Disjunction of Regular Timing Diagrams

by

Yu Feng

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Oct 2010

APPROVED:

Professor Kathi Fisler, Thesis Advisor

Professor Dan Dougherty, Thesis Reader

Professor Michael Gennert, Head of Department

Abstract

Timing diagrams are used in industrial practice as a specification language of circuit components. They have been formalized for efficient use in model checking. This formalization is often more succinct and convenient than the use of temporal logic. We explore the relationship between timing diagrams and temporal logic formulas by showing that closure under disjunction does not hold for timing diagrams. We give an algorithm that returns a disjunction (if any) of two given timing diagrams. We also give algorithms that tell satisfaction of timing diagram and exact time separation between events in timing diagram. An Alloy specification for timing diagrams with one waveform has also been built.

Keywords: timing diagrams, temporal logic, model checking, disjunction, Alloy.

Acknowledgements

I spent three wonderful years at Worcester Polytechnic Institute. I am indebted to my advisor Kathi Fisler. She introduced me to model checking and timing diagrams and taught me that presentations of research results should be easy for people to understand. I thank her for her numerous comments and revisions on this thesis. I learnt a lot from her about how to write. I also thank her for her patience on me.

Dan Dougherty has been a superb mentor. He taught me logic. His logic course is the best course I've taken at WPI. The second to it is his foundation course. I thank Danny for his advice on timing diagrams and NFA. I thank Tim for study groups on algorithms and logic courses. I thank Theo for discussions with him about logic and math, and his help on my job-hunting.

I owe much to my parents. They have been caring me and giving me advice during these years. I'm regretful that I have been away from them for 7 years.

Yu Feng

Contents

List of Figures

Chapter 1

Background

Timing diagrams are used in industrial practice as a specification language of circuit components. Nina Amla *et al*[2] introduced the class of *Regular Timing Diagrams* which have formal syntax and semantics. They gave a model checking algorithm with regular timing diagrams which runs in cubic size of regular timing diagrams. The notation of timing diagrams is often more succinct and convenient than the use of temporal logic. Fisler and Chockler[6] proved that each timing diagram can be translated to a LTL formula. More importantly, they gave two modalities, forgettable past and unforeseen future[6], which allow a temporal logic to capture timing diagrams more succinctly than LTL. But there have been no results showing whether each LTL formula can be translated to a timing diagram. In this thesis, we solve this problem by showing closure under disjunction does not hold for timing diagrams.

1.1 RTD as a Specification Language

Formal verification is a technique of mathematically proving whether a system design satisfies properties. It contains three elements: (1) a specification method of

a system design, (2) a kind of logic describing properties, and (3) an algorithm checking whether the system design satisfies properties. One approach to formal verification is model checking. It represents system designs and properties as automata and temporal logics, and it checks whether system designs satisfy properties by checking whether the language of the automaton is a subset of language of the temporal logic formula.

In industrial practice, timing behaviors for circuit components are often described by timing diagrams[2]. Nina Amla *et al* introduce the class of *Regular Timing Diagrams* which has formal syntax and semantics. They also gave an efficient model checking algorithm on Regular Timing Diagrams[2]. They decompose a Regular Timing Diagram into isolated waveforms and timing constraints among waveforms. Each waveform or constraint can be represented as a NFA and the whole Regular Timing Diagram can be represented as a ∀FA. A ∀FA is an automaton which accepts a word iff every run of the automata along the word ends at some accepting state. Their model checking algorithm runs in the worst case of size "cubic in the size of the diagram and the largest time constant represented in unary".

There are many systems, for example real communication systems, whose specifications are satisfied by infinite computations. For the problem of how an infinite computation satisfies a timing diagram, Fisler [5] considers two kinds of semantics: invariant semantics and iterative semantics. In invariant semantics, a timing diagram is satisfied from every position in a computation. In iterative semantics, each computation satisfying a timing diagram is a concatenation of infinitely many computation of finite length each of which satisfies a timing diagram starting from the first position and ending at the last position. Nina Amla's regular timing diagrams use the iterative semantics.

In order to understand the formal connections between timing diagrams and tex-

tual temporal logics so that temporal specifications can be made designer-friendly, Fisler and Chockler[6] considered the problem of translating timing diagrams to LTL formulas. They gave an algorithm that translate timing diagrams to LTL formulas with *forgettable past* (N) modality and *unforseeable future* ($\widetilde{N}$) modality. $N\widetilde{N}$LTL is equivalent to LTL[6].

1.2 Contributions of the Thesis

This thesis use the syntax of Nina Amla's Regular Timing Diagram and our one-step semantics. In chapter 2, we prove that for every word w and every regular timing diagram RTD, there is unique assignment π such that $w \models_\pi RTD$ and we give two algorithms to rewrite regular timing diagrams. In chapter 3, we give an Alloy specification for RTDs with one waveform, and we check some theorems using Alloy. In chapter 4, we prove the relationship between satisfiability of RTD and existence of negative cycles in event graph. We also give an algorithm that compute the exact time separation between each events in RTDs. In the last chapter, we give an algorithm that decide existence of disjunctions of given two regular timing diagrams, and we prove the correctness of the algorithm.

We refer to Regular Timing Diagram as RTD or timing diagram henceforth.

Chapter 2

Regular Timing Diagrams

Because of the practical use of timing diagrams in hardware specification, there are many ways to formally define them. Among those definitions, we use Amla's syntax [1][2] and our own semantics called one-step semantics which is close to Amla's semantics.

After we introduce RTD's syntax and semantics in the first two sections, we define the disjunction problem of well-formed timing diagrams in section 2.3 and give several ways of transforming RTD without changing its language in section 2.4.

2.1 Syntax

Figure 2.1 shows an example of RTD: This RTD has three **waveforms**: a, b, and c. Events of interest on each waveform are called **points**. In figure 2.1, there are four points on waveform a (labeled $(a, 0)$, $(a, 1)$, $(a, 2)$, and $(a, 3)$), five points on waveform b (labeled $(b, 0)$, $(b, 1)$, $(b, 2)$, $(b, 3)$, and $(b, 4)$) and three points on waveform c (labeled $(c, 0)$, $(c, 1)$ and $(c, 2)$). Vertical lines specify **concurrent dependencies** between points on different waveform. In figure 2.1, there are four concurrent dependencies: the first points and last points over all waveforms each

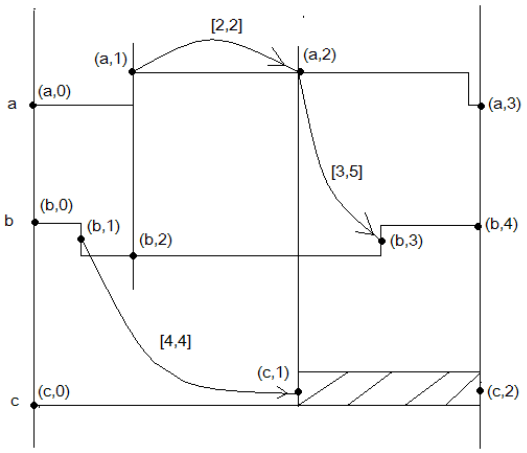

Figure 2.1: RTD example

form a concurrent dependency by default. In addition, the example adds concurrent dependencies $\{(a,1),(b,2)\}$ and $\{(a,2),(c,1)\}$. Curve lines labeled with pairs $[x,y]$ of natural numbers tell the allowed time difference between two points; these are called **sequential dependencies**. In figure 2.1, there are three sequential dependencies: $(a,1) \xrightarrow{[2,2]} (a,2)$, $(a,2) \xrightarrow{[3,5]} (b,3)$ and $(b,1) \xrightarrow{[4,4]} (c,1)$.

Formally, the syntax of RTD can be defined as follow:

Definition 2.1.1. *A RTD is a tuple (Point,WF,Value,SD,CD) in which*

- *Point is the set of all points in RTD.* (p,i) *means the* i^{th} *point on signal* p.
- WF*, the set of waveform names, is:* $\{p \mid \exists i.(p,i) \in Point\}$
- *Value:*$Point \rightarrow \{0,1,X\}$ *assigns a value to each point.*
- $SD \subseteq \{(p,i) \xrightarrow{[a,b]} (q,j),\ (p,i) \xrightarrow{[c,+\infty)} (q,j) \mid (p,i),(q,j)$ *are points,* a, b, c *are natural numbers and* $a \leq b\}$
- CD *is a set of sets of points.* $\{(p,0) \mid p \in WF\}$[1] $\in CD$ *and* $\{(p,m) \mid p \in$

[1]this is the set of first points on all waveforms.

$WF,\ \forall i.((p,i) \in Point \rightarrow m \geq i) \wedge (\forall j \forall i.\ (p,i) \in Point \rightarrow j \geq i \rightarrow (m \leq j))\}^2 \in CD$

Example 2.1.1. *For the RTD in figure 2.1:*

$Point = \{(a,0),(a,1),(a,2),(a,3),(b,0),(b,1),(b,2),(b,3),(b,4),(c,0),(c,1),(c,2)\},$

$WF = \{a,b,c\}$

$Value : (a,0) \rightarrow 0,\ (a,1) \rightarrow 1,\ (a,2) \rightarrow 1,\ (a,3) \rightarrow 0,\ (b,0) \rightarrow 1,\ (b,1) \rightarrow 0,$

$(b,2) \rightarrow 0,\ (b,3) \rightarrow 1,\ (b,4) \rightarrow 1,\ (c,0) \rightarrow 0,\ (c,1) \rightarrow X,\ (c,2) \rightarrow X,$

$SD = \{(a,1) \xrightarrow{[2,2]} (a,2),\ (b,1) \xrightarrow{[4,4]} (c,1),\ (a,2) \xrightarrow{[3,5]} (b,3)\},$

$CD = \{\{(a,0),(b,0),(c,0)\},\{(a,1),(b,2)\},\{(a,2),(c,1)\},\{(a,3),(b,4),(c,2)\}\}$

In RTD, a point is a rise or a fall if it has different value from its preceeding point's value. When a point (p,i) is a rise or a fall, it's not hard to tell where it will be located on a system behavior. For example, when (p,i) is a rise, it will be located to the position where the first 1 occurs after where $(p,i-1)$ is located on a system behavior. When a point is not a rise or a fall, it's difficult to tell where this event is located in a system behavior. For example, for points (p,i) which has value X, we don't know whether it should be located at a position where signal p is low, or high, or both. This will lead to a problem that there can be multiple ways to tell a system behavior satisfies a RTD. We want to avoid this and we want to know where precisely a point having value X is located in system behavior. One way to solve this problem is requiring every point to be an event.

We will define events after we define rises and falls as following:

Definition 2.1.2. *Point (p,i) is a rise (fall) if:*

1. $i \neq 0$ and

[2]This is the set of last points on all waveforms.

2. $Value((p,i)) = 1$ *and* $Value((p,i-1)) = 0$ ($Value((p,i)) = 0$ *and* $Value((p,i-1)) = 1$).

In figure 2.1, points (a, 1), (b, 3) are rises and points (a, 3), (b, 1) are falls.

Definition 2.1.3. *A point* (p,i) *is an event if it satisfies one of the following rules:*

- $i = 0$.
- (p,i) *is a rise or fall.*
- (p,i) *belongs to a concurrent dependency that contains an event.*
- *There exists a sequential dependency* $(q,j) \xrightarrow{[a,a]} (p,i)$ *in which* (q,j) *is an event.*

In figure 2.1, point (a, 2) is an event because there is a sequential dependency $(a,1) \xrightarrow{[2,2]} (a,2)$ and (a, 1) is an event. Point (b, 2) is an event because (b, 2) and (a, 1) are in the same concurrent dependency and (a, 1) is an event. Point (c, 1) is an event because there exists a sequential dependency $(b,1) \xrightarrow{[4,4]} (c,1)$ and (b, 1) is an event. Points (c, 2) and (b, 4) are events because (c, 2), (b, 4), and (a, 3) are in the same concurrent dependency and (a, 3) is an event. All other points are events either because they are the first points on waveforms or they are rises or falls.

A RTD specifies a timing relation on events in a system. On each waveform p, event (p,j) occurs later than (p,i) if $j > i$. For each sequential dependency like $(p,i) \xrightarrow{e} (q,j)$, event (q,j) occurs later than event (p,i) and their time difference satisfies e. It makes no sense that an event (q,j) occurs later than event (p,i) and (p,i) also occurs later than (q,j). In order to avoid this, we need to define formally the timing relation on events.

Definition 2.1.4. *We can define a relation* $\leq$ *on subset of Point.* p *and* q *are two points,* $(p,i) \leq (q,j)$ *if*

- $p = q$ *and* $i < j$, *or*
- (p, i) *and* (q, j) *are in the same concurrent dependency, or*
- *There exists a sequential dependency* $(p, i) \xrightarrow{[a,b]} (q, j)$ *or* $(p, i) \xrightarrow{[c,\infty)} (q, j)$.

The transitive closure of $\leq$ tells the timing relation on events. If $(p, i) \leq^+ (q, j)$, then in the system specified by the RTD, event (p, i) occurs no later than event (q, j). We can avoid the problem before Definition 2.1.4 by requiring $\leq^+$ is irreflexive. In figure 2.1, (b, 1) $\leq^+$(a, 2) because (b, 1)$\leq$(b, 2), (b, 2)$\leq$(a, 1) and (a, 1)$\leq$(a, 2).

Definition 2.1.5. *A RTD is well-formed if it satisfies:*

1. *All points are events.*
2. *The transitive closure* $\leq^+$ *of relation* $\leq$ *is irreflexive.*

We can see that relation $\leq^+$ in the timing diagram in figure 2.1 is irreflexive. We have showed above that all points in this RTD are events. The timing diagram in figure 2.1 is therefore a well-formed RTD.

In this thesis, we only focus on well-formed timing diagrams.

2.2 Semantics

RTD describes sets of words over an alphabet that assigns a sequence of 0 and 1 to each waveform. The semantics tells us whether a word satisfies the given RTD. Intuitively, a word satisfies a RTD when values on the words satisfy all points, and relative positions of points satisfy time bounds of sequential dependencies and concurrent dependencies.

Since RTD has only finitely many points on each waveform and many systems

have infinite computations, an important question in RTD semantics is how an infinite word satisfies a RTD. There are many kinds of semantics [5]:

1. In the invariant semantics, RTD is satisfied from every position in a word.

2. In the one-step semantics, RTD is satisfied starting from the first position of a word and ending at the last position of a word.

3. In the iterative semantics, each word satisfying a RTD is a concatenation of infinite many words that satisfy the RTD under one-step semantics.

We will use one-step semantics in this thesis. We believe we can use the result of operations[3] on RTDs under one-step semantics to solve those operations on RTDs under invariant or iterative semantics.

A point having value X means we don't care whether it maps to position where there is 0 or 1. A point having value 0(1) can only be located to position where there is 0(1). This can be described by a relation $\sqsubseteq$ on {0, 1, X}. The left side of $\sqsubseteq$ is the value of a point and the right side of $\sqsubseteq$ is the value on position in word to which the point corresponds.

Definition 2.2.1. *$a \sqsubseteq b$ if $a = X$ or $a = b$.*

Definition 2.2.2. *A word or model of a RTD is (WF, signal-trace) in which:*

$$\textit{signal-trace} \in \begin{pmatrix} p_0 : & \Sigma^* \\ p_1 : & \Sigma^* \\ \vdots & \vdots \\ p_{n-1} : & \Sigma^* \end{pmatrix} \textit{ in which } \Sigma = \{0, 1\}$$

Definition 2.2.3. *Given a word, $|word_p|$ is the length of signal trace p in word.*

[3]Including disjunction, conjunction and complement etc.

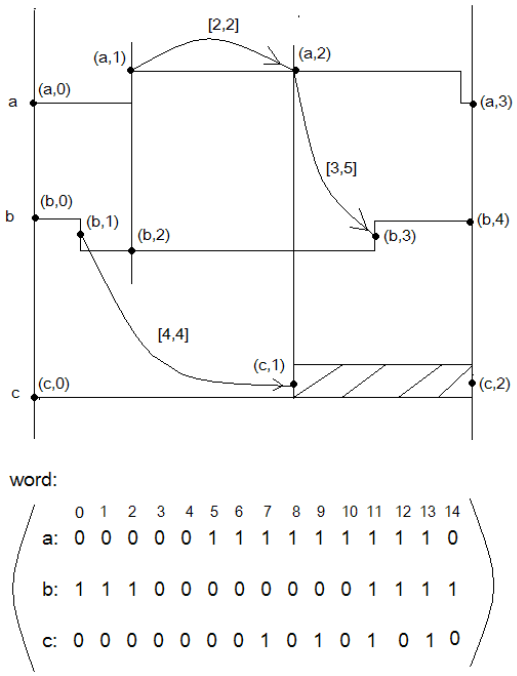

Figure 2.2: RTD and one of its words

Figure 2.2 shows a word that satisfies the RTD in figure 2.1. The word satisfies the RTD because there is a way to assign all points in RTD to appropriate positions such that points, sequential dependencies and concurrent dependencies are satisfied.

- Points (a, 0), (b, 0), (c, 0) are mapped to position 0.
- Points (a, 3), (b, 4), (c, 2) are mapped to position 14.
- Points (a, 1) and (b, 2) are mapped to position 5.
- Points (a, 2) and (c, 1) are mapped to position 7.

So all concurrent dependencies are satisfied by this word.

Point (b, 3) is mapped to position 11 and point (a, 2) is mapped to position 7, their difference is 4 which is in [3, 5], so sequential dependency $(a, 2) \xrightarrow{[3,5]} (b, 3)$ is satisfied.

These mapping ends up to be deterministic. That is, for every point (p, i), it will be mapped to only one position in a word satisfying the RTD. We will come back to this in Theorem 2.2.1.

We formally define the one-step semantics as follows:

Definition 2.2.4. *Given a RTD, an assignment* $\pi : Point \rightarrow \mathbb{N}$ *is a mapping from points in the RTD to positions in a word.*

Definition 2.2.5. *Given a RTD=(Point, WF, Value, SD, CD), a word=(WF, signal-trace) and an assignment* π, $p \in WF$ *and* $(p, i) \in Point$, $word_p(\pi((p, i)))$ *is the digit at position* $\pi((p, i))$ *on signal trace* p *in word.*

Definition 2.2.6. *Given a RTD=(Point, WF, Value, SD, CD), a word=(WF, signal-trace) and a function* π, $word \models_\pi RTD$ *if:*

1. $\forall p \in WF$, *let* m *be the number of points on waveform* p, $\pi((p, 0)) = 0$ *and* $\pi((p, m-1)) = |word_p| - 1$
2. *Point consistency:* $\forall (p, i) \in Point$, $Value((p, i)) \sqsubseteq word_p(\pi((p, i)))$
3. *Waveform consistency:* $\forall p \in WF$, *if* (p, i) *is not the last point on waveform* p, *then* $\forall j \in [\pi((p, i)), \pi((p, i+1)) - 1]$ $Value((p, i)) \sqsubseteq word_p(j)$
4. *Dependency consistency: For every sequential dependency* $(p, i) \xrightarrow{[a,b\rangle} (q, j)$, $\pi((q, j)) - \pi((p, i)) \in [a, b\rangle$. *For every concurrent dependency cd,* $\forall p \in cd$ $\forall q \in cd$, $\pi(p) = \pi(q)$.

Definition 2.2.7. *Given a RTD and a word,* $word \models RTD$ *if there exists an assignment* π *s.t.* $word \models_\pi RTD$.

Definition 2.2.8. *The language of RTD, denoted* $\mathcal{L}(RTD)$, *is* $\{word \mid word \models RTD\}$.

We can see that given a RTD=(Point, WF, Value, SD, CD) and $word \in \mathcal{L}(RTD)$, $\forall p, q \in WF$, $|word_p| = |word_q|$.

We've said above that for well-formed RTDs, their semantics is deterministic. Given any well-formed RTD and any word that satisfies it, there is only one assignment that meets Definition 2.2.6. The reason is: for every point which is a rise or a fall, there is only one way to locate it on words. For every point (q, j) which is neither a rise nor a fall, there exists a rise or a fall (p, i) and some sequential dependencies (or a sequential dependency) $(p, i) \xrightarrow{[k_0,k_0]} (p_0, i_0)$, $(p_0, i_0) \xrightarrow{[k_1,k_1]} (p_1, i_1)$, ..., $(p_{n-1}, i_{n-1}) \xrightarrow{[k_n,k_n]} (p_n, i_n)$, $(p_n, i_n) \xrightarrow{[k_n,k_n]} (q, j)$ which means there is only one position where (q, j) can be located. We will formally prove it in the following theorem.

Theorem 2.2.1. *Given a well-formed RTD=(Point, WF, Value, SD, CD), a word=(WF, signal-trace), if* $word \models RTD$*, then there exists only one assignment* π *s.t.* $word \models_\pi RTD$.

Proof. We prove by contradiction. Suppose there exists π_0 and π_1 s.t. $word \models_{\pi_0} RTD$ and $word \models_{\pi_1} RTD$ when $word \models RTD$. According to semantics, $\forall p \in WF$, $\pi_0((p, 0))=\pi_1((p, 0))=0$ and $\pi_0((p, m))=\pi_1((p, m))=|word_p| - 1$.

Since π_0 and π_1 are not the same, there exists a point (p, i) s.t. $\pi_0((p, i)) \neq \pi_1((p, i))$. We choose the first of such points on waveform p, and let it be (p, i). Without loss of generality, let $\pi_0((p, i)) < \pi_1((p, i))$.

- If (p, i) is a rise or fall, by definition, $Value((p, i)) \neq Value((p, i - 1))$ which means $word_p(\pi_0((p, i))) = 1 - word_p(\pi_0((p, i)) - 1)$ and $word_p(\pi_1((p, i))) = 1 - word_p(\pi_1((p, i)) - 1)$. Since (p, i) is the first point which is mapped to different position by π_0 and π_1, $\pi_0((p, i - 1)) = \pi_1((p, i - 1)) < \pi_0((p, i)) < \pi_1((p, i))$. Since $word \models_{\pi_1} RTD$, waveform consistency is satisfied which means $word_p(\pi_0((p, i))) = Value((p, i - 1))$. Point consistency must also be

satisfied which means $word_p(\pi_0((p,i))) = Value((p,i))$. So $Value((p,i)) = Value((p,i-1))$ which contradicts point (p,i) is a rise or fall.

- If (p,i) is not a rise or fall, by definition of well-formed timing diagram, there exists a sequential dependency $(q,j) \xrightarrow{[k,k]} (p,i)$. Since (p,i) is the first point which is mapped to different positions by π_0 and π_1, $\pi_0((q,j)) = \pi_1((q,j))$. Since $\pi_0((p,i)) < \pi_1((p,i))$, $\pi_0((p,i)) - \pi_0((q,j)) \neq \pi_1((p,i)) - \pi_1((q,j))$ which contradicts $word \models_{\pi_0} RTD$ and $word \models_{\pi_1} RTD$ because $(q,j) \xrightarrow{[k,k]} (p,i)$ cannot be satisfied by both π_0 and π_1.

So we have proved this theorem. □

Another interesting property of RTD semantics is RTD cannot be satisfied by any prefix of its words. This can be prove using similar technique as above.

Theorem 2.2.2. *Given any RTD=(Point, WF, Value, SD, CD), and any word* $\in \mathcal{L}(RTD)$*, if pre-word is a prefix of word and* $|pre\text{-}word| < |word|$*, then pre-word* $\notin \mathcal{L}(RTD)$.

Proof. We prove by contradiction. Suppose there exists an assignment π_p s.t. *pre-word* $\models_{\pi_p} RTD$. Since $word \models RTD$, let the assignment mapping points to position in word be π which means $word \models_{\pi} RTD$. $\forall p \in WF\ \exists i.\ \pi_p((p,i)) < \pi((p,i))$ because $|pre\text{-}word| < |word|$. We choose one waveform p and choose the first point of this kind. Let it be (p,i).

We first prove (p,i) has to be either rise or fall. If (p,i) is neither rise nor fall, then by definition of well-formed timing diagram, there exists a sequential dependency $(q,j) \xrightarrow{[k,k]} (p,i)$. Since $word \models_{\pi} RTD$ and $pre\text{-}word \models_{\pi_p} RTD$, this sequential dependency should be satisfied in both word and *pre-word*. So $\pi_p((p,i)) - \pi_p((q,j)) = \pi((p,i)) - \pi((q,j))$ which means $\pi_p((q,j)) < \pi((q,j))$. This

contradicts (p, i) being the first point which is mapped to different position by π and π_p. So (p, i) has to be either rise or fall.

Since (p, i) is either a rise or fall, $Value((p, i)) = 1 - Value((p, i-1))$. We have $\pi_p((p, i-1)) = \pi((p, i-1)) < \pi_p((p, i)) < \pi((p, i))$ because (p, i) is the first point mapped differently by π and π_p. Since $word \models_\pi RTD$, waveform consistency is satisfied which means $Value((p, i-1)) \sqsubseteq word_p(\pi_p((p, i)))$ which means $Value((p, i-1)) = word_p(\pi_p((p, i)))$. Since $pre\text{-}word \models_{\pi_p} RTD$, point consistency must be satisfied by *pre-word*. So $Value((p, i)) \sqsubseteq pre\text{-}word_p(\pi_p((p, i)))$ which means $Value((p, i)) \sqsubseteq word_p(\pi_p((p, i)))$ which means $Value((p, i)) = word_p(\pi_p((p, i)))$. So we have $Value((p, i)) = Value((p, i-1))$ which contradicts (p, i) is a rise or fall.

So we have proved the theorem. □

2.3 The Disjunction Problem

Given two RTDs RTD_0 and RTD_1, we want to know under what conditions there exists a RTD RTD_2 s.t. $\mathcal{L}(RTD_0) \bigcup \mathcal{L}(RTD_1) = \mathcal{L}(RTD_2)$.

Definition 2.3.1. *Given RTD_0 and RTD_1, if there exists a RTD_2 such that $\mathcal{L}(RTD_0) \bigcup \mathcal{L}(RTD_1) = \mathcal{L}(RTD_2)$, then RTD_2 is called disjunction of RTD_0 and RTD_1.*

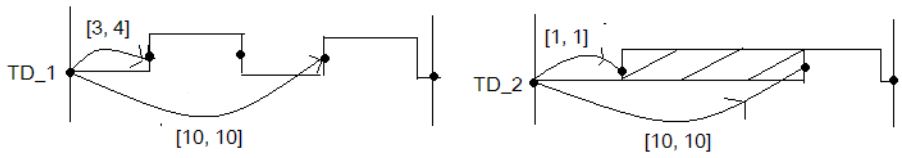

Figure 2.3: Two RTDs having disjunction

The two RTDs in figure 2.3 have their disjunction which can be drawn as TD_2. This is case when $\mathcal{L}(TD\_1) \subset \mathcal{L}(TD\_2)$.

There is no RTD whose language captures $\mathcal{L}(RTD_1) \bigcup \mathcal{L}(RTD_2)$ in Figure 2.4.

Figure 2.4: Two RTDs having no disjunction

Because the first digit of all words satisfying RTD_1 is 0 and the first digit of all words satisfying RTD_2 is 1. The last digit of all words satisfying RTD_1 is 1 and the last digit of all words satisfying RTD_2 is 0. Suppose such their disjunction RTD exists, its first and last point should both have value X. But in the language of disjunction, there are also words whose first digit and last digit are the same. These words can satisfy neither RTD_1 nor RTD_2 which contradicts definition of disjunction.

2.4 RTD transformation

In order to simplify the proofs about algorithms that appear in following chapters, we transform given RTD into another one without changing its language. For example, in Chapter 4 when we are going to compute exact time separations between any pair of points in a RTD. For that, we treat any concurrent dependency as a special form of sequential dependency. In our algorithm which computes disjunction of two given RTDs, we found that we can rewrite SD of any RTD into SD′ in which for any $(p, i) \xrightarrow{e} (q, j)$ (p, i) is a rise or fall. This rewriting will save us lots of trouble in proving the algorithm.

We present two transformations: CD Elimination and SD Rewriting.

2.4.1 CD Elimination

Given a RTD={Point, WF, Value, SD, CD}, we replace each concurrent dependency $\{(p, i_0), (q, i_1), (r, i_2), ...\}$ involving n points with $\binom{n}{2}$ sequential dependencies of the

form $(p, i_m) \xrightarrow{[0,0]} (q, i_n)$, ... After we do this for all concurrent dependencies in CD, we get a RTD with an empty CD. Note that when we replace every concurrent dependency with sequential dependencies, we do not change the language of RTD. For the RTD in figure 2.5, we can replace concurrent dependency $\{(p, 0), (q, 0)\}$

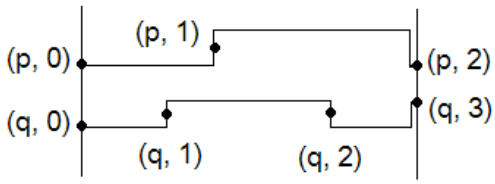

Figure 2.5: Before CD elimination

with a sequential dependency $(p, 0) \xrightarrow{[0,0]} (q, 0)$ and $\{(p, 2), (q, 3)\}$ with a sequential dependency $(p, 2) \xrightarrow{[0,0]} (q, 3)$. So we get a RTD in figure 2.6 whose CD is an empty

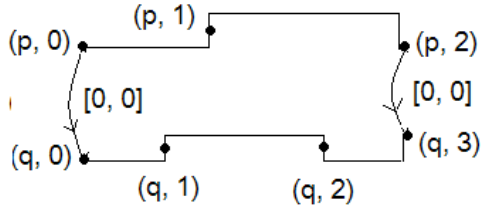

Figure 2.6: After CD elimination

set.

2.4.2 SD Rewriting

Given an RTD=(Point, WF, Value, SD, CD), we can rewrite it into another timing diagram RTD′=(Point, WF, Value, SD′, CD) in which for every sequential dependency $(p, i) \xrightarrow{e} (q, j)$ in SD′, (p, i) is either a rise/fall or the first point on waveform p.

Intuitively, we pick any sequential dependency $(p, i) \xrightarrow{[a,b]} (q, j)$ in which (p, i) is neither a rise/fall nor the first point on waveform p. Since RTD is well-formed,

there exists a sequential dependency $(r,l) \xrightarrow{[k,k]} (p,i)$, we can replace $(p,i) \xrightarrow{[a,b]} (q,j)$ with $(r,l) \xrightarrow{[a+k,b+k]} (q,j)$. We know (r,l) is an event, it may be or may not be a rise/fall or the first point on waveform r. If not, we repeat this process to replace $(r,l) \xrightarrow{[a+k,b+k]} (q,j)$ to some other sequential dependency; if yes, we replace other sequential dependencies in the same way.

The process is as follows:

for each $(p,i) \xrightarrow{\langle a,b\rangle} (q,j)$ in which neither (p,i) is a rise/fall nor $i=0$ **do**

$SD \leftarrow SD - \{(p,i) \xrightarrow{\langle a,b\rangle} (q,j)\}$

for each point (r,l) s.t. $(r,l) \xrightarrow{[k,k]} (p,i)$ exists **do**

if there exists a sequential dependency $(r,l) \xrightarrow{e} (q,j)$ **then**

$SD \leftarrow SD \bigcup \{\ (r,l) \xrightarrow{[a+k,b+k]\cap e} (q,j)\ \}$

else

$SD \leftarrow SD \bigcup \{\ (r,l) \xrightarrow{[a+k,b+k]} (q,j)\ \}$

This process will terminate, because the number of sequential dependencies $(p,i) \xrightarrow{e} (q,j)$ in which (p,i) is neither rise/fall nor the first point decreases in the process and the number of points is finite.

SD rewriting does not change language of RTDs. That is, for a given RTD_0, after we apply SD rewriting on it, it returns RTD_1 and $\mathcal{L}(RTD_0) = \mathcal{L}(RTD_1)$. The reason is: In RTD_0, there is $(p,i) \xrightarrow{[a,b\rangle} (q,j)$[4] and $(r,l) \xrightarrow{[k,k]} (p,i)$ which means all words satisfy these two sequential dependencies. This is the same as saying all words satisfy sequential dependencies $(r,l) \xrightarrow{[k,k]} (p,i)$ and $(r,l) \xrightarrow{[a+k,b+k\rangle} (q,j)$[5]. So each step of SD rewriting does not change language of $RTDs$ which means SD rewriting does not change their language.

Example 2.4.1.

[4]When b is an integer, the bound is $[a,b]$; when b is $+\infty$, the bound is $[a,+\infty)$.

[5]When b is an integer, the bound is $[a+k,b+k]$; when b is $+\infty$, the bound is $[a+k,+\infty)$.

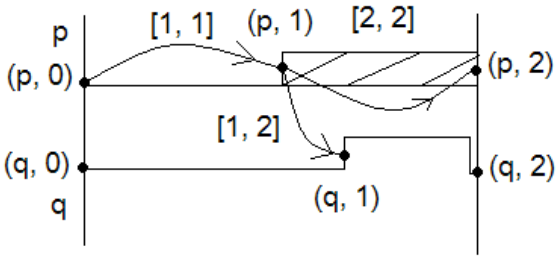

Figure 2.7: RTD before SD rewriting

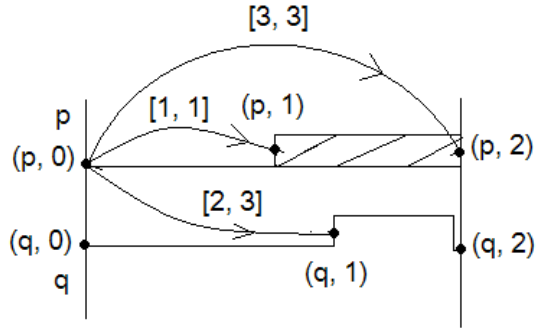

Figure 2.8: RTD after SD rewriting

Figure 2.7 shows an RTD on which we will apply SD rewriting. $(p, 1)$ is neither a rise/fall nor the first point on waveform p. We remove $(p, 1) \xrightarrow{[2,2]} (p, 2)$ and $(p, 1) \xrightarrow{[1,2]} (q, 1)$, then we add two sequential dependencies $(p, 0) \xrightarrow{[3,3]} (p, 2)$ and $(p, 0) \xrightarrow{[2,3]} (q, 1)$. Finally we get a RTD as in Figure 2.8. All sequential dependencies in Figure 2.8 is in the form of $Point_0 \xrightarrow{e} Point_1$ in which $Point_0$ is either a rise/fall or the first point on some waveform. The RTD in Figure 2.7 and the RTD in Figure 2.8 have the same language which contains only one word.

Chapter 3

Modeling RTD in Alloy

Alloy is a declarative specification language for expressing structural constraints and behavior in software systems. It provides a simple structural modeling tool based on first-order logic [8]. Alloy Analyzer is a tool [3] to find models of a specification written in Alloy language under certain bound. We have used Alloy Analyzer to successfully check some theorems about RTD with only one waveform.

3.1 Alloy Specification for One-Waveform RTD

Our full Alloy specification for RTD is in Appendix A. An Alloy specification consist of signatures[1], functions on signatures, facts and predicates. We model values as signatures which can be either ValueZero, ValueOne or ValueX meaning a value can be 0, 1, or X, as shown below.

[1] A set of relation symbols and constant symbols

```
one sig ValueZero extends NonXValue{}
one sig ValueOne extends NonXValue{}
one sig ValueX extends Value{}
sig NonXValue extends Value {}
sig Value{}
```

We model points, one-waveform RTDs, sequential dependencies, words, and assignments as signatures $Point$, $SimpleTimingDiagram$, $SequentialDependency$, $Word$, and Pi respectively, as shown below.

```
sig Point {hasvalue: one Value, index: one Int}
sig SimpleTimingDiagram {haspoint:set Point, hassd:set SequentialDependency}
{#haspoint>1}
sig SequentialDependency {SDis: Point -> Point -> Int -> Int}
sig Word{position: seq NonXValue}{#position>1}
sig Pi {positionis: SimpleTimingDiagram->Word->one Point2Position}
sig Point2Position {is: seq Int}
```

Relations are defined implicitly in signatures declarations. For example, relation *hasvalue* is defined implicitly in signature Point's declaration as a binary relation from signature *Point* to signature *Value*.

- Relation $hasvalue : Point \rightarrow One\ Value$ tells us the value of every point.
- Relation $index : Point \rightarrow int$ tells us the index of any point on the waveform. Since RTD has only one waveform, any pair of points are in relation $\leq^+$ which is the transitive closure of the relation defined in Definition 2.1.4. If point $p \leq^+ q$, then $p.index < q.index$.
- Relation $haspoint : SimpleTimingDiagram \rightarrow set\ Point$ tells us the set of points a RTD has. Relation $hassd : SimpleTimingDiagram \rightarrow set\ Sequential\text{-}Dependency$ tells us the set of sequential dependencies a RTD has. Since every

RTD has only one waveform, there is no concurrent dependency in RTD presumably means no concurrent dependency modeled at all!

- Relation $SDis : \ SequentialDependency \rightarrow Point \rightarrow Point \rightarrow Int \rightarrow Int$ tells us the set of sequential dependencies a signature $SequentialDependency$ has. $SequentialDependency.SDis$ is a subset of SD. In every Alloy specification, the number of one-waveform RTDs and the number of words are bounded. So sequential dependency $p \xrightarrow{[a,+\infty)} q$ can be seen as $(SequentialDependency, p, q, [a, b])$ when

$$Max_{i=0}^{i=n-1}(\pi_{word_i,RTD}(q) - \pi_{word_i,RTD}(p)) \leq b \ ^2$$

  in which $word_i \models_{\pi_{word_i,RTD}} RTD$. n is the number of words that satisfies the RTD and the bound of Alloy specification specified in Alloy language is the upper bound of n.

- Relation *position* maps signature Word to a sequence of values on $\{0, 1\}$, because every word that satisfies one-waveform RTD has only one signal-trace, which means every word is a sequence of integers from $\{0, 1\}$.

- Relation $Positionis$ maps signature Pi to $(SimpleTimingDiagram, Word,$ $Point2Position)$ in which $Point2Position$ tells the position in $Word$ where each point in $SimpleTimingDiagram$ is mapped by Pi.

There are also some interesting facts and predicates as below:

[2]This means for every sequential dependency $p \xrightarrow{[a,b]} q$, if b is greater than distance between where point p and point q are mapped in any word in an Alloy specification that satisfies the specification, then b can be viewed as $+\infty$.

- ```
  pred PointInTDIsRiseFall [td:SimpleTimingDiagram, p:Point]
  {some q:Point | (p.index>0) and (p.hasvalue != ValueX) and
  (q in td.haspoint) and (TDPointNext[td, q, p]) and (p.hasv
  alue !=q.hasvalue) and (q.hasvalue != ValueX)
   }
  ```

  This predicate is true when point p is either a rise of a fall in td. Predicate TDPointNext[td, q, p] is true when q is the preceding point of p.

- ```
  fact TDWellFormed {
  all td:SimpleTimingDiagram | all q:Point| some sd:td.hassd | some p:Point |
  let PrevOfqInsd=(((sd.SDis).Int).Int).q |
  let LowerBoundOfsdp2q=(q.(p.(sd.SDis))).Int |
  let HigherBoundOfsdp2q=Int.(q.(p.(sd.SDis))) |
  (
     (q in td.haspoint)
     and not (PointInTDIsRiseFall[td, q])//when q is neither a rise nor fall
     and (q.index>0) //point q is not the first point.
  )
  =>
  (
     p in td.haspoint and
     q in ToPointOfSD[sd] and p in PrevOfqInsd and
     (
        (p.index=0) //point p is the first point
        or
        PointInTDIsRiseFall[td, p] //point p is a rise/fall
     )
     and //lower bound and higher bound are the same
     (
        LowerBoundOfsdp2q=HigherBoundOfsdp2q and #LowerBoundOfsdp2q=1
        and #HigherBoundOfsdp2q=1
     )
  )
  }
  ```

  According to the SD rewriting process we described in section 2.4.2, we can rewrite every RTD into RTD' in which for every sequential dependency $p \xrightarrow{e}$

q, point p is either a rise/fall or the first point on its waveform. So an equivalent definition of event is: (p, i) is an event if it satisfies one of the following rules:

1. $i = 0$[3].
2. (p, i) is either a rise or a fall[4].
3. (p, i) belongs to a concurrent dependency that contains **a rise or a fall**[5].
4. There exists a sequential dependency $(q, j) \xrightarrow{[a,a]} (p, i)$ in which (q, j) is **a rise or a fall**[6].

By definition of event and well-formed RTD in section 2.1.3 and 2.1.5 respectively, a one-waveform RTD is well-formed iff every point is an event. So for well-formed one-waveform RTD, we only need to make sure for every point (p, i) which is neither a rise/fall nor the first point, there exists a dependency $(q, j) \xrightarrow{[a,a]} (p, i)$ in which (q, j) is either a rise/fall or the first point and a is an nonnegative integer.

The predicate **PointInTDIsRiseFall[td, q]** is true when q is either a rise or a fall in td. **q in ToPointOfSD[sd] and p in PrevOfqInsd** says there exists a dependency $p \xrightarrow{e} q$, and **LowerBoundOfsdp2q=HigherBoundOfsdp2q** says the lower bound of e is the same as the higher bound of e. $p \xrightarrow{e} q$ should be the only dependency between point p and q if td is satisfiable. Because if there exists more than one dependencies, for example $p \xrightarrow{[a,a]} q$ and $p \xrightarrow{[a',a']} q$ $(a \neq a')$, then if there exists a word $w \models_\pi td$, we have $\pi(q) - \pi(p) = a$ and $\pi(q) - \pi(p) = a'$ which contradicts $a \neq a'$. So we have **#LowerBoundOfsdp2q=1 and #HigherBoundOfsdp2q=1** saying that there exists only

[3]It's same as the first part of definition of event in section 2.1.3

[4]It's same as the second part of definition of event in section 2.1.3

[5]Text in bold is the part that is different with the third part of definition of event in section 2.1.3

[6]Text in bold is the part that is different with the last part of definition of event in section 2.1.3

one dependency between point p and q.

This fact says that if for all point q which is neither a rise/fall nor the first point, there exists a point p which is either a rise/fall or the first point such that there exists one dependency $p \xrightarrow{[a,a]} q$, then td is a well-formed RTD.

- ```
  pred PointValueMatch [td:SimpleTimingDiagram, pi:Pi, word:Word]
  {all p:td.haspoint |
  (p.hasvalue!=ValueX) =>
  (p.hasvalue=word.position[pi.PositionPointMappedByPi[td, p, word]])
  }
  ```

  This predicate is true when point consistency [7] is satisfied. It says for every point p, if p does not have value X, then p's value is the same as the value in word at position where p is mapped by pi.

- ```
  pred WaveformConsistencySatisfaction[td:SimpleTimingDiagram, pi:Pi,
                                       word:Word]
  {all p, q:td.haspoint | all j:Int |
  (
     TDPointNext[td, p, q] and (j<pi.PositionPointMappedByPi[td, q, word])
     and (j>=pi.PositionPointMappedByPi[td, p, word]) =>
        ((p.hasvalue != ValueX)=>word.position[j]=p.hasvalue)
  )
  }
  ```

  This predicate is true when waveform consistency is satisfied. It says for every two successive points p and q, values at positions between where p and q are mapped by pi are the same as value of p as long as p's value is not X.

[7] It's defined in Definition 2.2.6

- ```
  pred SequentialDependencySatisfaction [td:SimpleTimingDiagram, pi:Pi,
                                         word:Word]
  {all sd:td.hassd | all p, q:td.haspoint | all i, j:Int |
  let LowerBoundOfSDp2q=(q.(p.(sd.SDis))).Int |
  let HigherBoundOfSDp2qLowerIsi=(i.(q.(p.(sd.SDis)))) |
  ( (p->q in ((sd.SDis).Int).Int) and (i in LowerBoundOfSDp2q) and
    (j in HigherBoundOfSDp2qLowerIsi) ) =>
  ( sub[pi.PositionPointMappedByPi[td, q, word],
        pi.PositionPointMappedByPi[td, p, word]]<=j
  and
  sub[pi.PositionPointMappedByPi[td, q, word],
      pi.PositionPointMappedByPi[td, p, word]]>=i )
  }
  ```

  This predicate is true when dependency consistency is satisfied. It says for every sequential dependency $p \xrightarrow{e} q$, $pi(q) - pi(p) \in e$.

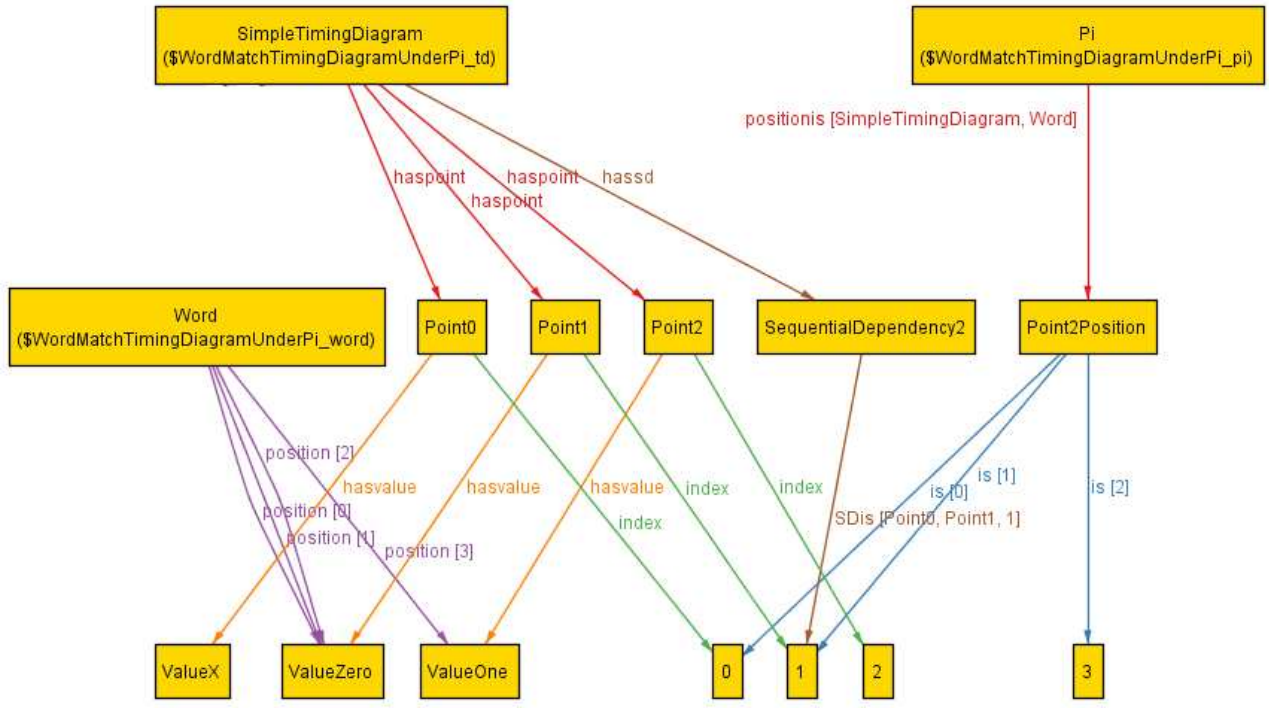

Figure 3.1: An Alloy Specification

Figure 3.1 illustrates an Alloy specification that satisfies our specification. The
  ```

top left box represents the RTD td; arrows labeled *haspoint* represent relation *haspoint*. Those arrows indicate td has three point: $Point_0$ which has value X shown by the arrow labeled hasvalue outgoing from it, $Point_1$ which has value 0 and $Point_2$ which has value 1. The arrow *hassd* indicates td has a sequential dependency. The arrow labeled *SDis[*$Point_0$*,* $Point_1$*, 1]* indicates the sequential dependency is $Point_0 \xrightarrow{[1,1]} Point_1$. The RTD is well-formed, because there exists a sequential dependency $Point_0 \xrightarrow{[1,1]} Point_1$ when $Point_1$ is neither a rise nor a fall.

The Word box represent a word and the arrow *position[0]* indicates value on position 0 is 0. So the word is 0001. The box labeled with Pi represent the assignment π mapping from points of td to position in word. In this model, it maps $Point_0$ to the first position, $Point_1$ to the second position, and $Point_2$ to the fourth position. So $word \models_{pi} td$

3.2 Properties about RTD Checked using Alloy

We have checked some interesting properties about RTD using Alloy Analyzer under certain bounds on number of RTDs, points, assignments, words, ... and so forth. This cannot be considered to be correctness proof of properties because there are infinite many RTDs and words, but it will gain us confidence about the correctness.

3.2.1 Unique Assignment

In Theorem 2.2.1, we have proved that given a RTD, for every word w that satisfies it, there is only one assignment π s.t. $w \models_\pi RTD$. We capture this in the following Alloy assertion:

```
assert PiUnique {
all td:SimpleTimingDiagram | all word:Word | all pi1, pi2:Pi |
(WordMatchTimingDiagramUnderPi[td, pi1, word] and
WordMatchTimingDiagramUnderPi[td, pi2, word])=>
((word.(td.(pi1.positionis))).is=(word.(td.(pi2.positionis))).is)
}
```

This assertion says for every RTD td, every word that satisfies td, if there exists two assignments pi_1 and pi_2 that satisfies every facts and predicates in our specification, then for every point in td, pi_1 and pi_2 map it to the same position in word.

We can check this assertion with the Alloy command:

```
check PiUnique for 6 but exactly 1 SimpleTimingDiagram, exactly 1 Word,
exactly 2 Pi
```

which means we check all models in which there is only one RTD, one word, and two assignments. Figure 3.2 shows the result returned by Alloy Analyzer after it checked assertion *PiUnique*.

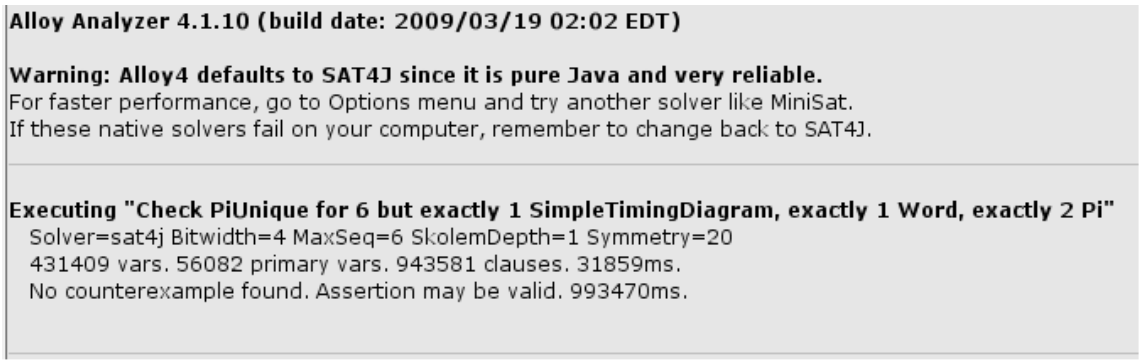

Figure 3.2: Result of Check of PiUnique

3.2.2 Checking Lack of Common Word

In Figure 2.4, we show for two RTDs each having two points, if none of their points has value X and if their first points don't have the same value, then they have no disjunction. In this section, we will show another interesting property about such pair of RTDs. That is, there is no word that satisfies both these two RTDs.

We can use Alloy Analyzer to check this. First we have a predicate saying each of two RTDs has two points having value 0 or 1, and the first points of two RTDs have different value, as follows:

```
pred FirstSecondPointDifferentValueAndNotX[td0, td1:SimpleTimingDiagram] {
    #td0.haspoint=2 and
    #td1.haspoint=2 and
    some p1td0, p1td1:Point |
    TDPointNext[td0, min[td0.haspoint], p1td0] and
    TDPointNext[td1, min[td1.haspoint], p1td1] and
    min[td0.haspoint].hasvalue !=ValueX and
    p1td0.hasvalue != ValueX and
    min[td1.haspoint].hasvalue !=ValueX and
    p1td1.hasvalue != ValueX and
    min[td0.haspoint].hasvalue != min[td1.haspoint].hasvalue and
    p1td0.hasvalue !=p1td1.hasvalue
}
```

Then we have an assertion called *FirstSecondPointDifferent*, as follows:

```
assert FirstSecondPointDifferent {all td0, td1:SimpleTimingDiagram
| all word:Word | all pi0, pi1:Pi |
FirstSecondPointDifferentValueAndNotX[td0, td1]=>
not
(
   WordMatchTimingDiagramUnderPi[td0, pi0, word] and
   WordMatchTimingDiagramUnderPi[td1, pi1, word]
)
}
```

This assertion says for all pair of RTDs td0 and td1, for all word, and for all pair of assignment pi0 and pi1, if td0 and td1 satisfy predicate FirstSecondPointDifferentValueAndNotX, then word cannot satisfy both td0 and td1.

Figure 3.3 shows the result returned by Alloy Analyzer after it checked assertion *FirstSecondPointDifferent*. The 'no counter-example found' indicates the assertion and predicates are unsatisfiable.

```
Executing "Check FirstSecondPointDifferent for 6 but exactly 2 SimpleTimingDiagram, 1 Word, 2 Pi"
  Solver=sat4j Bitwidth=4 MaxSeq=6 SkolemDepth=1 Symmetry=20
  435435 vars. 56124 primary vars. 953668 clauses. 40979ms.
  No counterexample found. Assertion may be valid. 4233ms.
```

Figure 3.3: Result of Check of FirstSecondPointDifferent

Chapter 4

Tight Bound Computation

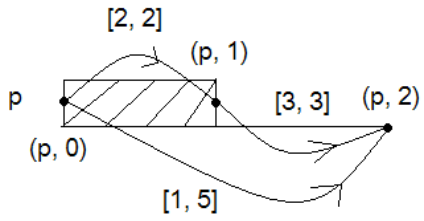

Figure 4.1: RTD having a sequential dependency that is not a tight bound

Sequential dependencies don't necessarily tell us the tightest time separations between each pair of points. That is, there may exist at least one value in some sequential dependency which does not represent the actual distance between the end points in any word. For example, the timing diagram in Figure 4.1 only allows words of length 5 due to the SDs between $(p, 0)$ & $(p, 1)$ and $(p, 1)$ & $(p, 2)$. Values 1-4 in time bound $[1, 5]$ of $(p, 0) \xrightarrow{[1,5]} (p, 2)$ do not occur in any word. The sequential dependency $(p, 0) \xrightarrow{[1,5]} (p, 2)$ is less restrictive than the constraints induced in the rest of the diagram.

Our disjunction algorithm needs time separations between each pair of points to be as tight as possible. In this chapter, we introduce a polynomial time algorithm

to find the tightest time separations[1] between each pair of points. This algorithm first computes an event graph from the given RTD, and computes the tight bound between each pair of points using Floyd-Warshall algorithm [4].

Definition 4.0.1. *Given a RTD=(Point, WF, Value, SD, CD), the tight bound between points* (p, i) *and* (q, j) *denoted* $tbound_{(p,i),(q,j)}$*, has the form of* $\langle a, b\rangle$[2] *in which* a *is an integer or* $-\infty$*,* b *is an integer or* $+\infty$ *such that*

1. *For all word* w*, if* $w \models_\pi RTD$ *then* $\pi((q, j)) - \pi((p, i)) \in tbound_{(p,i),(q,j)}$.
2. *If* $e \in tbound_{(p,i),(q,j)}$ *then there exists a word* w *s.t.* $w \models_\pi RTD$ *and* $\pi((q, j)) - \pi((p, i)) = e$.

In Figure 4.1, $tbound_{(p,0),(p,1)} = [2, 2]$, $tbound_{(p,1),(p,2)} = [3, 3]$ and $tbound_{(p,0),(p,2)} = [5, 5]$.

Computing tight bounds is a problem of finding the maximal achievable time separation [9, 7]

$$d_{ij} = max(\tau(j) - \tau(i))$$

between pairs of events i and j, occurring at times $\tau(i)$ and $\tau(j)$ respectively. The limits on the occurrence times of these events are specified by a system of timing constraints. Different kinds of constraints require different algorithms.

- When timing constraints are of the form $\tau(j) - \tau(i) \leq s_{ij}$ in which s_{ij} is a fixed bound for event i and j, they are called *linear constraints*.
- When timing constraints are of the form $\tau(j) = min(\tau(i) + \delta_{ij})$ or $\tau(j) = max(\tau(i) + \delta_{ij})$ where δ_{ij} is the delay from event i to event j and $l_{ij} \leq \delta_{ij} \leq$

[1] It will be called tight bounds henceforth.

[2] When $a = -\infty$ and $b = +\infty$, it is (a, b);When $a = -\infty$ and $b \neq +\infty$, it is $(a, b]$; When $a \neq -\infty$ and $b = +\infty$, it is $[a, b)$; When $a \neq -\infty$ and $b \neq +\infty$, it is $[a, b]$.

u_{ij} for fixed upper and lower bounds u_{ij} and l_{ij}, they are called *min/max constraints*.

Kenneth McMillan and David Dill have proved that the problem of finding the maximal achievable time separation under min/max constraints is NP-Complete [9].

Timing constraints of RTDs are linear constraints. In chapter 5 of [7], Eduard Cerny *et al.* mentioned that the all pair shortest path algorithm can be use to solve the maximum time separation problem in system of linear constraints. But we haven't found any detailed algorithm nor its correctness proof. So in this chapter, we give an algorithm to solve this problem and its correctness proof.

4.1 Event Graph

Given a RTD, we can interpret its timing relationship in a graph-theoretic point of view by translating it into an event graph. Event graphs are complete directed weighted graphs. Every point in RTD can be viewed as a vertex in its event graph G_{RTD}. When there is a sequential dependency $p \xrightarrow{\langle a,b \rangle} q$, there is a weighted edge from p to q with weight b, and a weighted edge from q to p with weight $-a$. The formal definition of the event graph is as follows:

Definition 4.1.1. *Given a RTD=(Point, WF, Value, SD, CD), its event graph* $G_{RTD}(Point, E, W)$ *is a complete directed weighted graph.* W *is a weight function. For all points* (p, i), (q, j) *in Point:*

- *If* $(p,i) \xrightarrow{[a,b]} (q,j) \in SD$, *then* $((p,i),(q,j)) \in E$, $W((p,i),(q,j)) = b$ *and* $((q,j),(p,i)) \in E$, $W((q,j),(p,i)) = -a$.
- *If* $(p,i) \xrightarrow{[a,+\infty)} (q,j) \in SD$, *then* $((p,i),(q,j)) \in E$, $W((p,i),(q,j)) = +\infty$ *and*

$((q, j), (p, i)) \in E$, $W((q, j), (p, i)) = -a$.

- *For points* (p, i) *and* (q, j) *between which there is no sequential dependency,* $((p, i), (q, j)) \in E$, $W((p, i), (q, j)) = +\infty$ *and* $((q, j), (p, i)) \in E$, $W((q, j), (p, i)) = +\infty$.

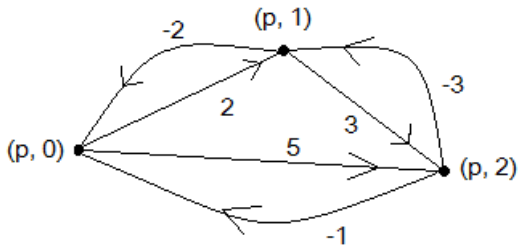

Figure 4.2: Event Graph from RTD in Figure 4.1

Figure 4.2 illustrates an event graph from the RTD in Figure 4.1.

We apply Floyd-Warshall algorithm on event graphs to compute the shortest paths between each pair of points. For any pair of points (p, i) and (q, j), if the shortest path from (p, i) to (q, j) is a and the shortest path from (q, j) to (p, i) is b, then $tbound_{(p,i),(q,j)}$ is $[-b, a]$. This is the idea for computing tight bounds and we will prove its correctness later in this chapter.

4.1.1 RTD Satisfiability and Event Graph

It turns out that we can check whether a given RTD is satisfiable by checking the existence of negative cycles in its event graph. The event graph is well-defined such that its negative cycles tells contradictions among sequential dependencies and concurrent dependencies. For this section, we will prove that given a RTD, it is satisfiable iff there exists at least one negative cycle in its event graph. We first prove there is no negative cycle in event graphs of satisfiable RTDs.

Theorem 4.1.1. *For any satisfiable RTD, there is no negative cycle in its event graph G_{RTD}.*

Proof.

If there exists an edge with weight $+\infty$ in this cycle, certainly this cycle is not

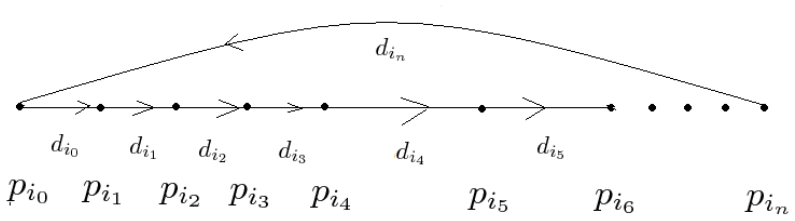

Figure 4.3: a cycle in G_{RTD}

negative. So we focus only on the case when no edge has a $+\infty$ weight.

In an event graph, the weight between two vertices (points) can be either positive or negative. When some edge from vertices p to q has weight $d_{p,q} \geq 0$, according to definition of event graph, there exists a sequential dependency with upper bound $d_{p,q}$. Since RTD is satisfiable, there exists a word w such that

$$V_q - V_p \leq d_{p,q} \text{ when } d_{p,q} \geq 0$$

in which V_p means the position where point p is mapped in w.

When some edge from vertices p to q has weight $d_{p,q}$ and $d_{p,q} \leq 0$, according to definition of event graph, there exists a sequential dependency with lower bound $-d_{p,q}$. Since RTD is satisfiable, we have

$$V_p - V_q \geq -d_{p,q} \text{ when } d_{p,q} \leq 0$$

which means

$$V_q - V_p \leq d_{p,q} \text{ when } d_{p,q} \leq 0$$

For any cycle, as illustrated by Figure 4.3, we have the following set of inequalities:

$$V_{P_{i_1}} - V_{P_{i_0}} \leq d_{i_0}$$
$$V_{P_{i_2}} - V_{P_{i_1}} \leq d_{i_1}$$
$$V_{P_{i_3}} - V_{P_{i_2}} \leq d_{i_2}$$
$$... \leq ..$$
$$... \leq ..$$
$$... \leq ..$$
$$V_{P_{i,0}} - V_{P_{i,n}} \leq d_{i_n}$$

When we add all these inequalities, we get

$$0 \leq \sum_{j=0}^{n} d_{i_j}.$$

Since the edges in equations above form a cycle in event graph, left sides of all inequalities above add up to 0. So for any cycle in event graph, the sum of weight is not negative. So we have proved the theorem. □

We then prove the theorem that given a RTD, if its event graph has no negative cycle then it is satisfiable. To prove this theorem, we first give a way to construct a word for RTD, then we prove the word we construct satisfies RTD.

Theorem 4.1.2. *For any regular timing diagram RTD, if there is no negative cycle in its event graph G_{RTD} then RTD is satisfiable.*

Proof. We construct the word satisfying RTD as follows:

1: Given a RTD whose event graph G_{RTD} has no negative cycle
2: Add a vertex v to G_{RTD}
3: **for all** vertex p other than v **do**
4: add an edge $v \to p$ with weight 0
5: Apply the Floyd-Warshall algorithm on the graph we get from above. Let the matrix returned by Floyd-Warshall algorithm be M
6: **for all** vertex p other than v **do**
7: Assign the integer $M_{v,p}$ to p //$M_{v,p}$ is the position where point p is mapped in the word we are constructing.
8: Assign appropriate values from {0, 1} at position $M_{v,p}$ for every point p.
9: **for all** consecutive points (p, i) and $(p, i+1)$ **do**
10: **if** (p, i) has value 0 or 1 **then**
11: Assign 0 or 1 to each position between $M_{v,(p,i)}$ and $M_{v,(p,i+1)}$
12: **else**
13: Arbitrarily assign values from $\{0, 1\}$ to positions between $M_{v,(p,i)}$ and $M_{v,(p,i+1)}$

We just need to prove the word we construct by the algorithm above satisfies RTD.

We first prove the assignment of $M_{v,p}$ to every point p in RTD satisfies all dependency consistencies. This is to prove that for every sequential dependency $p \xrightarrow{[a,b\rangle} q$, $M_{v,q} - M_{v,p} \in [a, b\rangle$ which is to prove $M_{v,q} - M_{v,p} \leq b$ and $M_{v,p} - M_{v,q} \leq -a$ [3]. By definition of event graph, when there is a sequential dependency $p \xrightarrow{[a,b\rangle} q$ in RTD, there are two edges in its event graph G_{RTD}: (p, q) with weight b and (q, p) with weight $-a$. For edge (p, q) with weight b, since $M_{v,p}$ and $M_{v,q}$ are the shortest distances from v to p and q respectively, we have

$$M_{v,q} \leq M_{v,p} + b$$

which means

$$M_{v,q} - M_{v,p} \leq b$$

For similar reasons, we can prove

[3]We assume all concurrent dependencies in RTD have been eliminated by the algorithm in section 2.4.1

$$M_{v,p} - M_{v,q} \leq -a$$

So the assignment of $M_{v,p}$ to every point p in RTD satisfies all dependency consistencies.

It's not difficult to see that the word returned by the algorithm above satisfies point consistency by line 8, and it satisfies waveform consistency by lines from line 9 to line 13.

So we have proved the theorem. □

Example 4.1.1. *Given RTD illustrated in Figure 4.1, there is no negative cycle in its event graph in Figure 4.2. By the algorithm on page 36,* $M_{v,(p,0)} = -5$, $M_{v,(p,1)} = -3$ *and* $M_{v,(p,2)} = 0$. *Since* $(p, 0)$ *has value* X, *we can assign either 0 or 1 at position* $M_{v,(p,0)}$. $(p, 1)$ *and* $(p, 2)$ *have value 0, so we can only assign 1 at position* $M_{v,(p,1)}$ *and* $M_{v,(p,2)}$. *At each position between* $M_{v,(p,0)}$ *and* $M_{v,(p,1)}$, *we can assign either 0 or 1. At each position between* $M_{v,(p,1)}$ *and* $M_{v,(p,2)}$, *we can only assign 0. So the word return by the algorithm on page 36 is:*

$$(\{p\}, p : 110000)$$

Theorem 4.1.3. *Given a regular timing diagram RTD, it is satisfiable iff there is no negative cycle in its event graph* G_{RTD}.

Proof. By Theorem 4.1.1 and Theorem 4.1.1, we can prove this theorem. □

4.2 Tight Bound Algorithm

Our tight bound algorithm on page 45 has three parts, the first part accepts an RTD and returns its event graph. The second part applies Floyd-Warshall algorithm on the event graph to find shortest paths between all pair of points. The third returns tight bounds between all pair of points. This algorithm runs in polynomial time. It

has complexity $O(|Point|^3)$.

From the tight bound algorithm on page 45, we can get the following lemma:

Lemma 4.2.1. *For all RTD, for any pair of points p and q from G_{RTD}, $bound_{p,q} = \langle a, b\rangle$ iff $bound_{q,p} = \langle -b, -a\rangle$, bound is the matrix returned by the tight bound algorithm on page 45.*

Proof. By tight bound algorithm, b is the shortest distance from p to q in event graph, and $-a$ is the shortest distance from q to p in event graph. So we have proved this lemma. □

4.2.1 Correctness Proof

By Definition 4.0.1, in order to prove the matrix *bound* returned by our tight bound algorithm produces tight bounds between any pair of points, we need to prove two theorems. The first (Theorem 4.2.1) is that every word corresponds to integers from *bound*; this theorem satisfies item 1 of Definition 4.0.1. The second (Theorem 4.2.2) is that for any pair of points p and q, every integer in $bound_{p,q}$ corresponds to at least one word; this satisfies item 2 of Definition 4.0.1.

Theorem 4.2.1. *For every word w, if $w \models_\pi RTD$ then $\pi(q) - \pi((p,i)) \in bound_{p,q}$ for every pair of points p and q, where bound is the matrix returned by the tight bound algorithm on page 45.*

Proof. By the tight bound algorithm, $bound_{p,q}$ can be defined by a table which is:

	when $D(q,p) = +\infty$	when $D(q,p) \neq +\infty$
when $D(p,q) = +\infty$	$bound_{p,q} = (-D(q,p), D(p,q))$	$bound_{p,q} = [-D(q,p), D(p,q))$
when $D(p,q) \neq +\infty$	$bound_{p,q} = (-D(q,p), D(p,q)]$	$bound_{p,q} = [-D(q,p), D(p,q)]$

We will only prove

$$\pi(q) - \pi(p) < D(p,q) \text{ or } \pi(q) - \pi(p) \leq D(p,q)$$

depending on whether $D(p,q)$ is $+\infty$ or not. The reason is by Lemma 4.2.1

$$\pi(q) - \pi(p) > -D(q,p) \text{ or } \pi(q) - \pi(p) \geq -D(q,p)$$

can be proved by

$$\pi(p) - \pi(q) < D(q,p) \text{ or } \pi(p) - \pi(q) \leq D(q,p)$$

When $D(p,q) = +\infty$, since word w is finite, we have $\pi(q) - \pi(p) < D(p,q)$ trivially.

When $D(p,q) \neq +\infty$, by our algorithm there exists a path $p \to r \to ... \to q$ in event graph G_{RTD} along which there is no edge having weight $+\infty$. In the proof of Theorem 4.1.1, we have proved in event graph G_{RTD}, for any edge $p \to r$ with weight $d_{p,r}$, for any word w that satisfies RTD, $\pi(r) - \pi(p) \leq d_{p,r}$. Each edge along the path corresponds to one inequality of this form. All left side of inequalities from the path $p \to r \to ... \to q$ add up to $\pi(q) - \pi(p)$ and all weights along the path add up to $D(p,q)$. So we have proved

$$\pi(q) - \pi(p) \leq D(p,q)$$

which proves this theorem. □

Theorem 4.2.2. *Given a satisfiable regular timing diagram RTD, after we run the tight bound algorithm on it, for any pair of points p and q, for any integer* $m \in bound_{p,q}$*, there exists a word* $w \models_\pi RTD$ *s.t.* $\pi(q) - \pi(p) = m$.

Proof. Let the event graph of RTD be G_{RTD}. The sketch of our proof is we first change the weight of edges (p,q) and (q,p) in G_{RTD} to m and $-m$ respectively to get another graph G', then we prove G' has no negative cycle. We prove G' has no negative cycle because we need to prove the corresponding timing diagram RTD' (whose event graph is G') is satisfiable[5].

We define graph G' as: $V_{G'} = V_{G_{RTD}}$, $E_{G'} = E_{G_{RTD}}$ and for each edge e

[5]By Theorem 4.1.1, G' is satisfiable iff there is no negative cycle in RTD'.

other than (p, q) and (q, p), $WF_{G'}(e) = WF_{G_{RTD}}(e)$ while $WF_{G'}((p, q)) = m$ and $WF_{G'}((q, p)) = -m$.

We then prove there is no negative cycle in G'. Since RTD is satisfiable, there is no negative cycle in its event graph G_{RTD}. For each cycle in G', if each edge in it is neither (p, q) nor (q, p), it is not negative cycle because this is a cycle that is also in G_{RTD}. For each cycle in G' which contains edges (p, q) or (q, p):

- If it contains (p, q), let the cycle be $p \to q \to r \to ... \to s \to p$ and let the sum of weight along path $q \to r \to ... \to s \to p$ be S. Since this cycle is also in G_{RTD} but with different weight of edge (p, q), we have:

  $$W_{G_{RTD}}((p, q)) + S \geq 0$$

  since $W_{G'}((p, q)) = m$ and $m \in bound_{p,q}$[6]. Let $bound_{p,q}$ be $[a, b]$. By the tight bound algorithm, $-a$ is the shortest distance from q to p in G_{RTD}. Therefore, $S \geq -a$. Since $m \in [a, b]$, $m - a \geq 0$. Therefore $m + S \geq 0$ which means this cycle is not a negative cycle in G'.

- Proof is similar when (q, p) is in the cycle.

So we have proved that G' has no negative cycle. Using the same technique as the algorithm on page 36, we first add a new vertex v to G' and compute the shortest distance between v and each vertex p in G' which is $M_{v,p}$. By definition of G', it keeps all dependencies in G_{RTD}. Therefore, by Theorem 4.1.1 the assignment of $M_{v,p}$ to each vertex p in G' satisfies all dependencies in G_{RTD}. Therefore, with $M_{v,p}$ for each vertex p, we can construct a word w by assigning appropriate values from $\{0, 1\}$ to positions in w by the algorithm on page 36. $w \models_\pi RTD$ and $\pi(q) - \pi(p) = m$.

So we have proved this theorem. □

[6]The matrix *bound* is the result we get after we run the tight bound algorithm on G_{RTD}

4.3 RTD Transformation using Tight Bound Computation

As we said in section 2.4, we transform given RTDs into another form without changing their language in order to simplify the design of our tight bound algorithm and the proof of its correctness and completeness.

Definition 4.3.1. *In RTD=*$\{$*Point, WF, Value, SD, CD*$\}$*, a point* $(p, i) \in$ *Point is a non-transition point if* (p, i) *is not the first point on waveform* p *and* $Value((p, i)) \neq Value((p, i-1))$.

In a well-formed RTD, there can be some points which are not transitions. For example in Figure 2.1, point $(a, 2)$, $(b, 2)$, $(b, 4)$, and $(c, 2)$ are non-transition point.

Eliminating non-transition points simplifies the disjunction algorithm and its proofs. So in this section, we present a method to eliminate non-transition points which are neither the first nor the last points.

Our algorithm to eliminate non-transition points in previous sentence is as follows:

1: Given *RTD*={*Point, WF, Value, SD, CD*}

2: Apply method in section 2.4.1 to eliminate all concurrent dependencies.

3: Apply method in section 2.4.2 to rewrite all sequential dependencies.

4: **for** each point p which is non-transition point and is neither the first nor the last point **do**

5: there exists a sequential dependency $t \xrightarrow{[k,k]} p$

6: **for** each sequential dependency $s \xrightarrow{\langle x,y \rangle} p$ **do**

7: Eliminate this sequential dependency by adding sequential dependency $s \xrightarrow{\langle x-k,y-k \rangle} t$

8: Get points q and r s.t. q is the preceding point of p and p is the preceding point of r

9: Add two sequential dependencies $t \xrightarrow{tbound_{t,q}} q$ and $t \xrightarrow{tbound_{t,r}} r$[7]

10: Remove point p

This algorithm will terminate because during each iteration, we eliminate one non-transition point without adding any point.

Example 4.3.1. *For the RTD in Figure 2.1, after we eliminate non-transition points which are neither the first nor the last points, we get another RTD as shown in Figure 4.4*

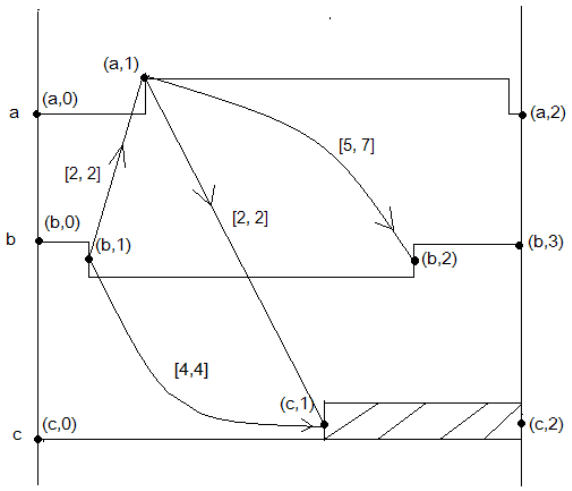

Figure 4.4: Result after Non-transition Points Elimination

Lemma 4.3.1. *Let $tbound_{t,q}$ and $tbound_{t,r}$ at line 9 of the non-transition elimination process be $\langle b, c\rangle$ and $\langle d, e\rangle$ respectively. Then $c < k$ and $d > k$.*

Proof. Since $\langle b, c\rangle$ and $\langle k, k\rangle$ are the tight bounds between t & q and t & p respectively, there exists a word $w \models_\pi RTD$ s.t. $\pi(q) - \pi(t) = c$ and $\pi(p) - \pi(t) = k$. Since $q \leq^+ p$, $\pi(p) > \pi(q)$ which means $\pi(p) - \pi(t) > \pi(q) - \pi(t)$ which means $k > c$.

We can use the similar way to prove $k > c$. □

Theorem 4.3.1. *Given RTD, if there is a point p which is neither the first nor the last point and it's a non-transition point, and if p is in only one sequential dependency which is $t \xrightarrow{[k,k]} p$, then after we add two sequential dependencies $t \xrightarrow{tbound_{t,q}} q$ and $t \xrightarrow{tbound_{t,r}} r$ in which q is the preceding point of p and p is the preceding point of r, we can remove p and sequential dependency $t \xrightarrow{[k,k]} p$ without changing its language.*

Proof. Given RTD, after we add two sequential dependencies $t \xrightarrow{tbound_{t,q}} q$ and $t \xrightarrow{tbound_{t,r}} r$, we get another timing diagram RTD′. For each word w in $\mathcal{L}$(RTD′), w is in $\mathcal{L}$(RTD) because RTD keeps the same points, sequential dependencies, and concur-

rent dependencies as RTD′ except for $t \xrightarrow{tbound_{t,q}} q$ and $t \xrightarrow{tbound_{t,r}} r$. For each word w in $\mathcal{L}$(RTD), w is in $\mathcal{L}$(RTD′) because by definition of tight bound, if $w \models_\pi RTD$, then $\pi(q) - \pi(t) \in tbound_{t,q}$ and $\pi(r) - \pi(t) \in tbound_{t,r}$ which means π also satisfies the only two sequential dependencies in RTD′ not in RTD.

We then get another timing diagram RTD″ after we remove point p and sequential dependency $t \xrightarrow{[k,k]} p$ from RTD′. We need to prove $\mathcal{L}$(RTD′)= $\mathcal{L}$(RTD″).

$\mathcal{L}$(RTD′)⊆ $\mathcal{L}$(RTD″) because RTD″ keeps all the same points, sequential dependencies and concurrent dependencies as RTD′ except for point p and sequential dependency $t \xrightarrow{[k,k]} p$.

We then prove $\mathcal{L}$(RTD″)⊆ $\mathcal{L}$(RTD′). Let $tbound_{t,q}$ be $\langle b, c\rangle$ and $tbound_{t,r}$ be $\langle d, e\rangle$. For every word $w \models_\pi$RTD″, we have $\pi(q) - \pi(t) \leq c$ which means

$$\pi(q) \leq \pi(t) + c$$

We also have $\pi(r) - \pi(t) \geq d$ which means

$$\pi(r) \geq \pi(t) + d$$

Since we have proved in Lemma 4.3.1 that $c < k < d$, we have

$$\pi(q) < \pi(t) + k < \pi(r)$$

which means the position $\pi(t) + k$ in w is always between $\pi(q)$ and $\pi(r)$. So we can build an assignment π' which is:

$$\pi'(p) = \begin{cases} \pi(p) & \text{if p is a point in RTD″} \\ \pi(t) + k & \text{if p is the only point in RTD′ not in RTD″} \end{cases}$$

in which $t \xrightarrow{[k,k]} p$ is the only sequential dependency in RTD′ not in RTD″ and $w \models_{\pi'}$RTD′.

So we have proved this theorem. □

Algorithm 1 Tight Bound Algorithm

Given a RTD=(Point, WF, Value, SD, CD), returns matrix *bound* on points.

//The first part

The event graph of it is $G_{RTD}(Point, E, W)$ in which W is a weight function.

$E \leftarrow \{\}$

for all $(p,i) \in Point$ **do**

  for all $(q,j) \in Point$ and $(q,j) \neq (p,i)$ **do**

    if $(p,i) \xrightarrow{[a,b]} (q,j) \in SD$ **then**

      $E \leftarrow E \bigcup \{((p,i),(q,j)),((q,j),(p,i))\}$

      $W((p,i),(q,j)) = b$, $W((q,j),(p,i)) = -a$

    else if $(p,i) \xrightarrow{[a,+\infty)} (q,j) \in SD$ **then**

      $E \leftarrow E \bigcup \{((p,i),(q,j)),((q,j),(p,i))\}$

      $W((p,i),(q,j)) = +\infty$, $W((q,j),(p,i)) = -a$

    else

      $E \leftarrow E \bigcup \{((p,i),(q,j)),((q,j),(p,i))\}$

      $W((p,i),(q,j)) = +\infty$, $W((q,j),(p,i)) = +\infty$

//The second part

D=Floyd-Warshall(G_{RTD})[4]

//The third part

for all $(p,i) \in Point$ **do**

  for all $(q,j) \in Point$ and $(q,j) \neq (p,i)$ **do**

    if $D((p,i),(q,j)) > 0$ **then**

      if $D((p,i),(q,j)) \neq +\infty$ **then**

        if $D((q,j),(p,i)) \neq +\infty$ **then**

          $bound_{(p,i),(q,j)} = [-D((q,j),(p,i)), D((p,i),(q,j))]$

        else

          $bound_{(p,i),(q,j)} = (-D((q,j),(p,i)), D((p,i),(q,j))]$

      else

        if $D((q,j),(p,i)) \neq +\infty$ **then**

          $bound_{(p,i),(q,j)} = [-D((q,j),(p,i)), D((p,i),(q,j)))$

        else

          $bound_{(p,i),(q,j)} = (-D((q,j),(p,i)), D((p,i),(q,j)))$

Return *bound*

Chapter 5

Disjunction Algorithm

Recall that in section 2.3, we illustrated that not every pair of regular timing diagrams has a disjunction that can be represented as a timing diagram. In this chapter, we give an algorithm that decides whether given pair of regular timing diagrams have a disjunction[1], and returns a disjunction if two given timing diagrams have one.

We first give an algorithm that decides the disjunction problem for timing diagrams with only one waveform. Then we show that the disjunction problem for timing diagrams with multiple waveforms can be solved using the result of one waveform case.

5.1 Disjunction of One-Waveform RTD

In this section, we focus on regular timing diagrams with only one waveform. So we assume all terms mentioned in this section as 'regular timing diagram' or 'timing diagram' really mean 'regular timing diagram with one waveform'.

[1]The word 'disjunction' means a timing diagram whose language is the union of languages of given timing diagrams. It is defined in Definition 2.3.1.

Regular timing diagrams contain points and timing constraints on pairs of points. In order to find a disjunction of two given timing diagrams, we therefore need to find a way to appropriately disjoin points and timing constraints from the given timing diagrams. Our approach is that we find pairs of points p and q from two given timing diagrams respectively. If there exists a disjunction of two given timing diagrams, then there exists points r such that for any word from one of given timing diagrams, r is mapped to the same position as p (in the timing diagram where point p is in) or q (in the timing diagram which point q is in) and the value of r is the disjunction of values of p and q. In regular timing diagrams, values of points can be either 0, 1, or X. We define disjunction on $\{0, 1, X\}$ as follow:

$\vee$	**0**	**1**	**X**
0	**0**	**X**	**X**
1	**X**	**1**	**X**
X	**X**	**X**	**X**

In the following sections, we formalize the notion of points above as *concrete points* and some examples illustrating we sometimes might need to add concretes points to given timing diagrams in order to find their disjunction.

5.1.1 Concrete Points

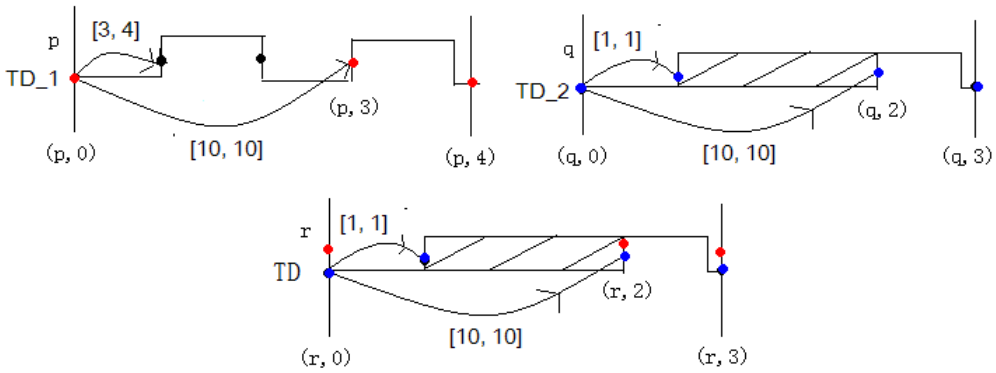

Figure 5.1: Concrete Point

Figure 5.1 shows two regular timing diagrams TD_1 and TD_2 and their disjunction timing diagram TD. We notice that there exists pairs of points p and q from TD_1 and TD_2 respectively such that there is a point r in TD that corresponds to them. By 'corresponds', we mean for any word w in $\mathcal{L}(TD_1)$ (or $\mathcal{L}(TD_2)$), point p (or q) is mapped to the same position in w as point r. For example, the first point $(p, 0)$ in TD_1 and the first point $(q, 0)$ in TD_2 correspond to the first point $(r, 0)$ in TD. The last point $(p, 4)$ in TD_1 and the last point $(q, 3)$ in TD_2 correspond to the last point $(r, 3)$ in TD. Point $(p, 3)$ from TD_1 and point $(q, 2)$ from TD_2 correspond to point $(r, 2)$ in TD. For any word w that satisfies TD_1, $(p, 3)$ is mapped to position 10 and $(r, 2)$ is mapped to position 10 in w. For any word w' that satisfies TD_2, $(q, 2)$ is mapped to position 10 and $(r, 3)$ is mapped to position 10 in w'.

We call points $(p, 0)$, $(p, 3)$ and $(p, 4)$ *concrete points* in TD_1 and we call points $(q, 0)$, $(q, 2)$ and $(q, 3)$ concrete points in TD_2.

Definition 5.1.1. *Given regular timing diagrams TD_0 and TD, for all word w such that $w \models_{\pi_0} TD_0$ and $w \models_{\pi} TD$, if for point p from TD_0 and r from TD, $\pi_0(p) = \pi(r)$ then p and r are concrete points and (p, r) is a concrete pair.*

The first and last points are always concrete points.

For regular timing diagrams with one waveform, we define their prefix as follow:

Definition 5.1.2. *Given RTD (Point, WF, Value, SD, CD) in which* $|WF| = 1$ *(without loss of generality, let* $WF = \{p\}$*), a prefix of it* $pre_{RTD,(p,i)}$ *is a tuple* $(Point_{(p,i)}, WF, Value_{(p,i)}, SD_{(p,i)}, CD_{(p,i)})$ *in which*

- *The nonnegative integer* i *is less than or equal to the number of points on* p*.*
- $Point_{(p,i)} = \{(p, j) \mid 0 \leq j < i\}$
- $Value_{(p,i)}$ *is defined on domain* $Point_{(p,i)}$ *and* $\forall 0 \leq j < i. Value_{(p,i)}((p, j)) = Value((p, j))$*.*
- $SD_{(p,i)} = \{(p, m) \xrightarrow{e} (p, n) \mid 0 \leq m, n < i,\ (p, m) \xrightarrow{e} (p, n) \in SD\}$
- $\{(p, i_0), (p, i_1), ..., (p, i_n)\} \in CD_{(p,i)}$ *iff* $\{(p, i_0), (p, i_1), ..., (p, i_n)\} \in CD$ *and* $\forall m \in [0, n].\ 0 \leq i_m \leq i$*.*

By the above definition, if a regular timing diagram TD is well-formed, then all of its prefixes except the one with only one point (the first point of TD) are well-formed.

Our algorithm starts from the first point, gradually enlarges the prefix of both TD_0 and TD_1 for which we have found a disjunction by disjoining concrete points and adding appropriate timing dependencies from prefix to the new added concrete points.

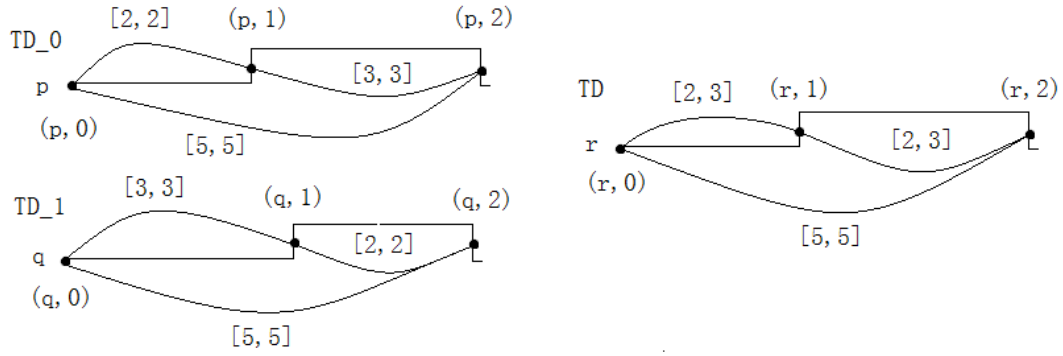

Figure 5.2: Disjunction algorithm example

Example 5.1.1. *Figure 5.2 shows an example on which we apply the disjunction algorithm. At the beginning, the prefix of TD_0 and TD_1 each contains only one point which is the first point.*

- *We add one point to TD whose value is the disjunction of values of $(p, 0)$ and $(q, 0)$.*
- *We extend prefixes of TD_0 and TD_1 to include $(p, 1)$ and $(q, 1)$ respectively and we add point $(r, 1)$ to TD whose value is the disjunction of values of $(p, 1)$ and $(q, 1)$.*
- *We add a sequential dependency $(r, 0) \xrightarrow{[2,3]} (r, 1)$ whose bound $[2, 3]$ is the union of that of sequential dependencies $(p, 0) \xrightarrow{[2,2]} (p, 1)$ and $(q, 0) \xrightarrow{[3,3]} (q, 1)$ from TD_0 and TD_1 respectively.*
- *We add point $(r, 2)$ and sequential dependencies $(r, 1) \xrightarrow{[2,3]} (r, 2)$, $(r, 0) \xrightarrow{[5,5]} (r, 2)$ to TD.*

TD is a disjunction of TD_0 and TD_1.

5.1.2 Adding Points

Our algorithm tries to find concrete points in TD_0 and TD_1, then generates a disjunction of TD_0 and TD_1 by disjoining values of concrete points and generating correct dependencies in TD. But sometimes we need to add some points to TD_0 or TD_1 to help us generate their disjunction TD.

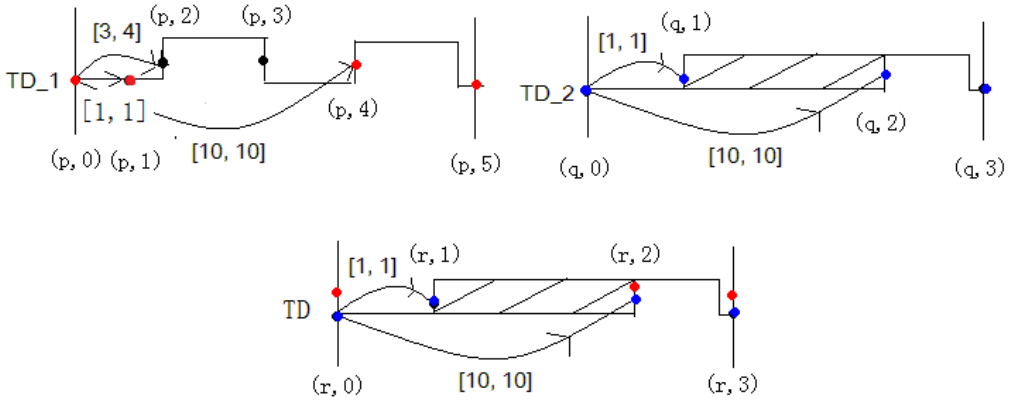

Figure 5.3: Adding point

For example: the timing diagrams in Figure 5.3 are similar to timing diagrams shown in Figure 5.1, except that we add a point with value 0 between the first and second points in TD_1 and a sequential dependency with bound $[1, 1]$. What TD_2 really means is that there is one 0 followed by nine uncertain values which can be either 0 or 1. What TD_1 really means before we add $(p, 1)$ is that there are either three or four 0 followed by some 1 and 0 and the total number of digits before the last rise should be 10. TD_1 can also be interpreted as there is one 0 followed by either two or three 0 followed by some 1 and 0 and the total number of digits before the last rise should be 10. If we interprete TD_1 in the second way, it's easy to find concrete points in it which are $(p, 0)$, $(p, 1)$, $(p, 4)$ and $(p, 5)$ and generate disjunction timing diagram TD.

This example shows that properly adding points can help us finding disjunction

of given regular timing diagrams. There are also examples showing that if we don't add any point, we cannot find a correct disjunction of given timing diagrams. A simple example of this kind is in Figure 5.4:

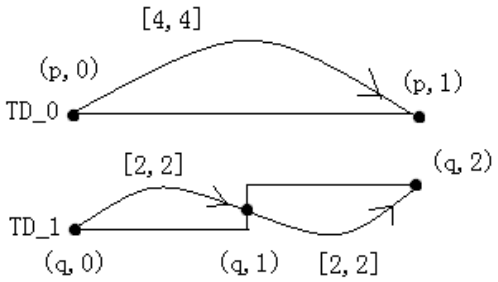

Figure 5.4: Example illustrating adding point is necessary

In this example, both TD_0 and TD_1 are well-formed, and there are no non-transition points between the first and the second points (not-inclusive). If we don't add any point between $(p, 0)$ and $(p, 1)$ in TD_0, we cannot find their disjunction. The reason is that after we find two points which capture disjunctions of $(p, 0)$, $(q, 0)$ and $(p, 1)$, $(q, 1)$ respectively, there is still a point $(q, 2)$ in TD_1 and we run out of points in TD_0.

5.1.3 Disjunction Type

If two regular timing diagrams TD_0 and TD_1 have disjunction TD, then there can be several relationships between $\mathcal{L}(TD_0)$ and $\mathcal{L}(TD_1)$.

- $\mathcal{L}(TD_0)$=$\mathcal{L}(TD_1)$. We call this type 'full-full'. In this case, $\mathcal{L}(TD)$=$\mathcal{L}(TD_0)$=$\mathcal{L}(TD_1)$.

- $\mathcal{L}(TD_0)\subset\mathcal{L}(TD_1)$ (or $\mathcal{L}(TD_1)\subset\mathcal{L}(TD_0)$). We call this type 'full-partial' (or 'partial-full'). In this case, $\mathcal{L}(TD)$=$\mathcal{L}(TD_1)$ (or $\mathcal{L}(TD)$=$\mathcal{L}(TD_0)$).

- $\mathcal{L}(TD_0) \not\subseteq \mathcal{L}(TD_1)$ and $\mathcal{L}(TD_1) \not\subseteq \mathcal{L}(TD_0)$. We call this type 'partial-partial'. In this case, $\mathcal{L}(TD_0)$ and $\mathcal{L}(TD_1)$ partition $\mathcal{L}(TD)$ but neither of them equals $\mathcal{L}(TD)$.

Definition 5.1.3. *We can define an operator* $*$ *on* $\{Full\text{-}Full, Full\text{-}Partial, Partial\text{-}Full, Partial\text{-}Partial\}$ *as follows:*

$*$	$Full\text{-}Full$	$Full\text{-}Partial$	$Partial\text{-}Full$	$Partial\text{-}Partial$
$Full\text{-}Full$	$Full\text{-}Full$	$Full\text{-}Partial$	$Partial\text{-}Full$	$Partial\text{-}Partial$
$Full\text{-}Partial$	$Full\text{-}Partial$	$Full\text{-}Partial$	UNDEFINED	UNDEFINED
$Partial\text{-}Full$	$Partial\text{-}Full$	UNDEFINED	$Partial\text{-}Full$	UNDEFINED
$Partial\text{-}Partial$	$Partial\text{-}Partial$	UNDEFINED	UNDEFINED	UNDEFINED

Remember that timing diagrams which have different syntax may have the same language. So the timing diagram returned by our disjunction algorithm might be different from any of given timing diagrams.

5.1.4 Disjunction Algorithm

Our disjunction algorithm accepts two well-formed regular timing diagrams and either returns a well-formed regular timing diagram, whose language is the disjunction of that of the given two, or returns 'NO DISJUNCTION' otherwise.

This algorithm starts from the first points of TD_0 and TD_1, and tries to enlarge the two prefixes on TD_0 and TD_1 to find disjunction of two enlarged prefixes. We apply algorithms on page 15, 17 and 42 before we call the disjunction algorithm. These guarantee that there are no non-transition points in TD_0 and TD_1. Given this, at each step we enlarge two prefixes on TD_0 and TD_1, we always enlarge each by one point. On page 55, we give a table which summarizes all possible cases when

two prefixes can be enlarged. In the table:

- The second column describes values of last points in prefixes and points to be included in enlarged prefixes. For example, in the second column of case 1, X X;X m means the last point in TD_0's prefix has value X and the point to be included in enlarged prefix in TD_0 has value X; the last point in TD_1's prefix has value X and the point to be included in enlarged prefix in TD_0 has value m. $m, n \in \{0, 1\}$ and $m \neq n$.

- The third column describes disjunction types. The last row describes constraints on the tight bound between last point of prefix and point to be included. $[a, b]$ and $[a', b']$ are tight bounds between last points of prefixes and points to be included in enlarged prefixes.

case 1	X X; X m	full-partial	$[a', b'] \subseteq [a, b]$
case 2	X m; X n	partial-partial	$[a', b'] = [a, b] = [1, 1]$
case 3	X m; X m	full-full or full-partial or partial-partial	$b \geq a'$ and $b' \geq a$
case 4	X X; m X	full-partial	$[a', b'] \subseteq [a, b]$
case 5	X m_1; m_0 m_1	full-partial	$[a', b'] \subseteq [a, b]$ and $m_0, m_1 \in \{0, 1\}$
case 6	m X; n X	partial-partial	$[a', b'] = [a, b] = [1, 1]$
case 7	m_0 m_1; n m_1	partial-partial	$[a', b'] = [a, b] = [1, 1]$ and $m_0 \neq n$ $(n, m_0, m_1 \in \{0, 1\})$
case 8	m X; m X	full-full or full-partial or partial-partial	$b \geq a'$ and $a \leq b'$
case 9	m_0 X; m_0 m_1	full-partial	$m_0, m_1 \in \{0, 1\}$ $[a', b'] \subseteq [a, b]$
case 10	m_0 m_1; m_0 n	partial-partial	$m_0, m_1, n \in \{0, 1\}$ $m_1 \neq n$ $[a, b] = [a', b'] = [1, 1]$
case 11	m n; m n	full-full or full-partial or partial-partial	$m \neq n$ $b \geq a'$ and $a \leq b'$
case 12	X X; m n	full-partial	$m \neq n$ $[a', b'] \sqsubseteq [a, b]$

On page 57, we give our disjunction algorithm for one-waveform regular timing diagrams.

5.1.5 Correctness Proof

By Definition 5.1.2, for any one-waveform timing diagram TD, if any of its prefix except the one with only one point is not well-formed, then TD is not well-formed. Line 12 of function *add-dependencies* on page 58 guarantees that if the disjunction algorithm returns a timing diagram, it should be well-formed.

To prove the disjunction algorithm for one-waveform RTDs is correct, we need to prove:

1. Given TD_0 and TD_1, if the disjunction algorithm returns a TD, then their disjunction exists and TD is one of them[2].

2. If the disjunction algorithm returns 'NO DISJUNCTION', then their disjunction does not exist.

The disjunction algorithm iterates over inputs TD_0 and TD_1. Each iteration accepts the prefix from the last iteration and extends the prefix to a larger one until all points and timing dependencies are considered. In each iteration, we have two choices: either add a new point that was not in original timing diagram TD_0 or TD_1 or extend prefixes without adding any point. So when the disjunction algorithm returns TD, there exists a sequence of choices. We can describe this sequence of choice as a sequence of point triples. The first and last point triples in this sequence contains the first and last points on TD_0, TD_1 and TD respectively. For example, the first point triple is $((p,0),(q,0),(r,0))$ in which $(p,0)$, $(q,0)$, and $(r,0)$ are the first points in TD_0, TD_1 and TD respectively. For each intermediate point triple,

[2]There can be multiple timing diagrams which have the same language.

Algorithm 2 A Disjunction Algorithm for RTD with One Waveform

1: one-waveform-disj($pre_{TD_0,(p,i)}$, $pre_{TD_1,(q,j)}$, mark, S)
//mark is the disjunction type of $pre_{TD_0,(p,i)}$ and $pre_{TD_1,(q,j)}$. S is a set: $((p,m),(q,n),(r,o)) \in S$ if (p,m) is a point in $pre_{TD_0,(p,i)}$, (q,n) is a point in $\in pre_{TD_1,(q,j)}$ and the value of (r,o) is disjunction of that of (p,m) and (q,n).
2: **if** $pre_{TD_0,(p,i)} \neq TD_0$ and $pre_{TD_1,(q,j)} \neq TD_1$ **then**
3: Let $bound_{(p,i),(p,i+1)} = [a,b]$ and $bound_{(q,j),(q,j+1)} = [a',b']$
4: L_0: Choose untried choice(s) from choice 1 and choice 2, if both are tried return 'NO DISJUNCTION'
choice 1 (this choice can only be tried when $b < b'$):
add a point (p,m) having the same value as (p,i) and sequential dependency: $(p,i) \overset{[a',b']}{\longrightarrow} (p,m)$
5: **if** (p,i) $(p,m);(q,j)$ $(q,j+1)$ isn't in one of 12 cases **then**
6: terminate this choice and return to L_0.
7: **else**
8: Let the disjunction type of (p,i) $(p,m);(q,j)$ $(q,j+1)$ be $mark'$
9: **if** ($mark$ contradicts $mark'$ AND NOT($mark$ = *Partial-Partial* AND $p(i) = 0$ *or* 1 AND $q(j) = 1$ *or* 0 AND $p(m) = q(j+1)$) **then**
10: terminate this choice and return to L_0.
11: **else**
12: $mark \leftarrow mark * mark'$
13: **if** $mark$ = *Partial-Partial* AND $p(i) = 0$ *or* 1 AND $q(j) = 1$ *or* 0 AND $p(m) = q(j+1)$ **then**
14: $mark \leftarrow$ *Partial-Partial*
15: call function *add-dependencies*($pre_{TD_0,(p,i)}, (p,i+1), pre_{TD_1,(q,j)}, (q,j+1), S, mark$) on page 58
16: choice 2: without adding point
17: **if** $[a,b] \cap [a',b'] \neq \phi$ **then**
18: **if** (p,i) $(p,i+1);(q,j)$ $(q,j+1)$ both with bound $[1,1]$ isn't in one of 12 cases **then**
19: terminate this choice and return to L_0.
20: **else**
21: get $mark'$ of (p,i) $(p,i+1);(q,j)$ $(q,j+1)$ both with bound $[1,1]$
22: **if** ($mark$ contradicts $mark'$ AND NOT($mark$ = *Partial-Partial* AND $p(i) = 0$ *or* 1 AND $q(j) = 1$ *or* 0 AND $p(m) = q(j+1)$) **then**
23: terminate this choice and return to L_0
24: **else**
25: mark $\leftarrow mark * mark'$
26: **if** $mark$ = *Partial-Partial* AND $p(i) = 0$ *or* 1 AND $q(j) = 1$ *or* 0 AND $p(m) = q(j+1)$ **then**
27: $mark \leftarrow$ *Partial-Partial*
28: call function *add-dependencies*($pre_{TD_0,(p,i)}, (p,m), pre_{TD_1,(q,j)}, (q,j+1), S, mark$) on page 58
29: **else**
30: terminate this choice and return to L_0.
31: **else**
32: **if** only one of $pre_{TD_0,(p,i)} = TD_0$ and $pre_{TD_1,(q,j)} = TD_1$ is true **then**
33: return 'NO DISJUNCTION'
34: **else**
35: return TD.

1: *add-dependencies*($pre_{TD_0,(p,i)}, (p, i+1), pre_{TD_1,(q,j)}, (q, j+1), S, mark$)
2: **for all** $((p, x), (q, y), (r, z)) \in S$ **do**
3: compute tight bound $bound_{(p,x),(p,i+1)} = [m, n]$ and $bound_{(q,y),(q,j+1)} = [m', n']$
4: **if** $n' < m$ or $n < m'$ **then**
5: terminate this choice and return to L_0.
6: **else**
7: add sequential dependency $(r, z) \overset{[m,n]\cup[m',n']}{\longrightarrow} (r, k+1)$, we get $pre_{TD_0,(p,i+1)}$ and $pre_{TD_1,(q,j+1)}$
//value of $(r, k+1)$ is disjunction of that of $(q, j+1)$ and $(p, i+1)$
8: **if** $[m, n]$ and $[m', n']$ don't contradict mark **then**
9: update mark according to $[m, n]$ and $[m', n']$.
10: **else**
11: mark ← Partial-Partial
12: **if** $(r, k+1)$ is not an event **then**
13: terminate this choice and return to L_0.
14: **else**
15: call function *equality*($pre_{TD_0,(p,i+1)}$, $pre_{TD_1,(q,j+1)}$, S, mark)
16: **if** the above function return false **then**
17: terminate this choice and return to L_0.
18: **else**
19: call function *one-waveform-disj*($pre_{TD_0,(p,i+1)}, pre_{TD_1,(q,j+1)}, mark, S \cup \{((p, i+1), (q, j+1), (r, k+1)\}$)

1: *equality*($pre_{TD_0,(p,i+1)}$, $pre_{TD_1,(q,j+1)}$, S, mark)
2: **for all** $((p, m), (q, n), (r, x)) \in S$,$(p, m'), (q, n'), (r, x')) \in S$ **do**
3: replace the original sequential dependency between (r, x) and $(r, k+1)$ to $tbound_{(p,m),(p,i+1)} - tbound_{(q,n),(q,i+1)}$.
4: replace the original sequential dependency between (r, x') and $(r, k+1)$ to $tbound_{(q,n'),(q,i+1)} - tbound_{(p,m'),(p,i+1)}$.
5: **if** there is no negative cycle in $G_{pre_{TD,(r,k+1)}}$ **then**
6: return false
7: **for all** $((p, m''), (q, n''), (r, x'')) \in S$ such that $((p, m''), (q, n''), (r, x'')) \neq ((p, i+1), (q, j+1), (r, k+1))$ **do**
8: replace the original sequential dependency between (r, x) and $(r, k+1)$ to $tbound_{(p,m),(p,i+1)} - tbound_{(q,n),(q,i+1)}$.
9: replace the original sequential dependency between (r, x') and (r, x'') to $tbound_{(q,n'),(q,n'')} - tbound_{(p,m'),(p,m'')}$.
10: **if** there is no negative cycle in $G_{pre_{TD,(r,k+1)}}$ **then**
11: return false
12: mark ← Partial-Partial
13: return true

points in it are supposed to be mapped to the same position inside each common word.

Theorem 5.1.1. *Given TD_0 and TD_1, if the disjunction algorithm returns TD, then $\mathcal{L}(TD_0) \cup \mathcal{L}(TD_1) \subseteq \mathcal{L}(TD)$.*

Proof. We only prove $\mathcal{L}(TD_0) \subseteq \mathcal{L}(TD)$. The proof that $\mathcal{L}(TD_1) \subseteq \mathcal{L}(TD)$ is similar.

Since the disjunction algorithm returns TD, there exists a sequence of point triples (p, q, r) as returned by the disjunction algorithm. We prove by induction on the sequence of point triples that for all word $w \models_{\pi_0} TD_0$, there exists an assignment π such that $w \models_{\pi} TD$ and for each point triple (p, q, r), $\pi_0(p) = \pi(r)$.

For the first point triple $((p, 0), (q, 0), (r, 0))$, by the disjunction algorithm, when the value of $(r, 0)$ is not X, it has the same value as $(p, 0)$. So $Value((r, 0)) \sqsubseteq w(\pi((r, 0)))$.

Suppose for the i^{th} point triple $((p, i), (q, i), (r, i))$, π satisfies point consistency, waveform consistency and dependency consistency for all points and dependencies before (r, i) (including (r, i)).

We then prove for the $(i+1)^{th}$ point triple $((p, i+1), (q, i+1), (r, i+1))$, π satisfies point consistency, waveform consistency and dependency consistency. There are two cases for this point triple: $(p, i+1)$ is a new added point or it is a point in the original TD_0.

1. When $(p, i+1)$ is a new added point, by the disjunction algorithm, if the value of $(r, i + 1)$ is not X, then $(r, i + 1)$ and $(p, i + 1)$ have the same value. So $Value((r, i + 1)) \sqsubseteq w(\pi((r, i + 1)))$ which means π satisfies point consistency with respect to $(r, i + 1)$ and w. For each digit $w(d)$ between where (r, i) and $(r, i + 1)$ are mapped in w, $Value((r, i + 1)) \sqsubseteq w(d)$ which means π satisfies

waveform consistency. By the disjunction algorithm, for each sequential dependency $(r, m) \xrightarrow{e} (r, i+1)$ between a previous point (r, m) and $(r, i+1)$, $tbound_{(p,m),(p,i+1)} \subseteq e$. By definition of tight bounds, π satisfies dependency consistency.

2. When $(p, i+1)$ is not a new added point, let $tbound((p, i), (p, i+2)) = [a, b]$ and $tbound((q, i), (q, i+1)) = [a', b']$. By the disjunction algorithm, $b' < b$ which means the new added point $(p, i+1)$ is always mapped between where (p, i) and $(p, i+2)$ are mapped, and has the same value as (p, i). Since if the value of $(r, i+1)$ is not X, it has the same value as $(p, i+1)$, π satisfies point consistency with respect to $(r, i+1)$ and w. For the similar reason as the first case, π satisfies waveform consistency and dependency consistency.

So we have proved this theorem. This proof also shows the disjunction algorithm correctly finds concrete points in TD_0 and TD_1. □

In order to prove TD is a disjunction of TD_0 and TD_1, we also need to prove TD is not satisfied by words that are in neither $\mathcal{L}(TD_0)$ nor $\mathcal{L}(TD_1)$.

Theorem 5.1.2. *Given TD_0 and TD_1, if the disjunction algorithm returns TD, then $\mathcal{L}(TD) \subseteq \mathcal{L}(TD_0) \cup \mathcal{L}(TD_1)$.*

Proof. We prove that for every word $w \models_\pi TD$, there exists an assignment π' such that $w \models_{\pi'} TD_0$ or $w \models_{\pi'} TD_1$.

Notice that when the disjunction algorithm returns TD, it also returns the disjunction type of TD_0 and TD_1. Their disjunction type can be Full-Full, Full-Partial, Partial-Full, or Partial-Partial.

When the disjunction algorithm returns Full-Full, by the disjunction algorithm, for each point triple (p, q, r), the value of p, q and r are the same. For every

two point triples $((p,i),(q,j),(r,k))$ and $((p,i'),(q,j'),(r,k'))$, $tbound_{(p,i),(p,i')} = tbound_{(q,j),(q,j')} = e$ in which e is the bound of sequential dependency $(r,k) \xrightarrow{e} (r,k')$. We prove $w \models_{\pi'} TD_0$ under the assignment π' defined as: for each point triple (p,q,r), $\pi(r) = \pi'(p)$. Since in each point triple (p,q,r), the value of p and the value of r are the same, π' satisfies point consistency and waveform consistency. For every sequential dependency $p \xrightarrow{d} p'$ in TD_0, by definition of tight bound, $tbound_{p,p'} \subseteq d$. Let the point triples where r and r' in TD be (p,q,r) and (p',q',r') and let the sequential dependency between r and r' in TD be $r \xrightarrow{e} r'$. By the disjunction algorithm $e = tbound_{p,p'} \subseteq d$. Since π satisfies $r \xrightarrow{e} r'$ and $\pi(r) = \pi'(p)$, $\pi(r') = \pi'(p')$, π' satisfies $p \xrightarrow{d} p'$. So π' satisfies dependency consistency.

When the disjunction type is Full-Partial, by the disjunction algorithm, for every point triple (p,q,r), either p has value X or p and q have the same value. When p has value X, r has value X; when p and q have the same value, r has the same value as them. For every point triples (p,q,r) and (p',q',r'), let the sequential dependency between r and r' in TD be $r \xrightarrow{e} r'$, $tbound_{q,q'} \subseteq tbound_{p,p'} = e$. We prove there exists an assignment π' such that $w \models_{\pi'} TD_0$ and for every point triple (p,q,r), $\pi(r) = \pi'(p)$. By the disjunction algorithm, for every point triple (p,q,r), p and r always have the same value. So we can prove similarly as we did in the first case that π' satisfies point consistency and waveform consistency. For every two point triples (p,q,r) and (p',q',r'), $tbound_{p,p'} = e$ in which e is the time bound of sequential dependency between r and r', so similarly we can prove that π' satisfies dependency consistency.

When the disjunction type is Partial-Partial, then there can be several cases:

1. There exists a point triple (p,q,r) in which p has value 0 and q has value 1. For each of the point triples other than this one, the three points in it have the same value which is guaranteed by line 9 and line 22. For each two point

triples (p', q', r') and (p'', q'', r''), $tbound_{p',p''} = tbound_{q',q''} = e$ in which e is the bound of sequential dependency $r' \xrightarrow{e} r''$.

2. For each point triple, the three points in it have the same value. There are at least two point triples (p, q, r) and (p', q', r') such that $tbound_{p,p'} - tbound_{q,q'} \neq \phi$ and $tbound_{q,q'} - tbound_{p,p'} \neq \phi$

We prove that there exists an assignment π' such that $w \models_{\pi'} TD_0$ or $w \models_{\pi'} TD_1$ and for every point triple (x, y, z), $\pi(z) = \pi'(x)$.

For the first case, without loss of generality, let $w(\pi(r)) = 0$ and $Value(p) = 0$. By definition of π', $\pi'(p) = \pi(r)$. So π' satisfies point consistency. We can prove π' satisfies waveform consistency and dependency consistency in the similar way as we did in Full-Full case.

For the second case, since in each point triple, the three points have the same value. We can prove π' satisfies point consistency and waveform consistency in similar way as we did in Full-Full case. We then prove π' satisfies dependency consistency. If for every two point triples (p, q, r) and (p', q', r'), $\pi(r) - \pi(r') \in tbound_{p,p'} \cap tbound_{q,q'}$, we can prove in the similar way as we did in the Full-Full case that $w \models_{\pi'} TD_0$ and $w \models_{\pi'} TD_1$. If there exists two point triples (p, q, r) and (p', q', r') such that $\pi(r) - \pi(r') \in tbound_{p,p'} - tbound_{q,q'}$ or $\pi(r) - \pi(r') \in tbound_{q,q'} - tbound_{p,p'}$, without loss of generality, $\pi(r) - \pi(r') \in tbound_{p,p'} - tbound_{q,q'}$. By function 'equality' called by the disjunction algorithm, there are no point triples (p'', q'', r'') and (p''', q''', r''') such that $\pi(r) - \pi(r') \in tbound_{q'',q'''} - tbound_{p'',p'''}$. Then we can prove $w \models_{\pi'} TD_0$.

So we have proved the theorem. □

By Theorem 5.1.1 and Theorem 5.1.2, given TD_0 and TD_1, if the disjunction algorithm returns TD, then TD is a disjunction of TD_0 and TD_1. We then prove

that if the disjunction algorithm returns 'NO DISJUNCTION', then their disjunction does not exists.

Notice that in the disjunction algorithm there are only two parts where it returns 'NO DISJUNCTION'. One is in line 4 and the other is in line 33. We first prove that given TD_0 and TD_1, if the disjunction algorithm returns 'NO DISJUNCTION' in line 4, then there is no disjunction of TD_0 and TD_1.

When we call function *one-waveform-disj* on input $pre_{TD_0,(p,i)}$, $pre_{TD_1,(q,j)}$, mark and S, there exists a disjunction of $pre_{TD_0,(p,i)}$ and $pre_{TD_1,(q,j)}$ and let it be $pre_{TD,(r,k)}$. There can be multiple disjunctions of $pre_{TD_0,(p,i)}$ and $pre_{TD_1,(q,j)}$. But there exists some relationship between all those disjunctions and $pre_{TD,(r,k)}$. Let one of those disjunction be $pre_{TD,(r,x)}$. By Theorem 4.3.1, we can remove all non-transition points in $pre_{TD,(r,x)}$. Let the result after we remove all non-transition points be $pre'_{TD,(r,x)}$. Then all points in $pre'_{TD,(r,x)}$ are all the transition points in $pre_{TD,(r,k)}$, because transition points describe the shape of a RTD.

In function *one-waveform-disj*, there are two choices. In each choice, the function can terminate its current execution and return to the other choice if that thoice has never been tried. Each choice tries to add a point triple $((p, i+1), (q, j+1), (r, k+1))$ to S. We will prove for each of the two choices, if the function terminates its current choice, then there is no disjunction of $pre_{TD_0,(p,i+1)}$ and $pre_{TD_1,(q,j+1)}$.

Theorem 5.1.3. *Given input $pre_{TD_0,(p,i)}$, $pre_{TD_1,(q,j)}$, mark, and S, if the disjunction algorithm terminates its current choice in choice 1, then there is no disjunction of $pre_{TD_0,(p,m)}$ and $pre_{TD_1,(q,j+1)}$.*

Proof. In choice 1, the algorithm can terminate its current choice at line 6 and line 10 in function *one-waveform-disj* and at line 5, line 12, and line 17 in function *add-dependencies*.

- If the algorithm terminates its current choice at line 6 in function *one-waveform-disj*, let the timing diagram whose points are (p, i) and (p, m) be $post_{TD_0,(p,i)}$, and let the timing diagram whose points are (q, j) and $(q, j+1)$ be $post_{TD_1,(q,j)}$. Since $post_{TD_0,(p,i)}$ and $post_{TD_1,(q,j)}$ does not satisfy any of the 12 cases in the table on page 55, there is no disjunction of $post_{TD_0,(p,i)}$ and $post_{TD_1,(q,j)}$. Suppose there exists a disjunction of $pre_{TD_0,(p,m)}$ and $pre_{TD_1,(q,j+1)}$, let it be $pre_{TD,(r,k+1)}$. Either there exists a concrete point (r, k) which corresponds to (p, i) and (q, j), or we can add such point without changing its language. The reason is all the transition points in $pre_{TD,(r,k)}$ which is constructed by the disjunction algorithm in the last step are in $pre_{TD,(r,k+1)}$. If $(r, k + 1)$ is a transition point, it is in $pre_{TD,(r,k+1)}$. If it is a non-transition point, there should be a sequential dependency $(r, x) \xrightarrow{[k,k]} (r, k + 1)$ in which (r, x) is a rise/fall because $pre_{TD,(r,k)}$ is well-formed. This implies that we can add (r, k) to $pre_{TD,(r,k+1)}$ without changeing its language. So the prefix of $pre_{TD,(r,k+1)}$ whose last point is (r, k) is the disjunction of $pre_{TD_0,(p,m)}$ and $pre_{TD_1,(q,j)}$. This means the timing diagram which has two points (r, k) and $(r, k + 1)$ and a sequential dependency $(r, k) \xrightarrow{tbound_{(r,k),(r,k+1)}} (r, k + 1)$ is the disjunction of $post_{TD_0,(p,i)}$ and $post_{TD_1,(q,j)}$. This contradicts the hypothesis that $post_{TD_0,(p,i)}$ and $post_{TD_1,(q,j)}$ has no disjunction. So there is no disjunction of $pre_{TD_0,(p,m)}$ and $pre_{TD_1,(q,j+1)}$.

- If the algorithm terminates its current choice at line 10 in function *one-waveform-disj*, $mark * mark'$ equals UNDEFINED in the table on Page 53. We prove by contradiction that there is no disjunction of $pre_{TD_0,(p,m)}$ and $pre_{TD_1,(q,j+1)}$. Let one of their disjunctions be $pre_{TD,(r,k+1)}$. Without loss of generality, let $mark = Full\text{-}Partial$ and $mark' = Partial\text{-}Partial$. As we have proved above in the first case, either there exists a concrete point (r, k)

which corresponds to (p, i) and (q, j), or we can add such point without changing the language of $pre_{TD,(r,k+1)}$. Let the prefix of $pre_{TD,(r,k+1)}$ whose last point is (r, k) be $pre_{TD,(r,k)}$, it is the disjunction of $pre_{TD_0,(p,i)}$ and $pre_{TD_1,(q,j)}$. Let the timing diagram which has two point (r, k) and $(r, k+1)$ and a sequential dependency $(r, k) \overset{tbound_{(r,k),(r,k+1)}}{\longrightarrow} (r, k+1)$ be $post_{TD,(r,k)}$, it is the disjunction of $post_{TD_0,(p,i)}$ and $post_{TD_1,(q,j)}$. Since $mark = Full\text{-}Partial$, there exists a word w_1 which satisfies $pre_{TD_0,(p,i)}$ but not $pre_{TD_1,(q,j)}$. Since $mark' = Partial\text{-}Partial$ and $tbound_{(p,i),(p,m)} = tbound_{(q,j),(q,j+1)}$[3], (p, m) and $(q, j+1)$ have value 0 or 1 but don't have the same value. Without loss of generality, let (p, m) has value 0 and $(q, j+1)$ has value 1. The word $w1$ should satisfies $pre_{TD,(r,k+1)}$, but it satisfies neither $pre_{TD_0,(p,m)}$ nor $pre_{TD_1,(q,j+1)}$ which contradicts the hypothesis that $pre_{TD,(r,k+1)}$ is a disjunction of $pre_{TD_0,(p,m)}$ and $pre_{TD_1,(q,j+1)}$. So there is no disjunction of $pre_{TD_0,(p,m)}$ and $pre_{TD_1,(q,j+1)}$.

- If the algorithm terminates its current choice at line 5 in function *add-dependencies*, there will be no bound of the sequential dependency between (r, k) and $(r, k+1)$ in $pre_{TD,(r,k+1)}$ in the form of $[a, b]$.

- If the algorithm terminates its current choice at line 12 in function *add-dependencies*, $pre_{TD,(r,k+1)}$ will not be well-formed.

- If the algorithm terminates its current choice at line 17 in function *add-dependencies*, the function *equality* returns false on input $pre_{TD_0,(p,i+1)}$, $pre_{TD_1,(q,j+1)}$, mark and S. By Theorem 4.1.1, there exists a word $w \models pre_{TD,(r,k+1)}$ and w satisfies $tbound_{(p,x'),(p,x'')} - tbound_{(q,y),(q,y'')}$ and $tbound_{(q,y'),(q,y'')} - tbound_{(p,x'),(p,x'')}$ for point triples $((p, x), (q, y), (r, z))$, $((p, x'), (q, y'), (r, z'))$ and $((p, x''), (q, y''), (r, z''))$. But w does not satisfies any of $pre_{TD_0,(p,m)}$ or $pre_{TD_1,(q,j+1)}$ which contra-

[3]This is because (p, m) is the new added point.

dicts the hypothesis that $pre_{TD,(r,k+1)}$ is a disjunction of $pre_{TD_0,(p,m)}$ and $pre_{TD_1,(q,j+1)}$.

So we have proved this theorem. □

Similarly we can prove the following theorem.

Theorem 5.1.4. *Given input $pre_{TD_0,(p,i)}$, $pre_{TD_1,(q,j)}$, mark and S, if the disjunction algorithm terminates its current choice in choice 2, then there is no disjunction of $pre_{TD_0,(p,i+1)}$ and $pre_{TD_1,(q,j+1)}$.*

We then prove that if both choices are unsuccessful, then there is no disjunction of given RTDs.

Theorem 5.1.5. *If the disjunction algorithm returns 'NO DISJUNCTION' in line 4, there is no disjunction of TD_0 and TD_1.*

Proof. We prove this theorem by contradiction. Suppose there exists a disjunction of TD_0 and TD_1, and let it be TD. Since two choices have been tried, by Theorem 5.1.3 and Theorem 5.1.4, $pre_{TD_0,(p,m)}$ and $pre_{TD_1,(q,j+1)}$[4] has no disjunction and $pre_{TD_0,(p,i+1)}$ and $pre_{TD_1,(q,j+1)}$ has no disjunction. Since $pre_{TD_0,(p,i)}$ and $pre_{TD_1,(q,j)}$ has a disjunction, we have proved above that there should be a concrete point (r,k) in TD which corresponds to (p,i) and (q,j). Since $(q,j+1)$ is a transition point, TD_2 change its shape at $(q,j+1)$[5]. There should be a point $(r,k+1)$ which captures this transition, because this transition may also occur in TD. $(r,k+1)$ can only correspond to either (p,m) or $(p,i+1)$ in TD_1. So $(r,k+1)$ is a concrete point. Since TD is a disjunction of TD_0 and TD_1, $pre_{TD,(r,k+1)}$ is a disjunction of either $pre_{TD_0,(p,m)}$ and $pre_{TD_1,(q,j+1)}$ or $pre_{TD_0,(p,i+1)}$ and $pre_{TD_1,(q,j+1)}$. This contradicts

[4] (p,m) is an added point between (p,i) and $(p,i+1)$ and it has the same value as (p,i).
[5] This is why we first remove all non-transition points.

$pre_{TD_0,(p,m)}$ and $pre_{TD_1,(q,j+1)}$ has no disjunction and $pre_{TD_0,(p,i+1)}$ and $pre_{TD_1,(q,j+1)}$ has no disjunction. So there should be no disjunction of TD_0 and TD_1. □

We then prove that if the algorithm returns 'NO DISJUNCTION' at line 33, then there is no disjunction of TD_0 and TD_1.

Theorem 5.1.6. *Given timing diagrams TD_0 and TD_1, if there exists a point (p, i) in TD_0 such that $pre_{TD_0,(p,i)}$ and TD_1 has a disjunction and (p, i) is not the last point in TD_0, then there exists no disjunction of TD_0 and TD_1.*

Proof. We prove this theorem by contradiction. Suppose there exists a disjunction of TD_0 and TD_1, and let it be TD'. Since there exists a disjunction of $pre_{TD_0,(p,i)}$ and TD_1 let it be TD. We can add a point (r, k) in TD' such that $pre_{TD',(r,k)}$ and TD have the same language. This means all words w that satisfy TD_1 satisfy both $pre_{TD',(r,k)}$ and TD'. This contradicts Theorem 2.2.1. □

By Theorem 5.1.3, 5.1.4, 5.1.5 and 5.1.6, we can prove that if the disjunction algorithm returns 'NO DISJUNCTION', then there exists no disjunction of given RTDs.

5.2 Disjunction of Multi-Waveform RTD

A multi-waveform RTD is a RTD which has more than one waveforms. Given two multi-waveform RTDs TD_0 and TD_1, if their exists a disjunction RTD of them, TD_0 and TD_1 must have the same number of waveforms. After we have computed tight bound between each pair of events, for any waveform p in TD_0, there exists a waveform p' in TD_1 such that there exists a disjunction RTD for p and p'. The contrapositive of this sentence is much more useful: after we have computed tight bound between each pair of events, if there exists a waveform p in TD_0 such that

there exists no disjucntion RTD of p and any waveform in TD_1, then there is no disjunction RTD of TD_0 and TD_1.

If for each waveform p in TD_0 there exists a waveform p' in TD_1 and a RTD TD which captures disjunction of p and p', we need to check whether there exist contradictions among disjunction types of waveforms from TD_0 and TD_1. For each pair of waveforms p_0 and p'_0 there exists a disjunction type t_0. So for TD_0 and TD_1, there exists a set of disjunction types $\{t_0, t_1, t_2, ..., t_n\}$. By applying the operator $*$ defined in Definition 5.1.3 on each pair of disjunction types t_i and t_j, if $t_i * t_j$ =UNDEFINED, then there exists no disjunction RTD of TD_0 and TD_1. If for any t_i and t_j, $t_i * t_j \neq$UNDEFINED, then we check whether there exsits any contradiction between tight bounds. If contradiction exists, then there is no disjunction RTD of TD_0 and TD_1; if not, there exists disjunction RTD.

Chapter 6

Conclusions

Timing diagrams are used in industrial practice as a specification language of circuit components. But the temporal logic based specification methods in model checking are not primarily used in practical design. So practical designers may want to know whether there exists a temporal logic formula expressing the same property as the timing diagram they draw. This thesis tries to explore relationship between linear temporal logic (LTL) and regular timing diagrams. It focuses on disjunction of regular timing diagrams. In order to solve the disjunction problem, we first find out all implicit timing relations between events. The tight bound algorithm in Chapter 4 computes exact time separations between each pair of events; we have presented a proof that the algorithm is correct. Then we focus on disjunction of one-waveform regular timing diagrams. The Algorithm 2 in section 5.1 decides the disjunction problem for one-waveform regular timing diagrams. In each iteration of the algorithm, we either add a concrete point or use the next point as a concrete point, and we check if there exists contradiction among timing constraints or values on waveforms. We used the one-waveform RTDs disjunction algorithm to decide disjunction of multi-waveform RTDs.

As we were exploring disjunctions of RTDs, we built an Alloy specification for one-waveform RTDs. We also checked a theorem proved by us in Alloy under a bound on the size of RTDs. We had hoped to use the Alloy specification to verify that algorithm under a useful bound on the size of the timing diagrams. But even for the smallest RTDs (RTDs with two points), verification of the disjunction algorithm requires far more memory and CPU time than our machine's. It would be useful to have some formal tool that can quickly identify counterexamples to our theorems and algorithms.

This thesis talks about disjunction of RTDs. For negation, RTD is not closed under negation. An example can be a RTD with three points each of which is either a rise or a fall. To further explore relationships between RTD and temporal logic formulas, we need to know an algorithm (if it exists) which decides existance of negation of RTD. Notice that the semantics used in this paper is different from either iterative semantics or invariant semantics. It would be useful to extend our result about disjunction to that under iterative semantics or invariant semantics.

Appendix A

An Alloy Specification for RTD

```
open util/sequniv
open util/ternary
open util/relation
open util/integer
open util/ternary
open util/ordering[Point]

one sig ValueZero extends NonXValue{}
one sig ValueOne extends NonXValue{}
one sig ValueX extends Value{}
sig NonXValue extends Value {}
sig Value{}
fact {NonXValue=ValueZero+ValueOne}
fact {Value=NonXValue+ValueX}

sig Point {hasvalue: one Value, index: one Int}

sig SimpleTimingDiagram {haspoint:set Point, hassd:set SequentialDependency}{
#haspoint>1}

fact TDPointIndex {all td:SimpleTimingDiagram | all p, q:td.haspoint |
(min[td.haspoint].index=0) and
(lt[p, q]=>(p.index<q.index)) and
(max[td.haspoint].index=#td.haspoint-1)
}

pred TDPointNext[td:SimpleTimingDiagram, p:Point, q:Point] {
p in td.haspoint and q in td.haspoint and q.index=p.index+1
```

```
}


pred PointInTDIsRiseFall [td:SimpleTimingDiagram, p:Point] {some q:Point |
(p.index>0) and //p is not the first point of td
(p.hasvalue != ValueX) and //value of p is not X
(q in td.haspoint) and //q is a point of td
(TDPointNext[td, q, p]) and //q is previous point of p
(p.hasvalue !=q.hasvalue)
//value of p is not the same as value of its previous point
and
(q.hasvalue != ValueX) //value of previous point of p is not X
}

sig SequentialDependency {SDis: Point -> Point -> Int -> Int}

fun FromPointOfSD [sd:SequentialDependency]:set Point
{(((sd.SDis).Int).Int).Point}

fun ToPointOfSD [sd:SequentialDependency]:set Point
{Point.(((sd.SDis).Int).Int)}

fact PointOfSDInTD {all td:SimpleTimingDiagram | all sd:td.hassd |
(FromPointOfSD[sd] in td.haspoint) and (ToPointOfSD[sd] in td.haspoint)
// For every SD, its first point and second point are both in the timing
//diagram where this SD is.
}
fact SDSecondPointLater {
all td:SimpleTimingDiagram | all sd:td.hassd | all p, q:Point |
(p->q in ((sd.SDis).Int).Int)=>TDPointNext[td, p, q]
//the second point should occurs later than the first point,
//and Point->Point in sd should be irreflexive
}

fact SDHigherNoLessThanLower {all sd:SequentialDependency | all i, j:Int |
(i->j in (Point.(Point.(sd.SDis)))) =>
(i>=0) and (i<=j) and ((i=0)=>(i!=j))
//lower bound of any SD of any timing diagram is no less than
//its higher bound and higher bound cannot be 0 when lower bound is 0
}

fact SDTimeBoundUnique {all sd:SequentialDependency | all p, q:Point |
let LowerBoundOfSDWhenFirstTwoArepq=(q.(p.(sd.SDis))).Int |
let HigherBoundOfSDWhenFirstTwoArepq=Int.(q.(p.(sd.SDis))) |
( p->q in ((sd.SDis).Int).Int)=>
```

```
( #LowerBoundOfSDWhenFirstTwoArepq=1 and #HigherBoundOfSDWhenFirstTwoArepq=1 )
}

fact TDWellFormed {
all td:SimpleTimingDiagram | all q:Point| some sd:td.hassd | some p:Point |
let PrevOfqInsd=(((sd.SDis).Int).Int).q |
let LowerBoundOfsdp2q=(q.(p.(sd.SDis))).Int |
let HigherBoundOfsdp2q=Int.(q.(p.(sd.SDis))) |
(
   (q in td.haspoint) //point q is not the first point in one waveform TD
   and //point q has value X or point q is not transition
   not (PointInTDIsRiseFall[td, q]) and
   (q.index>0)
)
=>
(
   p in td.haspoint and
   q in ToPointOfSD[sd] and p in PrevOfqInsd and
   (
      (p.index=0) //point p is the first point
      or //if p is not the first point
      PointInTDIsRiseFall[td, p] //point p is rise/fall
   )
   and //lower bound and higher bound are the same
   (
      LowerBoundOfsdp2q=HigherBoundOfsdp2q and #LowerBoundOfsdp2q=1
      and #HigherBoundOfsdp2q=1
   )
)
}

sig Word{position: seq NonXValue}{#position>1}
//the length of every word shoule be no less than 2

sig Pi {positionis: SimpleTimingDiagram->Word->one Point2Position}
//all pi has only one Point2Position, but can have more than one
//SimpleTimingDiagram

sig Point2Position {is: seq Int}

pred AllPointMappedInsidePi [td:SimpleTimingDiagram, pi:Pi, word:Word] {
(td->word in ((pi.positionis).Point2Position))=>
(#((word.(td.(pi.positionis))).is)=#td.haspoint)
}
```

```
pred Point2PositionNoDuplicate [td:SimpleTimingDiagram, pi:Pi, word:Word] {
!((word.(td.(pi.positionis))).is).hasDups
}

fun PositionPointMappedByPi[pi:Pi, td:SimpleTimingDiagram, p:Point,
word:Word]:Int {((word.(td.(pi.positionis))).is)[p.index]}
//we have to be sure p is in td

pred PointMappedToPositivePosition[td:SimpleTimingDiagram, pi:Pi, word:Word]
{
all i:Int | (i in ((word.(td.(pi.positionis))).is).elems)=> (i>=0)
//Every point is mapped to positive position
}

pred PointMappedInsideWord [td:SimpleTimingDiagram, pi:Pi, word:Word]
{all p:td.haspoint | all i:Int |
(i in pi.PositionPointMappedByPi[td, p, word])=>(i < #word.position)
//any point in TD mapped inside any word
}

pred FirstPointFirstPosition [td:SimpleTimingDiagram, pi:Pi, word:Word] {
((word.(td.(pi.positionis))).is)[0]=0
//first point of any timing diagram is mapped to the first position
//of any word that matches this timing diagram
}

pred LastPointLastPosition [td:SimpleTimingDiagram, pi:Pi, word:Word] {
(pi.PositionPointMappedByPi[td, max[td.haspoint], word])=#word.position-1
//last point of any timing diagram is mapped to the last position of
//any word that matches this timing diagram
}

pred MappingFunctionIncreasing [td:SimpleTimingDiagram, pi:Pi, word:Word]
{all p, q:td.haspoint |
(TDPointNext[td, p, q])=>
(pi.PositionPointMappedByPi[td, p, word]<pi.PositionPointMappedByPi[td, q, word])
//mapping function increase
}

pred PointValueMatch [td:SimpleTimingDiagram, pi:Pi, word:Word]
{all p:td.haspoint |
(p.hasvalue!=ValueX) =>
   (p.hasvalue=word.position[pi.PositionPointMappedByPi[td, p, word]])
//For every point, its value should be the same as the value on
//position where it's mapped
```

```
}

pred WaveformConsistencySatisfaction[td:SimpleTimingDiagram, pi:Pi, word:Word]
{all p, q:td.haspoint | all j:Int |
(
   TDPointNext[td, p, q] and (j<pi.PositionPointMappedByPi[td, q, word]) and
   (j>=pi.PositionPointMappedByPi[td, p, word]) =>
      ((p.hasvalue != ValueX)=>word.position[j]=p.hasvalue)
)
//any digit in word shoule have the same value as its nearest previous digit
//which is mapped by some point.
}

pred SequentialDependencySatisfaction [td:SimpleTimingDiagram, pi:Pi, word:Word]
{all sd:td.hassd | all p, q:td.haspoint | all i, j:Int |
let LowerBoundOfSDp2q=(q.(p.(sd.SDis))).Int |
let HigherBoundOfSDp2qLowerIsi=(i.(q.(p.(sd.SDis)))) |
( (p->q in ((sd.SDis).Int).Int) and (i in LowerBoundOfSDp2q) and
  (j in HigherBoundOfSDp2qLowerIsi) ) =>
( sub[pi.PositionPointMappedByPi[td, q, word],
      pi.PositionPointMappedByPi[td, p, word]]<=j
and
sub[pi.PositionPointMappedByPi[td, q, word],
    pi.PositionPointMappedByPi[td, p, word]]>=i )
//for every SD, the difference of positions where its first point and second
//point are mapped should be no less than its lower bound and no greater than
//its upper bound.
}

pred WordMatchTimingDiagramUnderPi [td:SimpleTimingDiagram, pi:Pi, word:Word] {
AllPointMappedInsidePi[td, pi, word] and
Point2PositionNoDuplicate[td, pi, word] and
SequentialDependencySatisfaction [td, pi, word] and
WaveformConsistencySatisfaction [td, pi, word] and
PointValueMatch [td, pi, word] and
MappingFunctionIncreasing [td, pi, word] and
LastPointLastPosition[td, pi, word] and
FirstPointFirstPosition [td, pi, word] and
PointMappedInsideWord [td, pi, word] and
PointMappedToPositivePosition [td, pi, word]
}

run WordMatchTimingDiagramUnderPi for 6 but 1 SimpleTimingDiagram, 1 Pi, 1 Word
```

Bibliography

[1] Nina Amla. Efficient model checking for timing diagrams. *PhD dissertation*, May 2001.

[2] Nina Amla, Kedar S. Namjoshi, and E.Allen Emerson. Efficient decompositional model checking for regular timing diagrams. *Conference on Formal Methods in Computer Aided Design (FMCAD)*, November 2000.

[3] Alloy Analyzer. `http://alloy.mit.edu/alloy4/`.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithm, Second Edition.* The MIT Press, 2001.

[5] Kathi Fisler. Timing diagrams: Formalization and algorithmic verification. *Journal of Logic, Language, and Information*, 8(3), July 1999.

[6] Kathi Fisler and Hana Chockler. Temporal modalities for concisely capturing timing diagrams. *CHARME*, 2005.

[7] Pierre Girodias, Eduard Cerny, and William J. Older. Solving linear, min and max constraint systems using clp based on relational interval arithmetic. *Theoretical Computer Science*, Volume 173, Issue 1:253–281, Feburary 1997.

[8] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[9] Kenneth L. McMillan and David L. Dill. Algorithms for interface timing verification. *Proc. IEEE Internat. Conf. on Computer Design, ICCD'92*, pages 48–51, 1992.