specify the behavior of synchronous systems. The model checking algorithms developed for SRTDs are linear in the size of the diagram. A tool, based on this framework, called RTDT, which allows a user to graphically create SRTD specifications and translate them into automata, is also part of this dissertation. RTDT has been used successfully in conjunction with the model checking tool *COSPAN* to verify that Lucent Technologies PCI Interface Core satisfied actual diagrams found in the PCI Local Bus specification.

The final part of the dissertation offers a way to cope with state explosion by employing a proof technique called *compositional reasoning* that reduces reasoning about the entire system to reasoning about individual components. The *assume-guarantee* paradigm, is a type of compositional reasoning, where each component guarantees properties based on assumptions made about the other components. Applying these proof rules, however, is not automatic; it requires non-trivial human effort to decompose a property into sub-properties and to then derive the appropriate assumptions. Additionally, such proof rules are generally not complete and must be applied differently for safety and liveness properties. A new sound and complete assume-guarantee proof rule is developed in this dissertation which can be applied to both safety and liveness properties. When the property is an SRTD, this rule can be applied in a fully automatic manner by using the fact that SRTDs have a natural decomposition into assume-guarantee pairs. The application of this rule to Lucent's PCI Core and other case studies yielded substantial reductions in the space and time required for model checking.

In summary, this dissertation introduces an alternative and visual way of specifying temporal properties, which makes model checking more accessible to the non-expert user. Furthermore, this work addresses the state explosion problem by presenting efficient model checking algorithms and a general assume-guarantee proof methodology that can be applied in a fully automated manner to specifications in this form.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The class of systems that are non-terminating and interact with their environment continuously are called *reactive systems* [HP85]. Operating systems, hardware controllers and network routers are well known examples of reactive systems. It is generally acknowledged that it is hard to verify the correctness of reactive systems using conventional validation techniques like testing and simulation. Moreover, formal techniques developed for terminating sequential programs are not applicable to reactive systems. Pnueli [Pnu77], proposed the use of temporal logic as a effective way to reason formally about the correctness of reactive systems. Model checking, introduced by Clarke and Emerson [CE81] (and independently by Quielle and Sifakis [QS82]), is a fully automated procedure that efficiently decides if a reactive system satisfies a temporal specification. The specification may be either a formula in a temporal logic, like CTL (Computation Tree Logic) [EH82] or LTL (Linear Temporal Logic) [Pnu77], or, specified as an automaton on infinite strings [VW86, Kur94]. The survey paper by Emerson [Eme91] presents a comparison of these specification methods and others in terms of efficiency and expressiveness.

Model checking has been applied successfully in the verification of many industrial hardware and software systems. In fact, model checking and other formal techniques are currently used in the design process at companies such as AMD, IBM, Intel and Motorola. Model checking, however, suffers in practice due to the *state explosion* problem: if system M is defined as the parallel composition of n sub-components, the global state transition graph may be exponential in n. This imposes severe limitations on the size of the systems that can be verified. As a result, ameliorating the state explosion problem is at the forefront of research in this area. Recently developed techniques, like symmetry reduction (cf. [ES93]) and compositional reasoning (cf. [dRdBH$^+$99]) that exploit the structure of the system, have been successful in coping with the state explosion problem. In *compositional reasoning*, one avoids reasoning directly about a system composed of many sub-components by decomposing the property and proving systematically that sub-components satisfy these sub-properties. A good survey of the main contributions in this area can be found in [dRLP97]. The most well studied compositional reasoning technique is *Assume-guarantee* reasoning [MC81, Jon81, Pnu85, Sta85, Kur87, CLM89, AL95, AH96, McM97, McM99] where one uses assumptions made about the environment to satisfy the requirements of a compositional proof. While this type of reasoning has been applied in practice [McM98, HQR98] there are, however, many difficulties in actually applying these "circular" proof rules. Firstly, many proof rules apply only to safety properties and restricted types of processes and/or temporal logic. Secondly, it has been shown [NT00] that many of these proof rule are not complete. Finally, decomposing the property and deriving the auxiliary assumptions must be done manually.

Another obstacle to the widespread use of model checking is the complex nature of the specification languages. Such specifications, based on temporal logics or automata, are not well understood in the design community. Visually intuitive specification methods – which are consistent with the users own notational conventions – provide an alternative way to specify temporal behavior. The inclusion of such notations into existing model checkers would make them accessible to the non-expert user and facilitate the wider application of model checking.

This dissertation addresses both issues: the incorporation of common specification methods and the state explosion problem. We introduce formal graphical specification languages, for both synchronous and asynchronous systems, which are based on an informal notation called *timing diagrams.* Timing diagrams are already widely used in the specification of hardware systems. Polynomial-time non-compositional and assume-guarantee style compositional model checking algorithms for these diagrams are presented.

In the first part of this dissertation, we will introduce a visual specification notation, that corresponds to regular languages, called Regular Timing Diagrams (RTDs). RTDs are an effective way to specify asynchronous behavior. We provide model checking algorithms, based on the automata-theoretic approach, that are polynomial in the size of the RTD specification. Next, we will present Synchronous Regular Timing Diagrams (SRTDs) that are tailored for synchronous systems. The model checking algorithms for SRTDs are linear in the size of both the system and the SRTD specification. The final part of the dissertation describes a sound and complete assume-guarantee proof rule that can be applied to both safety and liveness properties. More interestingly, we can use this rule in a fully automated manner to properties specified in

SRTD notation. These algorithms have been implemented in a tool called RTDT which is described in the dissertation.

Much of the work done for this dissertation has been published in the following papers: [AE98], [AEN99], [AEKN00], [AENT01] and [AEKN01]. The rest of this section contains a more in-depth discussion of the problems involved and justifications for our methods.

1.1 Regular Timing Diagrams

Asynchronous timing diagrams are used to specify the behavior of asynchronous handshaking protocols like bus arbitration and memory access. The key attribute of an asynchronous timing diagram is the absence of explicit timing with respect to a global system clock. We introduce a class of timing diagrams for asynchronous systems, called *Regular Timing Diagrams* (RTDs), that have a formal syntax and semantics. The key observation that leads to efficient model checking 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 build a single, *monolithic* ω*-NFA* (Non-deterministic Finite state Automaton on infinite strings) or a temporal logic formula that corresponds to the entire diagram, we decompose the diagram into properties of isolated waveforms and their interactions. This results in a conjunction of simple properties that can be conveniently represented by a succinct ω*-NFA* for the complement of the diagram. The resulting ω*-NFA* can be used as the property in the language containment paradigm to yield a model checking algorithm that is linear in the system size and polynomial in the size of diagram. We describe how these algorithms can be applied, with the model checker VIS [BHSV$^+$96], to verify a master-slave memory system. This work was published in [AE98, AEN99] and is described in Chapter 3.

1.2 Synchronous Regular Timing Diagrams

It is more common, however, to have a *synchronous* timing specification where the changes along a signal waveform are bound to a global system clock. The encoding of such synchronous properties as RTDs introduces a large number of dependency edges between each transition of the clock and each waveform, resulting in RTDs that are visually cluttered and increasing the complexity of model checking. The Synchronous Regular Timing Diagram (SRTD) notation is, therefore, tailored to describe synchronous timing specifications in a visually clean manner. More importantly, we exploit the regular structure of SRTDs to provide model checking algorithms that are more efficient than that for RTDs. We present *decompositional* translation algorithms that construct ω-automata of size that is linear in the size of the SRTD (compared with a polynomial size complexity in [AEN99] for RTDs). This algorithm has been implemented in a tool – the *Regular Timing Diagram Translator* (RTDT) – which is described in Chapter 6. RTDT has been used in conjunction with the model checker *COSPAN* [HHK96] to verify timing diagram properties of two systems: a synchronous master-slave system and Lucents' PCI Interface Core [BL96]. This work is presented in Chapter 4 and is based on results presented in [AEKN00].

1.3 Assume-Guarantee Reasoning for SRTDs

In this work we present a new rule for assume-guarantee reasoning which generalizes several earlier proof rules (cf. [Pnu85, AL95, AH96, McM99, NT00]) 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 assume-guarantee proof rule with an additional check for soundness. As the new rule does not discriminate between processes and properties, it fits in well with a top-down approach to designing systems. We show that this new rule is complete, to the extent that if the composed system satisfies a property, then it also satisfies the property with the new rule.

Next, we explore the benefits of applying this rule in the case where the property is specified as an SRTD. We show that not only is task decomposition a relatively simple matter for SRTDs, but also that it is possible to automatically generate assumptions directly from the specification. Furthermore, we identify a class of SRTDs for which the soundness check of the rule is always satisfied, and for which the generation of the assumptions is efficient. This leads to a model checking process that is efficient (linear in the size of the diagram and the system). These algorithms have been incorporated into RTDT, which uses *COSPAN* 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. This research was published in [AENT01] and is described in Chapter 5.

1.4 The Rtdt Tool

The Regular Timing Diagram Translator (Rtdt) tool provides a user-friendly graphical editor to create and edit SRTDs and a translator that implements the compositional and non-compositional model checking algorithms. Rtdt forms a formal and efficient timing diagram interface to the model checker *COSPAN*. The key features of Rtdt are described in Chapter 6 and has appeared in [AEKN00, AEKN01].

Chapter 2

Background

In this Chapter, we will present some background on automata theory, temporal logic, model checking and timing diagrams.

2.1 Automata on Finite Strings

Definition 0 (Nondeterministic Finite state Automata (*NFA*)) *An* automaton *on finite strings $\mathcal{A}$ is a tuple $(\Sigma, Q, \delta, Q^0, F)$, where Σ is finite input alphabet, Q is a finite set of states, $\delta \subseteq Q \times \Sigma \times 2^Q$ is a transition relation, $Q^0 \subseteq Q$ is a non-empty set of start states, and $F \subseteq Q$ is a set of accepting states.*

The automaton $\mathcal{A}$ is *deterministic* (*DFA*) if $|Q^0|$=1 and $|\delta(q, a)| \leq 1$, for all $q \in Q$ and $a \in \Sigma$. A *run* r of $\mathcal{A}$ on a finite string $w = a_0, a_1, ..., a_{n-1} \in \Sigma^*$ is a sequence of states $q_0, q_1, ..., q_n$ in Q such that $q_0 \in Q^0$, and $q_{i+1} \in \delta(q_i, a_i)$ for $O \leq i < n$. A run is *accepting* if $q_n \in F$. The *language* of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of finite strings that are accepted by $\mathcal{A}$.

Automata on finite strings are closed under union, intersection and complementation [RS59]. Deterministic automata can be complemented easily by complementing the acceptance condition. However, complementing a non-deterministic automaton involves determinization and results in a construction that is exponential.

2.2 Automata on Infinite Strings

Definition 1 (Nondeterministic ω-automata (ω-*NFA*)) *An automaton on infinite strings $\mathcal{A} = (\Sigma, Q, \delta, q^0, \Phi)$ has a finite input alphabet Σ, finite state set Q, transition relation $\delta \subseteq Q \times \Sigma \times 2^Q$, start state q_0 and 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 q_0 is an initial state, and for each i, $(q_i, x_i, q_{i+1}) \in \delta$. $\mathcal{A}$ accepts x if *some* run r on x satisfies the acceptance condition Φ.

An ω-automaton is *deterministic* (ω-*DFA*) if $|\delta(q, x)| \leq 1$ for all states $q \in Q$ and symbols $x \in \Sigma$. A run r is accepting by the *Büchi acceptance* criteria if there is an accepting state that repeats in r infinitely often. In this dissertation, we consider Φ to be Büchi acceptance.

Büchi automata are closed under union, intersection [Cho74] and complementation [Buc62]. The constructions are, however, much more involved than those for the automata on finite strings. The complexity of complementation is singly exponential [SVW87].

Definition 2 (Dual Run Automata ($\forall$*FA*)) *A $\forall$FA on infinite strings $\mathcal{A}$ = $(\Sigma,\ Q, \delta, q_0, \Phi)$ has a finite input alphabet Σ, finite state set Q, transition* *relation $\delta \subseteq Q \times \Sigma \times Q$, start state q_0 and 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.

The complement of the language accepted by a $\forall$*FA* $\mathcal{A}$ is accepted by an ω-*NFA* $\overline{\mathcal{A}}$, that has the identical structure but a complemented acceptance condition. This property is formalized in the following theorem. We define $\mathcal{L}_{NFA}(\mathcal{A})$ as the language accepted by a $\exists$-acceptance criteria and $\mathcal{L}_{\forall FA}(\mathcal{A})$ as the language accepted by a $\forall$-acceptance criteria.

Theorem 0 *([MP87, Var87]) For any $\forall$FA $\mathcal{A}$, $\neg\mathcal{L}_{\forall FA}(\mathcal{A}) = \mathcal{L}_{NFA}(\overline{\mathcal{A}})$.*

2.3 Linear Temporal Logic (LTL)

We will present the syntax and semantics of Linear Temporal Logic (LTL) [Pnu77]. Formulas of LTL are built from a set of atomic proposition $\mathcal{AP}$. An LTL formula is defined as follows:

1. If $f \in \mathcal{AP}$ then f is a formula.
2. If both f and g are formulas then $f \wedge g$, $f \vee g$ and $\neg f$ are formulas.
3. If f and g are formulas then $\mathsf{X}f$, $\mathsf{G}f$, $\mathsf{F}f$ and $f\mathsf{U}g$ are formulas.

Where X is "Next time", G is "Always", F is "Eventually" and U denotes "Until".

An LTL formula is interpreted over *computations*, where a computation is a function $\pi : N \rightarrow 2^{\mathcal{AP}}$ that assigns truth values to the elements in $\mathcal{AP}$ at each time instant. For a computation π and a time instant $i \in \omega$, we have:

- $\pi, i \models f$ iff $p \in \pi(i)$, for $f \in \mathcal{AP}$
- $\pi, i \models f \wedge g$ iff $\pi, i \models f$ and $\pi, i \models g$
- $\pi, i \models f \vee g$ iff $\pi, i \models f$ or $\pi, i \models g$
- $\pi, i \models \neg f$ iff not $\pi, i \models f$
- $\pi, i \models \mathsf{X} f$ iff $\pi, i+1 \models f$
- $\pi, i \models f \mathsf{U} g$ iff for some $j \geq i$, we have $\pi, j \models g$ and for all k, $i \leq k < j$, we have $\pi, k \models f$

Thus the formula, $\mathsf{F} f$ is an abbreviation for $\mathit{true}\,\mathsf{U} f$ and $\mathsf{G} f$ is abbreviation for $\neg\mathsf{F}\neg f$. An LTL formula can be any boolean combination or arbitrary nesting of the above operators, therefore one can express $\mathsf{GF}p$ ("infinitely often p") and $\mathsf{FG}p$ ("almost everywhere p). A computation π satisfies a formula f, written $\pi \models f$, iff $\pi, 0 \models f$.

The following theorem relates LTL and Büchi automata.

Theorem 1 *([VW94]) Given an LTL formula f, one can build a Büchi automaton $A_f = (\Sigma, Q, \delta, q_0, \Phi)$, where $\Sigma = 2^{\mathcal{AP}}$ and $|Q|$ is in $2^{O(|f|)}$, such that $L(A_f)$ is exactly the set of computations satisfying the formula f.*

2.4 Model Checking

Model checking [CE81, QS82, CES86] is an automated verification technique to analyze and verify hardware and concurrent reactive systems. In model checking, one checks that a system M satisfies a specification T (written as $M \models T$). Typically the system is a circuit or program and the specification is a formula in a temporal logic, like CTL [EH82] or LTL [Pnu77]. The model checking algorithm performs searches in the transition graph of the system in a systematic manner to determine the truth of sub-formulae. For the temporal logic CTL, the algorithm uses the Tarski-Knaster theorem [Tar55], to compute the set of states that define the least fix-point. The time complexity of this method is linear in both the size of the structure and the formula.

The *language containment* paradigm [VW86, Kur94, LP85] is an approach to model checking, where both the system and the property are specified as automata on infinite strings. For the system M and specification T, the verification check $M \models T$ can be cast as $\mathcal{L}(M) \subseteq \mathcal{L}(T)$. This is equivalent to $L(M) \cap \neg\mathcal{L}(T) = \emptyset$. The algorithm for checking non-emptiness proceeds by computing the strongly connected components of the product automaton and then checking if there is a path from an initial state to a strongly connected component containing an accepting state. Language inclusion may be decided in PSPACE [LP85, VW86], and the non-emptiness problem for Büchi automata is decidable in linear time [EL85a, EL85b]. The model checking algorithm for LTL [VW86] uses Theorem 1, to build a Büchi automaton $\mathcal{A}_{\neg T}$ for the negation of formula T and then checks $\mathcal{L}(M) \cap \mathcal{L}(\mathcal{A}_{\neg T})$ for emptiness. The time complexity of model checking that a finite state program M satisfies an LTL formula T is linear in size of M but is exponential in the size of formula T. The Lichtenstein-Pnueli thesis [LP85] argues that an upper time bound that is exponential in the size of the specification is considered reasonable since the specification is usually short.

By the results in [SVW87], we know that complementing a Büchi automaton results in an exponential blowup. As a result, an approach that first constructs the Büchi automaton $\mathcal{A}_T$ (for LTL formula T) and then complements it, would result in a double exponential blow-up. In the automata-theoretic approach, therefore, it is key that the automaton for the specification be easy to complement. We observe, as a consequence of Theorem 0, that $\forall FA$'s are trivial to complement and we will exploit this fact in our work.

2.5 Timing Diagrams

A *timing diagram*, in its most basic form, consists of a number of waveforms. Each waveform depicts the behavior of a signal or variable over a finite period of time. The value of a waveform at any point in the diagram is chosen from a pre-defined domain; generally this domain is the boolean set $\{0, 1\}$. A change in the value of a waveform is known as an *event.* There are several ways that a waveform may interact with other waveforms; these interactions are called *dependencies.* A *concurrent* dependency specifies that an event depends on other events occurring at the same time. Concurrent dependencies express properties like "b is low when a rises". A *sequential* dependency relates two events in the diagram, by specifying that one event occurs within a specified time interval of the other. A sequential dependency can state properties like "event a occurred within 5 time units of event b" or " event a precedes event b. These intervals determine the type of the resulting timing diagram language. Allowing integer constants, variables and arithmetic expressions in the intervals results in a non-regular timing diagram language and restricting the interval to just integer constants and ∞ yields a regular language.

A timing diagram, 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

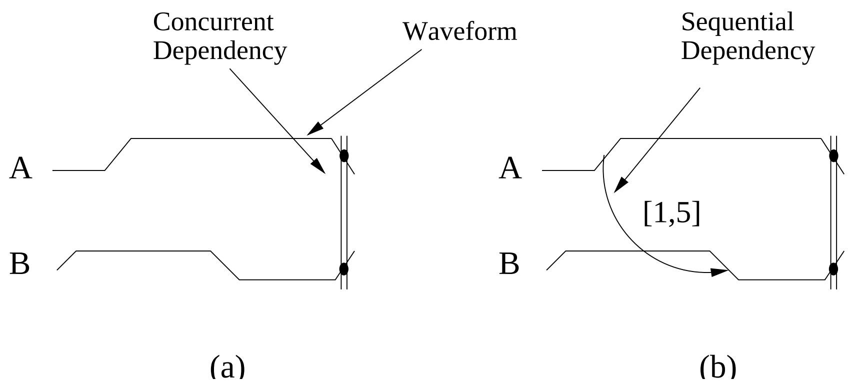

Figure 2.1: (a) Ambiguous Diagram (b) Unambiguous Diagram

is determined up to the clock period. Synchronous diagrams are used to specify timing requirements of clocked systems. On the other hand, *asynchronous* diagrams do not have a clock. Asynchronous timing diagrams are used to specify handshaking protocols like bus arbitration and memory access.

Another feature of timing diagrams, identified by Fisler [Fis96], is that the ordering between events is partial in general; such diagrams are called *ambiguous*. In Figure 2.1 (a), for example, the exact ordering between the rising event on waveform A and the falling event on waveform B is unknown. On the other hand, an *unambiguous* timing diagram has a total ordering on events. In Figure 2.1 (b), the sequential dependency between waveforms A and B enforces an ordering on those events. In general, synchronous timing diagrams have less ambiguity and more structure than asynchronous diagrams.

A timing diagram is defined for a finite time period and a key issue is an appropriate extension to infinite computations. Fisler [Fis96] addressed this question by considering 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.

Chapter 3

Regular Timing Diagrams

3.1 Introduction

Asynchronous timing diagrams are characterized by the absence of a global systems clock. These diagrams are generally used to specify handshaking protocols, like bus arbitration, memory access, etc. In this Chapter, we introduce a class of timing diagrams, for asynchronous systems, called *Regular Timing Diagrams* (RTDs). RTDs have a simple and precise semantics and efficient, decompositional model checking algorithms. These diagrams describe changes of signal values over a finite time period, and precedence and timing dependencies between such events; an *event* is defined as a change in signal value. RTDs can express properties like "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 integer constants, ensuring that the diagram defines a *regular* language.

RTDs, like other timing diagrams, may be *unambiguous*, there is a total ordering on events, or *ambiguous*, the ordering between events can be partial (see Figure 3.1). Since an RTD is defined for a finite time period, an important question that arises in defining the semantics is the manner in which an infinite computation satisfies a timing diagram? Recall that there are two kinds of semantics [Fis96]: 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, at points satisfying a precondition of the diagram. In our model, the precondition is a state property. Our semantics is a reformulation of the basic iterative semantics, where we permit a system to satisfy diagrams that express the correctness of different aspects of its operation. For ambiguous diagrams, we further classify this 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* [CE81, QS82, CES86] 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 build the single, *monolithic* ω-*NFA* or the temporal logic formula that corresponds to the entire diagram, we demonstrate that it is possible to decompose the 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 ($\forall FA$) [MP87, Var87]. A $\forall FA$ (also known as "dual-run" or "universal" automaton) 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 and naturally express properties that are conjunctive in nature.

Moreover, this conjunctivity can be exploited to verify smaller components of the timing diagram in isolation, thus avoiding the construction of the entire $\forall$-automaton. We present efficient algorithms that convert RTDs under the various semantics into $\forall$*FA*'s that are in the worst case of size polynomial in the size of the diagram and the largest time constant represented in unary (note that the unary size is exponential in the binary size). These constants are generally performance bounds and tend to be small; thus, we feel justified in claiming polynomial complexity. The use of $\forall$*FA*'s permits the efficient use of the automata-theoretic approach [VW86, Kur94, LP85] to model checking. For a 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 and $\mathcal{L}(X)$ denotes the language of X. This is equivalent to $\mathcal{L}(M) \cap \neg\mathcal{L}(\mathcal{A}_T) = \emptyset$. The complement language of a $\forall$*FA* is accepted by a *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 polynomial in 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 $\forall$*FA*'s, which may be checked independently with M. It is also simple to produce a description of $\neg\mathcal{L}(\mathcal{A}_T)$ that can be input to a symbolic model checker.

The algorithm is linear in the structure size, polynomial in the number of diagram points and dependencies and in the unary size of the constants. The polynomial complexity of our decompositional algorithm is a significant improvement over the earlier monolithic approaches (cf. [Fis96, DJS94]), where the size may be exponential in the worst case. Not withstanding the Lichtenstein-Pnueli thesis [LP85], in practice, as one reaches the limits of applicability of symbolic model checking tools, the size of the specification is of importance. A detailed discussion of these points is in Section 3.6.

The rest of the Chapter proceeds as follows. In Section 3.2, we give a precise syntax and semantics for Regular Timing Diagrams. Section 3.3 presents the algorithms that convert RTDs into $\forall FA$'s. The model checking procedure is presented in Section 3.4. Section 3.5 describes how the algorithms are used with with the model checking tool *VIS* [BHSV$^+$96] for the verification of a master-slave system. We conclude with a discussion of related work in Section 3.6.

3.2 Regular Timing Diagrams - Syntax and Semantics

A Regular Timing Diagram (henceforth referred to as an RTD or diagram) is specified by a number of finite waveforms, each defined over a set of "symbolic" values $\mathcal{SV}$, and timed dependencies between points on the waveforms. The set of symbolic values $\mathcal{SV}$ is an user-defined domain of values plus the value X, that is used to specify that the value is unspecified or unknown. For boolean signals, the set $\mathcal{SV}$ is $\{0, 1, X\}$. However, $\mathcal{SV}$ could be either an enumerated type, or all the values of an address bus. The set $\mathcal{SV}$ is partially ordered by $\dot{\sqsubseteq}$, where $a \dot{\sqsubseteq} b$ iff $a = X$ or $a = b$.

3.2.1 Syntax

Definition 3 (RTD) *A* RTD *is a tuple* (S, WF, SD, CD), *where*

- S *is a non-empty set of signal names.*

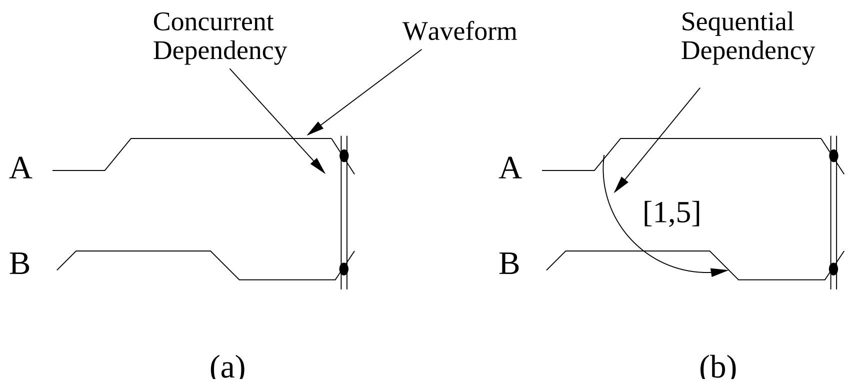

Figure 3.1: (a) Ambiguous RTD (b) Unambiguous RTD

- *WF is a collection of waveforms; for each signal $A \in S$, its associated* waveform *is a function $A : [0, n) \to \mathcal{SV}$ where $n > 1$ is an integer referred to as the* size *of the waveform. If $A \in WF$ and $i \in [0, size(A))$ then the pair (A, i) is called a* point *of A.*[1] *$(A, 0)$ is the* initial *point and $(A, size(A)$-1$)$ is the* final *point of A.*

- *SD is the set of* sequential *dependencies on the points of WF. Each dependency is specified as $(A, i) \xrightarrow{[a,b)} (B, j)$, where $a \in \mathbf{N}, b \in \mathbf{N} \cup \{\infty\}$, $1 \le a$ and $a < b$. For convenience, $[k, \infty)$ is often written as $\ge k$, $[1, k]$ as $\le k$ and $[k, k]$ as $= k$.*

- *CD is a collection of mutually disjoint, non-empty* concurrent *dependencies. Each concurrent dependency is a set of points with at most one point from each waveform in WF. The sets of initial and final points of the diagram form predefined concurrent dependencies.*

[1] *A point (A, i) is also a point of WF and the RTD.*

Definition 4 (Event) *The smallest set of points closed under the following rules are the* events *of an RTD $T = (S, WF, SD, CD)$.*

1. *For every waveform A in WF, $(A, 0)$ is an event.*

2. *Let (A, i) be an event with $(A, i) \neq X$ and let (A, j) be the first successor of (A, i) such that $A(i) \neq A(j)$. If $A(j) \neq X$ then (A, j) is 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 $(A, i) \xrightarrow{=k} (B, j)$ is a sequential dependency, 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 *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 set containing a point that is not in any concurrent dependency.

Definition 5 (Symbolic Point) *p is a* symbolic point *of an RTD iff either $p \in CD$ or p contains only one point e, such that for each $C \in CD$, $e \notin C$.*

The set of symbolic points is denoted by $\mathcal{SP}$. Informally, events in a symbolic point should occur simultaneously. The sequential dependencies of an RTD induce the following ordering relation $\prec$ on symbolic points.

Definition 6 (Ordering on Symbolic Points ($\prec$)) *Given symbolic points p and q, $p \prec q$ iff*

- *for some waveform $A \in WF$, the point $(A, i) \in p$ and point $(A, i+1) \in q$, or*
- *there exist $e \in p$ and $f \in q$ such that $e \xrightarrow{\alpha} f$ is a sequential dependency.*

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

Definition 7 (Well-formed RTD) *An RTD is* well-formed *iff (i) every point of the RTD is an event and (ii) the transitive closure of $\prec$ ($\prec^+$) is irreflexive.*

The annotated RTD in Figure 3.2 can be expressed notationally as follows.

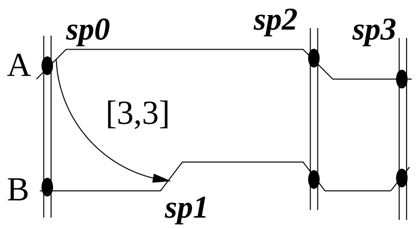

Figure 3.2: RTD T Annotated with Symbolic Points

$$\begin{aligned}
WF &: \{A, B\} \\
A &: 0 \mapsto 1, 1 \mapsto 0, 2 \mapsto 0 \\
B &: 0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 0, 3 \mapsto 1 \\
SD &: \{(A, 0) \xrightarrow{[3,3]} (B, 1)\} \\
CD &: \{\{(A, 0), (B, 0)\}, \{(A, 1), (B, 2)\}, \{(A, 2), (B, 3)\}\}
\end{aligned}$$

There are four symbolic points in this RTD: the singleton $\{(B, 1)\}$ and the three concurrent dependencies, $\{(A, 0), (B, 0)\}$, $\{(A, 1), (B, 2)\}$ and $\{(A, 2), (B, 3)\}$. The pre-defined concurrent dependencies at the initial and final points of the RTD are shown in Figure 3.2, however, for visual clarity, we will not always show these concurrent dependencies in the diagrams for the remainder of this Chapter.

3.2.2 Semantics

The semantics of an RTD is a set of infinite computations over *states*; each state is a vector indexed by the waveforms of the timing diagram. The set of states is denoted by Σ. The partial order $\dot{\sqsubseteq}$ defined earlier is extended to states as follows: $u \dot{\sqsubseteq} w$ iff for each i, $u(i) \dot{\sqsubseteq} w(i)$. A computation of the system to be verified consists of an infinite sequence of states from Σ. Since the syntax of an RTD describes only finite sequences of events, a key question is the appropriate extension to infinite computations.

The predefined initial and final concurrent dependencies can be viewed as the begin- and end- conditions of the finite sequence of events described by the RTD syntax; the initial concurrent dependency is a state predicate and the final concurrent dependency is a path predicate. For example, the begin-condition for the RTD in Figure 3.2 is $\langle A = 1, B = 0 \rangle$ and the end-condition is the concurrent dependency at the state $\langle A = 0, B = 1 \rangle$. As another example, if the diagram represents the behavior for a "memory-read" transaction, the begin- and end- conditions indicate the states that define the extent of this transaction. Clearly, this diagram should be checked only on the finite sub-computation that starts at a state satisfying the begin-condition and ends with a state satisfying the end-condition. One may thus consider an infinite sequence to satisfy a timing diagram iff the dependencies of the diagram are satisfied in each finite sub-sequence defined by the begin- and end- conditions. This statement, though, is still open to many interpretations, some of which are considered below. We first define what it means for a finite sequence of states to satisfy a timing diagram. Recall that the relation $\prec^+$ partially orders the set of symbolic points, $\mathcal{SP}$. In the following definitions $\mathcal{P}$ denotes the set of points in the diagram.

Definition 8 (Assignment) *Given a string of length n, an* assignment π *is a function $\pi : \mathcal{SP} \to [0, n)$, that is strictly monotonic w.r.t. $\prec$ ($p \prec q$ implies $\pi(p) < \pi(q)$) and maps the initial symbolic point to* 0.

Definition 9 (Equivalent Assignments) *Two assignments $\pi : \mathcal{SP} \to [0, n)$ and $\xi : \mathcal{SP} \to [0, m)$ are* equivalent *iff for all $p, q \in \mathcal{SP}$, $\pi(p) < \pi(q)$ iff $\xi(p) < \xi(q)$.*

Any assignment π induces the function $\hat{\pi} : \mathcal{P} \to [0, n)$ 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 of π, it follows that all points in a concurrent dependency are assigned a common position.

Definition 10 (RTD satisfaction) *An RTD $T = (S, WF, SD, CD)$ is satisfied by a finite sequence $z \in \Sigma^+$ w.r.t. an assignment $\pi : \mathcal{SP} \to [0, |z|)$ (written as $z \models_\pi T$) iff the following conditions hold.*

1. *Point consistency: For every point (A, i), if $\hat{\pi}((A, i)) = k$, then $A(i) \mathrel{\dot{\sqsubseteq}} z_k(A)$, where $z_j(A)$ is z_j projected onto the coordinates for A.*
2. *Waveform consistency: Let $\hat{\pi}((A, i)) = k$ and $\hat{\pi}((A, i + 1)) = l$. For every $j \in [k, l)$, $A(i) \mathrel{\dot{\sqsubseteq}} z_j(A)$.*
3. *Dependency consistency: For every sequential dependency $e \xrightarrow{[a,b)} f$, $(\hat{\pi}(f) - \hat{\pi}(e)) \in [a, b)$.*

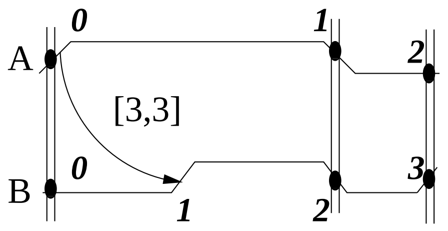

Figure 3.3: RTD Annotated with Points

We will use the following notation to denote sequences: the angle brackets denote the vector of values at a given state, ";" denotes succession in time and the superscript n on a state s is a shorthand for n successive copies of s. We will also use $\langle 1, 1\rangle$ to represent the state $\langle A = 1, B = 1\rangle$. Consider the finite sequence $y[0..6] = \langle 1, 0\rangle^3; \langle 1, 1\rangle; \langle 0, 0\rangle; \langle 0, 0\rangle; \langle 0, 1\rangle$. For RTD T in Figure 3.2, the assignment π maps: sp_0 to 0, sp_1 to 3, sp_2 to 4 and sp_3 to 6. The function $\hat{\pi}$ is as follows: $(A, 0) \to 0$, $(A, 1) \to 4$, $(A, 2) \to 6$, $(B, 0) \to 0$,

$(B, 1) \rightarrow 3$, $(B, 2) \rightarrow 4$ and $(B, 3) \rightarrow 6$. The RTD in Figure 3.3 is annotated with the points. Note that y satisfies the conditions in Definition 10, with respect to assignment π, hence $y \models_\pi T$.

For many systems, it is the case that the begin- condition for the timing diagram does not recur before the end- condition holds. For such non-overlapping 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 satisfaction of the begin- and end-conditions respectively and $\Delta = \Sigma \backslash \{\#, \$\}$. The sequence of the form $\#\Delta^+\$$ is called a *transaction*.

Definition 11 (Weak Iterative Semantics) *An infinite sequence z satisfies an 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 for which $\#y\$ \models_\pi T$.*

Definition 12 (Strong Iterative Semantics) *An infinite sequence z satisfies an 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 $\#y\$ \models_\pi T$.*

Consider the ambiguous RTD T in Figure 3.4 and a finite sequence $y = \langle 1, 0\rangle$; $\langle 1, 1\rangle$; $\langle 0, 1\rangle$; $\langle 0, 0\rangle$; $\langle 1, 0\rangle$; $\langle 0, 0\rangle$; $\langle 0, 1\rangle$; $\langle 0, 0\rangle$. Let z be an infinite sequence where the y repeats forever. In sequence y, there are two transactions, one where A falls before B rises and another where B rises before A falls. The definition of the weak iterative semantics allows a fresh ordering of events to be chosen on each transaction, therefore, $z \models_w T$. On the other hand, $z \not\models_s T$, since the ordering used in the two transactions is different.

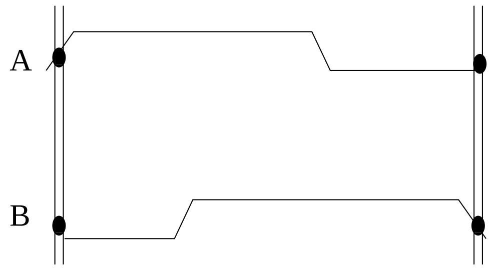

Figure 3.4: Ambiguous RTD T

We consider now an alternative formulation of Definition 10, which forms the basis for the decompositional algorithms for model checking. If $\#y\$$ satisfies the timing diagram, each event, by Definition 4, 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.

Theorem 2 *Let pt be the partial function that defines the location of events on a finite sequence. For an RTD $T = (S, WF, SD, CD)$, and any finite transaction $z = \#y\$$, there exists an assignment π such that $z \models_\pi T$ iff each of the following conditions holds:*

(a) *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 $A(i) \dot{\sqsubseteq} z_k(A)$.*

(b) *Let $pt(z, (A, i)) = k$ and $pt(z, (A, i + 1)) = l$. For every j in $[k, l)$, $A(i) \dot{\sqsubseteq} z_j(A)$.*

(c) *For each sequential dependency $e \xrightarrow{[a,b\rangle} f$, $(pt(z, f) - pt(z, e)) \in [a, b\rangle$.*

(*d*) *For each pair of events e, f in a concurrent dependency, $pt(z,e) = pt(z,f)$.*

Proof. ($\Rightarrow$) $z \models_\pi T$ implies, by Definitions 8 and 10 and the precise location property, that pt is total. Point consistency, in Definition 10, implies that $pt(z,(A,i)) = k$ then $A(i) \dot{\sqsubseteq} z_k(A)$. Condition ($c$) follows directly from waveform consistency in Definition 10. Dependency consistency in Definition 10 implies $(pt(z,f) - pt(z,e)) \in [a,b\rangle$. Definition 8 implies that each pair of events e, f in a concurrent dependency, are assigned by π to the same location, hence $pt(z,e) = pt(z,f)$.
($\Leftarrow$) If $\hat{\pi}((A,i)) = k$ then, by conditions (a) and (d), $A(i) \dot{\sqsubseteq} y_k(A)$ (point consistency). Conditions a and b ensure waveform consistency (Definition 10). Dependency consistency follows directly from c. $\square$

Notice that the theorem essentially transforms the existential ($\exists$) condition of Definitions 11 through 12 into a universal ($\forall$) condition; this forms the basis for the decompositional check.

3.3 Translation Algorithms

Theorem 2 is fundamental to decomposing RTDs into a conjunction of properties of individual waveforms, and ordering or timing restrictions on their interactions, which is the key to efficient model checking. In this section, we provide algorithms that translate an RTD under, both strong and weak iterative semantics, into a $\forall FA$. For clarity, we often describe the ω-*NFA* for the complement language instead of the $\forall FA$.

3.3.1 Translating RTDs with Weak Iterative Semantics

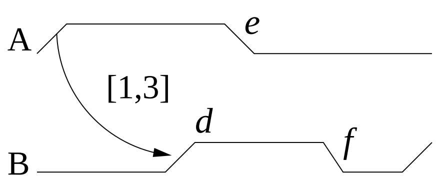

Figure 3.5: RTD T Annotated with Unordered Events

Recall, that we can construct a *DFA* called $locator(e)$ that accepts at the position on an input string where the event e holds. We now describe the ω-*NFA* $A_{\overline{T}}$ that accepts the complement of the weak-iterative language of an RTD $T = (S, WF, SD, CD)$.

Algorithm 1

1. Construct a finite string automata for each waveform and dependency as follows:

    - Waveform: The automaton $\mathcal{A}_B$ for a waveform B is constructed as follows: if $(B, i+1)$ is defined in terms of (B, i), then $locator((B, i))$ is extended to ensure that the signal values up to the change of value that defines $(B, i+1)$ are above $B(i)$ in $\dot{\sqsubseteq}$ order. Otherwise, $locator((B, i))$ is used to determine that the value at the position where (B, i) holds is above $B(i)$ in $\dot{\sqsubseteq}$ order.
    - Sequential dependency: The automaton $\mathcal{A}_{sd}$, for a sequential dependency $e \xrightarrow{[a,b\rangle} f$, is a parallel composition of $locator(e)$ and $locator(f)$

that accepts iff the time between the acceptance of the locator *DFA*'s is within $[a, b\rangle$.

- Concurrent dependency: The $\forall FA$, $\mathcal{A}_{cd}$, 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.

2. The ω-*NFA* $\mathcal{A}_{\overline{T}}$ operates as follows on an infinite input sequence: it non-deterministically "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.

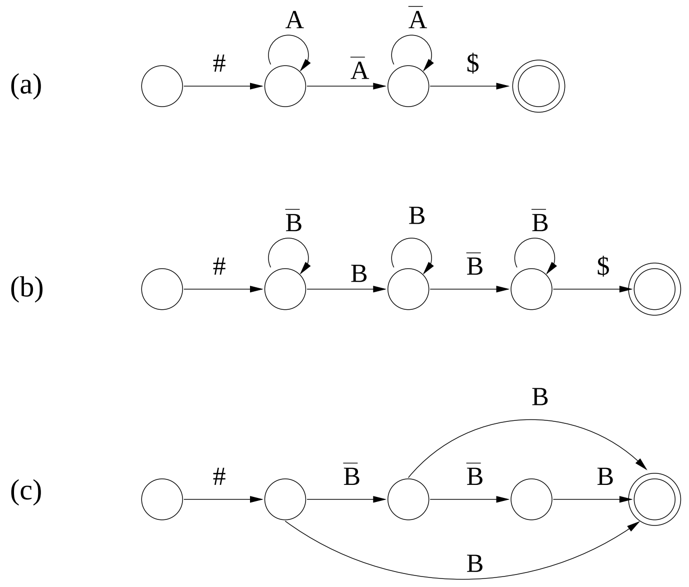

Figure 3.6: Automata for (a) Waveform A (b) Waveform B (c) Sequential Dependency

Notice that each automaton defined above is either a *DFA* or a $\forall FA$, both of which can be trivially complemented. The $\forall$*FA* $\mathcal{A}_T$ obtained from this ω-*NFA* $\mathcal{A}_{\overline{T}}$ by complementing the acceptance condition defines the language of the RTD under the weak iterative semantics.

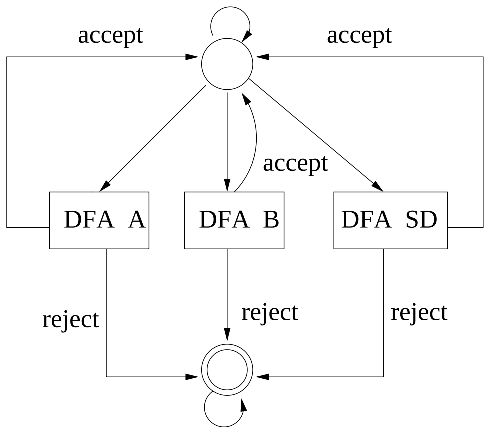

Figure 3.7: ω-*NFA* $\mathcal{A}_{\overline{T}}$ for Weak Iterative Semantics

Theorem 3 (Correctness) *For any RTD T and $x \in \Sigma^\omega$, $x \models_w T$ iff $x \in \mathcal{L}(\mathcal{A}_T)$.*

Proof. ($\Rightarrow$) $x \models_w T$ implies (by definition 11) that for every transaction $\#y\$$ on x, there exists an assignment π such that $\#y\$ \models_\pi T$. Let us assume that $x \in \mathcal{A}_{\overline{T}}$. We know, by the construction of $\mathcal{A}_{\overline{T}}$, that there must be a transaction z along x such that some *DFA* (for a waveform or dependency) $\mathcal{A}_d$ rejects on z. Therefore, by the construction of *DFA*'s $\mathcal{A}_d$, there is no assignment π such that $z \models_\pi T$ (i.e. z must violate the constraints on some waveform or dependency). Since $x \models_w T$, such a z transaction does not exists; thus $x \in \mathcal{L}(\mathcal{A}_T)$.

($\Leftarrow$) $x \in \mathcal{L}(\mathcal{A}_T)$, by definition 2, iff every run of $\mathcal{A}_T$ on x is accepting. In the construction of $\mathcal{A}_T$, this implies that every transaction along

x is accepted by all the *DFA*'s (for the waveforms or dependencies). Let us assume that $x \not\models_w T$, this means that x has a transaction z, such that $\neg(\exists\pi : (z \models_\pi T))$. Therefore, by Definition 10, z violates either (1) point consistency (2) waveform consistency, or (3) dependency consistency. However, the *DFA*'s constructed for waveforms or dependencies satisfy these conditions by construction. Thus, we have a contradiction, and $x \models_w T$. $\Box$

For the diagram $T = (S, WF, SD, CD)$, 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 4 (Complexity) *For any RTD T, the size of the corresponding $\forall$FA $\mathcal{A}_T$ is polynomial in $|T|$ and the unary length of the largest constant in T.*

Proof. The size of an RTD is $T = e+s+c$, where e is the number of events in T, $s = |SD|$ and $c=|CD|$. Let l be the largest constant in unary and w be the number of waveforms. We assume that the transitions in $\mathcal{A}_T$ are labeled with boolean formulas over the w 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 $v + t$, where v is the number of states and t is the transition size.

The number of states in each locator automaton is bounded by $k = (e + l)$. In the construction above, the number of states in the *DFA* for a waveform is $O(k)$. Since each transition encodes the values of the signals at each point, the size of each transition is $O(w)$, while the number of transitions is bounded by e. Thus, the transition size of each locator automaton is $O(k)$. The size of the *DFA* for each sequential dependency is $O(k^2)$, as it consists of two locators in parallel. The size of the $\forall$*FA* for each concurrent dependency is $O(w.k^2)$, as each concurrent dependency can have at most w events. The size of $\mathcal{A}_T$ is $w.O(k) + s.O(k^2) + c.O(k^2)$, which is cubic in $|T|$. $\square$

3.3.2 Translating RTDs with Strong Iterative Semantics

Under the strong iterative semantics, every transaction on an input computation has to satisfy the RTD T with respect to a single event ordering. The ω-*NFA* $\mathcal{A}_{\overline{T}}$ for the complemented language accepts a computation iff

- Some transaction violates a waveform or dependency constraint, or
- There is a transaction and a pair of events that occur in a different order from that in the first transaction.

The ω-*NFA* $\mathcal{A}_{\overline{T}}$ for the complement of the RTD T under the strong-iterative semantics is constructed as follows:

Algorithm 2

1. Construct the ω-*NFA* $\mathcal{A}_{weak}$ as defined for the weak-iterative semantics.
2. For each pair of unordered events e and f, construct an automaton A_{ef} as follows: A_{ef} first executes the locator *DFA*'s for events e and f in parallel on the first transaction to determine their relative order. A_{ef} then chooses a subsequent transaction and executes the locator *DFA*'s of the same events on that transaction to determine the new order, and accepts if the orders differ.

3. The ω-*NFA* $\mathcal{A}_{ord}$, at the initial state, nondeterministically "chooses" events e and f that are unordered by $\prec^+$, runs automaton A_{ef} and accepts if A_{ef} accepts.

4. The ω-*NFA* $\mathcal{A}_{\overline{T}}$ accepts if either $\mathcal{A}_{weak}$ or $\mathcal{A}_{ord}$ accepts.

Figure 3.8 depicts the automaton $\mathcal{A}_{de}$ for events d and e in the RTD shown in Figure 3.5. In Figure 3.8, Σ is the alphabet, $\Delta = \Sigma \backslash \{\#, \$\}$, $\Sigma^{\#}$ denotes $\Sigma \backslash \{\#\}$, Δ^{de} denotes $\Delta \backslash \{d, e\}$, Δ^{d} denotes $\Delta \backslash \{d\}$ and Δ^{e} denotes $\Delta \backslash \{e\}$.

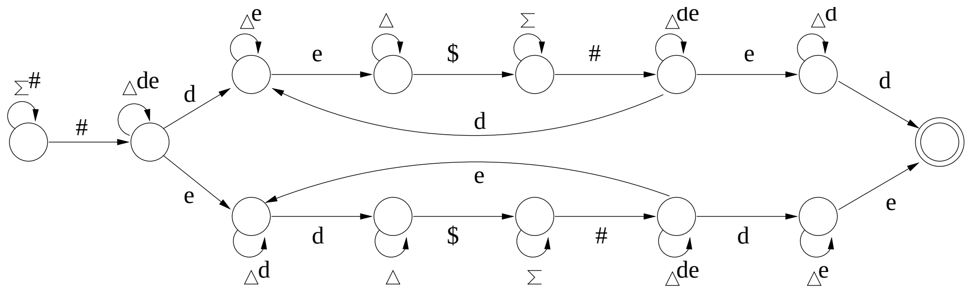

Figure 3.8: *NFA* $\mathcal{A}_{de}$ for Events d and e in Figure 3.5

Let $\mathcal{A}_T$ denote the $\forall FA$ obtained from the ω-*NFA* $\mathcal{A}_{\overline{T}}$ by complementing the acceptance condition. The size of $\mathcal{A}_T$ is polynomial in $|T|$ and l for the first case ($\mathcal{A}_{weak}$); for the second ($\mathcal{A}_{ord}$), it is quadratic in $|T|$ and l with a multiplicative factor of the number of event pairs (which is bounded by $(\#points)^2$).

Theorem 5 (Correctness) *For any RTD T and $x \in \Sigma^\omega$, $x \models_s T$ iff $x \in L(\mathcal{A}_T)$.*

Proof. ($\Rightarrow$) $x \models_s T$ iff (by definition 12) there exists an assignment π, such that for every transaction z along x, $z \models_\pi T$. Let us assume that $x \in \mathcal{A}_{\overline{T}}$; by the construction of $\mathcal{A}_{\overline{T}}$, there must be a transaction z along x where either (i) some *DFA* (for a waveform or dependency) $\mathcal{A}_d$ rejects on z, or (ii) there exists events e and f in z that differ in relative ordering from the initial transaction. In the first case, by Theorem 3, we have a contradiction. For the second case, the automaton $\mathcal{A}_{ef}$ accepts, indicating that e and f occur in a different order. This implies that a different assignment, ξ is used on transaction z. Such an assignment, however, is not possible, since $x \models_s T$; thus $x \in L(\mathcal{A}_T)$.

($\Leftarrow$) $x \in L(\mathcal{A}_T)$, by construction, implies that every transaction along x is accepted by the *DFA*'s for each waveform, dependency and ordering. Let us assume that $x \not\models_w T$, this means that x has a transaction z, such that either (i) $\neg(\exists \pi : (z \models_\pi T))$ or (ii) the ordering between two events e and f in z differs from ordering in the first transaction. In case (i), we appeal to the result in Theorem 3 to show $z \models_\pi T$. In the second case, by construction, each *DFA* $\mathcal{A}_{ef}$ that checks the relative ordering between the events e and f must reject. Hence, an assignment π used on the first transaction is used on every subsequent transaction. We get a contradiction in both cases, thus, $x \models_s T$. $\square$

Theorem 6 (Complexity) *For any RTD T, the size of the corresponding $\forall$FA $\mathcal{A}_T$ is polynomial in $|T|$ and the largest constant in unary.*

Proof. The size of an RTD T is $e+s+c$, where e is the number of events in T, s is the size of *SD* and c is the size of *CD*. Let l be the largest constant in unary and w be the number of waveforms. We assume that the transitions in $\mathcal{A}_T$ are labeled with boolean formulas over the w 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 $v+t$, where v is the number of states and t is the transition size.

Recall from Theorem 4 that the size of each locator automaton is bounded by $k=(e+l)$. The automaton $\mathcal{A}_{weak}$ for the first check is essentially the same as the automaton for the weak iterative semantics and the number of states is cubic in $|T|$ and l. The automaton $\mathcal{A}_{ord}$ for the second part has size proportional to the product of two locator *DFA*'s for each choice, and there are e^2 such choices; thus, the number of states overall is $e^2.O(k^2)$. The size of $\mathcal{A}_T$ is $O(|T|^3)$ + $e^2.O(|T|^2)$, which is polynomial in $|T|$. $\Box$

3.4 Decompositional Model Checking

The translation of an RTD to a small $\forall$*FA* implies that the language containment approach to model checking based on [VW86] gives an efficient algorithm. We need to check that $\mathcal{L}(M) \subseteq \mathcal{L}(\mathcal{A}_T)$, where M is the system to be verified and $\mathcal{A}_T$ is the $\forall$*FA* for the RTD T. This is equivalent to $\mathcal{L}(M) \cap \neg\mathcal{L}(\mathcal{A}_T) = \emptyset$. Complementation (the $\neg\mathcal{L}(\mathcal{A}_T)$ term) is trivial for a $\forall$*FA*; the complemented automaton (an ω-*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 [SVW87], is logarithmic in the sizes of both M and T.

Theorem 7 (Model Checking Complexity (Weak)) *For a transition system M and an RTD T with the weak iterative semantics, the time complexity of model checking is linear in the size of M and cubic in the size of T and the unary size of the largest constant in T.*

Proof. The size of an RTD is $T= e+s+c$, where e is the number of events in T, s is the size of *SD* and c is the size of *CD*. We know that checking $M \models_w T$ is equivalent to checking that $\mathcal{L}(M) \cap \neg\mathcal{L}(T) = \phi$. The size of the ω-*NFA* $\mathcal{A}_{\overline{T}}$ that accepts $\neg\mathcal{L}(T)$, by Theorem 4, is cubic in the size of T. Therefore, the time complexity of checking $\mathcal{L}(M) \cap \mathcal{L}(\mathcal{A}_{\overline{T}}) = \phi$ is $O(|M|.|T|^3)$. $\Box$

Theorem 8 (Model Checking Complexity (Strong)) *For a transition system M and an RTD T with the strong iterative semantics, the time complexity of model checking is linear in the size of M and a small polynomial in the size of T and the unary size of the largest constant in T.*

Proof. The size of an RTD is $T= e+s+c$, where e is the number of events in T, s is the size of *SD* and c is the size of *CD*. We know that checking $M \models_s T$ is equivalent to checking that $\mathcal{L}(M) \cap \neg\mathcal{L}(T) = \phi$. The size of the ω-*NFA* $\mathcal{A}_{\overline{T}}$ that accepts $\neg\mathcal{L}(T)$, by Theorem 6, is polynomial in the size of T. Therefore, the time complexity of checking $\mathcal{L}(M) \cap \mathcal{L}(\mathcal{A}_{\overline{T}}) = \phi$ is $O(|M|.|T|^4)$. $\Box$

An alternative way of utilizing the $\forall FA$ construction is to note that, $\mathcal{A}_T$ 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 0 *For finite-string languages L_i $(i \in [0,n))$ which are subsets of Δ^+, $(\Delta^+ \vee \#(\bigwedge_i L_i)\$)^\omega = \bigwedge_i(\Delta^+ \vee \#L_i\$)^\omega$.*

Proof. Let $\Sigma = \Delta \cup \{\#, \$\}$, $\mathcal{A}_m = (\Sigma, Q_m, \delta_m, q_0, \Phi_m)$ be the ω-automaton that accepts $\mathcal{L}((\Delta^+ \vee \# \bigwedge_i(L_i)\$)^\omega)$ and $\mathcal{A}_c = (\Sigma, Q_c, \delta_c, r_0, \Phi_c)$ be the ω-automaton that accepts $\mathcal{L}(\bigwedge_i(\Delta^+ \vee \#L_i\$)^\omega)$.
($\Rightarrow$) Let $\mathcal{A}_l = \mathcal{A}_0 \times \mathcal{A}_1 \times ... \times \mathcal{A}_n$ be the *DFA* that accept the language $L_0 \cap L_1 \cap ... \cap L_n$ and $\mathcal{A}_\Delta$ be the *DFA* for Δ^+. Let x be an infinite string accepted

by $\mathcal{A}_m$. We observe that $\mathcal{A}_m$ has a transition of the form $\delta_m((s_0, s_1, ..., s_n), \#) = (t_0, t_1, ..., t_n)$, where each t_i is the unique start state for each $\mathcal{A}_i$. We also know that every $\mathcal{A}_i$ transitions on \$ to an accepting state. It follows that all the $\mathcal{A}_i$ automata accept at the same point and therefore x is accepted by $\mathcal{A}_c$.
($\Leftarrow$) Consider an infinite string x that is accepted by $\mathcal{A}_c$ and rejected by $\mathcal{A}_m$. This implies that some $\mathcal{A}_i$ and $\mathcal{A}_j$ accept at different points in x. But we know that every transition on # goes to the unique start state in $\mathcal{A}_i$ and $\mathcal{A}_j$. Thus both $\mathcal{A}_i$ and $\mathcal{A}_j$ must start together. We also know that both $\mathcal{A}_i$ and $\mathcal{A}_j$ have a transition on \$ which goes back to the start state of *DFA* $\mathcal{A}_\Delta$; hence they also end simultaneously. This implies that both must accept at the same point and contradicts the assumption that x is not accepted by $\mathcal{A}_m$. $\Box$

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 $\mathcal{L}(M)$. 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.

Theorem 9 (Decompositional Model Checking (Weak)) *For a transition system M and an RTD T under the weak semantics, the time complexity of decompositional model checking is linear in the size of M and cubic in the size of T.*

Proof. The problem of checking $M \models_w \mathcal{A}_T$ can be decomposed into $\bigwedge_i M \models \mathcal{A}_i$, where $\mathcal{A}_i$ is the automaton for a waveform or dependency. We can check $M \models \mathcal{A}_i$ in time linear in the size of M and $\mathcal{A}_i$, which by Theorem 4 is $O(|M|.|T|^2)$. But we have $|T|$ such verification tasks, thus the time complexity of checking $M \models \mathcal{A}_T$ is $O(|M|.|T|^3)$. $\Box$

Theorem 10 (Decompositional Model Checking (Strong)) *For a transition system M and an RTD T, under the strong semantics, the time complexity of decompositional model checking is linear in the size of M and a small polynomial in the size of T.*

Proof. The complexity of checking $M \models_w \mathcal{A}_T$, by Theorem 9, is $O(|M|.|T|^3)$. The size of A_{ef}, the automaton that checks ordering between events e and f, is quadratic in $|T|$, and there may be $|T|^2$ such automata. Thus the time complexity of checking $M \models_s \mathcal{A}_T$ is $O(|M|.|T|^4)$. $\Box$

We will demonstrate in following section that the decompositional approach to model checking yields non-trivial savings in space and time.

3.5 Applications

We demonstrate the use of these algorithms in the verification of a master-slave memory system using the model checker VIS [BHSV$^+$96].

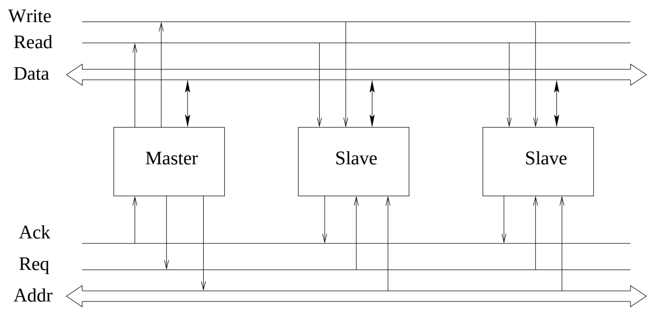

Figure 3.9: Master-Slave Architecture

In the master-slave system (Figure 3.9), 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 memory read (Figure 3.10) and write (Figure 3.11) cycles are specified as RTDs, interpreted under the weak iterative semantics.

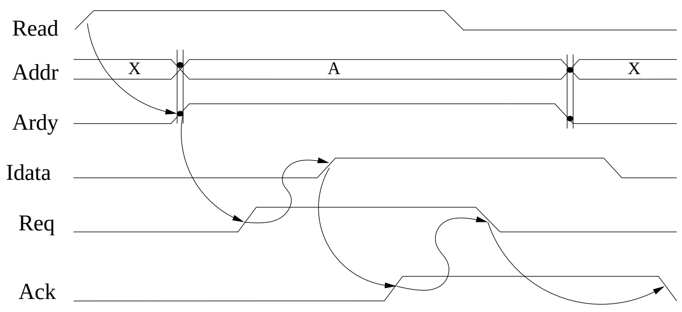

Figure 3.10: RTD for the Memory Read Cycle

The master-slave system was simplified by abstracting away inessential details. First, the address bus was simplified to the tag of the slaves. Since VIS does not allow shared variables, the bidirectional data bus is represented as two 1-bit boolean variables, *Idata* and *Odata* that denote the input and output data buses respectively. The begin-condition for the read RTD is the state that has *Ardy*, *Idata*, *Req*, *Ack* and *Write* being assigned 0 (low), the value of the address bus *Addr* is unknown and the *Read* signal is asserted. The end-condition for the read RTD is the state following the diagram where all the signals are low and *Addr* is X. The RTDs have a high degree of ambiguity since there is ordering specified for most of the de-assertion events in the diagram.

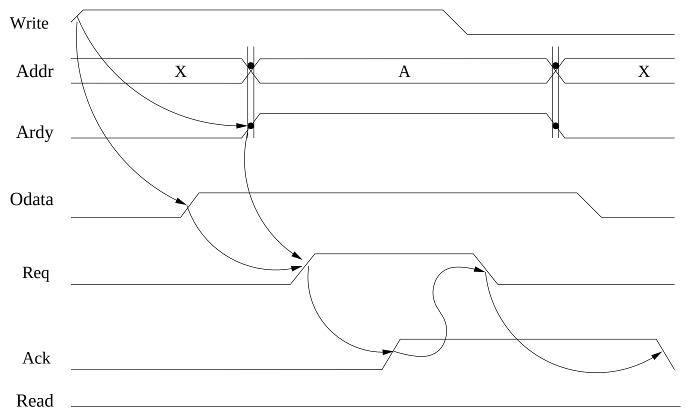

Figure 3.11: RTD for the Memory Write Cycle

The simplified master-slave system is represented in Verilog, which is the input language of VIS. For both RTDs, we created (as Verilog modules) both the complement of the $\forall FA$ and the complement *NFA*'s for individual dependencies and waveforms.

We verified that the master-slave system satisfied read and write RTDs, using both the decompositional and monolithic model checking approaches. The language emptiness check passed for both the read and write RTDs. In Table 3.5, the rows with the suffix *(D)* correspond to a verification check involving the master-slave system and a single waveform (or dependency) module. The suffix *(M)* refers to a verification check with the master-slave system and the product of all the waveform and dependency modules. We observe that the monolithic verification is significantly more expensive, in terms of BDD size and space, than a single decompositional check. For the read RTD, there

Design under verification	Number of BDD variables	Reachable state space	Number of BDD nodes	Verification Time (seconds)
Master−Slave and 1 waveform module (D)	91	11049	3223	22
Master−Slave and 1 dependency module (D)	91	10818	3316	29
Master−Slave and all read RTD modules (M)	145	8.5 x e 6	12509	1135
Master−Slave and all write RTD modules (M)	163	9.3 x e 6	32846	1970

Table 3.1: Verification Statistics for Master-Slave Design

were 11 such checks and for the write RTD, there were 14 decompositional checks. However, the total amount of time taken to check the entire diagram decompositionally was still less than time needed for the single monolithic check. The results in Table 3.5 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.

3.6 Related Work and Conclusions

Several researchers have investigated timing diagrams and their use in automated verification. Boriello [Bor92a] 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 [AL92, Thu96, RMM$^+$93, Cin93] have formalized timing diagrams and translated them to other formalisms (interval logics, trigger graphs etc.).

Cerny et al. present a procedure [KC98] for verifying whether the finite behavior of a set of action diagrams (timing diagrams) is consistent; [JC98] uses constraint logic programming to check if a system satisfies finite action diagram specifications. Formal notions of timing diagrams have also proved to be useful in test generation and logic synthesis (cf. [Tie92, GGL$^+$95, FS96]).

Fisler [Fis96, Fis97] proposes a highly expressive 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 [Fis97] provides a decision procedure that determines whether a regular language is contained in an unambiguous timing diagram language. This decision procedure [Fis97] has a high complexity (in PSPACE), while our algorithms have *polynomial* time complexity in the diagram size. They also provide algorithms that translates a certain class of timing diagrams into CTL [Fis96] and ω-automata [Fis00]. A key difference with our work is that these algorithms are restricted to a subset of unambiguous timing diagrams under the invariant semantics, while our algorithms are defined for all types of diagrams.

An important contribution in this area is the work done by Damm and colleagues at the University of Oldenburg on Symbolic Timing Diagrams (STD's) [DJS94, Sch95, DHKS94, HSD$^+$93, DH94]. STD's may be compiled into first-order temporal logic formulae which are then used for model checking. STD's are extended in [FJ97, Fey94] to RTSTD's (Real-time STD's), where a translation into a timed propositional temporal logic TPTL is provided. Both these research efforts consider infinite languages and ambiguity. A key difference with our work lies in the fact that their translation is monolithic, in the sense that all dependencies are considered together; this can result in an exponential blowup in the size of the resulting formulae when the diagram is highly ambiguous. While it is possible to model check the first order temporal logic presented in [DJS94, Fey94], the procedure is not very efficient.

In this chapter, we introduced Regular Timing Diagrams (RTDs) that can be used to specify temporal properties of asynchronous systems. We presented polynomial time, decompositional algorithms for model checking RTD specifications, which are based on a decomposition of the RTD semantics into properties of each waveform and the way they interact. Such decompositions may also provide a way of composing RTDs and thereby building new RTDs hierarchically. Our algorithms generate a $\forall FA$ (ω-NFA) corresponding to the RTD (the negation of the RTD). We can choose to use either the $\forall FA$ (by splitting it into smaller automata) or its complement ω-NFA in verifying that a system satisfies an RTD. These algorithms are a significant improvement over the earlier possibly exponential, monolithic translations. We have shown how our algorithms may be used in conjunction with a symbolic model checker, such as VIS, to verify systems with specifications formulated as RTDs.

Chapter 4

Synchronous Regular Timing Diagrams

4.1 Introduction

In Chapter 3, we proposed a class of timing diagrams called RTDs (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 synchronous specifications can be encoded as RTDs, the encoding introduces a large number of dependency edges between each transition of the clock and each waveform, which results in RTDs that are visually cluttered and have (unnecessarily) increased complexity for model checking. Hence, the initial motivation for introducing a new notation for synchronous timing properties was expressiveness.

Another key issue in using timing diagrams for model checking is whether the algorithms that translate timing diagrams into more basic specification formalisms 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 [DJS94, Ben98, BW98b], with non-regular timing diagrams [Fis97] and with Presburger arithmetic [ABHL97] provides algorithms that are, in the worst-case, of exponential or higher complexity in the size of the diagram. The regular structure of the synchronous timing diagrams used in practice led us to believe that more efficient translation procedures were indeed possible.

The SRTD notation proposed in this Chapter is, therefore, tailored towards describing synchronous timing specifications in a visually clean manner. We precisely define the class of timing diagrams called *Synchronous Regular Timing Diagrams* (SRTDs). We provide a formal syntax and semantics that corresponds closely to the informal usage. We present *decompositional* model checking algorithms that construct an ω-automaton of size linear in the timing diagram size (compared with a polynomial size complexity in [AEN99] for RTDs). 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. This results in a model checking procedure that is linear in the size of both the system and the SRTD specification.

This algorithm has been implemented in a tool - the *Regular Timing Diagram Translator* (RTDT). RTDT provides a user-friendly graphical editor for creating and editing SRTDs and a translator that compiles SRTDs to the input language of the formal verification tool *COSPAN*. The details of the main features of RTDT can be found in Chapter 6. We used RTDT and *COSPAN* to verify several SRTD properties of two systems, a synchronous master-slave memory access system and Lucent's synthesizable PCI Core. We verified that the master-slave system satisfied the read and write transactions, which were specified as SRTDs. The second example, the PCI Core, was considerably larger. In this case, the SRTD properties were formulated by looking at the actual timing diagrams in the PCI Local Bus specification [Gro95] and the PCI Core User's manual [BL96].

The rest of the Chapter is organized as follows. Section 4.2 presents the syntax and semantics of SRTDs. In Section 4.3, we describe the decompositional translation algorithms that convert SRTDs into ω-automata. Section 4.4 illustrates applications of the RTDT tool to 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 4.5.

4.2 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 SRTDs, an event must occur at either a rising edge of the clock (rising edge triggered) or at a falling edge (falling edge triggered). In the SRTD in

Figure 4.1, signals p and r are falling edge triggered while q is triggered on either edge.

4.2.1 Syntax

A waveform in an SRTD is defined over a pre-defined domain of values. This domain may, for example, be an enumerated type or all the possible values of an address bus. In Figure 4.1, the waveforms P and R are defined over the set of booleans $\mathcal{B}$ and waveform Q is defined over a set of values that includes the value "a". In addition to representing these values, it is useful to be able 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 4.1, the don't-care values on waveform Q are used to state that value of signal Q is unspecified. In order to specify properties such as "if signal B rises then signal 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 SRTDs, 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 only difference between don't-care transitions and don't-care values. In Figure 4.1, 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 always be necessary

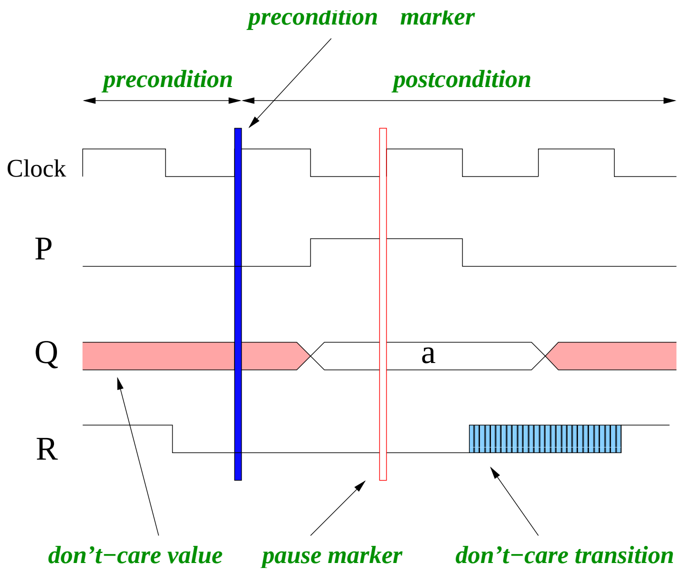

Figure 4.1: Annotated Synchronous Regular Timing Diagram

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 4.1, the pause at the end of the second clock cycle indicates that the state $\langle P = 1, Q = a, R = 0\rangle$ stutters for a finite period until P changes at a falling edge (the angle brackets indicate the tuple of values of the signals at a clock edge, while ";" indicates succession in time, measured by clock edges). The pauses allow us to express richer properties like "if *req* is asserted and stays *high* then eventually *grant* is asserted".

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 disjoint 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. Given that the domain of waveform Q is the set $\{a, b\}$, then the precondition of the diagram in Figure 4.1 is a path precondition, given by the path $\langle P = 0, Q = a + b, R = 1\rangle$; $\langle P = 0, Q = a + b, R = 0\rangle$;$\langle P = 0, Q = a + b, R = 0\rangle$.

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 from our discussion with engineers, we are led to believe that SRTDs correspond closely to informal usage and are expressive enough for industrial verification needs.

We now define SRTDs formally. A waveform A is defined over a set of *symbolic values*, $\mathcal{SV}_A$= $\mathcal{V}_A \cup \{X, D\}$, where X is a don't-care value, D indicates a don't-care transition and $\mathcal{V}_A$ is the domain of A. The set $\mathcal{SV}$ is ordered by $\sqsubseteq$, where $a \sqsubseteq b$ iff either a=b or $a \in \{X, D\}$ and $b \in \mathcal{V}$. The alphabet of an SRTD, defined over a set of signals S=$\{p, q, ..., r\}$, is $\mathcal{SV}(S)$=$\{(a_p a_q ... a_r) | a_p \in \mathcal{SV}_p \wedge ... \wedge a_r \in \mathcal{SV}_r\}$.

Definition 13 (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 $WF_A : [0, c) \rightarrow \mathcal{SV}_A$, while the associated waveform for the clock is $WF_{clk} : [0, c) \rightarrow \mathcal{B}$.*

- *M is a finite (non-empty) ascending sequence $0 \leq M_0 < M_1 < \ldots < M_{k-1} < c - 1$ 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 sub-intervals $I_0=[0, M_0]$, $I_1=(M_0, M_1]$,...,$I_{k-1}= (M_{k-2}, M_{k-1}]$, $I_k=(M_{k-1}, c-1]$. 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 14 (Segment) *The segment Seg_i ($i \in [0, k]$) that corresponds to the interval I_i of length m_i is defined to be a function $Seg_i : S \times [0, m_i) \rightarrow \mathcal{SV}$, where for each $j \in [0, m_i)$ and $A \in S$, $Seg_i(A)(j) = WF_A(j)$ when $i = 0$ and $Seg_i(A)(j) = WF_A(M_{i-1} + 1 + j)$ when $i > 0$.*

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 (Seg_0) represents the precondition, while segments $Post_i(Seg_i)$, for $i > 0$, represent successive post-condition segments. For instance, the SRTD in Figure 4.1 has three segments, one precondition segment and two postcondition segments. For each signal A, $Seg_i(A)$ is a function from $[0, m_i) \rightarrow \mathcal{SV}_A$ which describes the waveform for signal A in the ith segment. This representation of an SRTD is useful in the sequel.

Definition 15 (Precisely Locatable) *An event of waveform A occurring at a clock point t is precisely locatable if and only if $WF_A(t-1) \notin \{X, D\}$ and $WF_A(t) \notin \{X, D\}$.*

In Figure 4.1, 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.

We will now describe the well-formedness criteria on SRTDs.

Definition 16 (Well-formed SRTD) *An SRTD $T = (Pre, Post_1, ..., Post_k)$ is well-formed iff*

1. *The precondition segment Pre does not have any don't-care transitions, i.e. Pre is defined over $\mathcal{SV} \backslash \{D\}$.*

2. *Each waveform in the precondition Pre of length m must either have no don't care values or all m values must be don't-care values.*

3. *For every pause marker M_i, there exists at least one precisely locatable event at either clock point $M_i + 1$ or $M_i + 2$.*

4. *For every maximal non-empty sequence of don't-care transitions of the form $(a; D^+; b)$ in a waveform A, $a, b \in \mathcal{V}_A$ and $a \neq b$.*

5. *Every event in a waveform designated as rising(falling) edge triggered must occur at a rising(falling) edge of the clock.*

We can relax the first two requirements, to obtain a *general* SRTD, and our translation algorithms are still applicable. In this case, however, the resulting translation may be exponential in the size of the Pre; this issue will be discussed in Section 4.3.

4.2.2 Semantics

An SRTD defines properties of *computations*, which are sequences of *states*, where a state is an assignment of values to each of the n waveform signals. A computation is defined over the alphabet $\mathcal{V} = \{(a_p, a_q, ..., a_r) |\, a_p \in \mathcal{V}_P \wedge ... \wedge a_r \in \mathcal{V}_r\}$, for signals $p, q, ..., r$. For any computation y, we use y_A to denote the projection of y on to the coordinate for signal A.

Definition 17 ($\dot{\sqsubseteq}$) *For a finite waveform segment $Seg_i(A) : [0, m_i) \to \mathcal{SV}_A$ and a projection y_A of computation y with length m_i ($y_A \in \mathcal{V}_A^{m_i}$), $Seg_i(A) \mathrel{\dot{\sqsubseteq}} y_A$ iff*

- *For every $p \in [0, m_i)$, $Seg_i(A)(p) \ \sqsubseteq\ y_A(p)$.*
- *For every p, q, if $Seg_i(A)[p..q]$ has the form $(a; D^+; b)$ then $y_A[p..q]$ has the form $(a^+; b^+)$.*

Definition 18 (Segment Consistency) *A segment Seg_i of length m_i is satisfied by a sequence $y \in \mathcal{V}_n^{m_i}$ iff for each signal A, $Seg_i(A) \mathrel{\dot{\sqsubseteq}} y_A$ holds.*

Let $y = \langle 0, b, 1\rangle$; $\langle 0, a, 0\rangle$;$\langle 0, a, 0\rangle$ denote the finite sequence where $\langle P = 0, Q = b, R = 1\rangle$; $\langle P = 0, Q = a, R = 0\rangle$;$\langle P = 0, Q = a, R = 0\rangle$. In Figure 4.1, the precondition segment Pre is satisfied by y. The postcondition segment $Post_1$ is satisfied by the sequence $\langle 1, a, 0\rangle;\langle 1, a, 0\rangle;\langle 1, a, 0\rangle;\ \langle 1, a, 0\rangle$. Observe that the pause allows the state $\langle 1, a, 0\rangle$ to stutter for a finite period. The final postcondition segment $Post_2$ is satisfies by a sequence $y = \langle 0, a, 0\rangle;\langle 0, b, 0\rangle;\langle 0, a, 1\rangle;\ \langle 0, b, 1\rangle$.

We will now construct regular expressions for the precondition Pre_T and the postcondition $Post_T$ of a SRTD T. By the definition 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 at the pause marker separating it from $Post_{i+1}$.

We use $\langle 0, a, (0 + 1)\rangle$ to mean $(\langle 0, a, 0\rangle + \langle 0, a, 1\rangle)$ in the following expressions. For the SRTD T shown in Figure 4.1, the regular expression for Pre_T is $(\langle 0, (a + b), 1\rangle;\langle 0, (a + b), 0\rangle;\langle 0, (a + b), 0\rangle)$. The regular expression for $Post_T$ is $(\langle 1, a, 0\rangle;\ \langle 1, a, 0\rangle\ ;\langle 1, a, 0\rangle^*;\langle 0, a, (0 + 1)\rangle;\langle 0, (a + b), (0 + 1)\rangle;\ \langle 0, (a + b), 1\rangle;\langle 0, (a + b), 1\rangle)$.

Definition 19 (Always followed-by) $\mathbf{G}(p \hookrightarrow q)$ *holds of a computation* σ *iff, for all* i, j *such that* $j \geq i$, *if sub-computation* $\sigma[i \ldots j] \models p$, *then there exists* k *such that* $\sigma[j + 1 \ldots k] \models q$.

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 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. This is formalized in following definitions.

Definition 20 (Overlapping Semantics) *An infinite computation σ satisfies an*
SRTD T ($\sigma \models_o T$) iff $\sigma \models \mathbf{G}(Pre_T \hookrightarrow Post_T)$.

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 Pre_T is satisfied by a prefix of the suffix sequence starting at point i.

Definition 21 (Non-overlapping Semantics) *An infinite computation σ satisfies an SRTD T under the non-overlapping semantics ($\sigma \models_n T$) iff every occurrence of Pre_T that does not overlap an occurrence of Pre_T or $Post_T$ is immediately followed by an occurrence of $Post_T$. This is true iff $\sigma \in ((\neg p)^*; Pre_T; Post_T)^\omega + ((\neg p)^*; Pre_T; Post_T)^*; (\neg p)^\omega$.*

Consider the SRTD T in Figure 4.2 and the infinite sequence $\sigma = y^\omega$, where $y = \langle 0,1\rangle;\langle 0,0\rangle;\langle 1,0\rangle;\langle 1,0\rangle;\langle 0,1\rangle;\langle 0,1\rangle;\langle 1,1\rangle$. The precondition of T is the state formula $\langle 0,1\rangle$ and this state occurs again at the start of the third clock cycle. Clearly $\sigma \models_n T$ but $\sigma \not\models_o T$, since the second occurrence of the precondition along σ violates the postcondition of the diagram.

Proposition 0 *For any SRTD T, $\sigma \models_o T$ implies $\sigma \models_n T$.*

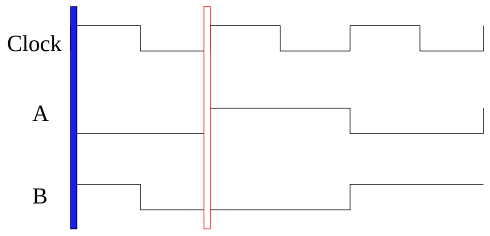

Figure 4.2: SRTD with an Overlapping Precondition

Proof.

$\sigma \models_o T$, by Definition 20, means that every occurrence of Pre_T is followed by $Post_T$. Let us now assume that $\sigma \not\models_n T$. Therefore, (by Definition 21) $\sigma \notin ((\neg p)^*; Pre_T; Post_T)^\omega$ and $\sigma \notin ((\neg p)^*; Pre_T; Post_T)^*; (\neg p)^\omega$. Clearly $\sigma \notin ((\neg p)^*; Pre_T; Post_T)^\omega$ violates the antecedent. $\sigma \notin ((\neg p)^*; Pre_T; Post_T)^*; (\neg p)^\omega$ is false, if Pre_T never holds along σ, and if Pre_T holds a finite number of times then the previous argument holds. Thus, we have a contradiction and $\sigma \models_n T$.
$\square$

4.3 Model Checking SRTDs

We first present an algorithm that translates an SRTD T with the overlapping semantics into an ω-automaton for the *negation* of the SRTD property. Next, we will present a similar translation algorithm for the non-overlapping semantics. We then present a decompositional model checking algorithm that make use of these automata.

4.3.1 Translation Algorithm for Overlapping Semantics

The algorithm constructs a ω-*NFA* that corresponds to the complement of the SRTD under the overlapping semantics. The algorithm proceeds by decomposing T into waveforms and producing sub-automata that track portions of each waveform. It consists of the following steps.

Algorithm 3

1. Construct a single deterministic automaton $\mathcal{A}_{pre}$ 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.

2. Construct a *DFA* $A_{\overline{post(i)}}$ for each signal i 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

3. Construct an *NFA* $\mathcal{A}_{\overline{T}}$ 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* $\mathcal{A}_{pre}$ at this point and if $\mathcal{A}_{pre}$ accepts it then "chooses" a postcondition *DFA*

$A_{\overline{post(i)}}$ and runs this automaton at the point where $\mathcal{A}_{pre}$ accepted and accepts if this automaton accepts. If $\mathcal{A}_{\overline{post(i)}}$ terminates (so the postcondition holds for signal i), $\mathcal{A}_{\overline{T}}$ returns to its initial state.

As a consequence of Theorem 0, an SRTD T can be represented succinctly by a $\forall$*FA* $\mathcal{A}_T$ that is obtained by complementing the acceptance condition of the *NFA* $\mathcal{A}_{\overline{T}}$.

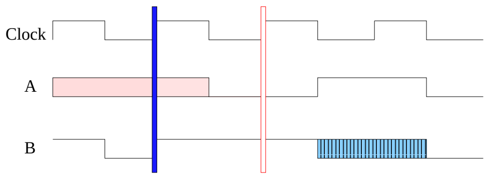

Figure 4.3: SRTD with Don't-Care Values in the Precondition

Consider the SRTD in Figure 4.3, the corresponding monolithic *DFA* $\mathcal{A}_{pre}$, and the *DFA*'s $\mathcal{A}_{post(A)}$ and $\mathcal{A}_{post(B)}$ for postcondition of waveforms A and B respectively, are shown in Figure 4.4 and Figure 4.5 respectively.

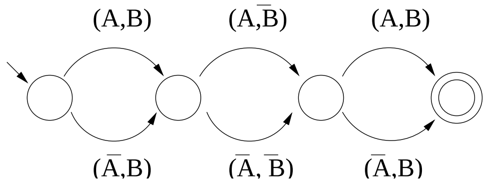

Figure 4.4: The *DFA* A_{pre} for the Overlapping Semantics

In Figure 4.5, we show the postcondition *DFA*'s, $\mathcal{A}_{post(A)}$ and $\mathcal{A}_{post(B)}$. These automata can be easily complemented to get $\mathcal{A}_{\overline{post(A)}}$ and $\mathcal{A}_{\overline{post(A)}}$.

A key attribute of the construction is the way the pauses are handled. The *NFA* shown in Figure 4.6, has a fairness constraint on state s in $\mathcal{A}_{\overline{post(A)}}$

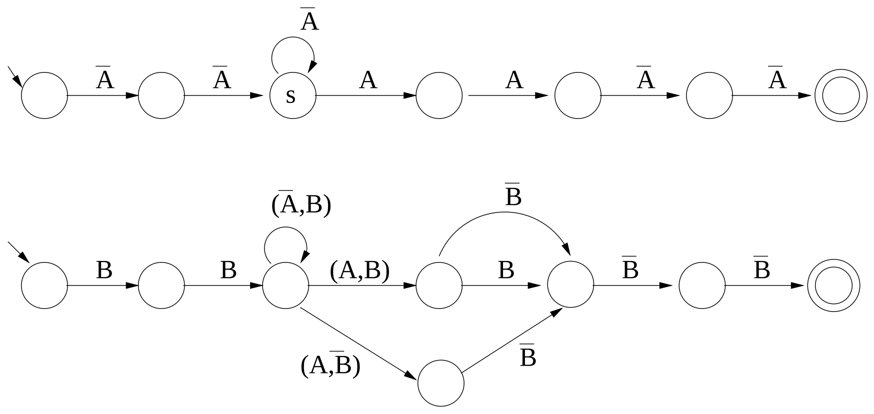

Figure 4.5: *DFA*'s for the Postcondition of Waveforms A (top) and B (bottom)

that prevents it from staying in this state forever. There are, however, no fairness constraints imposed on $\mathcal{A}_{\overline{post(B)}}$. The ω-*NFA* $\mathcal{A}_{\overline{T}}$ is shown in Figure 4.6.

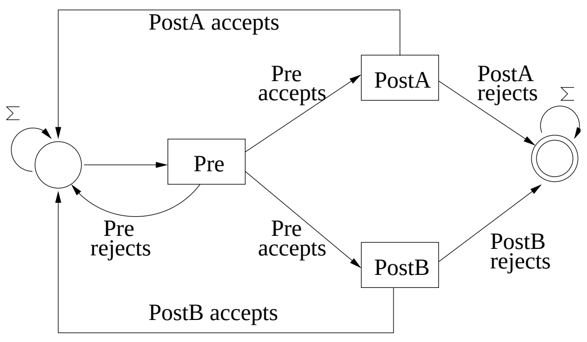

Figure 4.6: ω-*NFA* for the Complement of the SRTD in Figure 4.3

Theorem 11 (Correctness) *For any SRTD T and $x \in \mathcal{V}^\omega$, $x \models_o T$ iff $x \in \mathcal{L}(\mathcal{A}_T)$.*

Proof.

$(\Rightarrow)$ Let us assume that $x \in \mathcal{L}(\mathcal{A}_{\overline{T}})$. Thus, there is a sub-sequence $x[m..p]$, where $x[m..n] \in \mathcal{L}(\mathcal{A}_{pre})$ and $x[n+1..p] \in \mathcal{L}(\mathcal{A}_{\overline{post_i}})$, for some signal $i \in S$ and some p. $x \models_o T$ is (by Definition 20) $x \models \mathbf{G}(Pre_T \hookrightarrow \bigwedge_i Post_i)$. Clearly, since $x \models_o T$, such a sub-sequence does not exists and we have a contradiction and $x \in \mathcal{L}(\mathcal{A}_T)$.

$(\Leftarrow)$ An accepting run of $\mathcal{A}_T$ corresponds to either $\mathcal{A}_{pre}$ accepting at point x_n and for some signal $i \in S$, $(x[n+1..p] \downarrow i) \in \mathcal{L}(\mathcal{A}_{post(i)})$ or $\mathcal{A}_{pre}$ never being satisfied along x. By definition 2, $x \in \mathcal{L}(\mathcal{A}_T)$ if every run of $\mathcal{A}_T$ on x is accepting. Thus, if $\mathcal{A}_{pre}$ accepts at point x_n then $x[n+1..p]$ is accepted by automaton for the product of the $\mathcal{A}_{post(i)}$ automata. Therefore, $x \models \mathbf{G}(Pre_T \hookrightarrow \bigwedge_i Post_i)$ and $x \models_o T$.

$\square$

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.

Theorem 12 (Overlapping Complexity) *For any SRTD T and the equivalent $\forall$FA $\mathcal{A}_T$, the size of $\mathcal{A}_T$ is linear 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 $\mathcal{A}_{pre}$, is bounded by the number of clock points in the precondition, therefore $s < c$. Since each transition encodes the values of the signals at each point, the size of each transition is $O(n)$ and the number of such transitions is bounded by c. Thus, the transition size is linear in $|T|$.

The number of states s in $A_{\overline{post(i)}}$ is bounded by the number of clock points c, therefore $s \leq c$. The transitions are labeled with constant size formulae, since by construction a pause transition is dependent on at most one other signal value. Thus, the overall transition size for $A_{\overline{post(i)}}$ is bounded by c; hence, $A_{\overline{post(i)}}$ has size linear in c.

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 $|\mathcal{A}_{pre}| + n.\ |A_{\overline{post(i)}}| = n.c + n.c = O(|T|)$.

$\square$

4.3.2 Translation Algorithm for Non-overlapping Semantics

We now present the algorithm that constructs an ω-*NFA* for the complement of the SRTD property under the non-overlapping semantics.

To construct an ω-*NFA* $A_{\overline{T}}$ for the complement of the timing diagram language of T, we proceed as follows.

Algorithm 4

1. Construct a deterministic automaton A_{pre} from Pre_T 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 $(\Sigma^*; Pre_T)$ and determinizing it, so that it enters an accepting state at every point on an input string where Pre_T holds. We then eliminate outgoing edges from accepting states of this automaton. 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. [CLR90]). For *general* SRTDs, the *DFA* A_{pre} may be exponential in the length of the precondition.

2. Construct an *DFA* $A_{\overline{post(i)}}$, for each signal 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.

3. The automaton $\mathcal{A}_{\overline{T}}$ 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.

Let us consider the SRTD in Figure 4.3, the constructed *DFA* $\mathcal{A}_{pre}$

for the precondition is shown in Figure 4.7. Note that this construction of $\mathcal{A}_{pre}$ accepts the language $(\Sigma^*; Pre_T)$ as opposed to Pre_T in the overlapping semantics. There is no change in the construction of the postcondition automata shown in Figure 4.5. There is a minor change in the *NFA* $A_{\overline{T}}$ as shown in Figure 4.8.

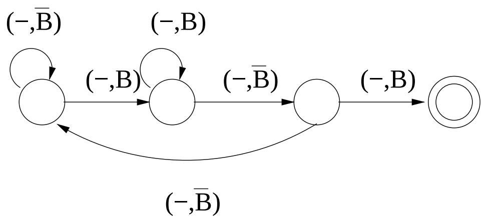

Figure 4.7: The *DFA* A_{pre} for Non-Overlapping Semantics

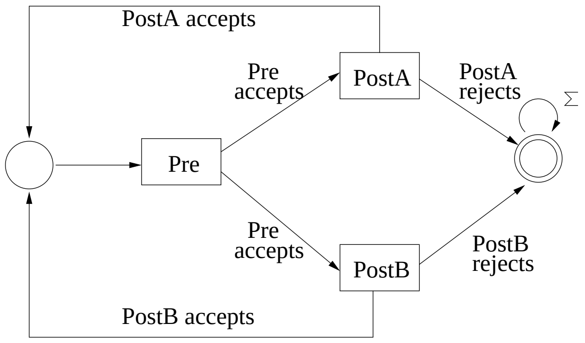

Figure 4.8: ω-*NFA* $A_{\overline{T}}$ for the Complement of the SRTD in Figure 4.3

Theorem 13 (Correctness) *For any SRTD* T *and infinite sequence* x*,* $x \models_n T$ *iff* $x \in L(\mathcal{A}_T)$*.*

Proof.

$(\Rightarrow)$ $x \models_n T$ iff (by definition 21) $x \in ((\neg p)^*; Pre_T;\ Post_T)^\omega + ((\neg p)^*; Pre_T;$

$Post_T)^*; (\neg p)^\omega$, where p is a proposition that is true at x_i iff $x[i..j] \in \mathcal{L}(Pre_T)$. Let us assume that $x \in \mathcal{L}(\mathcal{A}_{\overline{T}})$. Thus, there is a sub-sequence $((\neg p)^*; x[m..n]; x[n+1..o])$, such that $x[m..n] \in \mathcal{L}(\mathcal{A}_{pre})$ and $x[n+1..o] \in \mathcal{L}(\mathcal{A}_{\overline{post_i}})$, for some $i \in S$ and some o. Since $x \models_n T$, such a sub-sequence does not exists, thus $x \in \mathcal{L}(\mathcal{A}_T)$.

($\Leftarrow$) An accepting run of $\mathcal{A}_T$ along x corresponds to either (i) p never being satisfied along x, or (ii) there is a sequence $((\neg p)^*; x[m..n]; x[n+1..o])$ where $x[m..n] \in \mathcal{L}(\mathcal{A}_{pre})$ and for some signal $i \in S$, $(x[n+1..o] \downarrow i) \in \mathcal{L}(\mathcal{A}_{post(i)})$. By definition 2, $x \in \mathcal{L}(\mathcal{A}_T)$ iff every run of $\mathcal{A}_T$ on x is accepting. Thus, if $\mathcal{A}_{pre}$ accepts at point x_n then $x[n+1..o]$ is accepted by automaton for the product of the $\mathcal{A}_{post(i)}$ automata. Clearly $x \in ((\neg p)^*; Pre_T; Post_T)^\omega + ((\neg p)^*; Pre_T; Post_T)^*; (\neg p)^\omega$, hence $x \models_n T$.

$\square$

Theorem 14 (Non-overlapping Complexity) *For any SRTD T and the equivalent $\forall FA$ $\mathcal{A}_T$, the size of $\mathcal{A}_T$ is linear in the size of Pre_T and $Post_T$.*

Proof.

The size of an SRTD T is $n * c$, where n is the number of waveforms and c is the number of clock points. The size of the transitions in $\mathcal{A}_T$ is the sum of the length of the boolean formulas labeling the transitions. The size of $\mathcal{A}_T$ is the sum of the number of states and the transition size.

Let p be the number of clock points in Pre_T where $p < c$. The monolithic automaton $\mathcal{A}_{pre}$ must recognize $\mathcal{V}^*; Pre_T$. Each waveform segment, by Definition 16, in the precondition must either contain all don't-care values or none at all. Therefore, $\mathcal{A}_{pre}$ must either track the waveform or not, so the number of states s in $\mathcal{A}_{pre}$ is bounded by the number of clock points c. Since each transition encodes the values of the signals at each point, the size of each transition is $O(n)$ and the number of such transitions is also bounded by c. Thus, the size of $\mathcal{A}_{pre}$ is linear in $|T|$.

The number of states s in $A_{\overline{post(i)}}$ is bounded by the number of clock points c, therefore $s \leq c$. The transitions are labeled with constant size formulae, since by construction a pause transition is dependent on at most one other signal value. Thus, the overall transition size for $A_{\overline{post(i)}}$ is bounded by c; hence, $A_{\overline{post(i)}}$ has size linear in c.

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 $|\mathcal{A}_{pre}| + n.\ |A_{\overline{post(i)}}|$. Therefore the size of $\mathcal{A}_T$ is is linear in the size of T.

□

We can relax conditions 1 and 2 in the definition of a well-formed SRTD (Definition 16) to obtain a *general* SRTD; that is we allow don't-care transitions and arbitrary don't-care values in the precondition. The size of the resulting $\forall FA$ $\mathcal{A}_T$ for a general SRTD T is linear in the size of the postcondition but is exponential in the size of the precondition.

Theorem 15 (Complexity for General SRTDs) *For a general SRTD T and the equivalent $\forall FA$ $\mathcal{A}_T$, the size of $\mathcal{A}_T$ is linear in the size of $Post_T$ and exponential in the size of Pre_T.*

Proof.

Let p be the number of clock points in Pre_T where $p < c$. $\mathcal{A}_{pre}$ must recognize the first occurrence and all subsequent non-overlapping occurrences of Pre_T. Hence $\mathcal{A}_{pre}$ must remember the actual values seen on the signals that have don't-care values. Overlapping don't-care transitions introduce a similar blowup since $\mathcal{A}_{pre}$ must now remember at each state whether each of the don't-care transitions has made the transition to the new value. Therefore, the number of states s in $\mathcal{A}_{pre}$, is bounded by $|\mathcal{V}|^{p+1}$. Since each transition encodes the values of the signals at each point, the size of each transition is $O(n)$ and the number of such transitions is bounded by $|\mathcal{V}|^{p+1}$. Thus, the size of $\mathcal{A}_{pre}$ is exponential in $|T|$.

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 $|\mathcal{A}_{pre}| + n.\ |A_{\overline{post(i)}}|$. Therefore the size of $\mathcal{A}_T$ is is exponential in the size of Pre_T and linear in the size of $Post_T$. $\square$

4.3.3 Model Checking

We can use the constructed *NFA* $\mathcal{A}_{\overline{T}}$ described in the previous Section directly in the automata-theoretic model checking. Recall that in the language containment paradigm, one model checks a system M with respect to a property P by checking $\mathcal{L}(M) \subseteq \mathcal{L}(P)$, which is equivalent to checking that $\mathcal{L}(M) \cap \neg\mathcal{L}(P) = \emptyset$.

In both the overlapping and non-overlapping cases, we can use the respective translation algorithms to obtain an *NFA* $\mathcal{A}_{\overline{T}}$ for the negation of the SRTD T which is linear in size of T. This yields a model checking algorithm which is effectively linear in both the size of the system and the SRTD T.

Theorem 16 (Model Checking Complexity) *For a transition system M and an SRTD T, the time complexity of model checking, under either the overlapping semantics and non-overlapping semantics, is linear in the size of M and T.*

Proof.

We know, by Theorems 12 and 14, that the constructed *NFA* $\mathcal{A}_{\overline{T}}$ for the negation of T is linear in the size of T. Therefore, by the results in [EL85a, EL85b], we know that checking $\mathcal{L}(M) \cap \mathcal{L}(\mathcal{A}_{\overline{T}}) = \emptyset$ is linear in the size of M and T.

$\square$

For general SRTDs, we have the following Theorem.

Theorem 17 (Model Checking Complexity for General SRTDs) *For a transition system M and a general SRTD T, the time complexity of model checking, under either the overlapping semantics and non-overlapping semantics, is linear in the size of M and exponential in the size of T.*

Proof.

We know, by Theorem 15, that the constructed *NFA* $\mathcal{A}_{\overline{T}}$ for the negation of general SRTD T may be exponential in the size of Pre_T. Thus, checking $\mathcal{L}(M) \cap \mathcal{L}(\mathcal{A}_{\overline{T}}) = \emptyset$ is linear in the size of M and $Post_T$ and exponential in the size of Pre_T.

4.3.4 Decompositional Model Checking

Decompositional model checking is an alternative way to use the constructed automata that exploits the conjunctive nature of the $\forall$*FA* $\mathcal{A}_T$. The property represented by the SRTD T is $\mathbf{G}(Pre_T \hookrightarrow Post_T)$. Since $Post_T = \bigwedge_i \mathcal{A}_{\overline{post(i)}}$, this property can be decomposed into the conjunction of individual checks $\mathbf{G}(\mathcal{A}_{pre} \hookrightarrow \mathcal{A}_{\overline{post(i)}})$. 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 $\mathcal{A}_{pre}$ accepts, followed by a computation where $A_{\overline{post(i)}}$ accepts. Hence, model-checking can be done with this decomposed representation of the postcondition.

Theorem 18 (Overlapping Decompositional Model Checking) *For a transition system M and an SRTD T, the time complexity of decompositional model checking, under the overlapping semantics, is linear in the size of M and quadratic in the size of T.*

Proof.

The $\forall FA$ $\mathcal{A}_T$, corresponding to T under either semantics, is the automaton for $\mathbf{G}(\mathcal{A}_{pre} \hookrightarrow \bigwedge_i \mathcal{A}_{\overline{post(i)}})$ where $\mathcal{A}_{pre}$ is the automaton for Pre and each $\mathcal{A}_{\overline{post(i)}}$ is the automaton for the postcondition segment of waveform i. The problem of checking $M \models \mathcal{A}_T$ can be decomposed into $\bigwedge_i M \models \mathcal{A}_i$, where $\mathcal{A}_i$ is the automaton for $\mathbf{G}(\mathcal{A}_{pre} \hookrightarrow \mathcal{A}_{\overline{post(i)}})$. We can check $M \models \mathcal{A}_i$ in time linear in the size of M and $\mathcal{A}_i$ which, by Theorem 12, is $O(|M|.|T|)$. There $|S|$ such verification tasks, thus the time complexity of model checking $M \models_o T$ decompositionally is $O(|M|.|T|^2)$.

□

Theorem 19 (Non-overlapping Decompositional Model Checking) *For a transition system M and an SRTD T, the time complexity of decompositional model checking, under the non-overlapping semantics, is linear in the size of M and quadratic in the size of T.*

Proof.

Let $\mathcal{A}_i$ be the automaton that accepts iff every sub-sequence accepted by the

non-overlapping automaton $\mathcal{A}_{pre}$ is followed by a sub-sequence that is accepted by $\mathcal{A}_{\overline{post(i)}}$. The size of $\mathcal{A}_i$ is linear in $|T|$ (by Theorem 14). The complexity of checking, $\mathcal{L}(M) \cap \mathcal{L}(\mathcal{A}_i) = \phi$, is linear in the size M and $\mathcal{A}_i$ (by Theorem 16. There are $|S|$ such checks; hence the complexity of model checking $M \models_n T$ decompositionally is $O(|M|.|T|^2)$.
□

Theorems 18 and 19 shows that decompositional model checking is more expensive (quadratic versus linear) than model checking in the size of the SRTD. However, efficiency with respect to space is often of more practical interest. In our experiments, that are presented in the following Section, we found that the decompositional approach does indeed yield non-trivial savings in space. Thus, we feel justified in trading time versus space in this manner. This topic will be addressed in detail in the following section.

4.4 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* to verify two systems. The first is a synchronous master-slave memory system and the second is the Lucents' PCI Interface Core.

4.4.1 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

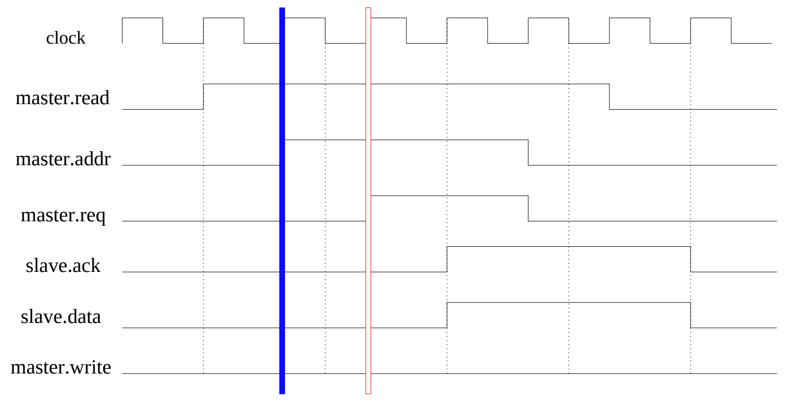

Figure 4.9: SRTD for the Read Transaction

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.

We verified that this system satisfied both read (see Figure 4.9) and write (see Figure 4.10) memory transactions formulated as SRTDs, with the overlapping semantics. The SRTDs were created with the RTDT editor and the translator was used to generate the corresponding *COSPAN* descriptions. We used *COSPAN* to model check the system with respect to these descriptions.

Recall that a monolithic translation of an SRTD yields an ω-*NFA* that is essentially the product (intersection) of the *DFA*'s for each waveform. In order to compare our decompositional algorithms with monolithic algorithms, we did the verification checks both decompositionally and monolithically. In Table 4.1, *read(M)* corresponds to the verification check on the master-slave design and the monolithic automaton for the read SRTD while *read(D)* corresponds

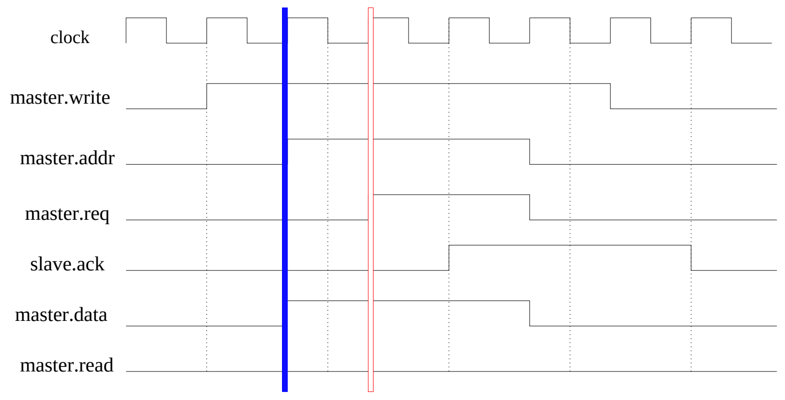

Figure 4.10: SRTD for the Write Transaction

to the verification check done on the master-slave design and automata for a single waveform. The numbers in Table 4.1 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 [LP85], 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.

Design	Number of BDD variables	Average BDD size	Average Space (MBytes)	Average Time (seconds)
read (D)	95	13433	0.86	0.32
read (M)	205	22079	1.46	3.19
write (D)	95	11542	0.86	0.31
write (M)	205	21915	1.45	2.51

Table 4.1: Verification Statistics for Master-Slave Design

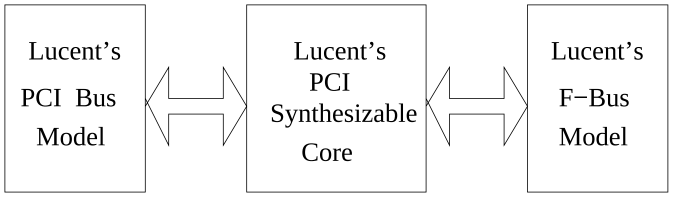

Figure 4.11: Block Diagram of Lucent's F-Bus with PCI Core

4.4.2 Lucent's PCI Synthesizable 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 implement a complete PCI interface. The PCI Interface Synthesizable Core is designed to be fully compatible with the PCI Local Bus specification [Gro95]. 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.

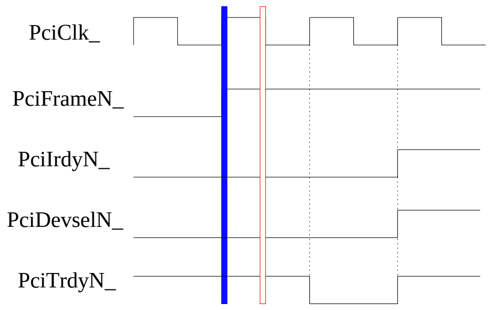

Figure 4.12: An SRTD Burst Property for the PCI Bus

We used Lucent's PCI Bus Functional Model shown in Figure 4.11, which is a sophisticated simulation environment that was developed to test the Synthesizable Core for functionality and compliance with the PCI specification [Gro95]. 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.

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

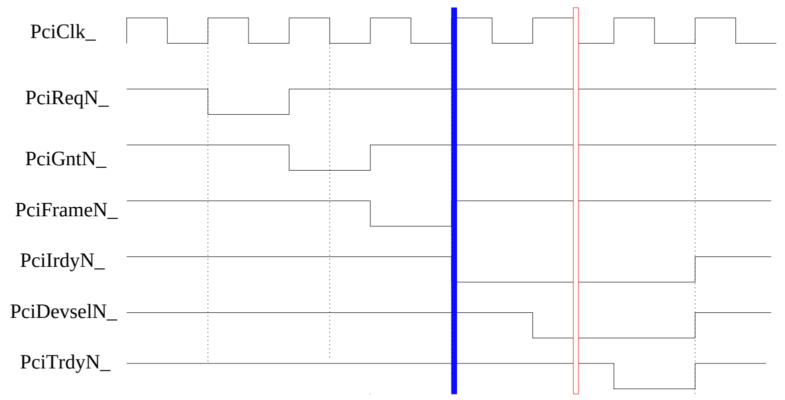

Figure 4.13: SRTD for the Non-burst Transaction of the PCI Bus

protocol are controlled by three signals *PciFrameN*, *PciIrdyN* and *PciTrdyN*. The master of the bus drives the signal *PciFrameN* to indicate the beginning and end of a transaction. *PciIrdyN* is asserted by the master to indicate that it is ready to transfer data. Similarly the target uses *PciTrdyN* to signal that it is ready for data transfer. Data is transferred between master and target on each rising clock edge for which both *PciIrdyN* and *PciTrdyN* are asserted. We verified that the PCI Core satisfied several timing diagram properties for both the burst and non-burst modes. We formulated the properties as SRTDs by looking at the actual timing diagrams that occurred in the PCI specification [Gro95] and the PCI Core User's Manual [BL96]. Figure 4.13 and Figure 4.12 are properties that we checked for the non-burst mode and burst mode respectively.

The verification was done both monolithically and decompositionally and Table 4.2 presents the verifications statistics. In Table 4.2, 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

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

Table 4.2: Verification Statistics for Lucent's Synthesizable PCI Core

the property. The suffix *(D)* refers to the average over the individual decompositional verification checks on the abstracted system and the automata for each waveform. Table 4.2 shows a savings of up to 30% in BDD size and corresponding savings 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 the decompositional approach is more space efficient than a monolithic one.

4.5 Related Work and Conclusions

Various researchers have investigated the formal use of timing diagrams. Damm et al. introduced a timing diagram notation, called Symbolic Timing Diagrams (STD's) [DJS94], that have a formal semantics. They [DJS94, Fey94, FS96, FJ97, BW98a] provide algorithms that translate STD's into various temporal logics, like CTL, TPTL [AH94] and a first order temporal logic TL [DJS94]. They have applied their algorithms successfully to a number of case studies [DHKS94, BW98b]. Unlike our work, their translation algorithms are monolithic and in general results in an exponential translation. Moreover, STD's are asynchronous in nature and cannot explicitly tie events to the clock. Fisler [Fis96, Fis97] provides a procedure to decide regular language containment of non-regular timing diagrams, but the model checking algorithms have a high complexity (PSPACE). Fisler's diagrams, like RTDs, can express synchronous properties, but the result is a visually cluttered diagram with unnecessary added complexity.

Cerny et al. present a procedure [KC98] for verifying whether the behavior of a set of action diagrams [CBGK98] (timing diagrams) is consistent; they do not consider infinite behavior. They [JC98] use constraint logic programming to check if a system satisfies finite action diagram specifications. Amon et al. [ABHL97, ABL98] use Presburger formulas to determine whether the delays and guarantees of an implementation satisfy constraints specified as a timing diagram. This work uses a commercial timing diagrams editor, called *Timing Designer* [KM97], 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.

Antoine and Le Goff [AL92] present a syntax and semantics of synchronous timing diagrams and translate them into CTL^* formulae; they only consider diagrams without any temporal ambiguity. Boriello [Bor92a, Bor92b] 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 [Thu96, RMM+93, Cin93] have formalized timing diagrams and translated them to other formalisms (interval logics, trigger graphs etc.). Formal notions of timing diagrams have also proved to be useful in test generation and logic synthesis (cf. [Tie92, GGL+95, Lut98, FS96]).

In contrast, for SRTDs, we have presented decompositional, efficient algorithms for model checking, which has time complexity that is linear in the 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 SRTDs suffices to express common timing properties, and is expressive enough for industrial verification needs.

Chapter 5

Compositional Reasoning with SRTDs

5.1 Introduction

Compositional reasoning [dRdBH$^+$99] – reduces 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.

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 Chapter, we first present a new rule for assume-guarantee reasoning, which generalizes several earlier rules (cf. [Pnu85, AL95, AH96, McM99, NT00]), 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 an SRTD. 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 SRTDs 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 the timing diagram analysis tool, RTDT [AEKN00, AEKN01], which uses the tool *COSPAN* [HHK96] 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 Chapter is as follows: we describe our new rule and prove its soundness and completeness in Section 5.2. The theory behind the application of this rule to timing diagrams is presented in Section 5.3. Our experiments with applying this rule are described in Section 5.4. We conclude the Chapter with a description of related work in Section 5.5.

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.

5.2.1 Preliminaries

Definition 22 (***V-state***) *For a non-empty set of typed variables V, an assignment of values to variables in V is called a V-state.*

Definition 23 (V-*sequence*) *A V-sequence $x = x_0, x_1, \ldots$ is a non-empty sequence (finite or infinite) of V-states.*

The length of a V-*sequence* x, written as $|x|$, is the number of states in 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 a set of variables V is a set of finite or infinite sequences of V-states.

Definition 24 (Satisfaction) *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 finite or infinite V-sequence satisfies L. Thus, for L_1 and L_2 over V, $[L_1 \Rightarrow L_2]$ denotes $L_1 \subseteq L_2$.

Definition 25 (Process) *A process P is specified by a tuple (V, I, R, F) where*

- *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 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. The variables $(V^m)'$, which are in 1-1 correspondence with V^m, represent values for V^m in the next state.*
- *$F(V)$ is a fairness condition.*

Definition 26 (Process Execution) *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 executions of a process P can be defined by the LTL formula $(I \wedge \mathsf{G}(R))$, interpreted over V-sequences. The set of finite executions is denoted by *finexec*(P).

Definition 27 (Language of a Process) *The language of a process P, $\mathcal{L}(P)$, is the set of finite executions of P together with those infinite executions of P that satisfy F. Thus $\mathcal{L}(P)$ can be expressed by the LTL formula $(I \wedge \mathsf{G}(R) \wedge F)$.*

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 Chapter, 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.

Definition 28 (Implements) *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.

Definition 29 (Process Composition) *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*

- $V = V_1 \cup V_2$, $V^p = V_1^p \cup V_2^p$ *and* $V^i = V_1^i \cup V_2^i$

- $I = I_1 \wedge I_2$
- $R = R_1 \wedge R_2$
- $F = F_1 \wedge F_2$

Definition 30 (Process Disjunction) *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*

- $V = V_1 \cup V_2 \cup \{c\}$, $V^p = V_1^p \cup V_2^p \cup \{c\}$ *and* $V^i = V_1^i \cup V_2^i$. *c is a private variable that serves to choose initially between the two processes.*
- $I = (c \wedge I_1) \vee (\neg c \wedge I_2)$
- $R = (c' = c) \wedge ((c \wedge R_1) \vee (\neg c \wedge R_2))$
- $F = (\mathsf{FG}(c) \wedge F_1) \vee (\mathsf{FG}(\neg c) \wedge F_2)$

The following Lemmas summarizes the properties of these constructions needed for the proofs in following Sections.

Lemma 1 *For processes P_1, P_2, $[\mathit{finexec}(P_1//P_2) \equiv \mathit{finexec}(P_1) \wedge \mathit{finexec}(P_2)]$*

Proof.

$$
\begin{array}{ll}
 & x \in \mathit{finexec}(P_1//P_2) \\
\equiv & \quad \text{(by Definition 26 (execution))} \\
 & I(P_1//P_2)(x_0) \wedge (\forall i : i+1 < |x| : R(P_1//P_2)(x_i, x_{i+1})) \\
\equiv & \quad \text{(by Definition 29 (composition))} \\
 & (I(P_1)(x_0) \wedge I(P_2)(x_0)) \wedge \\
 & (\forall i : i+1 < |x| : R(P_1)(x_i, x_{i+1}) \wedge R(P_2)(x_i, x_{i+1})) \\
\equiv & \quad \text{(rearranging the terms)}
\end{array}
$$

$(I(P_1)(x_0) \wedge (\forall i : i+1 < |x| : R(P_1)(x_i, x_{i+1}))) \wedge$

$(I(P_2)(x_0) \wedge (\forall i : i+1 < |x| : R(P_2)(x_i, x_{i+1})))$

$\equiv$ (by Definition 26 (execution))

$(x \in \mathit{finexec}(P_1)) \wedge (x \in \mathit{finexec}(P_2))$

□

Lemma 2 *For processes* P_1, P_2*,* $[\mathcal{L}(P_1//P_2) \equiv \mathcal{L}(P_1) \wedge \mathcal{L}(P_2)]$

Proof.

$\mathcal{L}(P_1//P_2)$

$\equiv$ (by Definition 27 (language))

$I(P_1//P_2)(v) \wedge \mathsf{G}(R(P_1//P_2)(v, v')) \wedge F(P_1//P_2)(v)$

$\equiv$ (by Definition 29 (composition))

$I(P_1)(v) \wedge I(P_2)(v)) \wedge \mathsf{G}(R(P_1)(v, v')) \wedge \mathsf{G}(R(P_2)(v, v')) \wedge$

$(F(P_1)(v) \wedge F(P_2)(v))$

$\equiv$ (rearranging the terms)

$(I(P_1)(v) \wedge \mathsf{G}(R(P_1)(v, v')) \wedge F(P_1)(v)) \wedge$

$(I(P_2)(v) \wedge \mathsf{G}(R(P_2)(v, v')) \wedge F(P_2)(v))$

$\equiv$ (by Definition 26 (language))

$\mathcal{L}(P_1) \wedge \mathcal{L}(P_2)$

□

Lemma 3 *For processes* P_1, P_2*,* $[\mathcal{L}^{\mathcal{O}}(P_1//P_2) \equiv \mathcal{L}^{\mathcal{O}}(P_1) \wedge \mathcal{L}^{\mathcal{O}}(P_2)]$

Proof. Let V_1^p and V_2^p be the private variables of P_1 and P_2, and $V^p = V_1^p \cup V_2^p$ be the private variables of $P_1//P_2$.

$\mathcal{L}^{\mathcal{O}}(P_1//P_2)$

$\equiv$ (by Definition 27 (language), Definition of $\mathcal{L}^{\mathcal{O}}$)

$(\exists V^p(P_1//P_2) : (I(P_1//P_2)(v) \wedge \mathsf{G}(R(P_1//P_2)(v,v')) \wedge$
$F(P_1//P_2)(v)))$

$\equiv$ (by Definition 29 (composition))

$(\exists V^p(P_1//P_2) : (I(P_1)(v) \wedge I(P_2)(v)) \wedge (\mathsf{G}(R(P_1)(v,v')) \wedge$
$\mathsf{G}(R(P_2)(v,v'))) \wedge (F(P_1)(v) \wedge F(P_2)(v)))$

$\equiv$ (rearranging the terms, Definition of composition)

$(\exists V_1^p(P_1) : (I(P_1)(v) \wedge \mathsf{G}(R(P_1)(v,v')) \wedge F(P_1)(v))) \wedge$
$(\exists V_2^p(P_2) : (I(P_2)(v) \wedge \mathsf{G}(R(P_2)(v,v')) \wedge F(P_2)(v)))$

$\equiv$ (by Definition 27 (language), Definition of $\mathcal{L}^{\mathcal{O}}$)

$\mathcal{L}^{\mathcal{O}}(P_1) \wedge \mathcal{L}^{\mathcal{O}}(P_2)$

□

Lemma 4 *For processes* P_1, P_2, $[(\exists\{c\} : \mathcal{L}(P_1 + P_2)) \equiv \mathcal{L}(P_1) \vee \mathcal{L}(P_2)]$.

Proof.

$(\exists\{c\} : \mathcal{L}(P_1 + P_2))$

$\equiv$ (by Definition 27 (language))

$(\exists\{c\} : I(P_1 + P_2)(v) \wedge \mathsf{G}(R(P_1 + P_2)(v,v')) \wedge F(P_1 + P_2)(v))$

$\equiv$ (by Definition 30 (disjunction))

$(\exists\{c\} : ((c \wedge I(P_1)(v) \vee (\neg c \wedge I(P_2)(v))) \wedge$
$((c' = c) \wedge ((c \wedge R(P_1)(v,v'))) \vee (\neg c \wedge R(P_2)(v,v'))$
$\wedge((\mathsf{FG}(c) \wedge F(P_1)(v)) \vee (\mathsf{FG}(\neg c) \wedge F(P_2)(v))$

$\equiv$ (rearranging the terms, logic)

$(\exists\{c\} : (c \vee (\neg c)) \wedge (c' = c) \wedge (\mathsf{FG}(c) \vee \mathsf{FG}(\neg c))) \wedge$
$(I(P_1)(v) \wedge R(P_1)(v,v') \wedge F(P_1)(v)) \vee$
$(I(P_2)(v) \wedge R(P_2)(v,v') \wedge F(P_2)(v))$

$\equiv$ (by Definition 27 (language))

$\mathcal{L}(P_1) \vee \mathcal{L}(P_2)$

$\square$

Definition 31 (Closure) *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 $x[0..i]; y \in L$.*

cl has the following properties.

Theorem 20 (Closure Properties) *([AS85]) Given languages L_1 and L_2,*

a. *L_1 specifies a safety property if and only if $cl(L_1) = L_1$.*

b. $[L_1 \Rightarrow cl(L_1)]$

c. $[cl(cl(L_1)) \Rightarrow cl(L_1)]$

d. $[cl(L_1 \cup L_2) = cl(L_1) \cup cl(L_2)]$

For any process P, there is a process $CL(P)$ such that the property $[\mathcal{L}^{\mathcal{O}}(CL(P)) \equiv cl(\mathcal{L}^{\mathcal{O}}(P))]$ holds. If P is finite-state, $CL(P)$ is formed from P by changing the fairness condition of P to *true*.

Definition 32 (Closure Process) *For any finite state process $P = (V, I, R, F)$, let $CL(P)$ be the process (V', I', R', F') where $V' = V$, $I' = I$, $R' = R$ and $F' = true$.*

Definition 33 (Non-blocking) *A process Q does not block process P iff*

- *Any initial state of P can be extended to an initial state of $P//Q$, and*

- *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 P is non-blocking if and only if, from any reachable state, P can make a transition on any external input.

Definition 34 (Machine Closure) *A process is machine closed iff every finite execution can be extended to an infinite fair execution.*

Machine closure indicates that it is possible at any point to break away from an infinite execution to one that is fair. A process that is machine closed satisfies the CTL property AGE(fair).

Lemma 5 *Given a finite state process $P = (V, I, R, F)$,*
$[\mathcal{L}^{\mathcal{O}}(CL(P)) \equiv cl(\mathcal{L}^{\mathcal{O}}(P))]$.

Proof. ($\Rightarrow$) Consider a sequence $x \in \mathcal{L}^{\mathcal{O}}(CL(P))$. Now, assume that $x \notin cl(\mathcal{L}^{\mathcal{O}}(P))$, so, for some i, $x[0..i]$ can not be extended to a sequence in $\mathcal{L}^{\mathcal{O}}(P)$. But, by construction (Definition 32), any execution of P is also an execution of $CL(P)$. Therefore, $x[0..i]$ can not be extended to a sequence in $\mathcal{L}^{\mathcal{O}}(CL(P))$. Thus, we have a contradiction and $x \in cl(\mathcal{L}^{\mathcal{O}}(P))$.
($\Leftarrow$) Consider a sequence x such that $x \in cl(\mathcal{L}^{\mathcal{O}}(P))$ which implies, by Definition 31, that, for all i, $x[0..i]$ can be extended into a sequence that is in $\mathcal{L}^{\mathcal{O}}(P)$. Thus, by Definition 25, x is an execution of P and, therefore, is an execution of $CL(P)$. $\square$

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.

5.2.2 Compositional Reasoning Rules

In compositional reasoning one avoids reasoning directly about a system, that is composed of many sub-components operating in parallel, by decomposing the property and attempting to prove that the system sub-components satisfy the sub-properties in a systematic manner. The following is an example of a "non-circular" compositional proof rule.

$$\frac{P_1 \models T_1 \qquad P_2 \models T_2}{P_1 // P_2 \models T_1 \wedge T_2}$$

These "non-circular" proof rules often do not work if the components are tightly coupled, since P_1 may satisfy T_1 only in the presence of P_2. For instance, in the following example, both $P_1 \| P_2 \models T_1 \wedge T_2$ and $P_2 \models T_2$ hold. However, in the absence of P_1, $P_1 \models T_1$ does not hold since input y is unconstrained.

Example 1 (Assume-Guarantee)

```
Process P1                           Process P2
  var x:  boolean;                     var y:  boolean;
  initially x=true;                    initially y=true;
  transition x'=y                      transition y'=true
end P1                               end P2


property T1:  Always(x)
property T2:  Always(y)
```

As illustrated in Example 1, it is unlikely that a system will satisfy any interesting property outside of its intended environment. Hence, the environment must be constrained/specified to some extent. Towards this end, several so-called "circular" proof rules have been proposed, of which this is an example. In the following rule [AH96], both the implementation and specification are processes. A *trace* is a sequence of states or events, and the semantics of a process is a set of traces. Parallel composition (//) is the intersection of the trace sets and the implements relation ($\models$) is trace set inclusion.

$$\frac{\begin{array}{c} P_1//T_2 \models T_1 \\ P_2//T_1 \models T_2 \end{array}}{P_1//P_2 \models T_1//T_2}$$

This rule is sound for safety properties (i.e. for finite computations), but the soundness depends on a number of additional semantic assumptions:

- The processes must be non-blocking.
- The processes must have non-empty trace sets.
- The output variables of the processes must be disjoint.

In Example 1, one can show that $P_2//T_1 \models T_2$ and $P_2//T_1 \models T_2$, and conclude, by the soundness of the rule, that $P_1//P_2 \models T_1//T_2$. This rule is, however, *unsound* for liveness properties. To see this, consider the following instantiation.

Example 2 (Liveness)

```
Process P1                          Process P2
  var x:  boolean;                   var y:  boolean;
  initially x=true or x=false;        initially y=true or y=false;
  transition x'=y                    transition y'=x
end P1                              end P2


property T1:  eventually(x)
property T2:  eventually(y)
```

Although both hypotheses, $P_1//T_2 \models T_1$ and $P_2//T_1 \models T_2$ 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. [AL95, AH96]) use the safety closure of one of the properties in the hypothesis as shown below.

$$\frac{\begin{array}{c} P_1//T_2 \models T_1 \\ P_2//CL(T_1) \models T_2 \end{array}}{P_1//P_2 \models T_1//T_2}$$

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.

Another issue concerning such rules is completeness. Namjoshi and Trefler [NT00] have explored completeness and have shown that many of these circular proof rules are indeed incomplete. The following example, taken from the paper [NT00], can be used to show that previous rules are not complete.

Example 3 (Completeness)

```
Process P1                          Process P2
  var l1,r1:  boolean;                var l2,r2:  boolean;
  initially l1=true, r1=true;         initially l2=true, r2=true;
  transition r1'=l1                   transition r2'=l2
  transition l1'=r2                   transition l2'=r1
end P1                              end P2


property T1:  Always(l1)
property T2:  Always(l2)
```

In the above example, $P_1//P_2 \models T_1//T_2$ holds, However, the hypothesis $P_1//T_2 \models T_1$ does not hold since variable r_2 is now unconstrained and l_1 may be assigned the value *false*. The second hypothesis also fails to hold by a symmetric argument.

We will now present a new assume-guarantee style proof rule that is both sound and complete and can be applied uniformly to both safety and liveness properties. For simplicity, we present this rule for the composition of two processes; it can be easily extended to apply to any finite composition.

Proof 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 $P_2//CL(T) \models (T + Q_1 + Q_2)$

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

We will first prove some preliminary lemmas that will be used later in the proof of the soundness and completeness of the above rule. In the following proof, let W be the private variables of $Q_1//Q_2$.

Lemma 6 $[\mathit{finexec}(P_1//P_2) \Rightarrow (\exists W : \mathit{finexec}(Q_1//Q_2))]$

Proof. A sequence x is in $\mathit{finexec}(P, k)$ iff $x[0..k]$ is a finite execution of process P. The property that process Q does not block P can be stated as follows: (i) $[I(P) \Rightarrow (\exists V^m(Q)\backslash V^m(P) : I(P//Q))]$, and (ii) for any $k > 0$, $[\mathit{finexec}(P, k) \wedge \mathit{finexec}(Q, k-1) \Rightarrow (\exists (V^m(Q)\backslash V^m(P))(k) : \mathit{finexec}(P//Q, k))]$, where x satisfies $(\exists V(k) : L)$ iff there is a sequence y in L, of the same length as x, that differs from x only in the values of the V-variables at the kth position.

The proof is by induction on the length of executions. Let $W = V^p(Q_1//Q_2)$, $U = V^m(Q_1)\backslash V^m(P_2)$ and $W_2 = V^p(Q_2)$.

Base case:

$\mathit{finexec}(P_1//P_2, 0)$

$\Leftrightarrow$ (definitions)

$I(P_1) \wedge I(P_2)$

$\Rightarrow$ (non-blocking from C0)

$(\exists U : I(P_2//Q_1)) \wedge I(P_1)$

$\Rightarrow$ (by C1)

$(\exists U : (\exists W_2 : I(Q_2))) \wedge I(P_1)$

$\Rightarrow$ (as U is disjoint from $V^m(Q_2)$ by C0)

$(\exists W_2 : I(Q_2)) \wedge I(P_1)$

$\Rightarrow$ (W_2 is a set of private variables)

$(\exists W_2 : I(P_1//Q_2))$

$\Rightarrow$ (by monotonicity of composition and C1)

$(\exists W : I(Q_2//Q_1))$

$\Leftrightarrow$ (definitions)

$(\exists W : \mathit{finexec}(Q_1//Q_2, 0))$

Inductive case: $k > 0$ and the result holds for $k - 1$ by assumption.

$\mathit{finexec}(P_1//P_2, k)$

$\Rightarrow$ (inductive hypothesis)

$\mathit{finexec}(P_1//P_2, k) \wedge (\exists W : \mathit{finexec}(Q_1//Q_2, k-1))$

$\Rightarrow$ (W is a set of private variables)

$(\exists W : \mathit{finexec}(P_1, k) \wedge \mathit{finexec}(P_2, k) \wedge \mathit{finexec}(Q_1, k-1)$

$\wedge\, \mathit{finexec}(Q_2, k-1))$

$\Rightarrow$ (non-blocking from C0)

$(\exists W : (\exists U(k) : \mathit{finexec}(P_2//Q_1, k)) \wedge \mathit{finexec}(P_1, k))$

$\Rightarrow$ (by C1)

$(\exists W : (\exists U(k) : (\exists W_2 : \mathit{finexec}(Q_2, k))) \wedge \mathit{finexec}(P_1, k))$

$\Rightarrow$ (U is disjoint from $V^m(Q_2)$ by C0, and $V^e(Q_2)$ is unconstrained)

$(\exists W : \mathit{finexec}(Q_2, k) \wedge \mathit{finexec}(P_1, k))$

$\Rightarrow$ (by C1)

$(\exists W : \mathit{finexec}(Q_1//Q_2, k))$

□

Theorem 21 (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 [AS85], 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 $[\mathcal{L}(P_1//P_2) \wedge cl(\mathcal{L}^{\mathcal{O}}(T)) \Rightarrow \mathcal{L}^{\mathcal{O}}(T)]$

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.

$$
\begin{array}{ll}
 & \mathit{finexec}(P_1//P_2) \\
\Rightarrow & \quad (\text{ by Lemma 6 }) \\
 & (\exists W : \mathit{finexec}(Q_1//Q_2)) \\
\Rightarrow & \quad (\text{ as } cl(\mathcal{L}(P)) \text{ includes } \mathit{finexec}(P)\text{ }) \\
 & (\exists W : cl(\mathcal{L}(Q_1//Q_2))) \\
\Rightarrow & \quad (\text{ by C2; monotonicity of } cl\text{ }) \\
 & (\exists W : cl(\mathcal{L}^{\mathcal{O}}(T))) \\
\Rightarrow & \quad (\text{ } W \text{ contains private variables not occurring in } T\text{ }) \\
 & cl(\mathcal{L}^{\mathcal{O}}(T))
\end{array}
$$

Next, we show the liveness part.

$$
\begin{array}{ll}
 & \mathcal{L}(P_1) \wedge \mathcal{L}(P_2) \wedge cl(\mathcal{L}^{\mathcal{O}}(T)) \\
\Rightarrow & \quad (\text{ by Lemma 5 })
\end{array}
$$

$\mathcal{L}(P_1) \wedge \mathcal{L}(P_2) \wedge \mathcal{L}^{\mathcal{O}}(CL(T))$

$\Rightarrow$ (by condition C3)

$\mathcal{L}(P_1) \wedge \mathcal{L}(P_2) \wedge \mathcal{L}^{\mathcal{O}}(T + Q_1 + Q_2)$

$\Rightarrow$ (by Lemma 4; $W \cup U \cup \{c\}$ consists of private variables)

$(\exists W \cup U \cup \{c\} : \mathcal{L}(P_1) \wedge \mathcal{L}(P_2) \wedge (\mathcal{L}(T) \vee \mathcal{L}(Q_1) \vee \mathcal{L}(Q_2)))$

$\Rightarrow$ (distributing $\wedge$ over $\vee$; Lemma 3 and condition C1)

$(\exists W \cup U \cup \{c\} : \mathcal{L}(T) \vee \mathcal{L}^{\mathcal{O}}(Q_1//Q_2))$

$\Rightarrow$ (distributing $\exists$ over $\vee$; condition C2)

$(\exists W \cup U \cup \{c\} : \mathcal{L}(T)) \vee (\exists W \cup U \cup \{c\} : \mathcal{L}^{\mathcal{O}}(T))$

$\Rightarrow$ ($W \cup \{c\}$ consists of private variables not in T)

$\mathcal{L}^{\mathcal{O}}(T)$

$\square$

Theorem 22 (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.

$\square$

Theorem 22 shows that the proof rule is complete for processes P_1 and P_2 that are non-blocking and have disjoint interface variables. Theorem 23 claims that the rule is complete for arbitrary processes. To show $P_1//P_2 \models T$ for arbitrary P_1, P_2 and T, the proof proceeds as follows.

- Syntactically transform processes P_1, P_2, and T into P'_1, P'_2, and T' such that (i) $P_1//P_2 \models T$ iff $P'_1//P'_2 \models T'$, and (ii) P'_1, P'_2 are non-blocking and have disjoint interface variables.
- Apply Theorem 22 to P'_1, P'_2, and T' which, by construction, satisfy the hypotheses for the theorem.

Thus, Theorem 23 shows that the rule is complete up to a syntactic transformation. This is a broader definition of completeness, which can be easily converted to the narrower syntactic sense by adding the transformation as an axiom. The proof alluded to above shows that the rule combined with the axiom yield a proof system that is complete in the syntactic sense. It is, of course, also sound by (i) and the soundness of the rule for arbitrary processes. We will now define the transformation in detail and give proofs of (i) and (ii).

Theorem 23 (Completeness-2) *The rule is complete for arbitrary processes.*

Proof. For simplicity, we consider first the case where P_1, P_2 have a shared interface variable y, but are non-blocking. Blocking processes are converted to non-blocking ones by a similar transformation, which is described later.

Processes with shared variables

Consider processes P_1, P_2 and T such that $P_1//P_2 \models T$. Let us assume that P_1 and P_2 have a shared interface variable y, but are non-blocking. First, if y is not an interface variable of T, let process T'' be obtained by declaring y as an interface variable, without changing anything else in T. Clearly, $P_1//P_2 \models T$ iff $P_1//P_2 \models T''$. Next, we transform processes P_1, P_2 and T'' by by adding a dummy initial state to each process where all variables have a fixed value, say $\perp$. Let P_1^I, P_2^I and T^I be the new versions of these processes. As the initial condition, transition relation, and fairness condition are unchanged, $P_1//P_2 \models T$ iff $P_1^I//P_2^I \models T^I$. For the rest of the proof, we assume P_1, P_2 and T satisfy the above conditions; that is, T has y as an interface variable, and P_1, P_2 and T have a dummy initial state where y has a single value, and that $P_1//P_2 \models T$.

We can transform P_1 into P_1' by by syntactically replacing every occurrence of y with y_1, which we represent as the substitution $[y \leftarrow y_1]$. Let x_1 represent the other variables of P_1. Thus, P_1' is defined as follows.

- $V(P_1') = (V(P_1)\backslash\{y\}) \cup \{y_1\}$,
- $I(P_1')(x_1y_1) = [y \leftarrow y_1]I(P_1)(x_1y)$,
- $R(P_1')(x_1y_1, x_1'y_1') = [y, y' \leftarrow y_1, y_1']R(P_1)(x_1y, x_1'y')$,
- $F(P_1')(x_1y_1) = [y \leftarrow y_1]F(P_1)(x_1y)$.

Likewise, we can transform P_2 into P_2' by replacing y with y_2. We now show a relationship between $P_1//P_2$ and $P_1'//P_2'$.

Lemma 7 $[\mathcal{L}(P_1//P_2) \equiv (\exists y_1, y_2 : \mathsf{G}(y = y_1) \wedge \mathsf{G}(y = y_2) \wedge \mathcal{L}(P_1'//P_2'))]$.

Proof.

$$
\begin{array}{ll}
 & (\exists y_1, y_2 : \mathsf{G}(y = y_1) \wedge \mathsf{G}(y = y_2) \wedge \mathcal{L}(P_1'//P_2')) \\
\equiv & \quad (\text{ definition of } \mathcal{L} \text{ }) \\
 & (\exists y_1, y_2 : \mathsf{G}(y = y_1) \wedge \mathsf{G}(y = y_2) \wedge \\
 & I(P_1')(x_1y_1) \wedge \mathsf{G}(R(P_1')(x_1y_1, x_1'y_1')) \wedge F(P_1')(x_1y_1) \wedge \\
 & I(P_2')(x_2y_2) \wedge \mathsf{G}(R(P_2')(x_2y_2, x_2'y_2')) \wedge F(P_2')(x_2y_2)) \\
\equiv & \quad (\text{ Leibnitz rule (using } \mathsf{G}(y = y_1) \wedge \mathsf{G}(y = y_2)\text{), definition of } P_1', P_2' \text{ })
\end{array}
$$

$(\exists y_1, y_2 : \mathsf{G}(y = y_1) \wedge \mathsf{G}(y = y_2) \wedge$

$I(P_1)(x_1 y) \wedge \mathsf{G}(R(P_1)(x_1 y, x_1' y')) \wedge F(P_1)(x_1 y) \wedge$

$I(P_2)(x_2 y) \wedge \mathsf{G}(R(P_2)(x_2 y, x_2' y')) \wedge F(P_2)(x_2 y))$

$\equiv$ (logic)

$(\exists y_1, y_2 : \mathsf{G}(y = y_1) \wedge \mathsf{G}(y = y_2)) \wedge$

$I(P_1)(x_1 y) \wedge \mathsf{G}(R(P_1)(x_1 y, x_1' y')) \wedge F(P_1)(x_1 y) \wedge$

$I(P_2)(x_2 y) \wedge \mathsf{G}(R(P_2)(x_2 y, x_2' y')) \wedge F(P_2)(x_2 y)$

$\equiv$ (logic)

$I(P_1)(x_1 y) \wedge \mathsf{G}(R(P_1)(x_1 y, x_1' y')) \wedge F(P_1)(x_1 y) \wedge$

$I(P_2)(x_2 y) \wedge \mathsf{G}(R(P_2)(x_2 y, x_2' y')) \wedge F(P_2)(x_2 y)$

$\equiv$ (definitions)

$\mathcal{L}(P_1 // P_2)$

$\square$

We modify the process T to T_1 by substituting y_1 for y; the following lemma relates T and T_1.

Lemma 8 $[\mathcal{L}^{\mathcal{O}}(T_1) \equiv (\forall y : \mathsf{G}(y = y_1) \Rightarrow \mathcal{L}^{\mathcal{O}}(T))]$

Proof.

$(\forall y : \mathsf{G}(y = y_1) \Rightarrow \mathcal{L}^{\mathcal{O}}(T))$

$\equiv$ (definitions)

$(\forall y : \mathsf{G}(y = y_1) \Rightarrow (\exists V^p(T) : I(T)(xy) \wedge \mathsf{G}(R(T)(xy, x'y')) \wedge$

$F(T)(xy)))$

$\equiv$ (Leibnitz rule; y, y_1 are not private variables of T)

$(\forall y : \mathsf{G}(y = y_1) \Rightarrow (\exists V^p(T) : I(T)(xy_1) \wedge \mathsf{G}(R(T)(xy_1, x'y_1')) \wedge$

$F(T)(xy_1)))$

$\equiv$ (rearranging)

$(\exists y : \mathsf{G}(y = y_1)) \Rightarrow (\exists V^p(T) : I(T)(xy_1) \wedge \mathsf{G}(R(T)(xy_1, x'y'_1)) \wedge F(T)(xy_1))$

$\equiv$ (definitions; logic)

$\mathit{true} \Rightarrow \mathcal{L}^{\mathcal{O}}(T_1)$

$\equiv$ (logic)

$\mathcal{L}^{\mathcal{O}}(T_1)$

□

We can now show the following lemma.

Lemma 9 $[\mathcal{L}(P_1//P_2) \Rightarrow \mathcal{L}^{\mathcal{O}}(T)]$ *iff* $[\mathcal{L}(P'_1//P'_2) \wedge \mathsf{G}(y_1 = y_2) \Rightarrow \mathcal{L}^{\mathcal{O}}(T_1)]$.

Proof.

$[\mathcal{L}(P_1//P_2) \Rightarrow \mathcal{L}^{\mathcal{O}}(T)]$

$\equiv$ (by Lemma 7)

$[(\exists y_1, y_2 : \mathsf{G}(y = y_1) \wedge \mathsf{G}(y = y_2) \wedge \mathcal{L}(P'_1//P'_2)) \Rightarrow \mathcal{L}^{\mathcal{O}}(T)]$

$\equiv$ (logic)

$[(\forall y_1, y_2 : \mathsf{G}(y = y_1) \wedge \mathsf{G}(y = y_2) \wedge \mathcal{L}(P'_1//P'_2) \Rightarrow \mathcal{L}^{\mathcal{O}}(T))]$

$\equiv$ (rearranging quantifiers: absorb y_1, y_2 into [], make y explicit)

$[(\forall y : \mathsf{G}(y = y_1) \wedge \mathsf{G}(y = y_2) \wedge \mathcal{L}(P'_1//P'_2) \Rightarrow \mathcal{L}^{\mathcal{O}}(T))]$

$\equiv$ (rearranging)

$[\mathcal{L}(P'_1//P'_2) \Rightarrow (\forall y : \mathsf{G}(y = y_1) \wedge \mathsf{G}(y = y_2) \Rightarrow \mathcal{L}^{\mathcal{O}}(T))]$

$\equiv$ (Leibnitz rule)

$[\mathcal{L}(P'_1//P'_2) \Rightarrow (\forall y : \mathsf{G}(y = y_1) \wedge \mathsf{G}(y_1 = y_2) \Rightarrow \mathcal{L}^{\mathcal{O}}(T))]$

$\equiv$ (rearranging)

$[\mathcal{L}(P'_1//P'_2) \Rightarrow (\mathsf{G}(y_1 = y_2) \Rightarrow (\forall y : \mathsf{G}(y = y_1) \Rightarrow \mathcal{L}^{\mathcal{O}}(T))]$

$\equiv$ (by Lemma 8)

$[\mathcal{L}(P_1'//P_2') \Rightarrow (\mathsf{G}(y_1 = y_2) \Rightarrow \mathcal{L}^{\mathcal{O}}(T_1))]$

$\equiv$ (rearranging)

$[\mathcal{L}(P_1'//P_2') \wedge \mathsf{G}(y_1 = y_2) \Rightarrow \mathcal{L}^{\mathcal{O}}(T_1)]$

$\square$

By Lemma 9, $[\mathcal{L}(P_1//P_2) \Rightarrow \mathcal{L}^{\mathcal{O}}(T)]$ holds iff $[\mathcal{L}(P_1'//P_2') \Rightarrow (\mathsf{F}(y_1 \neq y_2) \vee \mathcal{L}^{\mathcal{O}}(T_1))]$. We now modify the structure of T_1 to T', which takes the $\mathsf{F}(y_1 \neq y_2)$ condition into account. Informally, T' is in the "normal" mode, where $y_1 = y_2$, and T' behaves like T_1. If $y_1 \neq y_2$ in the next state, T' transitions to an "abnormal" mode, and stays in that mode from that point on. The distinction between normal and abnormal mode is expressed by a single private variable, n.

Formally, if $T_1 = (V, I, R, F)$, then $T' = (V', I', R', F')$, where

- $(V')^p = V^p \cup \{n\}$, $(V')^i = V^i \cup \{y_2\}$, and $(V')^e = V^e$. Let z refer to all the variables of V, V' other than y, y_1, y_2, n.
- $I'(zy_1y_2n) = (z = a) \wedge (y_1 = b) \wedge (y_2 = b) \wedge n$, where $I(zy_1) = (z = a \wedge y_1 = b)$. Recall there is a single initial state for T_1.
- $R'(zy_1y_2n, z'y_1'y_2'n') = (n \wedge (y_1' = y_2') \wedge n' \wedge R(zy_1, z'y_1')) \vee (n \wedge (y_1' \neq y_2') \wedge \neg n') \vee (\neg n \wedge \neg n')$
- $F'(zy_1y_2n) = (\mathsf{FG}(n) \wedge F(zy_1)) \vee \mathsf{FG}(\neg n)$

Lemma 10 *For process* $P = P_1'//P_2'$,
$[\mathcal{L}(P) \wedge \mathsf{G}(y_1 = y_2) \Rightarrow \mathcal{L}^{\mathcal{O}}(T_1)]$ *iff* $[\mathcal{L}(P) \Rightarrow \mathcal{L}^{\mathcal{O}}(T')]$

Proof. ($\Rightarrow$) Consider any execution π of P. We have to show that it belongs to $\mathcal{L}^{\mathcal{O}}(T')$. There are two cases:

- if $\mathsf{G}(y_1 = y_2)$ holds for π, by hypothesis, π belongs to $\mathcal{L}^{\mathcal{O}}(T_1)$; so there is a run of T' on π that stays within normal states. Thus, π belongs to $\mathcal{L}^{\mathcal{O}}(T')$.
- Otherwise, if eventually $(y_1 \neq y_2)$ holds in π, consider the first point i at which this happens. Then, $i > 0$, as the initial state of P satisfies $y_1 = y_2$. Thus, the prefix $\pi[0..i-1]$ is in $\mathcal{L}(P)$, and satisfies $\mathsf{G}(y_1 = y_2)$, so there is a run of T' on it. As $\pi[i]$ satisfies $y_1 \neq y_2$, T' has a transition to an abnormal state from the end state of this run, and accepts π.

($\Leftarrow$) Consider any execution π of P that satisfies $\mathsf{G}(y_1 = y_2)$. We have to show that it belongs to $\mathcal{L}^{\mathcal{O}}(T_1)$. This execution belongs to $\mathcal{L}^{\mathcal{O}}(T')$ by assumption. As $\mathsf{G}(y_1 = y_2)$ holds of π, the witnessing run of T' on π must stay in the normal part of T'. By construction of T', this is a run of T_1 on π, so π belongs to $\mathcal{L}^{\mathcal{O}}(T_1)$.

$\square$

Processes that are blocking

Suppose processes P_1 and P_2 are blocking. We transform P_1 to P_1', by adding a blocking variable b_1, and making the following modifications:

$I(P_1')(x_1 b_1) = I(P_1)(x_1) \wedge \neg b_1$,

$R(P_1')(x_1 b_1, x_1' b_1') = (b_1 \wedge b_1') \vee (\neg b_1 \wedge (\forall x_1' : \neg R(P_1)(x_1, x_1')) \wedge b_1')$

$\vee (\neg b_1 \wedge R(P_1)(x_1, x_1') \wedge \neg b_1')$,

$F(P_1')(x_1 b_1) = (\mathsf{G}(\neg b_1) \wedge F(P_1)(x_1)) \vee \mathsf{F}(b_1)$.

The variable b_1 is initially false. P_1' behaves like P_1 as long as b_1 is false; it transitions to a state satisfying b_1 on any condition for which P_1

has no enabled transition. P_2 can be similarly modified to P_2'. Thus, by this definition, P_1' and P_2' are non-blocking. We will now show, using a proof similar to that of Lemma 7, that $[\mathcal{L}(P_1//P_2) \equiv (\exists b_1, b_2 : \mathcal{L}(P_1'//P_2') \wedge \mathsf{G}(\neg b_1 \wedge \neg b_2))]$ holds.

Lemma 11 $[\mathcal{L}(P_1//P_2) \equiv (\exists b_1, b_2 : \mathcal{L}(P_1'//P_2') \wedge \mathsf{G}(\neg b_1 \wedge \neg b_2))]$.

Proof.

$(\exists b_1, b_2 : \mathsf{G}(\neg b_1 \wedge \neg b_2) \wedge \mathcal{L}(P_1'//P_2'))$

$\equiv$ (definition of $\mathcal{L}$)

$(\exists b_1, b_2 : \mathsf{G}(\neg b_1 \wedge \neg b_2) \wedge$

$I(P_1')(x_1 b_1) \wedge \mathsf{G}(R(P_1')(x_1 b_1, x_1' b_1')) \wedge F(P_1')(x_1 b_1) \wedge$

$I(P_2')(x_2 b_2) \wedge \mathsf{G}(R(P_2')(x_2 b_2, x_2' b_2')) \wedge F(P_2')(x_2 b_2))$

$\equiv$ (Leibnitz rule using $\mathsf{G}(\neg b_1 \wedge \neg b_2)$)

$(\exists b_1, b_2 : \mathsf{G}(\neg b_1 \wedge \neg b_2) \wedge$

$I(P_1')(x_1 \mathit{false}) \wedge \mathsf{G}(R(P_1')(x_1 \mathit{false}, x_1' \mathit{false})) \wedge F(P_1')(x_1 b_1) \wedge$

$I(P_2')(x_2 \mathit{false}) \wedge \mathsf{G}(R(P_2')(x_2 \mathit{false}, x_2' \mathit{false})) \wedge F(P_2')(x_2 b_2))$

$\equiv$ (definition of P'; logic)

$(\exists b_1, b_2 : \mathsf{G}(\neg b_1 \wedge \neg b_2)) \wedge$

$I(P_1)(x_1) \wedge \mathsf{G}(R(P_1)(x_1, x_1')) \wedge F(P_1)(x_1) \wedge$

$I(P_2)(x_2) \wedge \mathsf{G}(R(P_2)(x_2, x_2')) \wedge F(P_2)(x_2)$

$\equiv$ (definition of $\mathcal{L}$)

$\mathcal{L}(P_1//P_2)$

$\square$

Lemma 12 $[\mathcal{L}(P_1//P_2) \Rightarrow \mathcal{L}^{\mathcal{O}}(T)]$ *iff* $[\mathcal{L}(P_1'//P_2') \wedge \mathsf{G}(\neg b_1 \wedge \neg b_2) \Rightarrow \mathcal{L}^{\mathcal{O}}(T)]$.

Proof.

$$[\mathcal{L}(P_1//P_2) \Rightarrow \mathcal{L}^{\mathcal{O}}(T)]$$

$\equiv$ (lemma 11)

$$[(\exists b_1, b_2 : \mathcal{L}(P_1'//P_2') \wedge \mathsf{G}(\neg b_1 \wedge \neg b_2)) \Rightarrow \mathcal{L}^{\mathcal{O}}(T)]$$

$\equiv$ (rearranging quantifiers)

$$[\mathcal{L}(P_1'//P_2') \wedge \mathsf{G}(\neg b_1 \wedge \neg b_2) \Rightarrow \mathcal{L}^{\mathcal{O}}(T)]$$

$\Box$

We now modify the structure of T to T', which takes the $\mathsf{G}(\neg b_1 \wedge \neg b_2)$ condition into account. Informally, when in the "normal" mode (i.e. $\neg b_1 \wedge \neg b_2$), T' behaves like T. If $b_1 \vee b_2$ holds in the next state, T' transitions to an "abnormal" mode, and stays in that mode from that point on. The distinction between normal and abnormal mode is expressed by a single private variable, n.

Formally, if $T = (V, I, R, F)$, then $T' = (V', I', R', F')$, where

- $(V')^p = V^p \cup \{n\}$, $(V')^i = V^i \cup \{b_1, b_2\}$, and $(V')^e = V^e$. Let z refer to all the variables of V, V' other than b_1, b_2, n.
- $I'(zb_1b_2n) = (z = a) \wedge (\neg b_1) \wedge (\neg b_2) \wedge n$,
- $R'(zb_1b_2n, z'b_1'b_2'n') = (n \wedge \neg(b_1' \vee b_2') \wedge n' \wedge R(z, z')) \vee (n \wedge (b_1' \vee b_2') \wedge \neg n') \vee (\neg n \wedge \neg n')$
- $F'(zb_1b_2n) = (\mathsf{FG}(n) \wedge F(zb_1b_2)) \vee \mathsf{FG}(\neg n)$

Lemma 13 *For process* $P = P_1'//P_2'$,
$[\mathcal{L}(P) \wedge \mathsf{G}((\neg b_1) \wedge (\neg b_2)) \Rightarrow \mathcal{L}^{\mathcal{O}}(T)]$ *iff* $[\mathcal{L}(P) \Rightarrow \mathcal{L}^{\mathcal{O}}(T')]$

Proof. ($\Rightarrow$) Consider any execution π of P. We have to show that it belongs to $\mathcal{L}^{\mathcal{O}}(T')$. There are two cases:

- If $\mathsf{G}(\neg(b_1) \wedge \neg(b_2))$ holds for π, by hypothesis, π belongs to $\mathcal{L}^{\mathcal{O}}(T)$; so there is a run of T' on π that stays within normal states. Thus, π belongs to $\mathcal{L}^{\mathcal{O}}(T')$.

- Otherwise, if eventually $(b_1 \vee b_2)$ holds in π, consider the first point i at which this happens. Then, $i > 0$, as the initial state of P satisfies $(\neg b_1) \wedge (\neg b_2)$. Thus, the prefix $\pi[0..i-1]$ is in $\mathcal{L}(P)$, and satisfies $\mathsf{G}(\neg(b_1) \wedge \neg(b_2))$, so there is a run of T' on it. As $\pi[i]$ satisfies $(b_1 \vee b_2)$, T' has a transition to an abnormal state from the end state of this run, and accepts π.

($\Leftarrow$) Consider any execution π of P that satisfies $\mathsf{G}((\neg b_1) \wedge (\neg b_2))$. We have to show that it belongs to $\mathcal{L}^{\mathcal{O}}(T)$. This execution belongs to $\mathcal{L}^{\mathcal{O}}(T')$ by assumption. As $\mathsf{G}((\neg b_1) \wedge (\neg b_2))$ holds on π, the witnessing run of T' on π stays in the normal part of T'. By construction, this gives an accepting run of T on π, so x belongs to $\mathcal{L}^{\mathcal{O}}(T)$.$\square$
$\square$

5.3 Compositional reasoning with Timing Diagrams

In the previous section, we presented 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 SRTDs, which were described

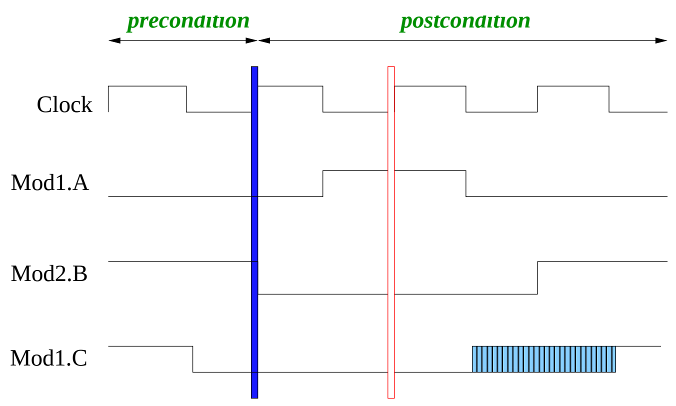

Figure 5.1: Augmented Synchronous Regular Timing Diagram

in detail in Chapter 4. By focusing on SRTDs, which are a highly regular specification formalism, we obtain several benefits. Firstly, for SRTDs with the non-overlapping semantics, 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 SRTDs have efficient model checking procedures. Finally, we also show that the generation of helper assertions is not only automatic but efficient for SRTDs.

In order to use SRTDs as a specification language in a compositional model checking paradigm we need to augment the definitions of SRTDs given in Chapter 4 with some information about the modularity of the design being verified. This is achieved by introducing the concept of an ownership function which is defined as follows.

Definition 35 (Ownership Function) *Given an SRTD $T = (c, S, WF, M)$ and a set of (implementation) process names N. An ownership function $O : S \to N$ maps each signal in S to the process in N that controls it.*

Note that the ownership function assumes that the signals are not controlled by more than one process. Thus, the ownership function O can be used to partition the SRTD T into disjoint *fragments*, $T_1, \ldots, T_n$. An SRTD fragment T_i consists of Pre_T, and only those waveforms in $Post_T$ that are owned by process i. Recall that an SRTD T is defined over a set symbolic values $\mathcal{SV}(S)$ $= \{(a_p a_q ... a_r) | a_p \in \mathcal{SV}_p \wedge ... \wedge a_r \in \mathcal{SV}_r\} \cup \{X, D\}$, where $S = \{p, q, ...r\}$ is the set of waveform names and $\mathcal{SV}_i$ is the domain of values for waveform i. An SRTD T is defined as $(Pre, Post_1, ..., Post_k)$, where the precondition Pre or any of the subsequent postcondition segments $Post_i$ of length m is a function $S \times [0, m) \rightarrow \mathcal{SV}(S)$.

Definition 36 (SRTD Fragment) *Given SRTD* $T = (Pre, Post_1, ..., Post_k)$ *and a ownership function* O*. Let* $S_A = \{s \in S | \, O(s) = A\}$ *be the signals in* T *controlled by implementation process* A*.* $T_A = (Pre', Post'_1, ..., Post'_k)$ *is a fragment of* T *with respect to* O *where*

- $Pre' = Pre$ *and*

- *Each postcondition segment* $Post'_i$*,* $Post'_i$ *is a function from* $S_A \times [0, m) \rightarrow \mathcal{SV}(S_A)$ *where, for all* $s \in S_A$*,* $Post'_i(s) = Post_i(s)$*.*

An SRTD fragment may not be a well-formed SRTD since a fragment may contain a pause whose pause owner is in another fragment. For example, in Figure 5.1, the ownership function O maps signals A and C to process Mod_1 and B to process Mod_2, and we have one fragment consisting of waveforms $Mod_1.A$ and $Mod_1.C$ and another with waveform $Mod_2.B$.

5.3.1 Translating SRTDs into Automata

In Chapter 4, we presented algorithms that translate an SRTD, with both the overlapping and non-overlapping semantics, into $\forall FA$. 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.

Algorithm 5

The algorithm that translates an SRTD T, relative to an ownership function O, into a $\forall FA$ $\mathcal{A}_T$ proceeds as follows.

- Use the ownership function O to partition the T into fragments $T_0, ..., T_n$.
- For each fragment T_i, construct a $\forall FA$ $\mathcal{A}_i$ using the algorithms (for either the overlapping and non-overlapping semantics) in Section 4.
- The $\forall FA$ A_T that corresponds to T is $\mathcal{A}_0 \times ... \times \mathcal{A}_n$.

Therefore, using this modified algorithm, an SRTD T with fragments $T_1, \ldots, T_n$ can be translated into an $\forall FA$ $\mathcal{A}_T = \mathcal{A}_1 \times \ldots \times \mathcal{A}_n$. As a consequence of Theorems 11 and 13, we know that $\mathcal{A}_T$ accepts the language of T.

5.3.2 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.

Algorithm 6

For each signal i in fragment T_j, process Q_j operates as follows.

- Sets signal i nondeterministically until the precondition holds, then it generates the values for i as specified in the postcondition of waveform i.
- If there is a don't-care value in waveform i, the output value is chosen nondeterministically from the domain $\mathcal{V}_i$.
- If there is a segment of don't-care transitions, the point at which the transition occurs is chosen nondeterministically as well. Q_j maintains the old value until this point and then generates the new value.
- If process Q_j 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.

Theorem 24 (Non-blocking) *For an SRTD fragment T_j, the corresponding helper process Q_j is non-blocking.*

Proof. In order to prove that Q_j is non-blocking, we must show that Q_j can make a transition from any reachable state on any external input. By construction, Q_j operates independent of its environment except in the case of a pause. In the case of a pause which is not owned by Q_j, if the pause breaking event never occurs Q_j may wait in this state forever, otherwise, once the event occurs Q_j continues to generate the postcondition. If Q_j owns the pause, there are fairness constraints that force this event to occur; thus Q_j is non-blocking.

□

It is easy to show that Q_j is just the completely chaotic process (with initial condition and transition relation both being *true*) composed with the automaton for T_j; hence, $(//j : Q_j)$ generates the non-overlapping language of T.

Theorem 25 (Correctness) *For any SRTD fragment T_j and the corresponding helper process Q_j, σ is a computation of $(//j : Q_j)$ iff $\sigma \models_n T$.*

Proof.
$(\Rightarrow)$ If σ is a computation of $(//j : Q_j)$ then (by construction) either the precondition Pre_T never holds or the first occurrence and all subsequent non-overlapping occurrences of Pre_T are followed by postcondition of T. Hence, by Definition 21, $\sigma \models_n T$.
$(\Leftarrow)$ $\sigma \models_n T$ iff $\sigma \in ((\neg p)^*; Pre_T; Post_T)^\omega + ((\neg p)^*; Pre_T; Post_T)^*; (\neg p)^\omega$ (by Definition 21). If Pre_T does not hold along σ, then σ is a computation of $(//j : Q_j)$. If there is sub-computation $\sigma[q..r]$ of $(//j : Q_j)$ that satisfies Pre_T then, by construction, $(//j : Q_j)$ generates $\sigma[r+1..s]$ that satisfies $Post_T$. We also know, by construction, that $(//j : Q_j)$ recognizes the first occurrence of Pre_T and ignores all overlapping occurrences of Pre_T. Therefore, σ is a computation of $(//j : Q_j)$.

□

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 26 (Non-interference) *For SRTD T with 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. T has a safety component, that specifies that the waveforms must not be violated and a liveness component which specifies that each pause must occur for arbitrary but finite period of time. If σ is in $cl(\mathcal{L}^{\mathcal{O}}(Q_1 // \ldots // Q_n))$, it must satisfy the waveform pattern at each point. If σ 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, by construction, σ is a computation of the Q_j's, $j \neq i$. $\square$

Theorem 27 (Complexity of a Helper Process) *Given an SRTD fragment T_j and the corresponding helper process Q_j, the size of Q_j is linear in the size of T_j.*

Proof. The size of process Q_j is $s + t$, where s is the number of states and t is the transition size. The transition size is the sum of the length of the boolean guards labeling the transitions. The size of T_j is $n * c$, where n is the number of signals and c is the number clock points.

The number of states in Q_j is bounded by c and is, therefore, linear in the size of the T_j. Each transition in Q_j is bounded by $O(n)$ and the number of transitions is bounded by c. Hence, the transition size is $n * c$, which is also linear in $|T_j|$. Thus, the size of Q_j is linear in the size of $|T_j|$. $\square$

5.3.3 Compositional Model Checking of SRTDs

In this section, we will describe a proof methodology that uses SRTDs as the property T in the proof rule in Section 5.2. 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 the construction in the previous Section, 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 26. Thus, the only condition to be checked is C1. The details of this check are described in the following section.

5.4 Applications

We have incorporated the algorithms described in the previous sections into the RTDT tool. We used RTDT to automatically generate the property automata and the helper processes. The verification tool *COSPAN* is used to discharge the proof obligations. *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 is more efficient, 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 12,14) model checking algorithms to discharge the obligation $P_1//Q_2 \models Q_1$ (similarly for the other obligation) in C1. We use Lemma 0 to replace the possibly 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.

5.4.1 Memory Access Controller

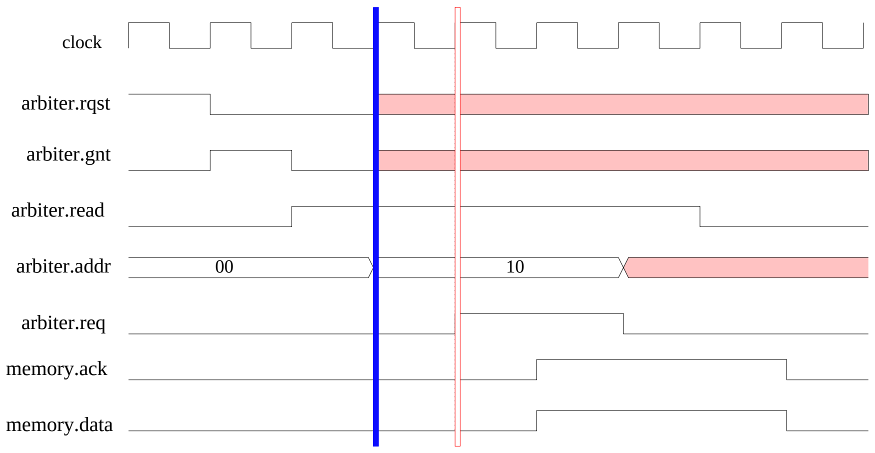

Figure 5.2: Read Transaction for the Memory Access Controller

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 putting an address the 2 bit 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 SRTDs as shown in Figure 5.2. Table 5.1 presents the verification statistics of both the compositional and non-compositional approaches. In Table 5.1, *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 *ω-NFA* for the SRTD fragment that corresponds to process *Arb* (*Mem*). Table 5.1 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 SRTDs for both the read and write transactions belonged to the arbiter.

5.4.2 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

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 5.1: Verification Statistics for Memory Access Controller Design

Interface Core is designed to be fully compatible with the PCI Local Bus specification [Gro95].

In the previous chapter, we used Lucent's PCI Bus Functional Model [BL96], 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,

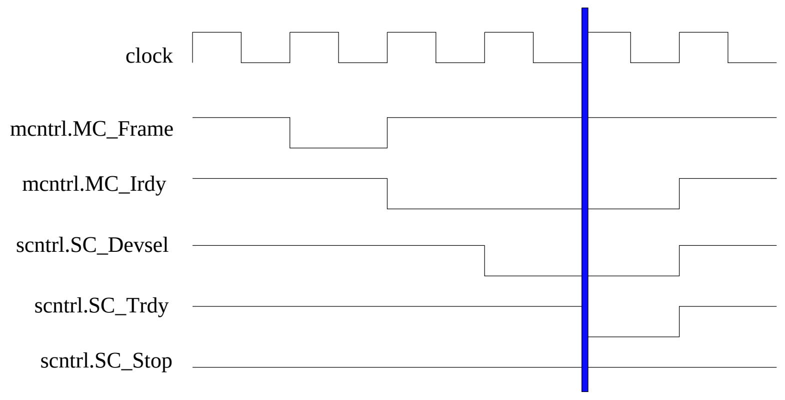

Figure 5.3: Non-burst Property for PCI Core

restricted our verification efforts to a part of this design called *pcim-core* that deals with basic PCI functionality. The *pcim-core* consists of the following processes acting in parallel: 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.

We formulated a number of properties as SRTDs by looking at the timing diagrams found in the PCI specification [Gro95] and the PCI Core User's manual [BL96]. These SRTDs 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 5.2, *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.

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

* did not complete due to shortage of space

Table 5.2: Verification Statistics for PCI Synthesizable Core Design

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.

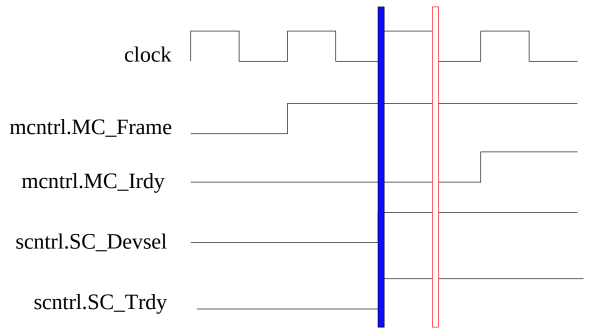

Figure 5.4: Burst Property for PCI Core

The first property in Table 5.2 stated that "in an ongoing 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, shown in Figure 5.4, specified that "if *PciFrame* is deasserted when both *PciIrdy* and *PciTrdy* are asserted then the data phase completes successfully ". The final property, shown in Figure 5.3, specified the non-burst property, "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 5.2 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 5.2 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.

5.5 Related Work and Conclusions

Compositional reasoning for concurrently active processes has been the subject of much work over the past three decades. The earliest work in assume-guarantee reasoning [MC81, Jon81] was concerned about reasoning about safety properties for networks of processes. Many other assume-guarantee proof rules, like those proposed in [Pnu85] [Sta85] [Kur87] [AH96] and [McM97], apply only to safety properties. There are more general proof rules, that can be applied to both safety and liveness properties, which are presented in the following: [Pnu85] [Jos87] [CLM89] [GL94] [AL95] [AH95] [AH96] [McM99] and [NT00]. 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. Unlike our rule, many other proof rules, like [AL95] [McM99] [AH95] [AH96] and [HQRT98], have been shown to be incomplete [NT00]. Moreover, most of the earlier work (cf. [Pnu85, AL95, AH96, McM99, NT00]) applies only to restricted kinds of processes or temporal logic formulas. In contrast, our process framework is very general and places far fewer restrictions on processes.

The possibility of using timing diagrams for compositional verification appears to have been first recognized in a paper by Josko [Jos87] on modular reasoning. This paper, however, uses timing diagrams only for illustrative purposes. In later work [HSD$^+$93], [DH94], [DHKS94], [BW98b], [BW98a], a compositional verification methodology proposed in [Jos93] is used to verify Symbolic Timing Diagram (STD) [DJS94] properties. 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 SRTDs 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.

Chapter 6

The RTDT Tool

6.1 Introduction

The Regular Timing Diagram Translator (RTDT) tool provides a user-friendly graphical editor, that is used to create and edit SRTDs, plus a translator that implements the compositional and non-compositional model checking algorithms. RTDT forms a formal and efficient timing diagram interface to the model checker *COSPAN* [HHK96].

The main features of the RTDT tool are as follows.

- A user friendly editor for graphically creating and editing SRTDs.
- A translator that implements the non-compositional algorithms and the compositional proof procedure described in Chapters 4 and 5.
- The user can execute *COSPAN* from within the RTDT tool.
- When a verification check fails, RTDT displays the resulting error trace as an SRTD and allows the user the option of editing this diagram.

6.2 RTDT Design Issues

Our design goals for the RTDT tool were: easy of use, efficiency, maintainability and portability. We chose *JAVA* as the programming language for two reasons, namely portability and the extensive graphical support. Unlike other timing diagrams editors (cf. [KM97]), we designed RTDT's Graphical User Interface to ensure that the diagram at any point in the editing process is well-formed. For instance, we use the user supplied clock triggering information in the editing process to ensure that a rising edge triggered waveform only changes state at the rising edge. The implementation is cleanly partitioned so that changes made to underlying model do not effect either the editor or translator.

RTDT makes use of the *JAVA Swing* API for the graphics. The core of the design is the intermediate representation of an SRTD, called *IR*, which is a record that contains the following information: number of clock cycles, number of waveforms, position of the precondition, a list denoting the pause markers, a list of column names and a list of waveforms. Each waveform is a list consisting of the waveform name, the triggering edge of the clock and the value at each clock point.

The editor reads and writes the *IR*. When the editor reads an *IR*, it creates a *Swing* component called a *JTable*. In order to display and edit an SRTD, instead of the table, we customized the *JTable* cell-editor and cell-renderer. The translator also inputs the *IR* and creates corresponding automata descriptions in S/R, which is the input language of *COSPAN*. The *IR* is written into a file with extension ".td" and the corresponding S/R translation is written into a file with the extension ".td.sr".