MODULAR SYNTHESIS AND VERIFICATION OF TIMED CIRCUITS USING AUTOMATIC ABSTRACTION

by

Hao Zheng

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Electrical and Computer Engineering

The University of Utah

May 2001

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Hao Zheng

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Chris J. Myers
	Reid R. Harrison
	Christian Schlegel
	Erik Brunvand
	Ganesh Gopalakrishnan

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Hao Zheng in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Chris J. Myers
Chair, Supervisory Committee

Approved for the Major Department

V. John Mathews
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

In order to increase performance, circuit designers are beginning to use more aggressive timed circuit designs instead of traditional synchronous static logic designs. Recent design examples have shown that significant performance gains are achieved when these aggressive circuit styles are used. Correct operation of these aggressive circuit styles is critically dependent on timing, and in industry they are typically designed by hand. To synthesize and verify timed circuits, the reachable state space of the circuit under the timing constraints needs to be explored. However, complete state space exploration is an exponential problem. State space explosion limits timed circuit designs to small sizes.

This dissertation presents a new automatic abstraction approach which enables modular design of large scale timed circuits. It attacks the state space explosion problem by avoiding the generation of a flat state space for the design. Instead, it partitions a design into blocks with manageable sizes, and performs synthesis and verification process on each of them. The results for the blocks are integrated as the solution to the whole design. This dissertation presents a series of theorems that supports modular synthesis and verification. The concepts of safe abstraction and transformations are also described. This dissertation presents techniques to partition a design and safe net reductions to simplify the complexity when designing each block. Results show that design processes using this method are orders of magnitude more efficient in design time and memory usage.

To my family.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the support and help from many other people. First, I own my deepest gratitude to my advisor, Chris Myers, for introducing me into the field of asynchronous circuit design and automation. He has been very patient, especially during my early period of time at the University of Utah when I had to learn a lot of things from ground. His guidance and encouragement helped me go through the most difficult time of my research when I felt lost and restless. I also would like to thank Genesh Gopalakrishnan and Erik Brunvand for their technical suggestions and comments to this dissertation and for serving in my supervisory committee. My thanks also extend to Christian Shelegle and Reid Harrison for serving in my supervisory committee.

This dissertation has also benefited in a great deal from the seminar given by Tomohiro Yoneda in the last summer. He introduced me to the timed trace theory which has formed the theoretical groundwork for this dissertation since then. The discussions with him throughout in the past year helped me clarify many ideas and concepts. His comments also helped me discover and fix several technical bugs occurred in this dissertation. I would like to thank my fellow student, Eric Mercer, for his contributions to my research work. I like to thank Ken Stevens at the Intel for donating the fast workstations, one of which was used by me, so that I could perform my research more smoothly and efficiently. I would also like to thank my officemates: Eric Perskin, Kip Killpack, and Scott Little for their so that I could focus on my research.

The love, advice, encouragement, and support from my family was essential in completing this dissertation. During my PhD study, my parents made me believe that obtaining the PhD is not only their dream for me, but also one of the most important and prestigious milestone a person could reach. My family has always been there for me, to encourage me and give me invaluable advice when things did not go well, to celebrate when I made progress. Even my niece and nephew have given me lots of fun and joy when I played with them. I would like specially to thank my sister and her family for the various and numerous support, both mentally and financially.

This research is supported by several grants: NSF CAREER award MIP-9625014, SRC contract 97-DJ-487 and 99-TJ-694, and a grant from the Intel Corporation.

CHAPTER 1

INTRODUCTION

Current VLSI circuits are getting faster and more complex. In order to continue to produce circuits of increasing speed, designers are moving away from pure static synchronous designs to more aggressive *timed circuit* design styles. Generally, timed circuits are a class of circuits that are optimized using explicit, bounded timing information throughout the design process. Using timed circuits enables designs to achieve high performance and low power consumption. One example is the Intel RAPPID instruction length decoder for a Pentium II instruction set [71]. The RAPPID design is an asynchronous implementation. It runs 3 times faster while dissipating only half the power of the synchronous implementation on the same process. The performance gain is derived from a highly timed asynchronous design. The second example is the *self-resetting* and *delayed-reset domino* circuits widely used in a gigahertz research microprocessor (guTS) at IBM [41]. The guTS microprocessor is the first microprocessor that runs over 1 Gigahertz on a $0.25\mu m$ CMOS process available in 1997. The performance gain is derived from a highly timed synchronous implementation. There are many timing assumptions made in both examples, and the correct operation of the examples are heavily dependent upon whether the timing constraints are satisfied. Therefore, extensive timing analysis and verification is necessary during the design process. Unfortunately, these new circuit styles cannot be efficiently and accurately synthesized, analyzed, and verified using traditional static timing analysis methods. This lack of efficient analysis tools is one of the reasons for the lack of mainstream acceptance of these design styles.

Most synthesis and verification methods require complete state space exploration which is an exponential problem. One common problem often encountered in state space exploration is the *state explosion* problem, which limits the size and complexity of timed circuit designs. There exist many methods to deal with state explosion. In this dissertation, a divide-and-conquer approach is proposed. This approach partitions a design into blocks, each of which has a constrained interface. Each block is designed

individually, and the integration of the results of all blocks gives the solution for the whole design. During design of each block, abstraction is applied to remove the irrelevant information to reduce the complexity of designing each block. In this way, a design with a large exponential state space is converted to a set of designs with small exponential state space. This approach not only substantially reduces the computational cost of synthesis and verification, but also solves large and complex design problems that cannot be handled before.

The first section of this chapter gives an overview of timed circuit design methodologies and the design flow of `ATACS`, our timed circuit design tool. The second section gives an overview of the previous work on specification and verification of timed circuits, and the methods to deal with the state explosion problem inherent in the synthesis and verification of large designs. The last two sections give the contributions and outline of this dissertation.

1.1 Design Flow for Timed Circuits

Designing a timed circuit involves the steps of specification, compilation, analysis, synthesis, and verification. Figure 1.1 shows the design flow for our timed circuit design tool `ATACS`. In `ATACS`, the design of timed circuits begins with a specification of circuit behavior in a hardware description language including VHDL, *timed handshaking expansions* (THSE) [63, 86], *asynchronous finite state machines* (AFSM) [42, 32, 84], and *signal transition graphs* (STG) [28]. These specification methods are able to describe sequencing, concurrency, and choice. Moreover, they support bounded timing information which is used to optimize the circuit implementations during various design stages. The timing parameters can come from the simulation of similar designs; or designers can make any timing assumptions that they think reasonable for the circuits. The timing parameters can be either bounded or unbounded.

The step of compilation translates the specification of a timed circuit to a *timed event/level structure* (TEL) [10]. A TEL structure is a new data structure for describing both event and level causality and timing behavior. The compilation step includes steps to decompose a specification into basic events. These events are then translated to simple TEL structures which are composed together in sequence, in parallel, and in conflict as dictated by the structure in the specification.

The timing analysis algorithms such as the one shown in [65, 9, 10, 55] are applied

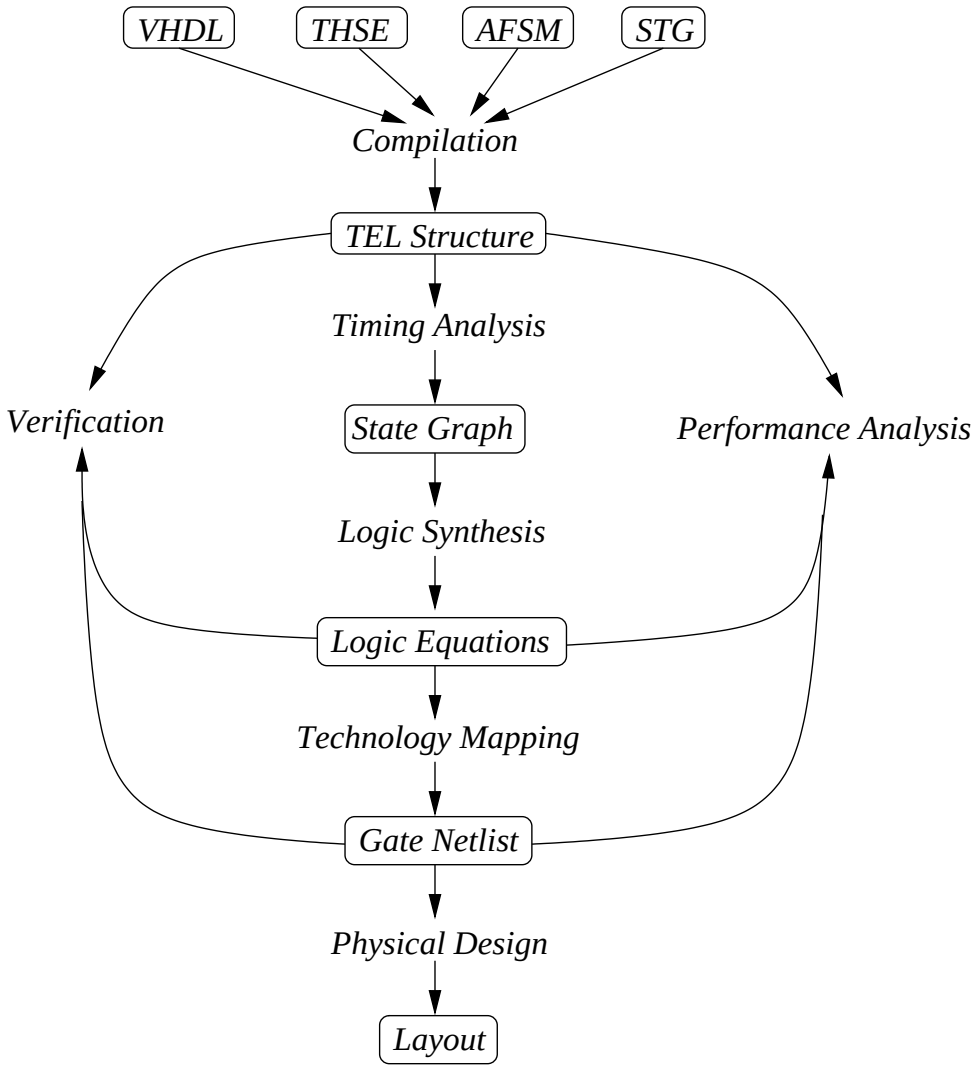

Figure 1.1. Design flow for timed circuit design.

to a TEL structure to find the reachable state space of the system. This step begins with an initial timed state which includes the set of all enabled causal relationships (or *rules*), the values of all signal wires, and timing relationships between all rules. From this timed state, all possible next timed states can be determined by firing the enabled rules. From these states, subsequent timed states are determined recursively. As timed states are found, they are stored to form a state graph. State space exploration is exponential in the size of the design, and timing makes it even more complex. On the other hand, timing information can help to identify states which are not reachable by the system, thus reducing the total number of states leading to an optimized logic implementation.

After the state space is generated, logic synthesis is applied to extract *excitation*

and *quiescent* regions from the state graph for each output signal. For each excitation region, logic equations are found that implement the region in a *hazard-free manner*. To accomplish this, correctness conditions are used to determine in which states the logic must evaluate to true, can evaluate to true, and must not evaluate to true. At this step, the implementation is largely technology independent. The logic equations may use gates that are either inefficient or non-existing. The step of technology mapping takes a set of logic equations and a gate library and determines a gate netlist to implement the logic equations.

Once the final circuit implementation is available, verification is applied to the circuit against the specification to show that the synthesized circuit is a reliable implementation of the specification [6]. The actual timing behavior of the implementation also needs to be verified that it is consistent with the specification.

The step of performance analysis is also applied to the specification and the synthesized circuit annotated with delay distribution information [54]. It is based on Monte-Carlo and Markov chain analysis, and can find steady-state probability distributions. From these distributions, the relative importance of every pin-to-pin delay in the circuit can be determined to show performance of the timed circuit implementation and pinpoint areas where optimization can lead to significant improvements in performances.

1.2 Related Work

All timing constraints of a timed circuit need to be checked to guarantee correct operation. Therefore, verification plays a critical role in timed circuit design. However, verification is not the only way to assure correctness An alternative is to design correct circuits in the first place. This requires correct synthesis. The key to the successful design of timed circuits and especially timed asynchronous circuits is a complete state space exploration. The major challenge of synthesis and verification is *state space explosion*. This problem happens in a system with many components that interact with each other or systems that have data structures that can assume many different values, such as the data path of a circuit. In such cases, the number of global states can be enormous. There has been a lot of successful work developed to deal with such a problem. This section gives an overview of the work to address the state explosion problem.

1.2.1 Circuit Specification Approaches

To automate the synthesis and verification of timed circuits, they need to be specified in a hardware description language. The expense and quality of the design of timed circuits depends on the type of specifications that are supported. In general, more flexible and expressive specifications allow the synthesis of faster and more complex circuits, but make the design process harder. More restricted specifications make the design process easier, but the derived circuits may be slow and redundant. Moreover, the specification method needs to provide an easy way to specify complex two-sided timing information needed for timed circuit design.

In general, the specification of timed systems can be loosely classified into two groups: those that use language-based specifications and those that use graph-based specifications. These two different groups require different design methods, and may generate different circuit implementations. The languages that are used to specify circuits include *CSP* [48], *Occam* [21], *Tangram* [13, 12], and *VHDL* [86]. Language-based approaches, such as those proposed by van Berkel [79] and Brunvand [22], often directly map language constructs to library blocks using syntax-directed translation. The advantage of these approaches is the ability to describe large complex systems hierarchically and combat the state explosion problem by mapping language constructs directly into fixed circuit modules. However, these approaches do not allow timing information to be specified, and the resulting circuits may be slow and redundant since optimizations are not always visible at such a high-level or are difficult to apply during syntax-directed translation. Another approach proposed by Martin in [48] translates a specification program into a self-timed circuit through a series of semantic preserving transformations. However, it does not support timing specification and needs a lot of human intervention to work effectively.

Graph-based approaches often specify circuit behavior in a lower level. It can often produce very efficient and fast circuits since timing information can be used to optimize the implementations. Graph-based methods include Petri Nets or STGs [28], I-nets [57], change diagrams [81], asynchronous finite state machines [42, 32, 84], and state graphs [60]. These methods often require complete state space exploration to find all reachable states in a design. Therefore, the state space explodes quickly as the complexity and size of the designs grow. Since specifications in these approaches are at the signal transition level, writing the specification is tedious and error prone for large designs.

Since hardware description languages are useful to organize large complex designs hierarchically, and timing analysis and synthesis algorithms are more easily applied to graphical representations, the specification method used in our synthesis and verification tool `ATACS` is a combination of language-based and graph-based approaches. The tool accepts VHDL or THSE descriptions [86] as well as AFSMs and STGs. Instead of synthesizing circuits directly from these descriptions, `ATACS` compiles them into a graphical representation, the TEL structure. Then, timing analysis algorithms are applied to find the reachable state space of the system, which is used to derive the circuit implementation.

1.2.2 Synthesis

There exist some systematic techniques for the design of timed circuits. In [17], Borriello describes a method which uses timing information in the design of *transducers*, interfaces between synchronous and asynchronous circuits. In [46], Lavagno develops a synthesis technique which uses methods similar to Chu [28] and Meng [53] to derive a complex gate-level implementation which is then mapped to a gate library using synchronous technology mapping techniques. In both methods, timing analysis is applied after synthesis to verify that the implementation is hazard-free. If hazards are detected, delay elements are added to avoid them, degrading the reliability and performance of the implementation. In [63, 64], myers first applied timed state space exploration to timed circuit synthesis. In his method, unreachable states of a design are eliminated based on the specified timing information. Therefore, this method produces optimized timed circuits. In [44], a direct synthesis method is proposed which synthesizes timed circuits directly from STGs. This method does not suffer from the state explosion problem since it does not explore the state space of the designs. The synthesized timed circuits using this method have nearly the same area compared with the results derived using other synthesis methods. However, this method can only be applied to a restricted class of free-choice STGs limiting its applicability. Furthermore, it can result in path explosion in the precedence graph derived for synthesis.

1.2.3 Verification

The purpose of verification is to give the designers confidence that resulting circuits operate correctly. Therefore, the most crucial issue in verification is the definition of correctness. In general, correctness is defined by two different approaches. One approach is model checking [29]. This approach explores the state space exhaustively and checks

if the specified properties are satisfied in every state. Another one is to check the conformance of the implementation to a *specification* [34]. Verification needs to show that the implementation exceeds the minimum requirements stated in the specification. These approaches raise another issue which is what properties need to be modeled and verified. Traditionally, there are two important properties to be modeled: *safety properties* and *liveness properties*. A safety property asserts that "nothing bad happens". A liveness property asserts that "something good happens". In the verification of a timed circuit, it is also important to verify timing properties in a specification.

One approach to circuit verification is model checking for finite state concurrent system. It checks whether a model of the behavior of the circuit satisfies a specification written as logical formulas. Most work in model checking is based on *temporal logic* [30, 36]. In general, a temporal logic is a propositional or first-order logic augmented with temporal modal operators which can assert how the behavior of the system evolves over time. Bugs have been discovered in several asynchronous circuits using this approach, and the modified designs have been shown correct [20, 56]. Although model checking has the advantage of being automated, it can only deal with small designs because the global state graph needs to be constructed before it can be checked and the state graph for large circuits can be very large.

Reachability analysis is widely used in protocol verification [16, 85]. It constructs a global state graph of a finite state system, then inspects the graph for errors. Verification using this approach checks the satisfaction of properties in each state: safety, absence of deadlock, liveness, timing constraints for timed circuits, and so on. The meaningfulness of these properties depends on the interpretation of the formal model being used and on the application. Reachability analysis cannot handle arbitrary liveness properties, because it does not consider infinite behavior.

In [34], Dill describes a hierarchical verification approach based on conformance checking using trace theory. In this approach, the circuit behavior is specified at different levels of abstraction. Specifications at one level of abstraction are treated as the descriptions of implementations at the higher levels of abstraction. If an implementation conforms to a specification, the implementation can safely replace the specification in any context while preserving correctness of the specification. In this hierarchical verification approach, irrelevant implementation details can be suppressed in moving from one level of abstraction to the next. Therefore, it can greatly reduce the computational complexity of verification.

Dill's approach can only be applied to speed-independent circuits. Verification methods based on timed trace theory are described in [25, 70, 83, 87].

1.2.4 Timed State Space Exploration

Both synthesis and verification of timed circuits require complete timed state space exploration. The state space can be found by exhaustively firing all events in a system. Since the growth of the reachable state space is highly dependent upon the representation of timing information, how to model timing behavior is a crucial issue in timed state space exploration.

There are two models to represent timing behavior of a system: discrete time and continuous time. In discrete time model [24, 18], time is broken into dicretization constants, and timers in the system can only advance in multiples of a discretization constant. Timing analysis using the discrete model is simpler and implicit methods can be applied to improve performance. The discretization constant needs to be set small enough to guarantee exact exploration of the state space. However, this can cause the state space to explode if the delay ranges are large [70].

In the continuous time model, timers in the system can take on any value between their lower bounds and upper bounds. A continuous time state space needs to be divided into equivalence classes, otherwise, the state space is infinite. All timing behaviors within an equivalence class must lead to the same state and do not need to be explored separately. Therefore, the size of equivalences should be as large as possible to reduce the number of timed states. In the *region* approach [1], timed states with the same integral clock values and a particular linear ordering of the fractional values of the clocks are equivalent. Although this approach eliminates the need to discretize time, the state space can explode if the delay ranges are large. Another approach to continuous time is to represent the equivalence classes as convex polyhedra called *zones* [33, 15, 47, 2]. The zones are represented by sets of linear inequalities (also know as *difference bound matrices* or DBMs). Although its worst complexity is worse than the discrete-time or region approaches, the zone approach often generates larger equivalence classes resulting in smaller state spaces when verifying real circuits. However, the number of zones can explode in highly concurrent systems. The reason for this explosion in the number of zones is that every possible sequence of concurrent events results in a different zone, even though these sequences result in the same untimed state. To solve this problem, a

POSET algorithm [65, 69] is proposed to consider partial ordered sets of events rather than the linear sequences. This algorithm can reduce the number of zones substantially. Belluomini extended the POSET algorithm to TEL structures and applied it to both synchronous and asynchronous designs [7, 6, 9]. In [55], Mercer described an enhanced version of the POSET method.

1.2.5 State Space Reduction

There exist many techniques and methods to deal with the state explosion problem. In [44], a direct synthesis method is presented where timed circuits are directly derived from signal transition graphs. It does not suffer from state space explosion at all since it does not explore state space. However, the class of specifications that can be handled is limited, and a similar approach cannot easily be applied to verification.

In systems with a large state space, explicit representations for the state graph can cost too much memory to be practical for realistic systems. In [26, 52], a symbolic representation for the state graph is presented. This symbolic representation is based on an *ordered binary decision diagram* [23]. It represents logic functions which express transition relations between states. OBDDs provide a canonical form for boolean functions which is substantially more compact, and very efficient algorithms have been developed for manipulating them. In [76], an implicit method using multiterminal BDDs was described. Because the symbolic representation captures some of the regularity in the state space of circuits, it can represent systems with an extremely large number of states [52]. Although implicit methods are able to represent systems with large state spaces, the exponential complexity of verification is still a serious problem.

In a concurrent system, such as an asynchronous circuit, different processes may perform independently without any synchronizations. Events executed concurrently often lead to the same state. The execution of concurrent events is represented by an *interleaving sequence* where the events are arbitrarily ordered with respect to one another. Most verification approaches explore all possible interleaving sequences which can result in an extremely large state space. In [37], a successful technique based on *partial order reduction* is presented to reduce the number of states by considering only a subset of the possible interleavings between events which is relevant to the property to be verified. *Stubborn sets* [78] and *unfolding* [50] are based on a similar idea. These techniques are targeted specifically at verification, because synthesis requires the information of the

complete state space. In [83], the partial order reduction approach is extended to timed systems.

1.2.6 Abstraction

Although the state reduction techniques in the last section are successful on some large systems, many realistic systems are still too large to be handled. To reduce the complexity incurred by state exploration, abstraction is necessary. In [4, 5, 68], safe approximations of internal signal behavior are presented to reduce the state space under consideration, but these methods suffer exponential complexity in the number of memory elements. In VIS [19], non-determinism is used to abstract the behavior of some circuit signals. It is often too conservative, and can introduce unreachable states which may exhibit hazards. In [67], a model checker is proposed based on hierarchical reactive machines. By taking advantage of the hierarchy information, it only tracks active variables so that the state space is reduced and verification time is improved. This approach, however, is best suited for software which has a more sequential nature. In [59], an abstraction technique is proposed for validation coverage analysis and automatic test generation. It removes all datapath elements which do not affect the control flow and groups the equivalent transitions together, thus resulting in a dramatic reduction in the state space. It is difficult, however, to distinguish the control from the datapath without help from the designers. In [45], an abstraction approach for the design of speed-independent asynchronous circuits from change diagrams is described. In this approach, each subcircuit is designed individually, and they are then recombined to produce the final circuit. This approach, however, does not address timing issues. In [66], Namjoshi and Kurshan describe an algorithm which constructs a finite state "abstract" program from a possibly infinite state "concrete" program by means of a *syntactic* program transformation. It can be applied to infinite state programs or programs with large data paths, and it allows other reduction methods to be applied for model checking. However, the iterative transformation may not finish. In [39], a divide-and-conquer method for synthesis of asynchronous circuits is described. This method breaks the state graph for a given problem into a number of simpler modular subgraphs for each output. Each modular subgraph is solved individually. The result of these small subgraphs are then integrated together contributing to the solution to the given problem. Although this makes synthesis and verification easier, the quality of the final solution may depend on the order in which the outputs are processed. Also, this

method generates a complete state graph before it breaks the state graph, which is highly undesirable for large complex designs. In [11], Belluomini described the verification of domino circuits using ATACS. She found out that verifying flat circuits even of a moderate size is too difficult to be done by ATACS, but with some hand abstraction, the verification is completed quickly. Although doing abstraction by hand is possible, it requires an expert user and methods must be developed to check that the abstractions are a reliable model of the underlying behavior. This is the major motivation of this research.

1.3 Contributions

This dissertation presents an automatic abstraction technique that is used to address state explosion problem in large designs. In timed circuit synthesis and verification, an environment needs to be provided to define the input behavior that the circuit must handle and output that the circuit should produce. During state space exploration, all states including the states of the environment need to be found. Since the function of an environment is to define the interface behavior for the circuit, the internal states of the environment have no impact on the bahavior of the circuit as long as the communication between the circuit and its environment remains the same. Based on the this observation, the internal details of the environment can be abstracted away, and the state space of the circuit and its environment can be reduced substantially.

When designing a circuit whose state space is too large to be represented, the divide-and-conquer method has to be used. In ATACS, a large circuit is partitioned into smaller blocks. For each block, the rest of blocks in the circuit and the environment for the whole circuit become the environment for the block. After the partitioning, the block and its environment contain the same amount of information that needs to be considered during state space exploration as before. From the above discussion, it is known that the internal states of the environment can be abstracted away. By using structural information, the internal details of the environment for the block can be identified and removed. After the environment is simplified, each block is synthesized or verified. This process is applied to all blocks in the circuit. Once the results for all blocks are available, they are integrated together to form the solution to the whole circuit. This idea can be illustrated by an example shown in Figure 1.2. In Figure 1.2(a), a circuit has four components, each has constrained interface. Suppose we would like to design component *M2*, components *M1, M3, M4* and the environment for the whole circuit E together become the environment

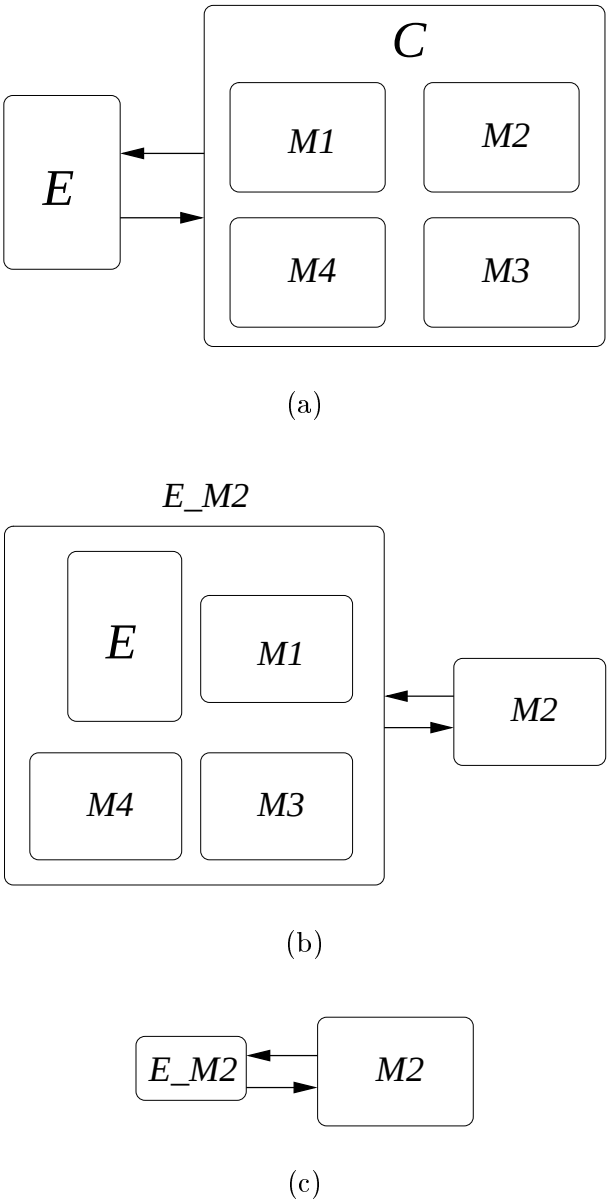

Figure 1.2. Illustration of modular design using abstraction.

for *M2* as shown in Figure 1.2(b). Since *M2* has a constrained interface, the environment for *M2*, *E_M2*, contains internal signals whose behavior has no impact on *M2*. After abstracting those signals away from the environment *E_M2* as shown in Figure 1.2(c), the complexity for designing *M2* can be substantially reduced. The same process can be repeated for *M1, M3* and *M4*. Although this approach does not solve the exponential

complexity inherent in the state space exploration, it improves the speed and memory usage of the design process by converting a large exponential problem into a set of small exponential sub-problems with a just little overhead for abstraction. This dissertation also gives theorems that prove the correctness of the design using our abstraction technique. This abstraction technique is implemented in a compiler [86] frontend to our timed circuit design tool `ATACS` and are applied to several examples. The results show that the design process with our abstraction technique is not only orders of magnitude faster and more memory efficient, but also successful on orders of magnitude more complex designs than can be designed without using it.

There are three major contributions of this dissertation. The first contribution is the theorems that give the theoretical support for the modular synthesis and verification of timed circuits using the abstraction techniques presented in this dissertation. The modular synthesis theorem asserts that if each block of a system is synthesized correctly with an abstracted version of the given environment, the composition of the results for all blocks is the correct solution to the whole system. Similarly, the modular verification theorem asserts that if each block of a system is verified correctly with an abstracted version of the given environment, the whole system is also correct. The successful application of these theorems is supported by *safe transformations* that reduce the complexity of a design while preserving its behavior.

The second contribution is the abstraction technique that is applied to the TEL structures. Since the state space of a design grows exponentially in size of a design, it is easier to use a divide-and-conquer approach to solve a complex design problem. The abstraction presented in this dissertation can aid designers to partition a circuit into blocks with manageable sizes, choose a block for synthesis and verification, group the rest of blocks and the environment for the whole circuit together as the environment for the selected block, and then identify the don't-care information for the block using specified hierarchical information.

The third contribution of this dissertation is the *safe net reductions* and *redundant rule checks* used by abstraction to remove the identified don't-care information in the environment while preserving the behavioral semantics of the block. These techniques reduce the design complexity of the block by removing the part of the environment specification which does not affect the operation of the block. These techniques have been proved to be safe according to the definition of safe transformations. Combination

of abstraction and safe transformations can make the design process much faster and more memory efficient than designing the flat system.

1.4 Dissertation Outline

This dissertation is organized as follows: Chapter 2 gives an overview of the specification method used in `ATACS` and its behavioral semantics. In `ATACS`, specifications in VHDL or THSE are compiled to TEL structures to specify the circuit behavior, and timed trace theory is used as the semantics. This chapter serves as the groundwork for the rest of this dissertation.

Chapter 3 defines the correctness of synthesis and verification, and safe transformations. This chapter also presents hierarchical synthesis and verification theorems which are mathematically proved. The significance of these theorems is that any design process using the abstraction in this dissertation is correct.

Chapter 4 describes the abstraction techniques. After a block is chosen for synthesis or verification, this chapter describes how abstraction identifies the don't-care information for the block by taking advantage of the structural information given in a specification. The abstraction technique for TEL structures with levels is different from that for TEL structures without levels. In this chapter, *safe abstraction* is defined to handle levels.

Since the don't-care information does not affect the behavior of the block, it is necessary to remove it to reduce the complexity of designing the block. Chapter 5 describes several safe net reduction techniques that remove the don't-care information while preserving the behavioral semantics of the block. Chapter 6 describes techniques to uncover and remove redundant rules in a TEL structure.

Chapter 7 gives experimental results using the techniques presented in this dissertation. The comparison between the abstraction technique and the traditional flat design approach is also given in this chapter.

Chapter 8 summarizes this dissertation, and discusses the future work and the necessary improvement on our abstraction technique.

CHAPTER 2

CIRCUIT SPECIFICATIONS AND SEMANTICS

Specification methods of timed circuits can be loosely divided into two groups: hardware description language (HDL) methods and graphical representations. HDL specifications are expressive in describing large and complex systems with a modular and hierarchical structure, while graphical representations are preferred during timing analysis and synthesis. Therefore, combining the two approaches provides an easy way to describe large and complex systems while the common timing analysis and synthesis algorithms can still be applied to optimize the circuit implementations. In [86], a new compiler frontend to `ATACS` is described. This compiler accepts inputs in VHDL or THSE, and compiles them into a graphical representation, the *timed event/level (TEL) structure* [8]. The behavioral semantics of a TEL structure is defined by using trace theory [34]. In this chapter, the first section gives an overview of the specification languages, namely, THSE and the synthesizable subset of VHDL. TEL structures are described in the next section. The third section gives a brief overview of the compilation procedure from a timed specification to a TEL structure. The fourth section describes the behavioral semantics of TEL structures, namely, *timed trace theory*. The last section describes how to derive a trace structure from a TEL structure.

2.1 Timed Specifications

The first step in any design method is to specify what is to be built. This section gives an overview of the syntax of THSE and a synthesizable subset of VHDL. Both languages allow a bounded timing constraint associated with each signal transition.

2.1.1 Timed Handshaking Expansions

A system can be specified structurally, behaviorally, or in a mixed manner. In Timed Handshaking Expansion (THSE), modules are the basic complete structures to specify

$$
\begin{array}{rcl}
\textit{module} & \Rightarrow & \textbf{module}\ \text{ID};\ \textit{declarations}\ \ \textit{stmts}\ \ \textbf{endmodule} \\
\textit{declarations} & \Rightarrow & \textit{declarations}\ \ \textit{sigdecl}\ \mid\ \textit{sigdecl} \\
\textit{sigdecl} & \Rightarrow & \textit{type}\ \text{ID} = \{\textit{initial},\ \textit{delay}\}; \\
\textit{stmts} & \Rightarrow & \textit{stmts}\ \ \textit{stmt}\ \mid\ \textit{stmt} \\
\textit{stmt} & \Rightarrow & \textit{cmpt\_stmt}\ \mid\ \textit{process}\ \mid\ \textit{gate}\ \mid\ \textit{constraint} \\
\textit{cmpt\_stmt} & \Rightarrow & \text{ID ID}(\textit{assoc\_list}) \\
\textit{assoc\_list} & \Rightarrow & \textit{assoc\_list},\ \text{ID} => \text{ID}\ \mid\ \text{ID} => \text{ID} \\
\textit{process} & \Rightarrow & \textbf{process}\ \text{ID};\ \textit{commands}\ \textbf{endprocess} \\
\textit{initial} & \Rightarrow & \textbf{true}\ \mid\ \textbf{false} \\
\textit{delay} & \Rightarrow & \langle \text{INT}, \text{INT};\ \text{INT}, \text{INT} \rangle\ \mid\ \langle \text{INT}, \text{INT} \rangle \\
\textit{commands} & \Rightarrow & \textit{commands}\ ;\ \textit{cmnd}\ \mid\ \textit{commands} \parallel \textit{cmnd}\ \mid\ \textit{cmnd} \\
\textit{cmnd} & \Rightarrow & \textit{action}\ \mid\ \textit{selection}\ \mid\ \textit{repetition}
\end{array}
$$

Figure 2.1. Modules, signal declarations, components and processes.

a system. The syntax of a subset of THSE is shown Figure 2.1. A module is composed of two parts: a set of *signal declarations* and a set of concurrent statements executing in parallel. Each declaration consists of a type (either **input** or **output**), a signal name, an initial value (either **true** or **false**), and a bounded delay associated with transitions on that signal. A delay is given in a form: $\langle l_r, u_r; l_f, u_f \rangle$ where l_r and u_r are the lower and upper bounds on a rising transition and l_f and u_f are the lower and upper bounds on a falling transition. If the delay is given in a form: $\langle l_r, u_r \rangle$, the delays are equal on rising and falling transitions. The lower bounds are non-negative integers and the upper bounds are an integer greater than or equal to the lower bound or ∞.

There are four kinds of concurrent statements: processes, gates, constraints, and component statements. Processes are used to specify the behavior of a system. A process consists a set of commands. The commands in the body of a process include actions, selection commands, and repetition commands. Commands can be executed in sequence (denoted $C_1; C_2$) or in parallel (denoted $C_1 \parallel C_2$). The actions are used to assign values to the output signals. Signals can only take two values: **true** and **false**. There are two actions associated with each signal x: $x+$ denotes that signal x changes from a low to high value, and $x-$ denotes that x changes from a high to low value. The language also includes a **skip** action that does nothing and terminates immediately.

*Selection*s and *repetition*s are used to control the flow of processes. A selection command has the following form:

$$[G_1 \ \rightarrow \ S_1 \ \mid \ \cdots \ \mid \ G_n \ \rightarrow \ S_n]$$

where G_1 through G_n are boolean expressions, S_1 through S_n are sequences of commands (G_i is called a "guard", and $G_i \ \rightarrow \ S_i$ is called a "guarded command"). The guard G_i of a guarded command is a boolean expression over a set of actions or signal values. The actions in this expression can be composed *conjunctively* (denoted $e_1 \ \& \ \cdots \ \& \ e_n$) in which the expression evaluates to true when all actions in the set has occurred. Mutually exclusive actions can be composed *disjunctively* (denoted $e_1 \mid \cdots \mid e_n$) in which the expression evaluates to true when exactly one action in the set has occurred. The guard may include a combination of conjunctive and disjunctive clauses. The guard can also be a **skip** action which evaluates to true immediately. If the expression is composed of signal values, s or $\neg s$, the guard evaluates to true when the signal s is high or low, respectively. Signals in the expression can also be composed conjunctively, disjunctively, or in a combination of both. A guard can simply be **true**. There is a subtle difference between a guard composed of actions and a guard composed of levels disjunctively. If composed of levels disjunctively, the expression evaluates to true when one or more levels evaluate to true. Signal levels in a disjunctive expression are not mutually exclusive while actions must be. For example, the guard command $[a+ \mid b+ \rightarrow c+]$ specifies that $c+$ can occur only after either $a+$ or $b+$ has occurred, but not both, while the guard command $[a \mid b \rightarrow c+]$ specifies that $c+$ can occur after the value of either a or b, or both are high.

When a process executes a selection command, all guards in that selection command are evaluated first. If one of the guards, G_i, is **true**, then a sequence of commands, S_i, following that guard is executed. If multiple guards evaluate to true, only one guard is chosen nondeterministically. There is a special form of selection command, $[G]$, which stands for $[G \ \rightarrow$ **skip**$]$, and is used to suspend the execution of a process until G evaluates to **true**.

A repetition command has the following form:

$$*[G_1 \ \rightarrow \ S_1 \ \mid \ \cdots \ \mid \ G_n \ \rightarrow \ S_n]$$

During execution, one of the guarded commands is chosen for execution, then the control loops back to the beginning of this command. If none of the guards evaluates to true, the execution of this command terminates and the control goes to the next command. There is a shorthand of this repetition command: $*[S]$ that stands for $*[$**true** $\rightarrow \ S]$ where S is

module *emptystage*;
input $xtin$;
input $xfin$;
input $ackin$;
output $xt = \{\langle 100, 200\rangle\}$;
output $xf = \{\langle 100, 200\rangle\}$;
output $ack = \{\mathbf{true},\ \langle 100, 200\rangle\}$;
process *datastage*;
$*[[xtin+ \rightarrow xt+ \mid xfin+ \rightarrow xf+] : [ackin-] :$
$[xtin- \rightarrow xt- \mid xfin- \rightarrow xf-] : [ackin+]]$
endprocess
process *ackstage*;
$*[[xt+ \mid xf+] : ack- : [xt- \mid xf-] : ack+]$
endprocess
endmodule

Figure 2.2. The THSE code for a single empty STARI stage.

a sequence of commands. This command causes S to be executed forever. This is usually used to define a reactive process:

$$*[[G_1 \ \rightarrow \ S_1 \ \mid \ \ldots \ \mid \ G_n \ \rightarrow \ S_n]]$$

When executing this command, the process waits until one of the guards is **true**, then executes the commands following that guard, and repeats. Another type of repetition construct is shown below:

$$[G_1 \ \rightarrow \ S_1 \ \mid \ \ldots \ \mid \ G_n \ \rightarrow \ S_n; *]$$

The operation of this construct is similar to that of the *selection* defined above except that after a guarded command followed by a '$*$' is executed, the control loops back to the beginning of the selection command. Otherwise, the control goes to the next command.

Figure 2.2 shows an example to illustrate how to use the language constructs to specify a circuit. The module in the example specifies that the circuit has three input signals and three output signals. It also contains two processes to define the behavior of the circuit.

Besides processes, a module can also contain a set of gate and constraint statements. Gates are used to describe the behavior of a system at the gate-level. A gate statement has the following form:

gate ID;
$G_1 \rightarrow x+$
$G_2 \rightarrow x-$
endgate

module *emptystage*;
input *xtin*;
input *xfin*;
input *ackin*;
output $xt = \{\langle 100, 200\rangle\}$;
output $xf = \{\langle 100, 200\rangle\}$;
output $ack = \{\mathbf{true}, \langle 100, 200\rangle\}$;
gate *ct*;
  $ackin\ \&\ \sim xtin \rightarrow xt+$
  $\sim ackin\ \&\ \sim xtin \rightarrow xt-$
endgate
gate *cf*;
  $ackin\ \&\ xfin \rightarrow xf+$
  $\sim ackin\ \&\ \sim xfin \rightarrow xf-$
endgate
gate *ack*;
  $xt\ \mid\ xf \rightarrow ack-$
  $\sim xt\ \&\ \sim xf \rightarrow ack+$
endgate
endmodule

Figure 2.3. The gate-level THSE code for a single empty STARI stage.

G_1 and G_2 are boolean expressions over a set of signal values. G_1 and G_2 defines the conditions when signal x can go high and low, respectively. Figure 2.3 shows the gate-level specification for the circuit shown in Figure 2.2. A gate statement is defined for each output signal. For example, in gate statement *ct*, signal xt goes high when the value of *ackin* is high and value of *xtin* is low. xt goes low when the value of both *ackin* and *xtin* is low.

Constraint statements are used to define timing properties among signals that are used for verification. A constraint statement has the following form:

constraint ID;
  $G \rightarrow \langle \mathrm{INT}, \mathrm{INT}\rangle\ \ action$
endconstraint

G can be a single action or a boolean expression over a set of signal values. The two integers define a lower and upper bound when a signal event is required to occur after another event has occurred or the boolean expression of G has evaluated to true. These two integers are optional. In such case, they are assumed to be 0 and ∞.

The component statements are used to specify the interconnection of the components, and provide a way to modulize the design and to manage the design complexity. This

```
module stari;
input clk = {⟨1200, 1200⟩};
input ack3 = {true, ⟨0, 100⟩};
input x0t = {⟨0, 100⟩};
input x0f = {⟨0, 100⟩};
efstage stage1(xtin => x0t, xfin => x0f, ackin => ack2,
               xt => x1t, xf => x1f, ack => ack1);
fullstage stage2(xtin => x1t, xfin => x1f, ackin => ack3,
                 xt => x2t, xf => x2f, ack => ack2);
process clk;
  *[clk+; clk−]
endprocess
process left;
  *[[clk+] : [skip → x0t+; [clk−/1]; x0t−
    |skip → x0f+; [clk−/1]; x0f−
   ]] :
endprocess
process right;
  *[[clk+] : ack3− : [clk−] : ack3+] :
endprocess
constraint notfull1;
  ack1 + @ → x0t+
endconstraint
constraint notfull2;
  ack1 + @ → x0f+
endconstraint
constraint notempty1;
  x2t + @ → ack3−
endconstraint
constraint notempty2;
  x2f + @ → ack3−
endconstraint
endmodule
```

Figure 2.4. The THSE code for a 2-stage STARI.

language construct is new since [86]. A component is an instance of a module. A component statement consists of a label, a type (i.e. module name) and an association list which connects the inputs and outputs of the component to the signals in the module. A component statement renames the signals declared in the component to the corresponding signals in the association list. The association list of a component statement consists of all input signals and a subset of output signals declared in the corresponding module. The output signals not on the association list are internal to the module.

Figure 2.4 shows the THSE specification for a circuit that consists of 2 stages. There

are two components in the module that specifies the structure of the circuit. The two processes specify the environment of the circuit. The constraints specify the timing properties that this circuit must satisfy. For example, constraint $ack1+ \rightarrow x0t+$ specifies that signal $x0t$ can go high only after $ack1$ goes high. If this constraint is not satified, the circuit is not correct.

2.1.2 A Synthesizable Subset of VHDL

The complete VHDL language contains many complicated language constructs that are not synthesizable in `ATACS`. This section introduces a synthesizable subset of VHDL to specify timed circuits.

The description of a system can be divided into two parts: the external view and the internal view. The external view describes the interface between the internal structure and the outside world. It specifies the number and types of the input and output signals. The internal view describes how the circuit implements its function. In VHDL, an *entity* describes the external interface, and one or more *architecture bodies* describe alternative internal implementations.

The syntax rules for entities and architecture bodies are shown in Figure 2.5. The identifier in an entity declaration names the module so that it can be referred to later. The port clause, which is optional, names each of the ports, which together form the interface to the entity. The ports can be thought of as being analogous to the pins of a circuit. Each port of an entity has a type, which specifies the kind of information that can be communicated. In this subset, the allowed data types are **bit** and **std_logic**. Each port also has a mode which specifies whether information flows into or out of the entity through the port. If the mode of a port is **in**, it means that the port can only read the information. If the mode is **out**, it means that the port can only output the information generated by the circuit. If the mode is **inout**, it means that the information can be both read and output by the port.

The internal operation of a module is described by an architecture body. In general, an architecture body applies some operations to the values on input ports, generating values to be assigned to output ports. The operations can be described either by processes, which contain sequential statements operating on values, or by a collection of components representing subcircuits, or by both. The identifier in an architecture body names a particular architecture body, and the entity name specifies which module is described by this architecture body. A single entity may have one or more different architecture bodies.

$$
\begin{array}{rcl}
\textit{entity\_declaration} & \Rightarrow & \textbf{entity}\ \text{ID}\ \textbf{is} \\
 & & \quad [\textbf{port}(\textit{interface\_list});] \\
 & & \textbf{end}\ [\textbf{entity}\ ]\ [\text{ID}]; \\
\textit{interface\_list} & \Rightarrow & (\text{ID}\{,\ldots\}:\ [\textit{mode}]\ \textit{type}\ [:= \textit{expression}])\{;\ldots\} \\
\textit{mode} & \Rightarrow & \textbf{in} \mid \textbf{out} \mid \textbf{inout} \\
\textit{architecture\_body} & \Rightarrow & \textbf{architecture}\ \text{ID}\ \textbf{of}\ \textit{entity\_ID}\ \textbf{is} \\
 & & \quad \textit{declarations} \\
 & & \textbf{begin} \\
 & & \quad \textit{concurrent\_stmts} \\
 & & \textbf{end}\ [\textit{architecture}]\ [\text{ID}]; \\
\textit{declarations} & \Rightarrow & \textit{signal\_declarations} \mid \textit{component\_declarations} \\
\textit{concurrent\_stmts} & \Rightarrow & \textit{process\_stmt} \mid \textit{component\_stmt}
\end{array}
$$

Figure 2.5. The syntax rules for entities and architecture bodies.

The declarations in an architecture body include signals and component declarations. The statements in the architecture body execute concurrently. In this synthesizable subset, process statements and component instantiation statements are allowed in an architecture body.

The syntax for signal and component declarations is shown in Figure 2.6. The signal declarations are used to specify signals used in an architecture and their attributes. Each signal declaration consists of a set of signal names, their data type, and an optional initial value. Signals declared in the entity of an architecture are also visible inside the architecture body and are used in the same way as signals declared in the architecture.

To synthesize a timed circuit, it is necessary to know how its environment behaves. The signals that connect the outputs of the environment to the inputs of the circuit are not synthesized. These signals are labeled by attaching a symbol "$--$@**in**" at the end of the declarations of those signals. This symbol is recognized by the `ATACS` synthesis engine but ignored by the simulator. This symbol is only used in the top level that contains the specifications for the whole circuit and the environment.

When designing a large and complicated system, a hierarchical approach is a good way to attack complexity. In this VHDL subset, component declarations and component instantiation statements are used for hierarchical design. The syntax of component declarations is shown in Figure 2.6. Similar to entity declarations, a component declaration simply specifies the external interface to the component.

$$
\begin{array}{rcl}
\textit{signal\_declarations} & \Rightarrow & \textbf{signal}\ \text{ID}\ \{,\ \cdots\}\ :\ \textit{type}\ [:= \textit{expression}]; \\
\textit{component\_declaration} & \Rightarrow & \textbf{component}\ \text{ID}\ [\textbf{is}] \\
 & & \quad \textbf{port}(\textit{interface\_list}); \\
 & & \textbf{end}\ [\textbf{component}]\ [\text{ID}];
\end{array}
$$

Figure 2.6. The syntax rules for signal and component declarations.

$$
\begin{array}{rcl}
\textit{component\_stmt} & \Rightarrow & [\textit{instantiation\_label}\ :] \\
 & & [\textbf{component}]\ \textit{component\_name} \\
 & & \quad \textbf{portmap}\ (\textit{association\_list}); \\
\textit{process\_stmt} & \Rightarrow & [\textit{process\_label}\ :]\ \textbf{process}\ [\textbf{is}] \\
 & & \quad \textit{variable\_declarations} \\
 & & \textbf{begin} \\
 & & \quad \textit{sequential\_stmts} \\
 & & \textbf{end}\ \textit{process}\ [\textit{process\_label}];
\end{array}
$$

Figure 2.7. The syntax rules for concurrent statements.

Concurrent statements in an architecture body are executed in parallel. A process specifies the behavior of a system, and a component instantiation statement specifies the interconnection between a subcircuit and the rest of the architecture body. The syntax rules for process and component instantiation statements are shown in Figure 2.7.

If a component is used in an architecture, it must be declared first, and instantiated by a component instantiation statement. A component instantiation statement specifies a usage of such a module in a design. The syntax rules show that we may simply name a component declared in the architecture body and provide actual signals to connect it to the ports in the entity. When the statement is used, all signals in the component are renamed to the corresponding actual signals in the association list. The label is necessary to identify the component instance. A process statement consists of a set of variable declarations and sequentially executed statements. The variable declarations in a process specify attributes of variables. Variable are only used in the processes specifying nondeterministic behavior of an environment. They are not synthesizable in `ATACS`. A process contains a set of sequential statements including *guard*, *assign*, *if*, and *while loop* statements. When the process is activated, it starts executing from the first sequential

$$
\begin{array}{rcl}
\textit{sequential\_stmts} & \Rightarrow & \textit{sequential\_stmts};\ \textit{stmt} \\
\textit{stmt} & \Rightarrow & \textit{if\_stmt} \mid \textit{loop\_stmt} \mid \textit{guard} \mid \textit{assign} \\
\textit{if\_stmt} & \Rightarrow & \textbf{if}\ \textit{boolean\_expression}\ \textbf{then} \\
 & & \quad \textit{sequential\_stmts} \\
 & & \textbf{elsif}\ \textit{boolean\_expression}\ \textbf{then} \\
 & & \quad \textit{sequential\_stmts} \\
 & & \textbf{else} \\
 & & \quad \textit{sequential\_stmts} \\
 & & \textbf{endif};
\end{array}
$$

$$
\begin{array}{rcl}
\textit{loop\_stmt} & \Rightarrow & \textbf{while}\ \textit{boolean\_expression}\ \textbf{loop} \\
 & & \quad \textit{sequential\_stmts} \\
 & & \textbf{endloop}; \\
\textit{guard} & \Rightarrow & \textbf{guard}\ (G_1, G_2); \\
 & & \mid \textbf{guard\_or}\ (G_1, \cdots, G_2); \\
 & & \mid \textbf{guard\_and}\ (G_1, \cdots, G_2); \\
\textit{assign} & \Rightarrow & \textbf{assign}(\textit{assign\_stmt}\{,\ \cdots\}); \\
\textit{assign\_stmt} & \Rightarrow & \mathrm{ID}, \textit{expression}, \mathrm{INT}, \mathrm{INT}
\end{array}
$$

Figure 2.8. The syntax rules for sequential statements.

statement and continues until it reaches the last one. It then starts again from the first one. This would be an infinite loop, and is desirable in electronic circuits because circuits typically operate continuously until the power is shut down. The syntax rules for sequential statements are shown in Figure 2.8.

An *if* statement consists of a set of **if** branches and an **else** branch. Each **if** branch contains a boolean expression over a set of signal values and a set of sequential statements. If one of expressions evaluates to true, the following statements are executed. If expressions of multiple branches evaluate to true, the first branch in the statement is chosen. If none of the expressions evaluates to true, the statements in the **else** branch are executed. A *while loop* statement is used to describe a piece of repetitively executed program. A *while loop* statement consists of a boolean expression and a set of sequential statements. If the expression evaluates to true, the statements in the *while loop* statement are executed repetitively until the expression evaluates to false. The expression can simply be a **true** that causes the *while loop* statement to execute infinitely.

Guard and *assign* are two procedures that come from the *handshake* package devel-

oped for ATACS. The procedure $guard(s, v)$ takes a signal, s, and a value, v, and stalls a process until the signal s has taken the value v. The VHDL code that implements $guard(s, v)$ is as follows:

if $(s \mathrel{/}= v)$ **then**
  wait until $s = v$;
endif

In VHDL, a *wait* statement stalls the process until the expression $s = v$ becomes true. However, if the expression is true when the *wait* is executed, the process stalls until the expression goes false and becomes true again. This can cause a system to deadlock. To address this problem, the expression is checked before the execution of the *wait* statement to make sure that the *wait* is ignored if the expression is true. The procedure $guard\_or(s_1, v_1, s_2, v_2, \cdots)$ takes a set of signals and values and stalls a process until some signal s_i has taken value v_i. Similarly, the procedure $guard\_and(s_1, v_1, s_2, v_2, \cdots)$ takes a set of signals and values and stalls a process until all signals s_i have taken value v_i. The procedure $assign(s, v, l, u)$ takes a signal, s, a value, v, a lower bound of delay, l, and an upper bound of delay, u, and changes the value of the signal s to v after after a random delay between l and u. This is a truly sequential statement in that the statement following an *assign* procedure can execute only after the signal event created by the *assign* procedure has occurred. The *assign* procedure also allows parallel assignments. For example, $assign(s_1, v_1, l_1, u_1, s_2, v_2, l_2, u_2)$ changes the values of s_1 and s_2 to v_1 and v_2 in parallel.

2.2 Timed Event/Level Structures

Timed event/level (TEL) structures are a variant of Myers' *timed event-rule (ER) structures* [63] with a boolean condition added to each rule in the rule set. Event structures were introduced by Winskel [82], and timing has been added to them in several ways. Subrahmanyam added timing to event structures using temporal assertions [72]. Burns introduced timing in a deterministic version, the event-rule (ER) system, in which causality is represented using a set of rules, and a single delay value is associated with each rule [27]. Myers introduced timed ER structures that extend ER systems with bounded timing constraints and add conflicts from event structures to model nondeterministic behavior (namely, environmental choice). TEL structures, introduced by Belluomini [8], extend timed ER structures by associating a boolean expression with each rule.

The formal definition of a TEL structure is given below in which $\mathcal{N} = \{0, 1, 2, 3, \ldots\}$:

Definition 2.2.1 *A timed event/level structure is* $N = \langle \Sigma, A, E, R, \#, S_0 \rangle$ *where*

1. Σ *is the set of signals;*
2. $A \subseteq \Sigma \times \{+, -\}$ *is the set of atomic actions;*
3. $E \subseteq A \times \mathcal{N} \cup \{\$\}$ *is the set of events;*
4. $R \subseteq E \times E \times \mathcal{N} \times (\mathcal{N} \cup \{\infty\}) \times z : \{0,1\}^N \rightarrow \{0,1\})$ *is the set of rules;*
5. $\# \subseteq E \times E$ *is the conflict relation;*
6. $S_0 = \{0,1\}^N \times R \times (R \rightarrow Q)$ *is the initial state.*

A TEL structure can also be expressed as a directed cyclic graph where the nodes in the graph are the corresponding events in the TEL, and the edges are the rules in the rule set of the TEL. Each edge is labeled by a timing constraint and level for the corresponding rule. Also in this dissertation, an edge may be labeled by a unique identifier that is used to represent the corresponding rule in the explaination. The description of conflicts is given later in this section.

In timed systems, the signal set, Σ, contains all input, output, and internal wires in the specification. The atomic action set, A, contains a rising transition and a falling transition for each signal $x \in \Sigma$, denoted by $x+$ and $x-$, respectively. The occurrence of an action is an event, and it is denoted (a, i) where a is the action and i is an *occurrence index* for the action. The first instance of this action has $i = 0$, and i increments with each subsequent instance. There is also a special kind of events: a *sequencing event* starting with '$'. Sequencing events do not represent any signal value changes in a system. They are place holders for timing information and boolean level evaluations. In the early stage of the VHDL compiler development [86], the introduction of sequencing events is for the convenience of compilation. During abstraction, sequencing events are created to replace the don't-care events in a system. The don't-care events are defined in a later chapter. Since sequencing events cause no signal value change in a system, they should be removed, whenever possible, to reduce the complexity of the design problem to be solved. Removal of sequencing events is a crucial step for abstraction to be successful, as described in a later chapter.

The rule set, R, is used to represent a causal dependence between two events. Each rule of the form $\langle e, f, l, u, z \rangle$ is composed of an *enabling event* $e \in E$, an *enabled event* $f \in E$, a *bounded timing constraint* $\langle l, u \rangle$ where $l \in \mathcal{N}$ and $u \in \mathcal{N} \cup \infty$, and a sum-of-product

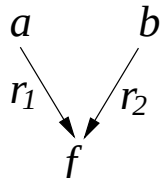

Figure 2.9. An example of conjunctive causality.

boolean expression z over the signals in the signal set N. Note that z is not shown in a rule if the value of z is **true**. Given a rule $r = \langle e, f, l, u, z \rangle$, $\texttt{enabling}(r) = e$, $\texttt{enabled}(r) = f$, $\texttt{lower}(r) = l$, $\texttt{upper}(r) = u$ and $\texttt{level}(r) = z$. For an event $e \in E$, the *preset* of e (denoted $\bullet e$) is the set of rules where e is the enabled event (i.e., $\texttt{enabled}(r) = e$ for all $r \in \bullet e$), and the *postset* of e (denoted $e\bullet$) is the set of rules where e is the enabling event (i.e., $\texttt{enabling}(r) = e$ for all $r \in e\bullet$). The size of the preset of an event e (denoted $\texttt{size}(\bullet e)$) is the number of rules in $\bullet e$. Similarly, the size of the postset of an event e (denoted $\texttt{size}(e\bullet)$) is the number of rules in $e\bullet$. For an event $e \in E$, the enabling set of e is the set of events that are the enabling events of the rules in the preset of e (i.e., $\texttt{enabling\_set}(e) = \{t = \texttt{enabling}(r) \mid r \in \bullet e\}$), and the enabled set of e is the set of events that are the enabled events of the rules in the postset of e (i.e., $\texttt{enabled\_set}(e) = \{t = \texttt{enabled}(r) \mid r \in e\bullet\}$). A rule r is *enabled* if $\texttt{enabling}(r)$ has fired and $\texttt{level}(r)$ evaluates to true in the current state. A timer is assigned to each rule when it becomes enabled. $\texttt{timer}(r)$ is initialized to zero when r is enabled. All timers of enabled rules increase uniformly. The bounded timing constraint $\langle l, u \rangle$ places a lower and upper bound on the timing of a rule. A rule r is said to be *satisfied* if r is enabled and $\texttt{timer}(r) \geq \texttt{lower}(r)$. A rule r is said to be *expired* if r is enabled and $\texttt{timer}(r) \geq \texttt{upper}(r)$. Ignoring conflicts, an event e cannot occur until r is satisfied for *all* $r \in \bullet e$. This causality requirement is termed *conjunctive*. An event e must always fire before *every* $r \in \bullet e$ is expired. Since an event may be enabled by multiple rules, it is possible that the difference in time between the enabled event and some enabling events exceed the upper bound of their timing constraints, but not for all enabling events. These timing constraints are the same as *max constraints* [51] and *type 2 arcs* [80]. Figure 2.9 shows an example that expresses this conjunctive causality where c is enabled by two rules r_1 and r_2. Given a rule $r = \langle e, f, l, u, z \rangle$, $\texttt{EFT}(f \leftarrow r)$ and $\texttt{LFT}(f \leftarrow r)$ indicate the *earliest* and *latest* firing time of f decided by r, and they are defined as follows:

$$\texttt{EFT}(f \leftarrow r) = t_r + l \quad \text{and} \quad \texttt{LFT}(f \leftarrow r) = t_r + u$$

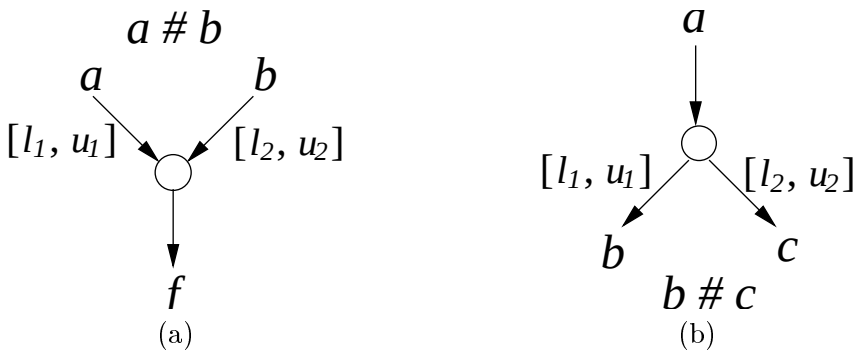

Figure 2.10. Examples of conflict places for disjunctive causality and conflict outputs.

where t_r is the time when r becomes enabled. If an event f is enabled by multiple rules, for example, $r_1 = \langle a, f, l_1, u_1, z_1 \rangle$ and $r_2 = \langle b, f, l_2, u_2, z_2 \rangle$, $\texttt{EFT}(f \leftarrow r_1, r_2)$ and $\texttt{LFT}(f \leftarrow r_1, r_2)$ are the *earliest* and *latest* firing time of f decided by r_1 and r_2, and they are defined as follows:

$$\begin{aligned} \texttt{EFT}(f \leftarrow r_1, r_2) &= max(t_{r_1} + l_1,\ t_{r_2} + l_2) \\ \texttt{LFT}(f \leftarrow r_1, r_2) &= max(t_{r_2} + u_1,\ t_{r_2} + u_2) \end{aligned}$$

where t_{r_1} and t_{r_2} are the times when r_1 and r_2 become enabled.

There are two possible semantics concerning the enabling of a rule. In one semantics, referred to as *non-disabling semantics*, once a rule becomes enabled, it cannot lose its enabling due to a change in the state. In the other semantics, referred to as *disabling semantics*, a rule can become enabled and then lose its enabling. This can occur when another event fires, resulting in a state where the boolean function is no longer true. A single specification can include rules with both types of semantics. Non-disabling semantics are typically used to specify environment behavior and disabling semantics are typically used to specify logic gates. For the purposes of verification, the disabling of a boolean expression on a disabling rule is assumed to correspond to a failure, since it corresponds to a glitch on the input to a gate.

The conflict relation in # is added to model *disjunctive* behavior and choice. When two events e and e' are in conflict (denoted $e\#e'$), this specifies that either e can occur or e' can occur, but not both. Taking the conflict relation into account, if two rules have the same enabled event and conflicting enabling events, then only one of the two mutually exclusive enabling events needs to occur to cause the enabled event. This models a form

of disjunctive causality. Choice is modeled when two rules have the same enabling event and conflicting enabled events. In this case, only one of the enabled events can occur. Figure 2.10 shows an example of disjunctive causality and an example of choice. The circles in the figure, similar to the places in Petri nets, are conflict places which represent conflicts among events. The events in the preset and postset of a place are in conflict. The conflict places are just for notational convenience. If it is impossible to display conflicts using conflict places, we label the conflicts using text in figures. The concept of conflict places can be extended to single rules. We say that a single rule has a conflict place implicitly. For example, if an event e has two rules in its preset, and their enabling events are in conflict, then there is only one conflict place in the preset of e. If the enabling events of the rules are not in conflict, we say that there are two conflict places in the preset of e. This concept is used when analyzing safe net reductions in Chapter 5.

If an event e is enabled by multiple rules and there are conflicts in $\texttt{enabling\_set}(\bullet e)$, we define a conflict-free set $\texttt{cfs}(\bullet e)$ to be the maximum subset of $\bullet e$ such that there is no conflicts among the enabling events of the rules in $\texttt{cfs}(\bullet e)$. $\bullet e$ can be divided into several different conflict-free sets. Firing e requires that all rules in a conflict-free set are satisfied. EFT and LFT can be extended to reflect the conflicts, correspondingly. Suppose an event e is enabled by $r_1 = \langle a, e, l_1, u_1, z_1 \rangle$, $r_2 = \langle b, e, l_2, u_2, z_2 \rangle$, and $r_3 = \langle c, e, l_3, u_4, z_4 \rangle$. Also, a and c are in conflict. Therefore, e has two conflict-free sets in its preset: $\texttt{cfs}_1(\bullet e) = \{r_1, r_2\}$ and $\texttt{cfs}_2(\bullet e) = \{r_2, r_3\}$. $\texttt{EFT}(f \leftarrow r_1, r_2, r_3)$ and $\texttt{LFT}(f \leftarrow r_1, r_2, r_3)$ are defined as follows:

$$\texttt{EFT}(y \leftarrow r_1, r_2, r_3) = \begin{cases} max(t_{r_1} + l_1,\ t_{r_2} + l_2) & \text{if } a \text{ and } b \text{ fire} \\ max(t_{r_2} + l_2,\ t_{r_3} + l_3) & \text{if } b \text{ and } c \text{ fire} \end{cases}$$

$$\texttt{LFT}(y \leftarrow r_1, r_2, r_3) = \begin{cases} max(t_{r_1} + u_1,\ t_{r_2} + u_2) & \text{if } a \text{ and } b \text{ fire} \\ max(t_{r_2} + u_2,\ t_{r_3} + u_3) & \text{if } b \text{ and } c \text{ fire} \end{cases}$$

where t_{r_1}, t_{r_2}, and t_{r_3} are the time of when r_1, r_2, and r_3 become satisfied.

A state S of a TEL structure N is a three-tuple $\langle s, M, \texttt{timer} \rangle$, where $s = \{0,1\}^{\Sigma}$ is the state bitvector of logic values of all signals in N, M is the marking, and timer is a function $R \rightarrow Q$. A marking $M \subseteq R$ where all rules $r \in M$ are marked. $S_0 = \langle s_0, M_0, \texttt{timer}_0 \rangle$ is the initial state of N, where the signal values in the *initial bitvector* s_0 are determined in the specification, the *initial marking* M_0 contains all rules which are initially enabled, and $\texttt{timer}_0(r) = 0$ for all $r \in R$. An event e is enabled to fire in a state S if a $\texttt{cfs}(\bullet e) \subseteq M$ and r is satisfied for all $r \in \texttt{cfs}(\bullet e)$. $\texttt{enabled}(S)$ is the set of all events which are enabled to fire in the state S. The states of N change if either time passes or an event fires. In

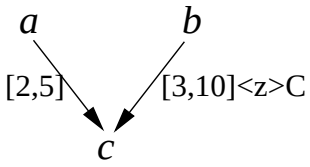

Figure 2.11. Example of a constraint rule

state $S = \langle s, M, \texttt{timer} \rangle$, time τ can pass if for each $e \in \texttt{enabled}(S)$, there is at least one $r \in \bullet e$ such that $\texttt{timer}(r) + \tau \leq \texttt{upper}(r)$. The state resulting from passing time τ in S is $S' = \langle s', M', \texttt{timer}' \rangle$, where

1. $s' = s$,
2. $\texttt{timer}'(r) = \texttt{timer}(r) + \tau$ for all $r \in M'$.

If $e \in \texttt{enabled}(S)$ where $S = \langle s, M, \texttt{timer} \rangle$, firing e leads the system to the next state. After firing e, the state changes to $S' = \langle s', M', \texttt{timer}' \rangle$, where

1. $s' = s$ after the corresponding signal value in s is changed,
2. $M' = (M - \bullet e) \cup e\bullet$, and
3. $\texttt{timer}'(r) = 0$ for all $r \in e\bullet$.

A TEL structure, N, is *safe* if every rule in N has at most one token in any reachable marking. In a marking M, if a rule r has a token, we say that $r \in M$. If r does not have a token, $r \notin M$. The safeness of N can be expressed as follows:

$$(M - \bullet e) \cap e\bullet = \emptyset$$

A TEL structure, N, is *live* if from every reachable marking, there exists a sequence of events such that any event can fire.

Timing properties of a system are specified using a set of *constraint* rules: $C \subseteq E \times E \times \mathcal{N} \times (\mathcal{N} \cup \{\infty\}) \times (b : \{0,1\}^N \rightarrow \{0,1\})$. These constraint rules are similar to the constraint places described by Rokicki in [69]. Constraint rules never actually enable an event to fire. Instead, the constraint rules are checked each time an event fires in a state. Failures caused by constraint rules arise due to following three conditions:

1. There exists a constraint rule $r \in \bullet e$ such that $r \notin M$ when firing e.
2. $\texttt{timer}(r)$ is not satisfied for any constraint rule $r \in \bullet e$ when firing e.
3. $\texttt{timer}(r)$ is expired for any constraint rule $r \in \bullet e$ before firing e.

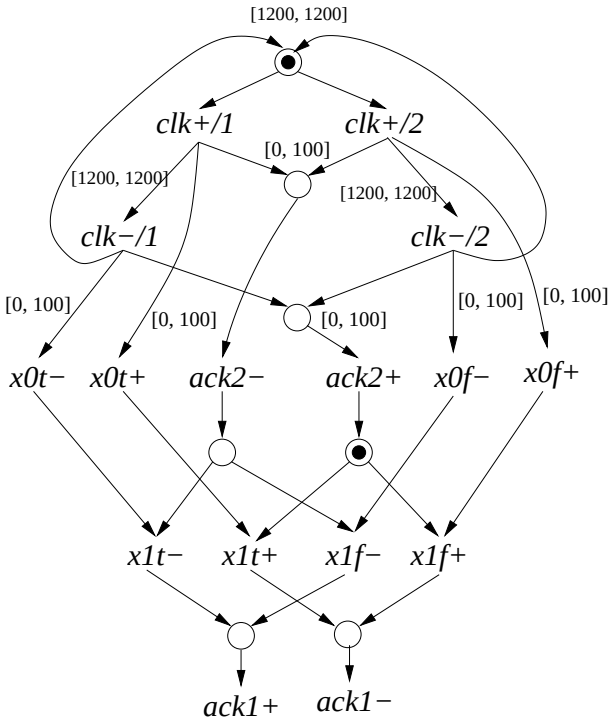

Figure 2.12. The TEL structure of a single stage STARI.

Figure 2.11 shows a TEL structure fragment which contains a constraint rule. This rule requires that the TEL structure must meet a number of requirements. The first requirement is that c must fire no more than 10 time units after the rule $\langle b, c, 3, 10, z\rangle$ becomes enabled. If c can ever fire later than this, the age of the constraint rule exceeds its upper bound and causes a failure. The next requirement is that b must fire at least 3 time units before c fires, and the level z must be high at least 3 time units before c fires. These conditions are necessary in order for the constraint rule to be satisfied when c fires. If the constraint rule is disabling, then the rule would also require that z remains high from the time it rises to the time that c fires. This single constraint rule specifies a rather complex set of requirements. Constraint rules, especially when combined with the ability to specify sequencing events, provide a reasonably powerful way in which to describe the behavior to be verified.

2.3 Translating Timed Specifications to TEL Structures

This section introduces the basic concepts of translating a timed specification to a TEL structure. The details of the translation procedure can be found in [86].

During translation, a timed specification in HSE is decomposed to actions. A TEL

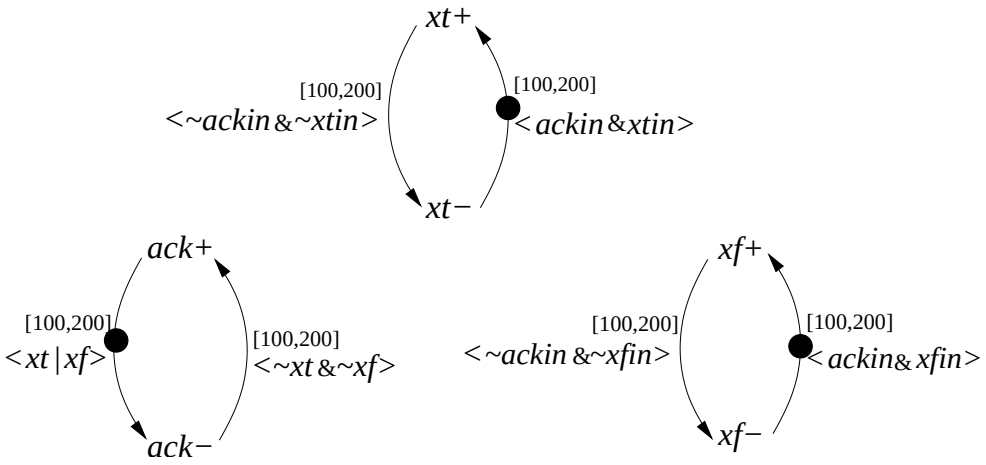

Figure 2.13. The TEL structure of a gate-level single stage STARI.

structure is created for each action. The TEL structures for the actions are composed in parallel, in conflict, or in sequence depending on whether they are are executed in parallel, in conflict, or in sequence. For the actions in the boolean expression of a guard, their TEL structures are composed in parallel if these actions are composed conjunctively in the expression, or their TEL structures are composed in conflict if the actions are composed disjunctively in the expression. For a guarded command, the TEL structures of the guard and the command are composed in sequence in that the command is executed after the actions in the guard have occurred. If a process is repetitive, the TEL structures for the actions executed last in the process are composed with the TEL structures for the actions executed first in the process in sequence to describe the repetitive feature. Since all processes in a module operate in parallel, their TEL structures are composed in parallel. If a module is instantiated in another module through a component statement, all signals in the instantiated module are renamed to the actual signals in the association list, then the renamed TEL structure for the called module is composed in parallel with the TEL structures for processes and other component statements in the calling module. For a constraint statement, it is directly translated to a constraint rule as defined in the last section. A gate is also directly translated to corresponding rules. Figure 2.12 shows the fragment of the TEL structure translated from the THSE code for the 2-stage STARI shown in the first section of this chapter. This fragment of TEL structure corresponds to a single empty stage of the STARI shown in Figure 2.2 that is instantiated in the 2-stage STARI. Figure 2.13 shows the TEL structure for the single empty stage STARI in the gate-level THSE shown in Figure 2.3.

A similar procedure is also applied to translate a timed specification in VHDL to a TEL structure. The specification is decomposed to *assign*s and *guard*s. Each branch of an *if* statement can be regarded as a guarded command in HSE and is equivalently decomposed as a *guard* procedure that interprets the boolean expression following **if** and a set of sequential statements. A TEL structure is created for each *guard* and *assign* procedure, then TEL structures for sequential statements are composed in sequence. For an *if* statement, A TEL structure is created for each branch by creating a TEL for the boolean expression and a TEL for the set of sequential statements in the branch and composing these two TELs in sequence. Then, the TELs for all branches are composed in conflict to reflect that only one of them can be chosen. If an *if* statement has an **else** clause, the boolean expression of the clause is implicitly the negation of the conjunctive composition of all boolean expressions in the leading **if** clauses. The TEL for the **else** can be created similarly. For a *while* statement

$$\textbf{while}\ \ b\ \ \textbf{loop}\ \ CMD\ \ \textbf{endloop};$$

where b is the boolean expression and CMD is a set of sequential statements. This *while* statement can be regarded as the following guarded command in THSE:

$$[\neg b\ \rightarrow\ \textbf{skip}\ \mid\ b\ \rightarrow\ CMD;\ *];$$

so the TEL for a *while* statement can be created in the same as for the above guarded command in THSE. Then, the TELs for all processes are composed in parallel. If the architecture body contains a component instantiation statement, the signals in the TEL for the component are renamed to the actual signals in the association list, then the renamed TEL structure for the component is composed with the TEL structures for processes and other component instantiation statements in the architecture body in parallel.

2.4 Timed Trace theory

The semantic behavior for TEL structures is defined using *timed trace theory* [83]. This section provides a brief overview of timed trace theory which provides the necessary mathematics for the proofs in the later chapters. Trace theory was first applied to the verification of speed-independent circuits by Dill [35]. Later, timing was added so that trace theory can be applied to the verification of timed circuits [25, 83].

A *timed trace*, x, for a circuit is a finite or infinite sequence of timed events (i.e., $x = e_0 e_1 \ldots$). Each timed event is of the form $e_i = (w_i, t_i)$ where w is a wire name in the circuit, which represents a logic value change on that wire, and t is a rational number indicating when that change happens. A timed trace must also satisfy the following two properties:

- *Monotonicity*: $t_i \leq t_{i+1}$ for all $i \geq 0$, and
- *Progress*: if x is infinite then for any time t there exists an i such that $t_i > t$.

Monotonicity states that time can only advance forward, and *progress* states that there is no limit on how long time can pass.

The following shows two useful operations on timed traces. Given a trace $x = e_1 e_2 ...$ and a set of signals, D, the function $\mathbf{del}(D)(x)$ is defined recursively as follows:

$$\mathbf{del}(D)(x) = e_1 y \qquad \text{if } w_1 \notin D$$
$$\mathbf{del}(D)(x) = y \qquad \text{if } w_1 \in D$$

where $y = \mathbf{del}(D)(e_2 e_3 ...)$ and $e_1 = (w_1, t_1)$. This function deletes all events of a trace whose wire names are in D. For example, given a trace $t = abcdbdac$, $\mathbf{del}(\{a, c\})(t) = bdbd$. It is extended naturally to sets of traces. Given a set of traces, X, the function inverse delete $\mathbf{del}^{-1}(D)(X)$ is the set $\{x' \mid \mathbf{del}(D)(x') \in X\}$. This function returns the set of traces which would be in X if all events with wire names in D are deleted. Intuitively, if x is a trace not containing symbols from D, $\mathbf{del}^{-1}(D)(x)$ is the set of all traces that can be generated by inserting events in D at any time into x. Some useful properties of these two functions are listed below:

$$\mathbf{del}(D)(X) = \emptyset \Leftrightarrow X = \emptyset \tag{2.1}$$

$$\mathbf{del}(D)(\mathbf{del}^{-1}(D')(X)) = \mathbf{del}^{-1}(D')(\mathbf{del}(D)(X)) \quad \text{when } D \cap D' = \emptyset \tag{2.2}$$

$$\mathbf{del}(D)(\mathbf{del}^{-1}(D)(X)) = X \tag{2.3}$$

$$\mathbf{del}(D)(X \cap X') \subseteq \mathbf{del}(D)(X) \cap \mathbf{del}(D)(X') \tag{2.4}$$

A *prefix-closed trace structure* T is a four-tuple $\langle I, O, S, F \rangle$. I is a set of input wires, and O is a set of output wires where $I \cap O = \emptyset$. $A = I \cup O$ is the *alphabet* of the structure. S is the *success set* which contains all successful traces of a system. F is the *failure set*

which contains all failure traces of a system. $P = S \cup F$ is the set of all possible traces of a system. The set S is prefix-closed, that means if a trace t is a success, all prefixes of t are also successes. A trace structure must be *receptive*, meaning that $PI \subseteq P$. Intuitively, this means a circuit cannot prevent the environment from sending an input.

Composition ($\|$) combines two circuits into a single circuit. Composition of two trace structures $T = \langle I, O, S, F \rangle$ and $T' = \langle I', O', S', F' \rangle$ is defined when $O \cap O' = \emptyset$. To compose two trace structures, the alphabets of both trace structures must first be made the same by adding new inputs as necessary to each structure. Inverse delete is extended to trace structures for this step as follows:

$$\mathbf{del}^{-1}(D)(T) = \langle I \cup D, O, \mathbf{del}^{-1}(D)(S), \mathbf{del}^{-1}(D)(F) \rangle \tag{2.5}$$

This is defined only when $D \cap A = \emptyset$.

After the two alphabets of the two structures are made to match, we need to find the traces that are consistent with the two structures. The intersection of these two trace structures is defined as follows:

$$T \cap T' = \langle I \cap I', O \cup O', S \cap S', (F \cap P') \cup (P \cap F') \rangle \tag{2.6}$$

This is defined only when $A = A'$ and $O \cap O' = \emptyset$. From this definition, a success trace in the composite must be a success trace in both components. A failure trace in the composite is a possible trace that is a failure trace in either component. The possible traces for the composite are $P \cap P'$.

Composition can now be defined as follows:

$$T \parallel T' = \mathbf{del}^{-1}(A' - A)(T) \cap \mathbf{del}^{-1}(A - A')(T') \tag{2.7}$$

In the following chapters, $\{T, T'\}$ is also used to indicate the composition of T and T'.

Another useful operation is **hide** which is used to make some wires *internal* to the circuit so that they can no longer be connected to other wires. Formally, given a trace structure T, **hide**(D)(T) is defined when $D \subseteq O$. $\mathbf{hide}(D)(T)$ is defined as follow:

$$\mathbf{hide}(D)(T) = \langle I, O - D, \mathbf{del}(D)(S), \mathbf{del}(D)(F) \rangle \tag{2.8}$$

where D is the set of wires to be hidden. Figure 2.14 shows a circuit with two components, and signals c and d connect the two components and are also outputs. Hiding signals efficiently and correctly is the key goal of abstraction. Figure 2.15 shows the result of

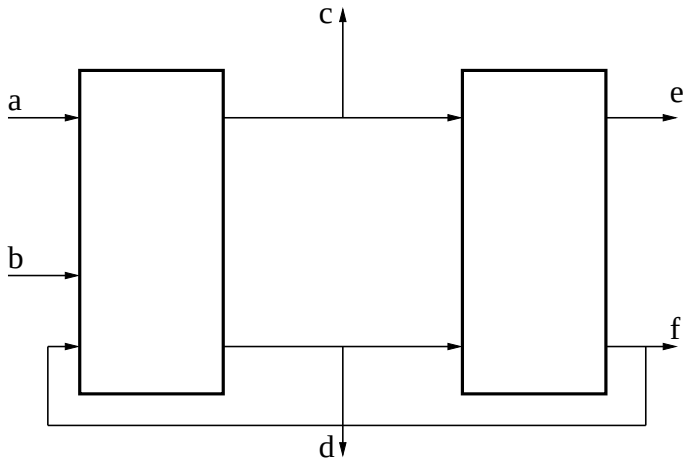

Figure 2.14. Block diagram of a circuit with two components.

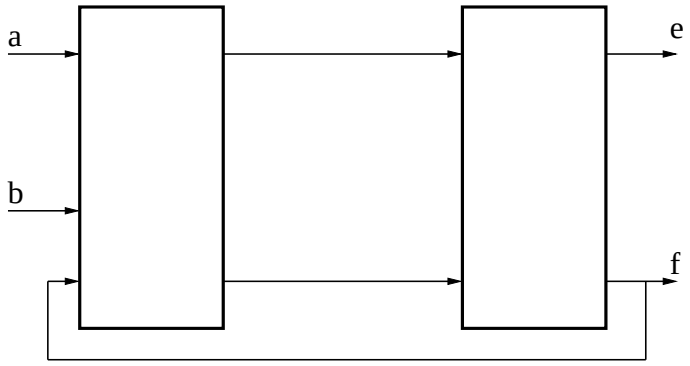

Figure 2.15. Signals c and d are hidden.

hiding the signals c and d in Figure 2.14. Hiding signals efficiently and correctly is the key goal of abstraction.

A trace structure is *failure-free* if its failure set is empty. Given two trace structures, T and T', we say T *conforms* to T' (denoted $T \preceq T'$) if $I = I', O = O'$, and for *all* environments E, if $E \parallel T'$ is failure-free, so is $E \parallel T$. Intuitively, if a system using T' cannot fail, neither can a system using T.

The following lemma gives a simple sufficient condition to determine conformance between two trace structures.

Lemma 2.4.1 *$T \preceq T'$ if $I = I', O = O', F \subseteq F'$, and $P \subseteq P'$.*

The condition $F \subseteq F'$ assures that if the environment does not cause a failure in T', it does not cause a failure in T. The condition $P \subseteq P'$ assures that if T' does not cause a

failure in the environment, T does not cause one.

The next lemma shows that if T conforms to T', this conformance is maintained in any environment.

Lemma 2.4.2 *If $T \preceq T'$ and T'' is any trace structure, then $T \parallel T'' \preceq T' \parallel T''$.*

Proofs of these lemmas can be found in [35].

The following example is a *C-element* to illustrate how the trace structure models a circuit behavior. A C-element is very useful in asynchronous designs. It is typically used to signal the completion of several concurrent computations. The output value of a C-element remains constant until *all* of its inputs are equal to the complement of the output; the output then changes to its complement after some delay. Figure 2.16 shows a two-input C-element and the state graph which can accept the S and F sets of the corresponding trace structure.

The states of the state graph are all possible logical valuations for all signals in a circuit and an additional state called a *failure state*. A *stable state* is a state in which the value of the boolean function of the circuit is the same as the actual value of the output wire. An *unstable* state is a state in which they are not the same. In the state graph of the C-element, The initial values of the wires are either $abc = 000$ or $abc = 111$. State 1, 2, and 3 are stable states, and state 4 is an unstable state.

A *hazard* occurs if the value of a boolean function changes and then reverts to its original value without waiting for the actual output of the gate to change. In a C-element, changes in the inputs are restricted so that when all inputs are equal to the complement of the output, they must remain constant until the output changes. Input transitions that violate this restriction may cause hazards on an output, and lead the circuit to the failure state, F.

2.5 From a TEL to a Trace Structure

Both TEL structures and trace theory can be used to model the circuit behavior. This section gives a brief description of how to derive the corresponding trace structure from a TEL structure.

After a system is modeled by a TEL structure, the state space of the system can be found by exhaustively firing all events in the system, and reachability analysis is used to study the behavior of the system. Firing an event leads the system to another state. Firing a sequence of events results in a sequence of states. A state S_j is said to be *reachable* from

another state S_i if there exists a sequence of event firings that changes S_i to S_j. A *firing sequence* or a *run* can be expressed as $\rho = S_0 \stackrel{e_1}{\rightarrow} S_1 \stackrel{e_2}{\rightarrow} S_2 \stackrel{e_3}{\rightarrow} \ldots \stackrel{e_n}{\rightarrow} S_n$, where S_0 is the initial state, and S_{i+1} is obtained from S_i by passing some time and then firing e_{i+1}. If an event e_{n+1} fires at the end of a firing sequence ρ, a new firing sequence $\rho' = \rho \stackrel{e_{n+1}}{\longrightarrow} S_{n+1}$. Therefore, a firing sequence is *prefix-closed*. Let $\texttt{time}_i(\rho)$ be the sum of time passed when the system reaches the state S_i from the initial state S_0 through the firing sequence ρ. It is true that $\texttt{time}_0(\rho) = 0$ and $\texttt{time}_{i+1}(\rho) = \texttt{time}_i(\rho) + \tau$ where $t_{min} \leq \tau \leq t_{max}$, where

$$\begin{aligned} t_{\texttt{cfs}_i} &= max(\{\texttt{lower}(r) \mid r \in M_i \text{ and } r \in \texttt{cfs}_i(\bullet e_{i+1})\}) \\ t_{min} &= min(\{t_{\texttt{cfs}_i} \mid \text{for all } \texttt{cfs}_i(\bullet e_{i+1}) \subseteq \bullet e_{i+1}\}) \\ t_{max} &= max(\{\texttt{upper}(r) \mid r \in M_i \text{ for } r \in \bullet e_{i+1}\}) \end{aligned}$$

Therefore, a run ρ produces a timed trace

$$(t_1, \texttt{time}_1(\rho))\ (t_2, \texttt{time}_2(\rho))\ \cdots$$

Since a timed system produces the timed traces by reachability analysis, the behavior of a timed system can also be studied using timed trace theory. A function **trace**(N) is defined to return a trace structure which contains the set of all possible timed traces produced by a TEL structure N. This function uniquely connects a TEL structure and its corresponding timed trace structure together. The following is the definition of **trace**(N).

Definition 2.5.1 *Function* **trace**(N) *takes a TEL structure* N*, and returns a trace structure* $T = \langle I, O, P \rangle$ *where*

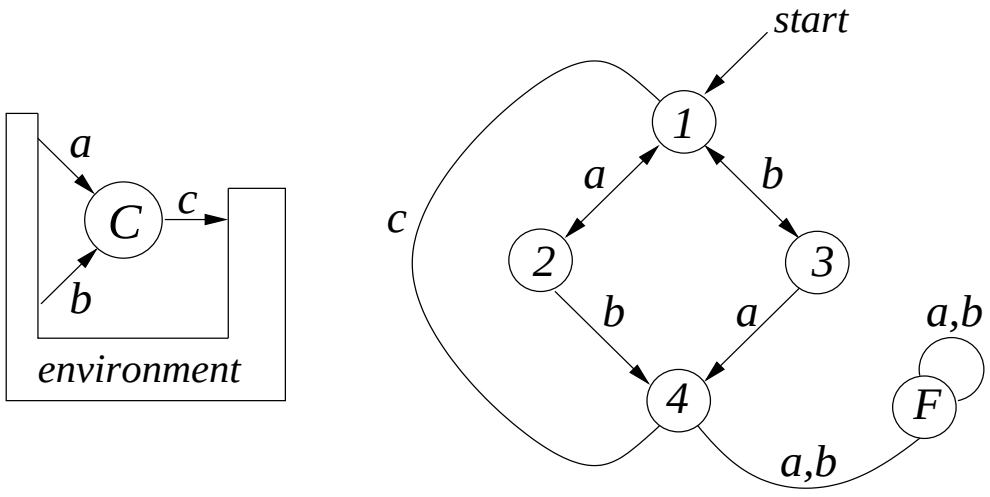

Figure 2.16. A C-element and its state graph.

1. *I is the set of input signals in N;*
2. *O is the set of output and internal signals in N;*
3. *P is the set of all possible timed traces produced by N.*

CHAPTER 3

MODULAR SYNTHESIS AND VERIFICATION

The purpose of synthesis is to generate the correct circuit implementation from a given specification, and that of verification is to determine whether the given circuit correctly implements its specification. Therefore, the obvious question is how to define correctness. In general, the correctness of a circuit can be derived in two ways. First, we can check properties of the circuit, and if all the properties are satisfied, the circuit is claimed to be correct. Second, the correctness of the circuit can be defined relative to its specification. If the specification is guaranteed to be correct, then the circuit is also correct. Again, the correctness of the specification is determined by whether certain properties are satisfied. Now, the question is what properties should be checked to assert the correctness. Properties can be classified as either *safety* or *liveness* properties. In general, a safety property is a condition on finite computations while a liveness property is a condition on the indefinite future. Liveness properties cannot be verified easily. Many methods either do not verify liveness properties, or do so in a limited way. In timed circuits, all timing constraints also need to be checked. In `ATACS`, general liveness properties are not checked. Instead, circuits are checked if they can deadlock. Timing is another important issue in the synthesis and verification of timed circuits design. Even when timing information is designed in from the beginning, it is necessary to verify that the physical implementation meets the requirements of the specification. These properties and timing constraints are checked during timed state space exploration to guarantee that the synthesized circuit or the circuit to be verified is correct. In this way, verification in `ATACS` is similar to model checking. This chapter describes the properties and the timing constraints to satisfy the correctness requirement. The safe transformations which preserve these properties are described next. In the last two sections, the theorems for modular synthesis and verification are described. These theorems are proved using trace theory.

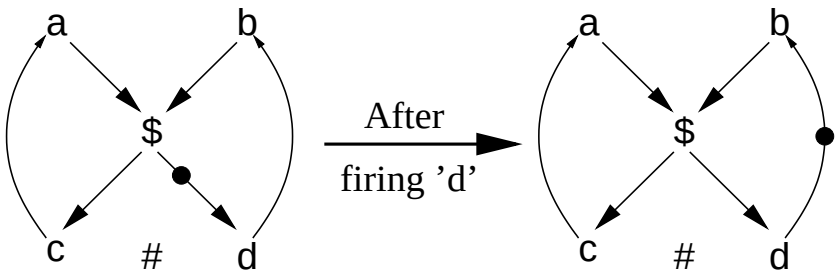

Figure 3.1. An example of deadlock.

3.1 Definition of Correctness

This section defines the correctness of circuits. Informally, a correct circuit keeps doing things as expected. There are two conditions for a correct circuits: it must produce the expected outputs for the given inputs; and the outputs must be produced in finite delay after the inputs are given. The first condition describes a safety property and the second one describes a liveness property. The expected outputs are determined by the function of the circuit and the inputs. Also, the output signals cannot contain glitches that are unexpected pulses on signal wires, and can cause the circuit either to malfunction or fail. A circuit that may produce glitches on its output wires is hazardous. A correct asynchronous circuit must be hazard-free. A correct circuit must also produce outputs in a finite delay. It is obvious that a circuit is useless if it does not produce any outputs. For timed circuits, we also impose timing properties on their correctness. For example, we want a circuit to produce an output event $b+$ between 5 and 10 time units after an input event $a+$ has occurred. The circuit is not correct if it produces a $b+$ beyond that timing bound after $a+$.

In `ATACS`, the behavior of circuits is modeled by TEL structures. The first requirement is that TEL structures must be safe. Also, circuits must be hazard-free. In Chapter 2, it is described that a hazard appears when a signal is enabled by a disabling rule, but the enabling condition is lost before the signal transition fully completes. This can result in either the signal transition not being produced, or a glitch. Any trace that leads to this situation can cause the circuit to fail. General liveness properties cannot be modeled in `ATACS`. Instead, correct circuits must not deadlock, a limited liveness property. Deadlock is the state where the circuit can halt for an infinite period of time and no outputs can

ever be produced. Deadlock occurs when firing a signal, for example a, depends on some other signal events which then depend on the firing of a. This situation is illustrated in Figure 3.1. Another very illustrative and interesting example of deadlock can be found in a law passed by the Kansas legislature early in this century. It said, in part, "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone." During state space exploration, a state causing deadlock is found if the enabled rules in the current state are not enough to fire an event. In other words, the enabled event set of a deadlock state is empty. Timing properties of timed circuits are defined by constraint rules as described in Chapter 2 and must be satisfied. Traces causing any constraint rule violations are failures. In summary, the above conditions that cause failures are shown in the following definition.

Definition 3.1.1 *Suppose σ is a success trace that leads S_0 of a TEL structure N to S_n. Firing $e \in E$ leads S_n to S_{n+1}. σe is a failure trace if one of the following conditions hold:*

1. $(M_n - {\bullet}e) \cap e{\bullet} \neq \emptyset$.
2. *firing e causes a disabling rule $r \in M_n$ to be disabled.*
3. $\texttt{enabled}(S_{n+1}) = \emptyset$.
4. *There exists a constraint rule $r_c \in {\bullet}e$ such that $r_c \notin M_n$ when firing e.*
5. $\texttt{timer}(r_c)$ *is not satisfied for any constraint rule $r_c \in {\bullet}e$ when firing e.*
6. $\texttt{timer}(r_c)$ *is expired for any constraint rule $r_c \in {\bullet}e$ before firing e.*

The first condition asserts that the TEL structure must be safe. The second condition asserts that firing an event must not cause an enabled disabling rule to be disabled. The third condition defines deadlock. The last three conditions asserts that firing an event causes a failure if any timing requirement is violated.

These conditions completely describe the failure set F of a trace structure. During state space exploration and timing analysis, all these conditions are checked, and a failure trace is generated whenever a firing satisfies one of the above conditions. The procedure that performes these checks is called **fail**. For example, given a TEL structure T, P_T is the set of all possible traces produced by T, $\textbf{fail}(P_T)$ returns a subset of P_T where the traces are failures. $F = \textbf{fail}(P_T)$ where F is the failure set of $\textbf{trace}(T)$.

As described in the first chapter, an environment is necessary to define the interface behavior of a circuit in the timed circuit design. It is essential for synthesis and verification to know how the circuit behaves and how it interacts with its environment. The internal details of the environment have no impact on the circuit. For modular synthesis and verification to succeed, the definition of $\mathbf{fail}(P)$ must satisfy two requirements. The first requirement asserts that during synthesis and verification of timed circuits, only the behavior of the circuit is checked, and the internal behavior of the environment which is not visible to the synthesis and verification is not checked. In other words, internal failures of the environment do not cause the circuit to fail. The second requirement asserts that if the relation between the possible trace sets of two trace structures is inclusive, this relation is preserved between their corresponding failure sets returned by **fail**. These two properties are given formally below. Suppose C is the trace structure defining the behavior of a circuit, E is the trace structure of the environment describing the input behavior for C, and X_E is the set of internal signals of E. In the following equations, P_1 is the possible trace set of $\{C, E\}$ and P_2 is the possible trace set of $\{C, \mathbf{hide}(X_E)(E)\}$.

$$\mathbf{del}(X_E)(\mathbf{fail}(P_1)) = \mathbf{fail}(P_2) \tag{3.1}$$

$$\mathbf{fail}(P_1) \subseteq \mathbf{fail}(P_2) \quad \text{if} \quad P_1 \subseteq P_2 \tag{3.2}$$

3.2 Definitions of Safe Transformations

In the design of a timed system, an environment must be provided. An environment has two functions. First, it defines and supplies the input behavior which the system must be able to process for correct operation. Second, the outputs of the system must not cause the environment to fail. In other words, a correct system operating in the specified environment does not cause any failure. Described in trace theory, $\mathbf{fail}(P_{\{C,E\}}) = \emptyset$ where C is a correct circuit and E is the specified environment described as trace structures. If the system operates in another environment, E', that produces a superset of timed traces of E, and if $\mathbf{fail}(P_{\{C,E'\}}) = \emptyset$, it is true that $\mathbf{fail}(P_{\{C,E\}}) = \emptyset$. This is a direct result of Equation 3.2 in the last section. This result is very useful in that if a given environment, E, is complex, it can be transformed to a simpler one, E', as long as it produces a superset of timed traces. For synthesis, the resulting circuit may contain a redundant part to deal with the extra behavior introduced by transformations, but the synthesized circuit definitely works correctly in the originally specified environment that supplies a subset of inputs that the synthesized circuit is able to process. For verification,

if a system is verified with E' without any failure, we can assert that the system is also failure-free with E. However, verifying a system with a transformed environment may result in a *false negative* answer. A false negative is that the violating states (i.e. states where an error has been detected) are found when verifying a system with the transformed environment, but these violating states may not be reachable if the system is verified with the originally specified environment. This approach never results in a *false positive* answer where the system is verified failure-free with E' while verification would detect violating states in the system with E. A false positive would happen only if the transformations reduce the behavior of the environment so that the violating states of the system may not be reachable during verification. If false negatives happen rarely, transformations can substantially reduce the complexity of the environment.

In the above discussion, transformations must satisfy one requirement: the environment after transformations must produce a superset of timed traces that are produced by the originally specified environment. Such transformations are defined to be safe. Given an environment, E, the one derived through a series of safe transformations is referred to as the "*abstracted*" environment of E. The following is the definition of safe transformations.

Definition 3.2.1 (Safe Transformations) *A system is described by a TEL structure N. N' is derived by applying a transformation on N. Suppose P_N and $P_{N'}$ are sets of possible timed traces produced by N and N', respectively. If $P_N \subseteq P_{N'}$, the transformation is safe.*

As described in the last section, internal signals of the environment can be removed for synthesis and verification as long as the interface behavior of the environment with the internal signals is preserved. Abstraction converts internal signals of the environment into sequencing events. Then safe transformations are applied to remove sequencing events from TELs under two conditions. First, removal of a sequencing event must not reduce the specified untimed behavior of the environment. Second, the timing information carried by the environment must be preserved in a conservative fashion. This can also be described by trace theory. Suppose N_E is the TEL structure describing the behavior of the environment, and T_E is its corresponding trace structure. The interface behavior of T_E is described by $\mathbf{del}(D)(P_E)$, where D is the set of signals internal to the environment, and P_E is the set of possible timed traces. In the abstracted environment, the internal

signals, D, are removed from N_E to obtain the trace structure $T_A = \mathbf{trace}(\mathbf{abs}(D)(N_E))$ where the function $\mathbf{abs}(D)(N_E)$ returns a TEL N'_E where the signals in D are abstracted away from N_E using safe transformations. Let X_1 and X_2 be the untimed trace sets produced by $\mathbf{abs}(D)(N_E)$ and N_E, respectively. To preserve the interface behavior, a safe transformation must satisfy that $\mathbf{del}(D)(X_2) \subseteq X_1$. Since timing information is preserved conservatively, it is true that $\mathbf{del}(D)(P_E) \subseteq P_A$, where D contains the internal signals of the environment to be removed and P_A is the possible trace set of T_A.

Calculating the interface behavior of an environment can be conducted by state space exploration to generate all possible timed traces and then applying function **del** to remove all internal signals from the possible timed traces. However, state space exploration is an exponential problem, which is computationally unfeasible for large systems. Instead, internal signals are removed from TEL structures using safe transformations before state space exploration, and the possible timed traces produced by the abstracted TEL structure include the specified interface behavior. The following lemma proves that the unabstracted environment conforms to the abstracted environment if only the interface behavior is considered. This conformation is used to prove the theorems of modular synthesis and verification in the next section.

Lemma 3.2.1 *A system is described by a TEL structure, N, its corresponding trace structure is T, O is the set of all output signals in N, and $D \subseteq O$. If the function* $\mathbf{abs}(D)(N)$ *uses only safe transformations, then* $\mathbf{hide}(D)(T) \preceq \mathbf{trace}(\mathbf{abs}(D)(N))$.

Proof: Let P and P' be the possible trace sets of $\mathbf{hide}(D)(T)$ and $\mathbf{trace}(\mathbf{abs}(D)(N))$, respectively. From the definition of safe transformations, we have $P \subseteq P'$. From property 3.2, we have $\mathbf{fail}(P) \subseteq \mathbf{fail}(P')$. Therefore, from Lemma 2.4.1, we have $\mathbf{hide}(D)(T) \preceq \mathbf{trace}(\mathbf{abs}(D)(N))$. □

3.3 Modular Synthesis and Verification

Given a specification, we can design the entire flat circuit at one time. This approach only works well for small circuits. When the circuit gets large and complex, divide-and-conquer approach is necessary. By partitioning a circuit into blocks, each with constrained complexity, the design process can finish much faster by designing each block individually for all blocks. The improvement in the speed of the design process comes from that all irrelevant details to the block being designed is removed and the total information under

consideration during the design process is substantially reduced. When a block is chosen to be verified or synthesized, the rest of the circuit becomes its environment. As described in Section 2, internal signals of the environment belong to those irrelevant details and need to be removed to simplify the design problem. In the following two sections, a series of theorems is formulated and proved that modular synthesis and verification are correct.

3.3.1 Modular Synthesis Theorems

Before giving the theorem for modular synthesis, it is necessary to define correct synthesis. In `ATACS`, the behavior of a circuit is defined in a specification with an environment to specify the operating environment of the circuit. A state graph is generated by exploring the timed state space of the circuit. If no error is found, an implementation is derived from the state graph. The synthesized circuit operates correctly in the specified environment. The correctly synthesized circuit consists of two parts of behavior. First, the circuit implements a partial set of behavior defined in the specification. Second, extra behavior may be introduced during synthesis to simplify the circuit impllementation. The extra behavior is produced for the inputs not defined in the environment. Since the inputs for the extra behavior are not defined in the environment, the circuit does produce the extra behavior when operating in the environment. The definition of correct synthesis is given as follows.

Definition 3.3.1 (Correct Synthesis) *B and E are the circuit and its environment specifications, respectively. If C is the circuit implementation correctly synthesized from $\{B, E\}$, it is true that*

$$\mathbf{fail}(P_{\{C,E\}}) = \emptyset \tag{3.3}$$

$$P_{\{C,E\}} \subseteq P_{\{B,E\}} \tag{3.4}$$

As described above, the internal signals of an environment can be abstracted away. The resulting environment has the same or more interface behavior. Hence, the circuit working in the environment after abstraction needs to be able to accept the newly introduced behavior. It is obvious that the synthesized circuit still operates correctly given a subset of the input that it can handle. The following lemma asserts that a circuit synthesized from a specification with the abstracted environment still operates correctly in the unabstracted environment.

Lemma 3.3.1 *B and E are the circuit and environment specifications, respectively. X_E is the set of internal signals of E. C' is the circuit implementation synthesized correctly from $T = \{B, \mathbf{abs}(X_E)(E)\}$. Let $T' = \{C', E\}$, it is true that $\mathbf{fail}(P_{T'}) = \emptyset$.*

Proof: First, let P_1 be the set of possible traces of $\{C', \mathbf{abs}(X_E)(E)\}$, and P_2 be the set of possible traces of $\{C', \mathbf{hide}(X_E)(E)\}$. Since C' is correctly synthesized from $\{B, \mathbf{abs}(X_E)(E)\}$, therefore

$$\mathbf{fail}(P_1) = \emptyset \tag{3.5}$$

From Lemma 3.2.1 and 2.4.2, we have

$$\{C', \mathbf{hide}(X_E)(E)\} \preceq \{C', \mathbf{abs}(X_E)(E)\} \tag{3.6}$$

Therefore,

$$\mathbf{fail}(P_2) = \emptyset \tag{3.7}$$

From Property 3.1, we have

$$\mathbf{del}(X_E)(\mathbf{fail}(P_{T'}) = \emptyset \tag{3.8}$$

And from Property 2.1, we have

$$\mathbf{fail}(P_{T'}) = \emptyset \tag{3.9}$$

□

An alternative to flat synthesis is to synthesize the circuit block by block. When a block is chosen to synthesize, the rest of the circuit is treated as its environment. When the results of all components are available, they are integrated together to determine the solution to the whole design. This idea is formulated in the following lemma.

Lemma 3.3.2 *A system $\{B_1, B_2\}$ has two components: B_1 and B_2. C_1 and C_2 are the circuits correctly synthesized from B_1 and B_2 with B_2 and B_1 as the corresponding environment. It is true that $\mathbf{fail}(P_{\{C_1, C_2\}}) = \emptyset$.*

Proof: Since C_1 is the circuit correctly synthesized from B_1 with B_2 as its environment, its behavior can be expressed as follows:

$$P_{C_1} = P_1 \cup P_1' \tag{3.10}$$

and $P_1 \subseteq P_{B_1}$ and $P_1' \cap P_{B_2} = \emptyset$ where P_{B_1} and P_{B_2} are the sets of possible traces of B_1 and B_2, respectively. Similarly, the behavior of C_2 can be expressed as follows:

$$P_{C_2} = P_2 \cup P_2' \tag{3.11}$$

and $P_2 \subseteq P_{B_2}$ and $P_2' \cap P_{B_1} = \emptyset$. The possible traces of $\{C_1, C_2\}$ is

$$P_{\{C_1,C_2\}} = P_{C_1} \cap P_{C_2} \tag{3.12}$$

Substitute Equation 3.10 and Equation 3.11 into Equation 3.12 and we have

$$P_{\{C_1,C_2\}} = (P_1 \cap P_2) \cup (P_1 \cap P_2') \cup (P_1' \cap P_2) \cup (P_1' \cap P_2') \tag{3.13}$$

Since

$$\begin{aligned} \mathbf{fail}(P_{\{C_1,B_2\}}) &= \emptyset \\ P_{\{C_1,B_2\}} &= (P_1 \cap P_{B_2}) \cup (P_1' \cap P_{B_2}) \end{aligned}$$

we have

$$\mathbf{fail}(P_1 \cap P_{B_2}) = \emptyset$$

Since $P_2 \subseteq P_{B_2}$, $\mathbf{fail}(P_1 \cap P_2) = \emptyset$. And also since $P_1 \subseteq P_{B_1}$, $P_1' \cap P_{B_2} = \emptyset$, and $P_{B_1} \cap P_2' = \emptyset$, we have

$$P_1 \cap P_2' = \emptyset \quad \text{and} \quad P_1' \cap P_2 = \emptyset$$

Therefore,

$$\mathbf{fail}(P_{\{C_1,C_2\}}) = \mathbf{fail}(P_1' \cap P_2')$$

P_1' and P_2' are the traces produced by C_1 and C_2 for the inputs not defined in their corresponding environments. The inputs to C_1 are the outputs from C_2, and the inputs to C_2 are the outputs from C_1. C_1 cannot produce P_1' until C_2 produces outputs defined by P_2', and the same for C_2. Sine both C_1 and C_2 must wait for each other mutually to produce a trace in P_1' or P_2', C_1 and C_2 can never produce P_1' and P_2'. Therefore, $P_1' \cap P_2' = \emptyset$, and from Property 2.1, we have

$$\mathbf{fail}(P_{\{C_1,C_2\}}) = \emptyset$$

□

Combining the above two theorems, we can derive an important theorem of modular synthesis. This theorem asserts that each block is synthesized with its corresponding environment of which internal signals are abstracted away, the integration of the results for all blocks is still the correct solution for the whole design. The theorem is shown as follows:

Theorem 3.3.1 *A system* $\{B_1, B_2\}$ *has two components:* B_1 *and* B_2. X_{B_1} *and* X_{B_2} *are sets of internal signals of* B_1 *and* B_2, *respectively.* C_1 *and* C_2 *are the circuits correctly synthesized from* $\{B_1$,$\mathbf{abs}(X_{B_2})(B_2)\}$ *and* $\{\mathbf{abs}(X_{B_1})(B_1)$,$B_2\}$. *It is true that* $\mathbf{fail}(P_{\{C_1,C_2\}}) = \emptyset$.

Proof: Since C_1 is the circuit correctly synthesized from $\{B_1, \mathbf{abs}(X_{B_2})(B_2)\}$, we have

$$\mathbf{fail}(P_{\{C_1,\mathbf{abs}(X_{B_2})(B_2)\}}) = \emptyset \tag{3.14}$$

From Lemma 3.3.1, we have

$$\mathbf{fail}(P_{\{C_1,B_2\}}) = \emptyset \tag{3.15}$$

Similarly, we have

$$\mathbf{fail}(P_{\{B_1,C_2\}}) = \emptyset \tag{3.16}$$

From the above two equations, C_1 and C_2 can also be thought of as circuits correctly synthesized from B_1 and B_2 with B_2 and B_1 as the corresponding environment. Therefore, according to Lemma 3.3.2 , we have

$$\mathbf{fail}(P_{\{C_1,C_2\}}) = \emptyset \tag{3.17}$$

□

This lemma is extended to a system with a number of components more than 2.

3.3.2 Modular Verification Theorems

Given a circuit M consisting of two blocks $M_1 = \{I_1, O_1, P_1\}$ and $M_2 = \{I_2, O_2, P_2\}$, the composition $M_1 \parallel M_2$ defines the behavior of M. P_1 and P_2 are the possible trace sets of M_1 and M_2, respectively. X_1 and X_2 are the set of internal signals of M_1 and M_2, respectively. The composition of M_1 and M_2 is defined when the following conditions are satisfied.

$$O_1 \cap O_2 = \phi$$
$$X_1 = O_1 - I_2$$
$$X_2 = O_2 - I_1$$
$$X_1 \cap X_2 = \phi$$

To verify M, we can verify M_1 and M_2, separately. If both are correct, M is also correct. When verifying one block, the other one behaves like the environment for the

former one. Therefore, the internal signals of the later one needs to be removed and the result of the verification is not affected. If a different block is chosen, a similar process is applied to its environment. This is formulated in the following theorem.

Theorem 3.3.2 *Let X_1 and X_2 be internal signal sets of M_1 and M_2, respectively. If $M_1 \parallel \mathbf{hide}(X_2)(M_2)$ is failure-free, and $\mathbf{hide}(X_1)(M_1) \parallel M_2$ is failure-free, then $M = M_1 \parallel M_2$ is failure-free.*

Proof: First, the failure set of $M_1 \parallel M_2$ is

$$(\mathbf{del}^{-1}(X_2)(\mathbf{fail}(P_1)) \cap \mathbf{del}^{-1}(X_1)(P_2)) \cup (\mathbf{del}^{-1}(X_1)(\mathbf{fail}(P_2)) \cap \mathbf{del}^{-1}(X_2)(P_1))$$

Suppose $M_1 \parallel \mathbf{hide}(X_2)(M_2)$ is failure-free. That means its failure set

$$(\mathbf{fail}(P_1) \cap \mathbf{del}^{-1}(X_1)(\mathbf{del}(X_2)(P_2))) \cup (P_1 \cap \mathbf{del}^{-1}(X_1)(\mathbf{fail}(\mathbf{del}(X_2)(P_2)))) = \phi \quad (3.18)$$

Therefore,

$$\mathbf{fail}(P_1) \cap \mathbf{del}^{-1}(X_1)(\mathbf{del}(X_2)(P_2)) = \phi \quad (3.19)$$

$$P_1 \cap \mathbf{del}^{-1}(X_1)(\mathbf{fail}(\mathbf{del}(X_2)(P_2))) = \phi \quad (3.20)$$

From Property 2.3, Equation 3.19 can be transformed as follows:

$$\mathbf{del}(X_2)(\mathbf{del}^{-1}(X_2)(\mathbf{fail}(P_1))) \cap \mathbf{del}^{-1}(X_1)(\mathbf{del}(X_2)(P_2)) = \phi$$

From Property 2.2,

$$\mathbf{del}(X_2)(\mathbf{del}^{-1}(X_2)(\mathbf{fail}(P_1))) \cap \mathbf{del}(X_2)(\mathbf{del}^{-1}(X_1)(P_2)) = \phi$$

From Property 2.4,

$$\begin{aligned}&\mathbf{del}(X_2)(\mathbf{del}^{-1}(X_2)(\mathbf{fail}(P_1)) \cap \mathbf{del}^{-1}(X_1)(P_2)) \\ &\quad \subseteq \mathbf{del}(X_2)(\mathbf{del}^{-1}(X_2)(\mathbf{fail}(P_1))) \cap \mathbf{del}(X_2)(\mathbf{del}^{-1}(X_1)(P_2)) = \phi\end{aligned}$$

Therefore,

$$\mathbf{del}(X_2)(\mathbf{del}^{-1}(X_2)(\mathbf{fail}(P_1)) \cap \mathbf{del}^{-1}(X_1)(P_2)) = \phi$$

Finally, from Property 2.1,

$$\mathbf{del}^{-1}(X_2)(\mathbf{fail}(P_1)) \cap \mathbf{del}^{-1}(X_1)(P_2) = \phi \quad (3.21)$$

Similarly, suppose $M_2 \parallel \mathbf{hide}(X_1)(M_1)$ is failure-free. Thus, its failure set

$$\mathbf{fail}(P_2) \cap \mathbf{del}^{-1}(X_2)\mathbf{del}(X_1)(P_1)) \cup (P_2 \cap \mathbf{del}^{-1}(X_2)(\mathbf{fail}(\mathbf{del}(X_1)P_1))) = \phi$$

$$\mathbf{fail}(P_2) \cap \mathbf{del}^{-1}(X_2)\mathbf{del}(X_1)(P_1)) = \phi \tag{3.22}$$

$$(P_2 \cap \mathbf{del}^{-1}(X_2)(\mathbf{fail}(\mathbf{del}(X_1)P_1))) = \phi$$

By applying the same steps above to Equation 3.20, we can derive that

$$\mathbf{del}^{-1}(X_1)(\mathbf{fail}(P_2)) \cap \mathbf{del}^{-1}(X_2)(P_1) = \phi \tag{3.23}$$

The union of Equations 3.21 and 3.23 is the failure set of $M_1 \parallel M_2$. Since both Equation 3.21 and 3.23 are empty, the failure set of $M_1 \parallel M_2$ is empty. □

This theorem is naturally extended to a circuit consisting of more than two blocks.

Calculation of P is an exponential problem. Instead, we can apply abstraction and safe transformations to its corresponding TEL structure to remove internal signals, then the state space is explored to generate the new trace structure. Suppose N is a TEL structure and T is the corresponding trace structure. From Lemma 3.3.1 we know that $\mathbf{hide}(D)(T)$ conforms to $\mathbf{trace}(\mathbf{abs}(D)(N))$. Suppose N_1 and N_2 are the TEL structures for M_1 and M_2, respectively. Therefore, combined with Lemma 2.4.2, we know $M_1 \parallel \mathbf{hide}(X_2)(M_2)$ conforms to $M_1 \parallel \mathbf{trace}(\mathbf{abs}(X_2)(N_2))$ and $\mathbf{hide}(X_1)(M_1) \parallel M_2$ conforms $\mathbf{trace}(\mathbf{abs}(X_1)(N_1)) \parallel M_2$. From above the conclusion, we show another very important theorem.

Theorem 3.3.3 *Let X_1 and X_2 be internal signal sets of M_1 and M_2, respectively. If $M_1 \parallel \mathbf{trace}(\mathbf{abs}(X_2)(N_2))$ is failure-free, and $\mathbf{trace}(\mathbf{abs}(X_1)(N_1)) \parallel M_2$ is failure-free, then $M = M_1 \parallel M_2$ is failure-free.*

Proof: From Lemma 3.2.1, we have

$$\mathbf{hide}(X_1)(M_1) \preceq \mathbf{trace}(\mathbf{abs}(X_1)(N_1))$$

$$\mathbf{hide}(X_2)(M_2) \preceq \mathbf{trace}(\mathbf{abs}(X_2)(N_2))$$

From Lemma 2.4.2, we have

$$\mathbf{hide}(X_1)(M_1) \parallel M_2 \preceq \mathbf{trace}(\mathbf{abs}(X_1)(N_1)) \parallel M_2$$

$$M_1 \parallel \mathbf{hide}(X_2)(M_2) \preceq M_1 \parallel \mathbf{trace}(\mathbf{abs}(X_2)(N_2))$$

Since $\mathbf{trace}(\mathbf{abs}(X_1)(N_1)) \parallel M_2$ and $M_1 \parallel \mathbf{trace}(\mathbf{abs}(X_2)(N_2))$ are failure-free, then $\mathbf{hide}(X_1)(M_1) \parallel M_2$ and $M_1 \parallel \mathbf{hide}(X_2)(M_2)$ are also failure-free. From Theorem 3.3.2, $M_1 \parallel M_2$ is failure-free. $\square$

CHAPTER 4

ABSTRACTION

Synthesis and verification of timed systems are typically based on a complete state space exploration. The state space can be derived by exhaustively firing all possible transition sequences in the system. The number of states grows exponentially as the complexity of the design grows in terms of the number of signals in the design. Therefore, synthesis and verification of large and complex systems is difficult or even impossible because of state explosion. In order to constrain the computational complexity, it is necessary to suppress certain details of the design while keeping the important system properties. While synthesizing or verifying a circuit, an environment needs to be provided to describe the input behavior for the circuit. The environment is also viewed as a testbench which supplies inputs to the circuit, and determine whether the output is expected. From the circuit's viewpoint, only the communications between the interfaces of the circuit and the environment has impact on the correctness of the circuit. Therefore, all behavior concerning the internal signals of the environment can be abstracted away. During abstraction, the interface behavior of the environment needs to remain the same or be extended conservatively. The environment after abstraction is referred to as the *abstracted* environment, and the one before abstraction is referred to as the *unabstracted* environment. It has been proven that if the circuit operates correctly in the abstracted environment, it also operates correctly in the unabstracted environment.

The environment for the whole system is referred to as the *system* environment. In practice, the system environment is usually designed very simply, typically containing no internal signals. Therefore, there is not much abstraction needed. In general, a large and complex system is not specified by a lump of declarations and statements. Instead, it often has a well defined structure and is organized in a number of components. Each component groups relative functions of the system and has a constrained interface, therefore, constrained size and complexity. The operation of a component depends on the conditions of the surrounding components. Therefore, the rest of the components

and system environment can be treated as the environment for the component, which is referred to as the *block* environment. Since the component has a limited interface, the block environment may contain a lot of internal signals. After these internal signals are abstracted away, the total number of signals under consideration can be much smaller than the number of signals in the whole system and the size of the state space can be dramatically reduced compared with the whole design. The abstraction approach described in this dissertation does not change the exponential complexity of state space exploration. Instead, with a little overhead it converts a big exponential problem into a set of small exponential problems.

The abstraction approach in this dissertation operates as two steps. First, the TEL structure for the whole system is found. If a component is chosen for synthesis or verification, then, the interface signals of the selected component are found and the signals not in the interface signals are internal signals of the block environment and all events on the internal signals are converted into sequencing events. This step is called *abstraction*. Second, these sequencing events are abstracted away using safe net reductions. This chapter describes how abstraction is applied to TEL structures without levels in the first section and ones with levels in the following section. The safe net reductions are described in the next chapter.

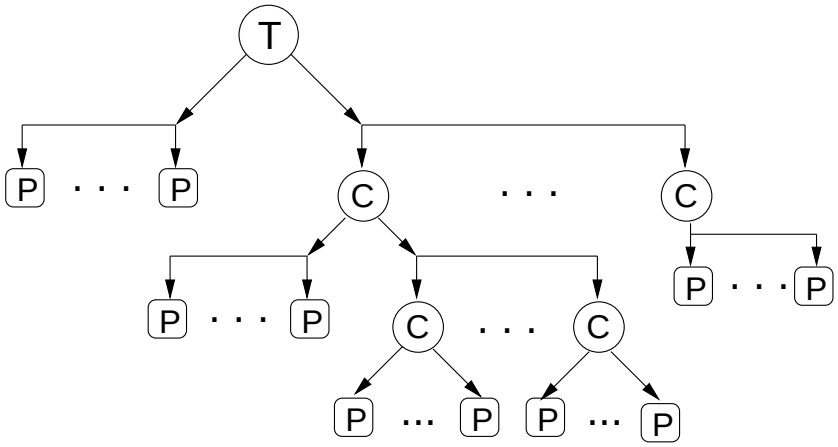

Figure 4.1. Hierarchical organization of a specification.

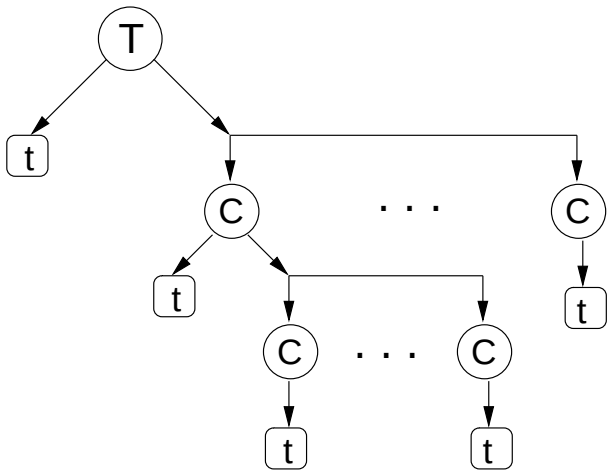

Figure 4.2. Organization of the TEL for the corresponding specification.

4.1 Abstraction for TEL Structures Without Levels

In ATACS, the specification of a circuit is typically composed of a number of processes defining its behavior and a number of components defining its structure, and each component has a similar structure. The components in the lowest level consist of only processes. The structure of a specification can be viewed as a tree shown in Figure 4.1. The circle with a T inside is the root node of the tree representing the specification on the top level. The circles with 'C' inside indicate the specifications of the components on the different levels. The squares with 'P' inside are the specifications for processes. The node for a component can contain a set of pointers to the other components that are used in this component. During compilation, the specification is decomposed into processes, each process is compiled to a TEL structure, then the TEL structures of the processes in the same component are composed in parallel. If the component contains a number of other components, pointers in the containing component are assigned to point to those components. After compilation, these TEL structures are also organized as a tree shown in Figure 4.2. Now, a component contains a TEL structure that is the parallel composition of the TELs for all processes in this component and a set of pointers to the nodes of the components that are included in this component.

In the tree, the node for a component also contains a list of signals that consists of interface signals and internal signals of the component. This signal list is referred to as

Algorithm 4.1.1 (Find the TEL from the tree)

```
⟨TELstructure, interface_list⟩ TEL(root, label){
    if(root.label==label)
        interface_list = root.interface_list;
    result.TELstructure=NULL;
    result=compose(result.TELstructure, root.t);
    foreach(pointer pc in root to another node){
        tmp = TEL(pc, label);
        result.interface_list = tmp.interface_list;
        result=compose(result.TELstructure, tmp.TELstructure);
    }
    return result;
}
```

Figure 4.3. Find the TEL for the whole design from the TEL tree.

Algorithm 4.1.2 (Change internal signal events to sequencing events)

```
replace_event(TELstructure, interface_list){
    if(interface_list is empty)
        return;
    foreach(e ∈ TELstructure.E)
        if(the signal of e ∉ interface_list)
            replace e with a sequencing event;
}
```

Figure 4.4. Replace internal signal events with sequencing events.

the *interface list*. Each component has two sets of signals: input set and output set. Input set consists of all signals connected to the component's input ports. Output set consists of all output signals to which the component's output ports connect and all internal signals of the component. The reason to include a component's internal signals in the output set is that the these signals are necessary during synthesis and verification, and we do not want them to be abstracted away during abstraction as determined by these two sets of signals. The union of these two sets forms the interface list that is the superset of the union of inputs and outputs of the component.

A procedure shown in Figure 4.3 is used to find the TEL structure for the whole design. This procedure takes two arguments: a pointer to the root of the tree of TEL structures and a label of a component. The label indicates which component in the system is selected. This procedure returns a pair. The first element of the pair is the TEL structure found for

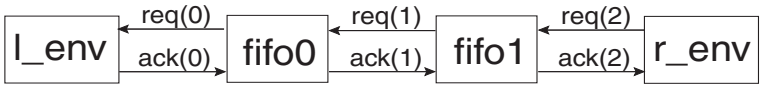

Figure 4.5. Block Diagram of a 2-stage FIFO.

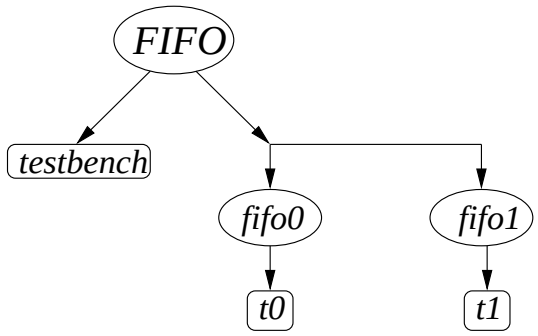

Figure 4.6. The organization of the TEL for a 2-stage FIFO.

the whole system and the second element of the pair is the interface list of the component selected by the label. This procedure traverses the tree from the root and composes all TEL structures it finds in parallel. In the meantime, it compares the label argument with the label stored in the nodes for the components. If the label argument matches the label stored in the node for a component, the interface list in that node is returned along with the TEL structure for the whole system. The label argument is optional, that means no component is chosen and the system is designed all at once. In such a case, the returned interface list is empty. After the TEL structure for the whole system and the interface list of the chosen component are available, a renaming procedure shown in Figure 4.4 is used to replace the events on the internal signals to sequencing events. This procedure takes two arguments: a TEL structure and an interface list of the chosen component, and it replaces all events in the TEL whose signals are not in the interface list to sequencing events.

The following figures illustrate how abstraction works. Figure 4.5 is the block diagram of a 2-stage FIFO. This FIFO consists of two components that are the same single *fifo* stage and two testbench processes serving as the environment for the components. Figure 4.6 shows the organization of the TELs for the testbench and components. In the figure, node *testbench* stores the TEL for the testbench processes as shown in Figure 4.7(a). Node *fifo0* stores the instantiated TEL $t0$ for a single *fifo* stage as shown in Figure 4.7(b) and the interface list $(ack(0), req(1) : \textbf{in};\ req(0), ack(1) : \textbf{out})$. Similarly,

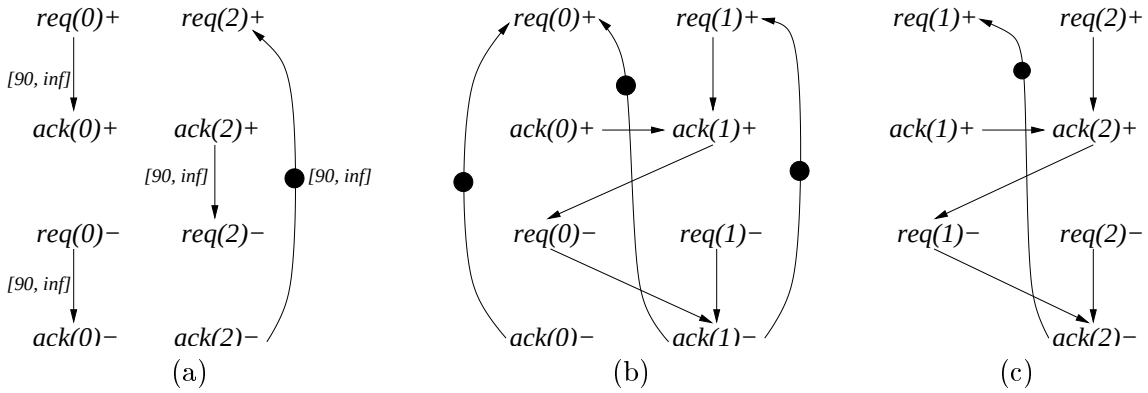

Figure 4.7. TEL structures for each block and environment.

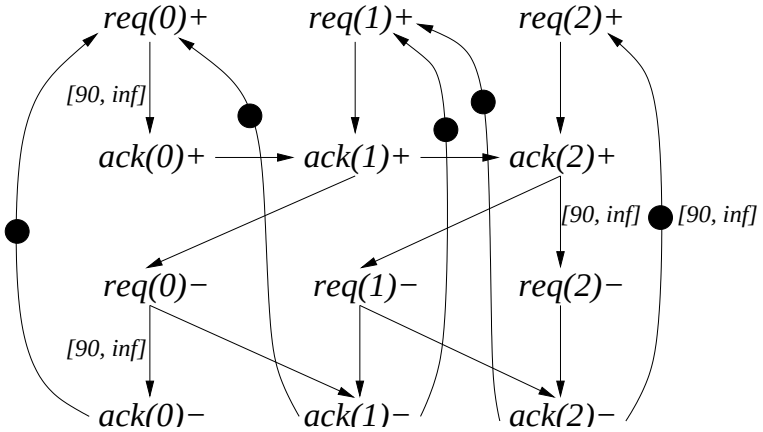

Figure 4.8. TEL of 2-stage FIFO before abstraction.

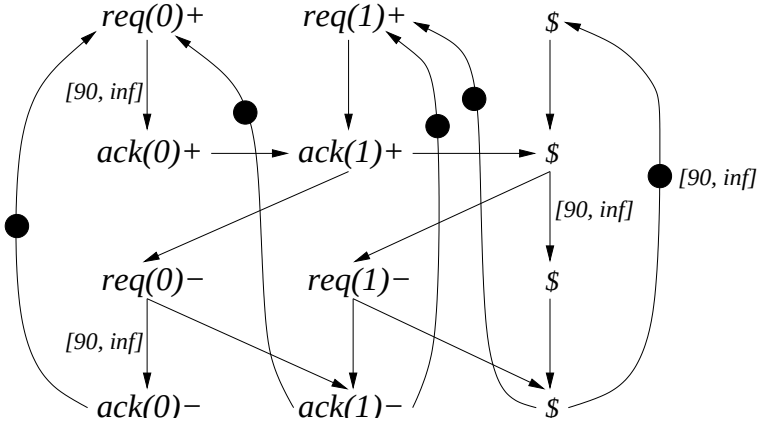

Figure 4.9. TEL of 2-stage FIFO after abstraction.

node *fifo1* stores the instantiated TEL $t1$ for a single *fifo* stage as shown in Figure 4.7(c) and the interface list $(ack(1), req(2) : \textbf{in}; req(1), ack(2) : \textbf{out})$. For simplicity, the rules in the TELs without labeled timing have the timing constraint [90,100]. Suppose we want to design *fifo0* first. The abstraction procedure goes through the tree storing the TELs starting at root *FIFO* and composes the TELs in parallel. When the node *fifo0* is reached, the interface list stored in that node is returned. The composition of the TELs for the *FIFO* is shown in Figure 4.8. Next, the abstraction procedure changes all events whose signals are not in the returned interface list to sequencing events. The signals not in the interface list of *fifo0* are $req(2)$ and $ack(2)$, that are the internal signals of the environment for *fifo0*. After abstraction is done, the TEL for the *fifo0* is shown in Figure 4.9 where the events on $req(2)$ and $ack(2)$ are changed to sequencing events.

4.2 Abstraction for TEL Structures With Levels

The result of the compilation of a specification to a TEL with levels is still organized as a tree as shown in Figure 4.2. Now the internal signals of the environment can appear both on events and in levels. The events on the these internal signal can still be abstracted away using the algorithms described in the last section. However, these internal signals in the levels must also be removed because the definition for their values has been abstracted away. Since the internal signals do not participate in the boolean evaluation of the new levels, the new levels evaluate to true at a different time than the previous ones. This section analyzes the change of the semantics of the TEL structures after removing the internal signals of the environment in levels, and how to compensate for the change.

It is described in the last chapter that an event t is enabled to fire when all rules in the preset of t are satisfied. A rule is satisfied if the enabling event of the rule has fired and the level of the rule evaluates to true, and time has passed the lower bound of the timing constraint of the rule since then. The level of a rule is a sum-of-product boolean expression. For example, a rule $r = \langle x, y, l, u, z \rangle$ where $z = ab + cd$. z becomes true when either both a and b switch to high or both c and d switch to high. Suppose a and c are internal signals of the environment, the level turns into $z' = b + d$ after removing a and c, and it becomes true when either b or d switches to high. The rule r becomes $r' = \langle x, y, l, u, z' \rangle$. Since it is possible that when z' evaluates to true is different than when z does, r and r' may become satisfied at a different time, that results in a change of the firing time of the enabled event y. Also, since the enabling condition of y has

changed after some signals are removed from the level, the timed traces produced by the system may be changed. In the last chapter, a transformation is defined to be safe if it preserves the system's behavior conservatively in terms of timed traces. This concept can be applied to abstraction similarly. If removing internal signals from a level does not reduce the untimed traces of the system and the specified firing time of an event is preserved after the level of its enabling rule is changed, removing the internal signals from the level is defined as a *safe abstraction*. This definition is formulated as follows:

Definition 4.2.1 (Safe Abstraction) *Suppose there exists a rule* $r = \langle x, y, l, u, z \rangle$ *in a TEL structure* N. *After abstracting some signals away from* z, r *becomes* $r' = \langle x, y, l', u', z' \rangle$ *and* N *becomes* N'. *Let* $\mathtt{T}$ *and* $\mathtt{T}'$ *be the ranges of firing time of* y *decided by* r *and* r', *respectively. The abstraction is safe if* N' *produces a superset of untimed traces of* N *and* $\mathtt{T}'$ *is a superset of* $\mathtt{T}$.

According to the definition, a safe abstraction must not reduce the untimed traces produced by the system. How the untimed traces are preserved after the level of a rule is changed? Suppose there exists a rule $r = \langle x, y, l, u, z \rangle$ where $z = ab$ and b is the signal that needs to be removed. To preserve the untimed traces, b in the level z must be replaced by **true**. This results in $z' = a$. In the system, N, containing r, y can fire only after x, a, and b have fired. While in the system N' containing r', y can fire after x and a have fired. b may or may not fire before firing y so N' produces a superset of untimed traces of N. Now suppose $z = a + b$ and b is the signal that needs to be removed. Again, b in the level z must be replaced by **true** to preserve the untimed traces so $z' =$ **true**. On the other hand, if b is simply removed and $z' = a$, this results in a loss of the untimed traces produced by N. The reason is explained as follows. In the new rule $r' = \langle x, y, l, u, z' \rangle$ where $z' = a$, y can fire after x and a have fired, while firing both x and b cannot cause y to fire because z' cannot evaluate to true. However, this firing sequence is allowed by the rule $r = \langle x, y, l, u, z \rangle$ where $z = a + b$. In summary, if a signal is removed from a level, this signal is replaced by **true**. This idea can be extended to a level that consists of a sum-of-product boolean expression. If all signals in a product term need to be removed, then the level simply becomes **true** based on the discussion for the level that consists of a sum boolean expression.

Next, it is necessary to study how to preserve the timing of the rule after the level of the rule has changed. To preserve the timing, the timing constraint of a rule with

the level changed can be always changed to $[l, \infty]$. However, this adjustment may be too conservative to be useful. Before we proceed to analyze the adjustment of timing constraints of rules in a refined way, minimum and maximum firing separation time needs to be defined. Firing separation time of a pair of events is the range of the difference of the firing times between these two events. The minimum firing separation time (denoted by `min_st`) is the lower bound of that range, and the maximum firing separation time (denoted by `max_st`) is the upper bound of that range. For two events a and b,

$$\texttt{min\_st}(a, b) = min(t_b - t_a) \quad \text{and} \quad \texttt{max\_st}(a, b) = max(t_b - t_a)$$

where t_a and t_b are the corresponding firing time of a and b. $\texttt{min\_st}(a, b)$ and $\texttt{max\_st}(a, b)$ can be positive or negative. A positive value indicates that a fires before b. The firing separation time can be used to indicate the firing time of an event relative to another event if the firing time of the latter one is known.

To simplify the discussion, levels with a single product term are analyzed, then levels with a sum expression, and last levels with a sum-of-products expression. Suppose $r = \langle x, y, l, u, z \rangle$ where the level $z = ab$. `min_st` and `max_st` can be used to indicate when a level becomes true relative to an event e by finding the separation times between the events that make the level become true and e. For example, in the above rule r, z becomes true when both a and b switch to high, therefore, z becomes true $max(\texttt{min\_st}(x, a), \texttt{min\_st}(x, b))$ time units after x has fired. If both $\texttt{min\_st}(x, a) \leq 0$ and $\texttt{min\_st}(x, b) \leq 0$, it indicates that both a and b fire before x does so z evaluates to true before x fires. y can fire after x has fired and time has passed l time units since then. If either $\texttt{min\_st}(x, a) > 0$ or $\texttt{min\_st}(x, b) > 0$, y fires after z becomes true and time has passed l time units since then because z evaluates to true after x has fired. A similar analysis can be applied if `min_st` is replaced with `max_st`, and a similar result can be derived. Therefore, the $\texttt{EFT}(y \leftarrow r)$ and $\texttt{LFT}(y \leftarrow r)$ are defined as follows:

$$\texttt{EFT}(y \leftarrow r) = max(t_x + max(\texttt{min\_st}(x, a), \texttt{min\_st}(x, b)) + l,\ t_x + l);$$

$$\texttt{LFT}(y \leftarrow r) = max(t_x + max(\texttt{max\_st}(x, a), \texttt{max\_st}(x, b)) + u,\ t_x + u);$$

where t_x is the firing time of x.

Figure 4.10 shows six cases of timing relations between a and b. The horizontal bars indicate the timing ranges that event a or b fires. From the figure, it is obvious that the new level z' after removing either a or b from z becomes true at the same time or sooner

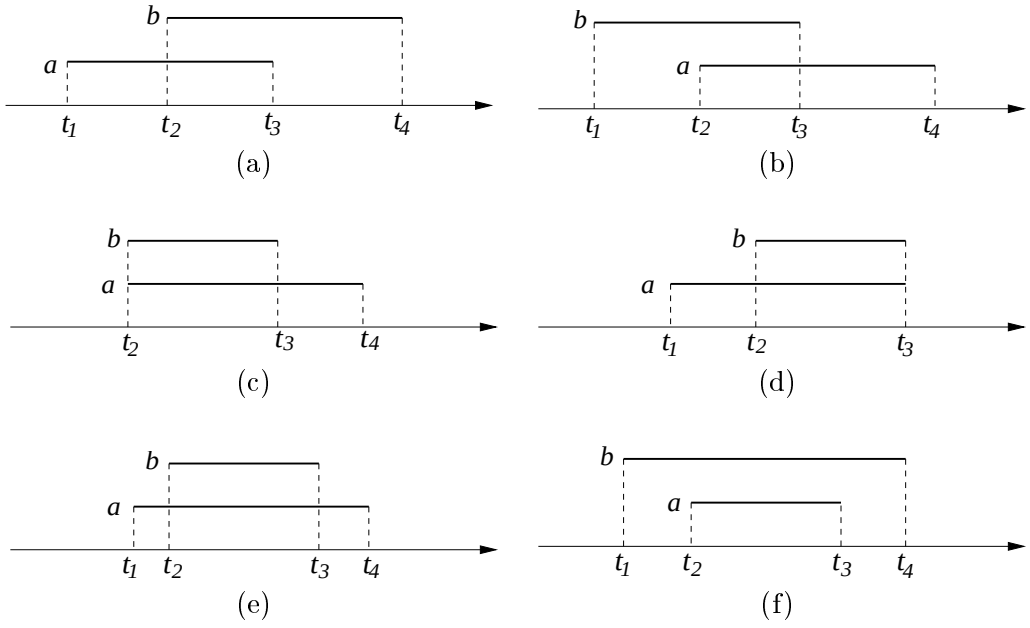

Figure 4.10. Timing relations of two events a and b.

than z. This means that y may fire sooner than specified. For example, if a is removed from z, then

$$\texttt{EFT}(y \leftarrow r') = max(t_x + \texttt{min\_st}(x, b) + l,\ t_x + l)$$

$$\texttt{LFT}(y \leftarrow r') = max(t_x + \texttt{max\_st}(x, b) + u,\ t_x + u)$$

where $r' = \langle x, y, l, u, z' \rangle$ and $z' = b$. If $\texttt{min\_st}(x, a) > \texttt{min\_st}(x, b)$ and $\texttt{max\_st}(x, a) > \texttt{max\_st}(x, b)$, then $\texttt{EFT}(y \leftarrow r') < \texttt{EFT}(y \leftarrow r)$ and $\texttt{LFT}(y \leftarrow r') < \texttt{LFT}(y \leftarrow r)$. The same result can be derived if b is removed from z. Obviously, the the range of firing time of y may change due to the removal of either a or b, therefore, this abstraction is not safe. To preserve the specified timing behavior of y, it is necessary to compensate for the loss of constraint from removing either a or b by adding a delay to $\texttt{EFT}(y \leftarrow r')$ and $\texttt{LFT}(y \leftarrow r')$ so that $\texttt{EFT}(y \leftarrow r') \leq \texttt{EFT}(y \leftarrow r)$ and $\texttt{LFT}(y \leftarrow r') \geq \texttt{LFT}(y \leftarrow r)$. In the case of removing a, the following equation must be satisfied to preserve the same timing behavior of y.

$$\texttt{EFT}(y \leftarrow r') = max(t_x + \texttt{min\_st}(x, b) + l + \Delta_{min},\ t_x + l)$$

$$\texttt{LFT}(y \leftarrow r') = max(t_x + \texttt{max\_st}(x, b) + u + \Delta_{max},\ t_x + u)$$

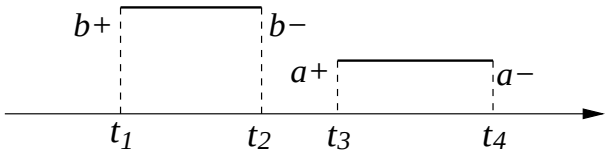

Figure 4.11. The special case of the level with a product always being **false**.

where

$$\Delta_{min} \leq max(\texttt{min\_st}(x,a) - \texttt{min\_st}(x,b),\ 0)$$

$$\Delta_{max} \geq max(\texttt{max\_st}(x,a) - \texttt{max\_st}(x,b),\ 0)$$

In the case of removing b, a similar result can be derived. The above discussion is based on the fact that part of the signals in a level are removed. If the whole product term is removed, the range of firing time of y decided by r is as follows:

$$\texttt{EFT}(y \leftarrow r') = t_x + l + \Delta_{min}$$

$$\texttt{LFT}(y \leftarrow r') = t_x + u + \Delta_{max}$$

where

$$\Delta_{min} = max(\texttt{min\_st}(x,a),\ \texttt{min\_st}(x,b),\ 0)$$

$$\Delta_{max} = max(\texttt{max\_st}(x,a),\ \texttt{max\_st}(x,b),\ 0)$$

It is obvious that $\texttt{EFT}(y \leftarrow r') = \texttt{EFT}(y \leftarrow r)$ and $\texttt{LFT}(y \leftarrow r') = \texttt{LFT}(y \leftarrow r)$. Therefore, l' and u' are defined as follows as z is changed to z':

$$l' = l + \Delta_{min} \quad \text{and} \quad u' = u + \Delta_{max} \tag{4.1}$$

where Δ_{min} and Δ_{max} are defined above. If 0 and ∞ are assigned to Δ_{min} and Δ_{max}, this abstraction is safe.

However, it should be noted that there is a special case where either a or b becomes true and then becomes false before the other one becomes true as shown in Figure 4.11. It translates to the level z always false. If z is abstracted as described above, the level may obtain a chance to become true, and the rule with such a level may actually fire the enabled event. This may change the untimed semantics of the rule. For the above analysis to be correct, it is necessary to determine if the following equation is true.

$$\texttt{max\_st}(x, \neg a) \geq \texttt{min\_st}(x,b) \quad \text{or} \quad \texttt{max\_st}(x, \neg b) \geq \texttt{min\_st}(x,a) \tag{4.2}$$

If Equation 4.2 is true, z is changed to **false**; otherwise, z is abstracted and the timing constraints of the rule are modified as described above.

Now, suppose $r = \langle x, y, l, u, z \rangle$ where $z = a + b$. z becomes true when either a or b switch to high. Therefore, z becomes true $min(\texttt{min\_st}(x, a), \texttt{min\_st}(x, b))$ time units after x has fired if both $\texttt{min\_st}(x, a) \geq 0$ and $\texttt{min\_st}(x, b) \geq 0$. The $\texttt{EFT}(y \leftarrow r)$ and $\texttt{LFT}(y \leftarrow r)$ are defined as follows:

$$\texttt{EFT}(y \leftarrow r) = max(t_x + min(\texttt{min\_st}(x, a), \texttt{min\_st}(x, b)) + l,\ t_x + l)$$

$$\texttt{LFT}(y \leftarrow r) = max(t_x + max(\texttt{max\_st}(x, a), \texttt{max\_st}(x, b)) + u,\ t_x + u)$$

where t_x is the firing time of x.

Since removing a signal in a sum boolean expression results in a level with **true**,

$$\texttt{EFT}(y \leftarrow r') = t_x + l \quad \text{and} \quad \texttt{LFT}(y \leftarrow r') = t_x + u$$

It is obvious that y fires sooner in the new rule. To make $\texttt{LFT}(y \leftarrow r) \geq \texttt{LFT}(y \leftarrow r)'$ and $\texttt{EFT}(y \leftarrow r) \leq \texttt{EFT}(y \leftarrow r)'$, the delay of the new rule must be adjusted as follows:

$$l' = l + \Delta_{min}$$

$$u' = u + \Delta_{max}$$

where

$$\begin{aligned} \Delta_{min} &= max(min(\texttt{min\_st}(x, a),\ \texttt{min\_st}(x, b)),\ 0) \\ \Delta_{max} &= max(\texttt{max\_st}(x, a),\ \texttt{max\_st}(x, b),\ 0) \end{aligned}$$

We have described a process how to abstract signals away from a level with a product or sum term. This process can be similarly applied to a level with a sum-of-product expression. If all signals in a product term are internal signals, the whole product term needs to be removed resulting in a new level **true**. The timing constraint of the rule with the level needs to be adjusted in the same way as the abstraction for a level with a sum expression. On the other hand, if some signals in a product term are abstracted away, the timing constraint of the rule with the level needs to be adjusted in the same way as the abstraction for a level with a product expression.

It is shown from the above analysis that the key to safe abstraction for levels is the knowledge of minimum and maximum separation times between two events in the system. However, finding minimum and maximum separation times is as hard as state space exploration which is an exponential problem. This results in a chicken-and-egg

situation. The purpose of abstraction is to reduce the complexity of the problem under consideration thus reducing the cost of state space exploration. If an approximation algorithm is available to find conservative minimum and maximum separation times, the abstraction can still be safe.

CHAPTER 5

SAFE NET REDUCTIONS

While analyzing a circuit, certain details need to be suppressed to constrain computational complexity. When the circuit is represented by a graph, this requires transformation of the graph to remove some nodes and adjacent edges while preserving the important system properties. There exists a lot of research work on simplifying Petri nets. Suzuhi and Murata [73, 74] presented a method of stepwise refinement of transitions and places into subnets. They show a sufficient condition that such subnets must satisfy, which are dependent on the structure and initial marking of the net. The resulting net has the same liveness and safety properties as that of the original net. However, this refinement process has to be repeated every time the initial marking is changed. This makes automating the refinement difficult. Berthelot [14] presented several transformations that depend only on the structure of the net. In [62, 43, 61], several transformations for marked graphs that are Petri nets without conflicts are presented. These transformations cut places and transitions in the graph while preserving liveness and safety. However, these transformations can only be applied to an untimed marked graph that has no choices and the timing issue is not addressed.

Chapter 4 describes how to identify internal signals in an environment and convert events on those signals into sequencing events, and also discusses the safe abstraction of signals in levels. The sequencing events need to be removed before state space exploration starts to reduce the computational complexity. This chapter describes net reduction techniques that remove the sequencing events from TEL structures safely. As defined in Chapter 3, a safe transformation must preserve the behavior of the system conservatively, or it may hide some design errors that may possibly be uncovered after the design is completed. The system's behavior is determined by the timed traces that it can generate. Therefore, it is required that the system after safe reductions must produce a superset of timed traces that are produced by the system before the reductions. To prove a reduction to be safe, we first show that the TEL structures after the reductions do not reduce the

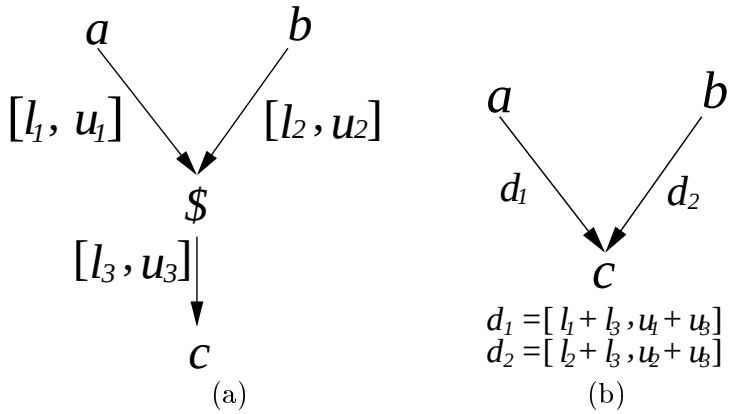

Figure 5.1. A case of Reduction 1.

untimed trace set; second, we show that the reductions preserve the timing conservatively. If these two requirements are satisfied, it guarantees that the TEL structure after safe reductions produces a superset of timed traces that can be produced by the original one. This chapter describes net reduction techniques that remove sequencing events with different topologies in TEL structures. These reductions have been proved to be safe according to the definition of safe transformations given in Chapter 3. We first present simple reductions applied to TEL structures with no levels and no conflicts. Next, we discuss how conflicts affect safe reductions and how to extend these reductions to TEL structures containing choices. In the last section, we describe the extensions of reductions to TEL structures with levels.

5.1 Safe Reductions for Conflict-Free TEL Structures

This section describes five safe net reductions for TEL structures without conflicts and levels. These reductions remove sequencing events from nets with different topologies. Reduction 1 is used when a TEL structure, N, contains a sequencing event, \$, where $\texttt{size}(\$\bullet) = 1$ as shown in the example in Figure 5.1(a). In this case, $\texttt{size}(\bullet\$) = 2$. A new TEL structure, N', is derived from N by applying Reduction 1 removing the sequencing event and all rules in its preset and postset from N. Then, for each $r_i = \langle e_i, \$, l_i, u_i \rangle \in \bullet\$$ and $r_j = \langle \$, e_j, l_j, u_j \rangle \in \$\bullet$, a new rule r is created as follows: $r = \langle e_i, e_j, l_i + l_j, u_i + u_j \rangle$. For the TEL structure shown in Figure 5.1(a), the new TEL structure after the sequencing event is removed is shown in Figure 5.1(b). This reduction preserves the system's behavior exactly in that the timed trace set produced by N' is the same as that produced by N.

Reduction 1 is naturally extended to sequencing events, \$, where $\texttt{size}(\bullet\$) > 2$. This reduction cuts the number of events and rules in a TEL by 1. To prove a reduction to be safe in the rest of the chapter, we first show that the untimed trace set produced by the TEL after the reduction includes that produced by the TEL before the reduction. Then, we show that the timing of the events enabled by the sequencing events are preserved conservatively in that they are the only events affected by the reduction. Lemma 5.1.1 asserts that Reduction 1 is safe.

Reduction 1 (For TEL structures without conflicts and levels) *If there exists a sequencing event* \$ *in a TEL structure* N *where* $\texttt{size}(\$\bullet) = 1$, *a new TEL structure* N' *can be derived from* N *as follows:*

- $E' = E - \{\$\}$,
- $R' = (R - \{r_i, r_j\}) \cup \{r\}$ *where* $r_i = \langle e_i, \$, l_i, u_i \rangle \in \bullet\$$, $r_j = \langle \$, e_j, l_j, u_j \rangle \in \$\bullet$, *and* $r = \langle e_i, e_j, l_i + l_j, u_i + u_j \rangle$.

Lemma 5.1.1 *Reduction 1 is a safe transformation.*

Proof: Consider the TEL N shown in Figure 5.1(a) and the abstracted TEL N' shown Figure 5.1(b). There are two possible untimed traces produced by N: $\{ab\$c, ba\$c\}$. These map to the traces $\{abc, bac\}$ produced by N', so the first condition is satisfied. Next, we must show that the set of timed traces produced by N' contains all the timed traces produced by N with the sequencing event deleted. Consider a timed trace $x = e_1 e_2 \ldots$ in which $e_i = (a, t_a)$, $e_j = (b, t_b)$, $e_k = (\$, t_\$)$, and $e_l = (c, t_c)$ with $i < k$, $j < k$, and $k < l$. The value of $t_\$$ falls in the following range:

$$\max\{t_a + l_1, t_b + l_2\} \leq t_\$ \leq \max\{t_a + u_1, t_b + u_2\} \tag{5.1}$$

The value of t_c comes from the range:

$$t_\$ + l_3 \leq t_c \leq t_\$ + u_3. \tag{5.2}$$

Substituting Equation 5.1 into Equation 5.2 yields:

$$\max\{t_a + l_1, t_b + l_2\} + l_3 \leq t_c \leq \max\{t_a + u_1, t_b + u_2\} + u_3. \tag{5.3}$$

In the abstracted TEL N', the value of t_c comes from the range:

$$\max\{t_a + l_1 + l_3, t_b + l_2 + l_3\} \leq t_c \leq \max\{t_a + u_1 + u_3, t_b + u_2 + u_3\}.$$

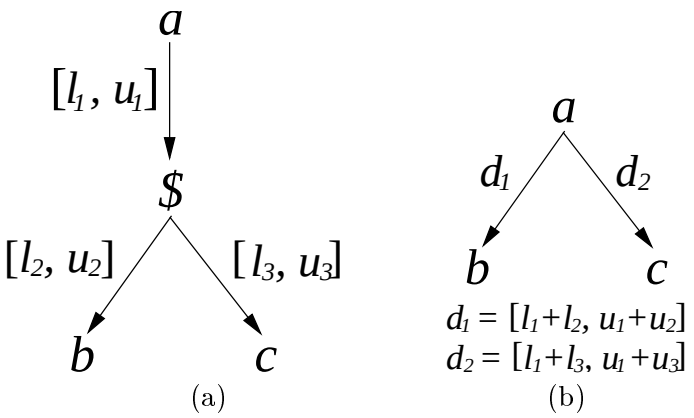

Figure 5.2. A case of Reduction 2.

This is equivalent to Equation 5.3, so the range of values for t_c before and after abstraction are equal. This means that the abstracted TEL N' produces the same timed traces as the unabstracted TEL, N. The same result can be obtained if the above analysis is applied to a sequencing event \$ where $\texttt{size}(\bullet\$) > 2$. According to the definition of safe transformation, Reduction 1 for TEL structure without conflicts is safe. □

Reduction 2 is applied to a TEL structure N when it contains a sequencing event \$ where $\texttt{size}(\bullet\$) = 1$ as one case shown in Figure 5.2(a). In this case, $\texttt{size}(\$\bullet) = 2$. Similar to Reduction 1, Reduction 2 removes the sequencing event \$ and all rules in its preset and postset. Then, for each $r_i = \langle e_i, \$, l_i, u_i \rangle \in \bullet\$$ and $r_j = \langle \$, e_j, l_j, u_j \rangle \in \$\bullet$, a new rule $r = \langle e_i, e_j, l_i + l_j, u_i + u_j \rangle$ is added to N. This reduction results in a new TEL structure, N'. For the TEL structure shown in Figure 5.2(a), the new TEL structure after the sequencing event is removed is shown in Figure 5.2(b). This reduction is naturally extended to sequencing events \$ where $\texttt{size}(\$\bullet) > 2$. This reduction cuts the number of events and rules in a TEL by 1. It needs to be pointed out that extra interleavings not seen before the reduction are created. In the case shown in Figure 5.2, after the reduction, N' could generate a trace $(a, t_a)(b, t_a + l_1 + l_2)(c, t_a + u_1 + u_3)$, where t_a is when a fires. This trace is impossible in the system before the reduction. Reduction 2 is defined and proved to be safe in the following definition and lemma.

Reduction 2 (For TEL structures without conflicts and levels) *If there exists a sequencing event* \$ *in a TEL structure* N *where* $\texttt{size}(\bullet\$) = 1$, *a new TEL structure* N' *can be derived from* N *as follows:*

- $E' = E - \{\$\}$,
- $R' = (R - \{r_i, r_j\}) \cup \{r\}$ *where* $r_i = \langle e_i, \$, l_i, u_i \rangle \in \bullet\$$, $r_j = \langle \$, e_j, l_j, u_j \rangle \in \$\bullet$, *and* $r = \langle e_i, e_j, l_i + l_j, u_i + u_j \rangle$.

Lemma 5.1.2 *Reduction 2 is a safe transformation.*

Proof: Consider the TEL N shown in Figure 5.2(a) and the abstracted TEL N' shown Figure 5.2(b). There are two possible untimed traces produced by N: $\{a\$bc, a\$cb\}$. These map to the untimed traces $\{abc, acb\}$ produced by N', so the first condition is satisfied. Next, we must show that the timed traces produced by N' contains all the timed traces produced by N with the sequencing event deleted. Consider a timed trace $x = e_1 e_2 \ldots$ in which $e_i = (a, t_a)$, $e_j = (\$, t_\$)$, $e_k = (b, t_b)$, and $e_l = (c, t_c)$ with $i < j$, $j < k$, and $j < l$. The value of t_b falls in the following range:

$$t_a + l_1 + l_2 \leq t_b \leq t_a + u_1 + u_2. \tag{5.4}$$

The value of t_c comes from the range:

$$t_b + l_3 - u_2 \leq t_c \leq t_b + u_3 - l_2. \tag{5.5}$$

After reduction, the value of t_b can still be drawn from Equation 5.4, but the value of t_c comes from the range:

$$t_b + (l_1 + l_3) - (u_1 + u_2) \leq t_c \leq t_b + (u_1 + u_3) - (l_1 + l_2).$$

This can be rewritten as follows:

$$t_b + (l_3 - u_2) + (l_1 - u_1) \leq t_c \leq t_b + (u_3 - l_2) + (u_1 - l_1).$$

Since $l_1 - u_1 \leq 0$ and $u_1 - l_1 \geq 0$, the range of values for t_c after abstraction is a superset of those before abstraction. This means that the abstracted TEL N' produces a superset of traces of the unabstracted net N. The same result can be obtained if the above analysis is applied to a sequencing event \$ where $\texttt{size}(\$\bullet) > 2$. According to the definition of safe transformation, Reduction 2 is safe. □

It seems to be natural to combine Reduction 1 and 2 to form a new reduction on a sequencing event where $\texttt{size}(\bullet\$) \geq 2$ and $\texttt{size}(\$\bullet) \geq 2$. Technically, this new reduction is safe according to the definition of safe transformations. However, this reduction can

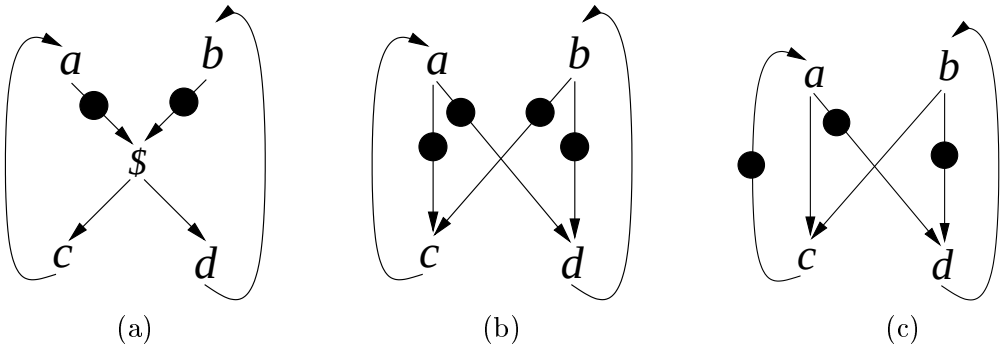

Figure 5.3. A reduction that causes a safety violation.

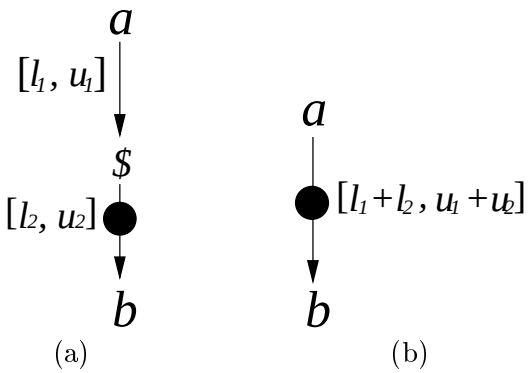

Figure 5.4. An example of Reduction 1 that changes the semantics if initially enabled rules are involved.

cause a safety violation in the reduced net. An untimed example is shown in Figure 5.3. In the TEL structure shown in Figure 5.3(a), $\texttt{size}(\bullet\$) = \texttt{size}(\$\bullet) = 2$. Figure 5.3(b) shows the new TEL after \$ is removed. In the marking shown in Figure 5.3(b), firing c results in a new marking shown in Figure 5.3(c). In this marking, firing a causes the rule $\langle a, d\rangle$ to have two tokens, which is a safety violation. Therefore, this reduction is usually avoided.

If an event is enabled by a rule in the initial marking, the event fires after a delay in the timing constraint of the rule. If the postset of a sequencing event has rules in the initial marking, Reduction 1 and 2 cannot be used because they change the timing semantics. In the example shown in Figure 5.4(a), event b fires between l_2 and u_2 time units after the system starts the execution, while in Figure 5.4(b), b fires between $l_1 + l_2$ and $u_1 + u_2$ time units after the system starts the execution. In other words, the firing time of b is delayed

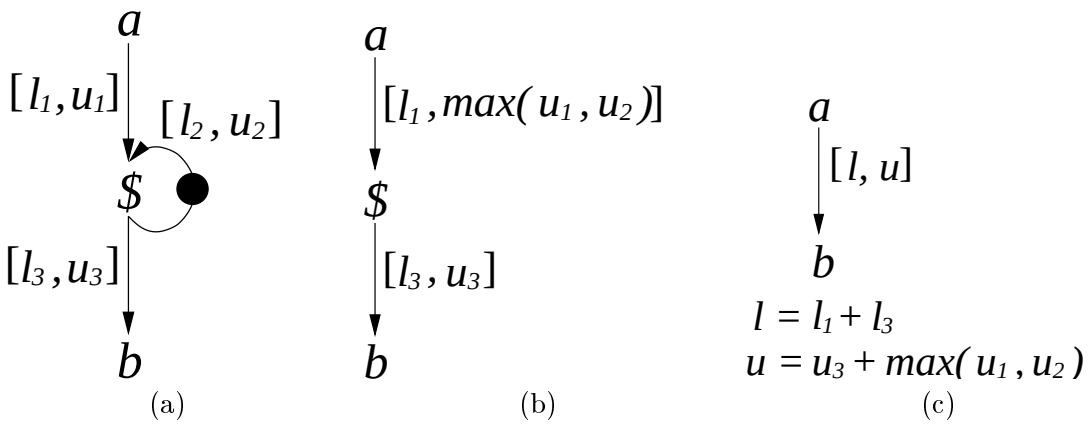

Figure 5.5. A case of Reduction 3.

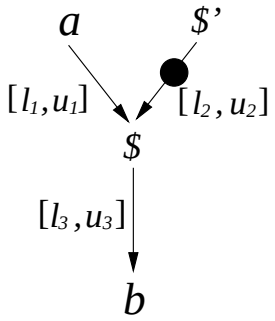

Figure 5.6. Unroll the self loop rule.

in the first execution cycle. Timing is changed instead of being preserved conservatively, so Reduction 1 and 2 cannot be used in these situations.

A more complicated reduction is when a self loop appears on a sequencing event. An example is shown in Figure 5.5(a). Self loops must be removed before the other safe reductions can be used. This reduction changes the upper bound of the delay on each rule in the preset of the sequencing event to the maximum of the original upper bound and the upper bound of the self loop rule. The lower bounds remain the same. This ensures that no matter when the last instance of the sequencing event occurred, the self loop rule would be expired when the other rules in the preset become expired. This makes the self loop redundant. The new TEL structure is shown in Figure 5.5(b). Reduction 3 is defined and proved to be safe in the following definition and lemma.

Reduction 3 (Remove self loops in TELs without conflicts and levels) *If there*

exists a sequencing event \$ *in a TEL structure* N *where a rule* $r = \langle \$, \$, l_2, u_2 \rangle$ *exists in both* $\bullet\$$ *and* $\$\bullet$. *A new TEL structure* N' *can be derived from* N *as follows:*

- $E' = E - \{\$\}$,
- $R' = R - \{r\}$,
- $\texttt{upper}(r_i) = max(u_i, u_2)$ *for all* $r_i \in \bullet\$$.

Lemma 5.1.3 *Reduction 3 is a safe transformation.*

Proof: Consider the TEL N shown in Figure 5.5(a) and the abstracted TEL N' shown in Figure 5.5(b). It is obvious that N and N' produce the same untimed traces. Next, we must show that the timed traces produced by N' contains all the timed traces produced by N. Consider a timed trace $x = e_1 e_2 \ldots$ in which $e_i = (\$, t_\$^{-1})$, $e_j = (a, t_a)$, $e_k = (\$, t_\$)$, and $e_l = (b, t_b)$, with $i < j < k < l$. The value of $t_\$$ falls in the following range:

$$\max\{t_a + l_1, t_\$^{-1} + l_2\} \leq t_\$ \leq \max\{t_a + u_1, t_\$^{-1} + u_2\} \tag{5.6}$$

where $t_\$^{-1}$ represents the firing time of the previous \$ event. Figure 5.5(a) is redrawn in Figure 5.6 to show this relationship where \$′ is the last \$ event. In N', the value of $t_\$$ falls in the following range:

$$t_a + l_1 \leq t_\$ \leq t_a + \max\{u_1, u_2\} \tag{5.7}$$

Since $x \leq \max\{x, y\}$ for any values of x and y, this means that $t_a + l_1 \leq \max\{t_a + l_1, t_\$^{-1} + l_2\}$. Since $t_a \geq t_\$^{-1}$, this means that $t_a + \max\{u_1, u_2\} \geq \max\{t_a + u_1, t_\$^{-1} + u_2\}$. Therefore, the range of values for $t_\$$ after abstraction is a superset of those before abstraction. This means that the abstracted net N' produces a superset of traces of N. The same result can be obtained if the above analysis is applied to a sequencing event \$ where $\texttt{size}(\bullet\$) > 2$ and $\texttt{size}(\$\bullet) > 2$. According to the definition of safe transformation, Reduction 3 is safe. □

After removal of the self loop, the TEL structure can be reduced to one of the cases shown above and the sequencing event can be removed using either Reduction 1 or 2, as described above. For the example shown in Figure 5.5(a), after Reduction 3, Reduction 1 can be applied to remove \$ and the final result is shown in Figure 5.5(c).

The first three reductions deal with a single sequencing event based on the sizes of its preset and postset. They do not check the graphical structures of its neighbors and

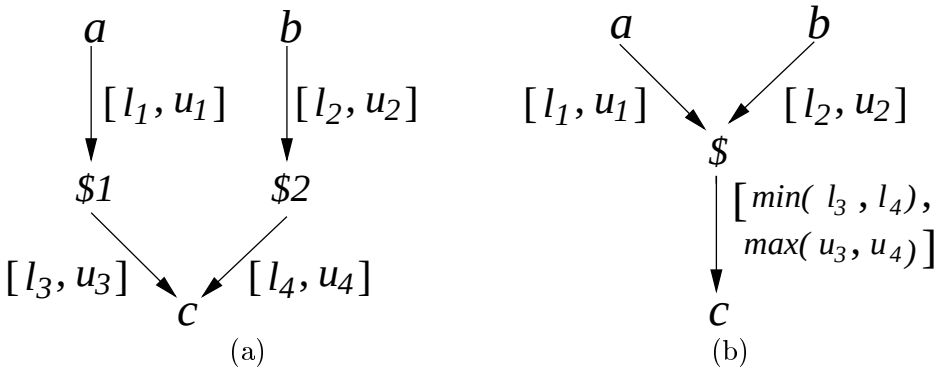

Figure 5.7. A case of Reduction 4.

how the sequencing event relates to them. The next two reductions fill this gap by first checking the graphical structures of a group of sequencing events. If they have a similar structure in a certain way, then all but one of them can be removed. Reduction 4 is applied to a TEL structure N when it contains two sequencing events \$1 and \$2 where `enabled_set`(\$1) = `enabled_set`(\$2) as in the case shown in Figure 5.7(a). In Figure 5.7(a), c is enabled by \$1 and \$2 which are enabled by a and b, respectively. This TEL structure can be regarded as c is enabled by a and b indirectly. Another way to view this situation is to treat this TEL structure as a black box which has two inputs a and b, and one output c. c occurs after both a and b have occurred. Reduction 4 merges \$1 and \$2 to \$ and rules in their presets and postsets. Assume `enabling_set`(\$1) $\cap$ `enabling_set`(\$2) $= \emptyset$, Reduction 4 cuts one sequencing event and the number of `size`(\$1$\bullet$) rules from the TEL. When merging two rules from the postsets of \$1 and \$2, respectively, the minimum of the lower bounds of the timing constraints of these two rules is assigned to the lower bound of the new rule, and the maximum of the upper bounds of the timing constraints of these two rules is assigned to the upper bound of the new rule. In this way, the range of firing time of the events in the `enabled_set`(\$1) and `enabled_set`(\$2) is preserved in the abstracted TEL structure, which produces a superset of timed traces of the unabstracted TEL structure. This result is proven in the following lemma. Reduction 4 is naturally extended to any number of sequencing events that have the same set of enabled events.

Reduction 4 (Merge sequencing events with the same enabled set) *If there exist two sequencing events* \$1 *and* \$2 *in a TEL structure* N *where* `enabled_set`(\$1) =

`enabled_set`($2), *a new TEL structure* N' *can be derived from* N *as follows:*

- $E' = E - \{\$1, \$2\} \cup \{\$\}$,
- *for each* $r_i = \langle e_i, \$1, l_i, u_i \rangle \in {}^\bullet\$1$ *and* $r_j = \langle e_j, \$2, l_j, u_j \rangle \in {}^\bullet\$2$, *they are changed to*

$$r_i' = \langle e_i, \$, l_i, u_i \rangle \quad \textit{and} \quad r_j' = \langle e_j, \$, l_j, u_j \rangle$$

- $R' = (R - \{r_m, r_n\}) \cup \{r\}$ *where* $r_m = \langle \$1, c, l_m, u_m \rangle \in \$1^\bullet$, $r_n = \langle \$2, c, l_n, u_n \rangle \in \$2^\bullet$, *and* $r = \langle \$, c, min(l_m, l_n), max(u_m, u_n) \rangle$.

Lemma 5.1.4 *Reduction 4 is a safe transformation.*

Proof: Consider the TEL structure N shown in Figure 5.7(a), it produces four possible untimed traces: $\{ab\$1\$2c, ba\$1\$2c, ab\$2\$1c, ba\$2\$1c,$ a\$1b\$2c, b\$2a\$1c$\}$. The untimed trace set produced by the abstracted TEL structure N' has two traces: $\{ab\$c, ba\$c\}$. It is obvious that N and N' have the same untimed traces after all sequencing events in the untimed traces are deleted, so the first condition is satisfied. Next, we must show that the timed traces produced by N' contains all the timed traces produced by N with all sequencing events deleted. Consider a timed trace $x = e_1 e_2 \ldots$ where $e_i = (a, t_a)$, $e_j = (b, t_b)$, $e_k = (\$1, t_{\$1})$, $e_l = (\$2, t_{\$2})$ and $e_m = (c, t_c)$ with $i < k$, $j < l$, and $k < m$, $l < m$. The value of $t_{\$1}$ and $t_{\$2}$ fall in the following ranges:

$$t_a + l_1 \leq t_{\$1} \leq t_a + u_1 \tag{5.8}$$

$$t_b + l_2 \leq t_{\$2} \leq t_b + u_2 \tag{5.9}$$

The value of t_c comes from the range:

$$max\{t_{\$1} + l_3, t_{\$2} + l_4\} \leq t_c \leq max\{t_{\$1} + u_3, t_{\$2} + u_4\} \tag{5.10}$$

Substituting Equation 5.8 and 5.9 into Equation 5.10 yields:

$$\max\{t_a + l_1 + l_3, t_b + l_2 + l_4\} \leq t_c \leq \max\{t_a + u_1 + u_3, t_b + u_2 + u_4\} \tag{5.11}$$

In the abstracted TEL structure, N', the value of $t_\$$ and t_c come from the ranges:

$$max\{t_a + l_1, t_b + l_2\} \leq t_\$ \leq max\{t_a + u_1, t_b + u_2\}. \tag{5.12}$$

$$t_\$ + min\{l_3, l_4\} \leq t_c' \leq t_\$ + max\{u_3, u_4\}. \tag{5.13}$$

Substituting Equation 5.12 into Equation 5.13 yields:

$$max\{t_a + l_1 + min\{l_3, l_4\}, t_b + l_2 + min\{l_3, l_4\}\} \leq t_c'$$

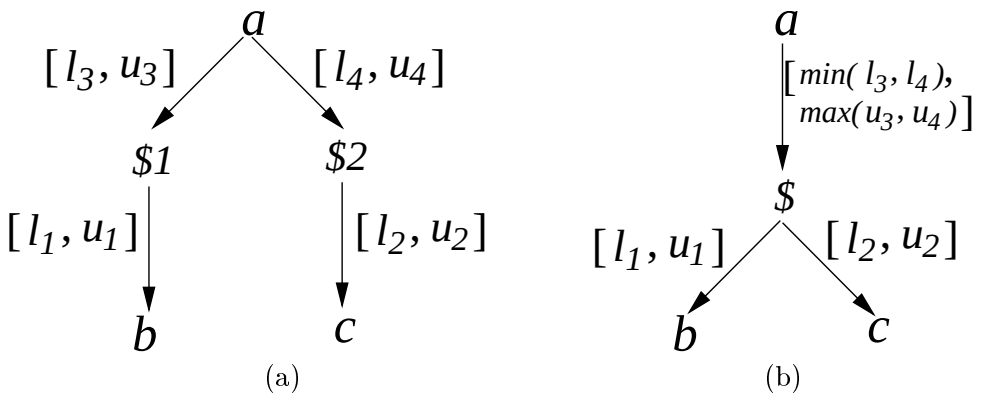

Figure 5.8. A case of Reduction 5.

$$\leq max\{t_a + u_1 + max\{u_3, u_4\}, t_b + u_2 + max\{u_3, u_4\}\}$$

Since $a \geq min\{a, b\}$ and $a \leq max\{a, b\}$, the following two equations are true:

$$max\{t_a + l_1 + min\{l_3, l_4\}, t_b + l_2 + min\{l_3, l_4\}\} \leq max\{t_a + l_1 + l_3, t_b + l_2 + l_4\}$$

$$\max\{t_a + u_1 + u_3, t_b + u_2 + u_4\} \leq max\{t_a + u_1 + max\{u_3, u_4\}, t_b + u_2 + max\{u_3, u_4\}\}$$

The range of values for t'_c in N' is a superset of t_c in N. This means that the abstracted TEL structure N' produces a superset of timed traces of the unabstracted TEL structure N. The same result can be obtained if the above analysis is applied to sequencing events where the sizes of their presets and postsets are greater than 2. According to the definition of safe transformation, Reudction 4 is safe. □

Reduction 5 is called when two sequencing events have the same set of enabling events. One case is shown in Figure 5.8(a) where $1 and $2 have the same enabling event. Similar to Reduction 4, Reduction 5 merges these two sequencing events. The enabling events of all rules in their postsets are merged together. For two rules coming from the presets of $1 and $2 respectively, since the sequencing events are merged into a single sequencing event, they are merged together to form a new rule. The minimum of the lower bounds of the timing constraints of these two rules is assigned to the lower bound of the new rule, and the maximum of the upper bounds of the timing constraints of these two rules is assigned to the upper bound of the new rule. The abstracted TEL structure for the one shown in Figure 5.8(a) is shown Figure 5.8(b). In this way, the range of firing time of the events in the `enabled_set`($1) and `enabled_set`($2) is preserved in the abstracted TEL

structure, which produces a superset of timed traces of the unabstracted TEL structure. This result is proven in the following lemma. Similar to Reduction 4, if \$1 and \$2 enable no common events, Reduction 5 cuts one sequencing event and `size`(•\$1) rules from the TEL. Reduction 5 is naturally extended to any number of sequencing events that have the same enabling set of events.

Reduction 5 (Merge sequencing events with the same enabling set) *If there exist two sequencing events* \$1 *and* \$2 *in a TEL structure* N *where* `enabling_set`(\$1) = `enabling_set`(\$2)*, a new TEL structure* N' *can be derived from* N *as follows:*

- $E' = E - \{\$1, \$2\} \cup \{\$\}$,
- *for each rule* $r_i = \langle \$1, e_i, l_i, u_i \rangle \in \$1\bullet$ *and* $r_j = \langle \$2, e_j, l_j, u_j \rangle \in \$2\bullet$*, they are changed to*

$$r_i' = \langle \$, e_i, l_i, u_i \rangle \quad \textit{and} \quad r_j' = \langle \$, e_j, l_j, u_j \rangle$$

- $R' = (R - \{r_m, r_n\}) \cup \{r\}$ *where* $r_m = \langle a, \$1, l_m, u_m \rangle \in \bullet\$1$, $r_n = \langle a, \$2, l_n, u_n \rangle \in \bullet\$2$, *and* $r = \langle a, \$, min(l_m, l_n), max(u_m, u_n) \rangle$.

Lemma 5.1.5 *Reduction 5 is a safe transformation.*

Proof: Consider the TEL structure N shown in Figure 5.8(a) and its abstracted counterpart N' shown in Figure 5.8(b). There are six possible untimed traces produced by N: $\{a\$1\$2bc, a\$1\$2cb, a\$2\$1bc, a\$2\$1cb,$ a\$2c\$1b, $a\$1b\$2c\}$. The untimed trace set produced by N' has two possible untimed traces: $\{a\$bc, a\$cb\}$. It is obvious that these two nets have the same untimed traces after all sequencing events in the traces are deleted, so the first condition is satisfied. Next, we must show that the timed traces produced by N' include all the timed traces produced by N with the sequencing event deleted. Consider a timed trace $x = e_1 e_2 \ldots$ where $e_i = (a, t_a)$, $e_j = (\$1, t_{\$1})$, $e_k = (\$2, t_{\$2})$, $e_l = (b, t_b)$, and $e_m = (c, t_c)$ with $i < j$, $i < k$, $j < l$, and $k < m$. The value of $t_{\$1}$ and $t_{\$2}$ fall in the following ranges:

$$t_a + l_3 \leq t_{\$1} \leq t_a + u_3 \tag{5.14}$$

$$t_a + l_4 \leq t_{\$2} \leq t_a + u_4 \tag{5.15}$$

The value of t_b and t_c comes from the range:

$$t_{\$1} + l_1 \leq t_b \leq t_{\$1} + u_1 \tag{5.16}$$

$$t_{\$2} + l_2 \leq t_c \leq t_{\$2} + u_2 \tag{5.17}$$

Substituting Equation 5.14 and 5.15 into Equation 5.16 and 5.17 yields:

$$t_a + l_1 + l_3 \leq t_b \leq t_a + u_1 + u_3$$

$$t_a + l_2 + l_4 \leq t_c \leq t_a + u_2 + u_4$$

After reduction, the value of $t_\$$ comes from the range:

$$t_a + min\{l_3, l_4\} \leq t_\$ \leq t_a + max\{u_3, u_4\}. \tag{5.18}$$

and the value of t_b and t_c still come from the range defined in Equation 5.16 and 5.17 where $t_{\$1}$ and $t_{\$2}$ are defined in Equation 5.18. Substituting Equation 5.18 into Equation 5.16 and 5.17 yields:

$$t_a + l_1 + min\{l_3, l_4\} \leq t'_b \leq t_a + u_1 + max\{u_3, u_4\} \tag{5.19}$$

$$t_a + l_2 + min\{l_3, l_4\} \leq t'_c \leq t_a + u_2 + max\{u_3, u_4\} \tag{5.20}$$

Since

$$t_a + l_1 + min\{l_3, l_4\} \leq t_a + l_1 + l_3$$

$$t_a + u_1 + u_3 \leq t_a + u_1 + max\{u_3, u_4\}$$

and

$$t_a + l_2 + min\{l_3, l_4\} \leq t_a + l_2 + l_3$$

$$t_a + u_2 + u_4 \leq t_a + u_2 + max\{u_3, u_4\}$$

the ranges of values for t'_b and t'_c are a superset of t_b and t_c. This means that the abstracted TEL structure N' produces a superset of timed traces in the unabstracted TEL structure N. The same result can be obtained if the above analysis is applied to sequencing events where the sizes of their presets and postsets are greater than 2. According to the definition of safe transformation, Reduction 5 is safe. □

One may ask the question: since Reduction 1 and 2 exist, why Reduction 4 and 5 are necessary to remove those sequencing events as illustrated in Figure 5.7 and 5.8. To answer the question, we need to take a look at the example shown in Figure 5.9. For simplicity, delays on all rules in this figure are not shown. In this figure, the sizes of the presets and postsets of both the sequencing events are two. Neither Reduction 1 nor 2

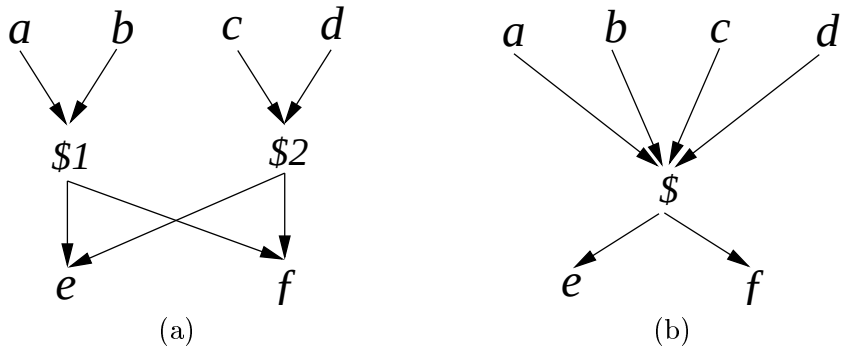

Figure 5.9. An example of the application of Reduction 4.

can be applied to remove either one of the sequencing events. However, Reduction 4 can be used in this example to merge the two sequencing events together. Therefore, the new TEL structure has one sequencing event and two rules less than the unabstracted one. The new TEL structure is shown in Figure 5.9(b). Similar situations can be found that only Reduction 5 is applicable.

5.2 Dealing With Conflicts

The TEL structures with the reductions described in the last section can be applied when there are no conflicts. However, conflict-free TEL structures are not very capable of modeling real systems. It is common that most TEL structures contain conflicts to model choice, so it is necessary to extend the reductions to be able to handle conflicts. Given an event in a TEL, there are two groups of conflicts involved with this event: conflicts among the events in its enabling set and enabled set, and those between the event itself and other events. The conflict relation between a pair of events must be carefully preserved if one of them is abstracted away, otherwise, the untimed semantics of the system may change.

Consider the example shown in Figure 5.10(a). In the last section, Reduction 1 can be applied, but now it has a conflict between a and b. For simplicity, the timing on the rules is not shown. According to the semantics of conflicts, either firing a or b, but not both, causes \$ to fire. Therefore, this TEL can be decomposed to two equivalent TEL structures as shown in Figure 5.10(b). During one execution cycle of the TEL, either one of the TELs in Figure 5.10(b) is active. As proved in Lemma 5.1.1, the system produces the same timed traces after Reduction 1 is applied to them. The abstracted TEL for the one shown in Figure 5.10(a) is shown in Figure 5.10(c). Another example is shown

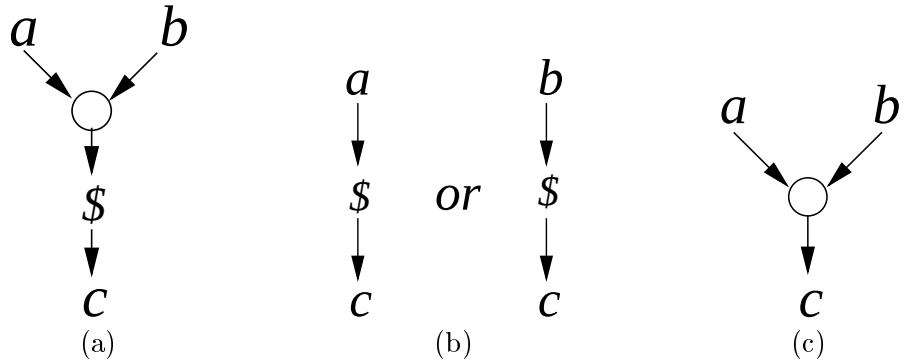

Figure 5.10. An example of a sequencing event with a conflict in its preset.

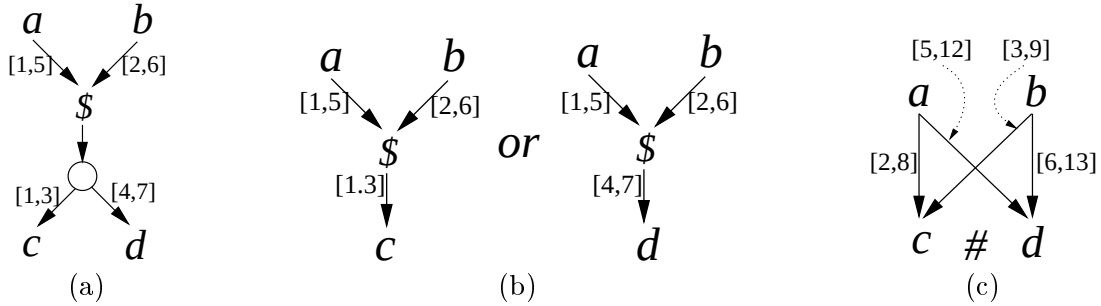

Figure 5.11. An example of a sequencing event where its postset has multiple rules but only one conflict place.

in Figure 5.11(a) where there are two rules in the postset of the sequencing event, but the enabled events of these two rules are in conflict. Similarly, this TEL fragment can be decomposed to two equivalent TELs as shown in Figure 5.11(b). Since Reduction 1 can be applied to both of them safely, the sequencing event in the TEL in Figure 5.11(a) can be abstracted away safely, and the abstracted TEL is shown in Figure 5.11(c). This reduction assumes that the choices between events are timing independent. If, instead the timing dependent choice semantics is applied, the reduction shown in Figure 5.11 creates extra behavior. For example, in Figure 5.11(a), event c has no chance to fire because the rule $\langle \$, c, 1, 3 \rangle$ always expires before the other rule becomes satisfied. In the TEL shown in Figure 5.11(c), there exists a chance that event c can fire. However, with the timing independent choice semantics, choice between firing c and d is made first, then the timing is considered. Therefore, no extra behavior is created. The semantics of choices in the

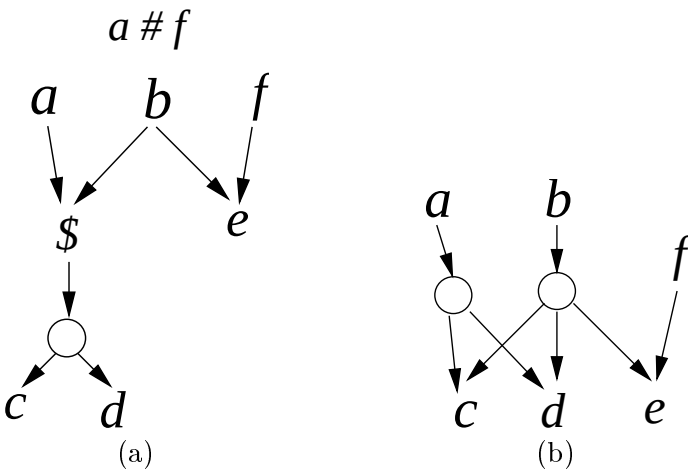

Figure 5.12. An example where the sequencing event conflicts with another event.

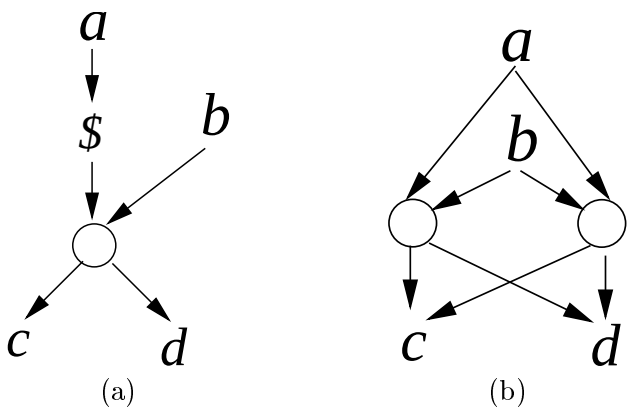

Figure 5.13. Another example where the sequencing event conflicts with another event.

TEL structures is timing independent. This discussion can be extended to a sequencing event where the postset of the sequencing event has a number of rules greater than 2 but all enabled events of the rules are in conflict. In other words, there is only one conflict place in the postset of the sequencing event. Combining the analysis for the above two examples, if there exists only one conflict place in the postset of a sequencing event, Reduction 1 in Lemma 5.1.1 can be applied to remove the sequencing event without any modification.

In the example shown in Figure 5.12(a), the sequencing event, \$, has a conflict place in its postset, and meanwhile the sequencing event itself conflicts with another event e,

and \$ and e have a common enabling event. During one execution cycle, either \$ or e can fire, but not both. Firing \$ causes either c or d to fire. This situation can be translated to the following: after firing a and b, or b and f, only one of c, d, and e can fire. In other words, c, d, and e are in conflict. To keep the same semantics, it is necessary to add two new conflicts after the sequencing event, \$, is abstracted away: $c\#e$ and $d\#e$. The abstracted TEL for the example shown in Figure 5.12(a) is shown in Figure 5.12(b). Another example where the sequencing event \$ conflicts with another event b is shown in Figure 5.13(a). In this example, \$ and b enable some common events. Firing either \$ or b causes one of c and d to fire. Since \$ is enabled by a, that means a and b must be in conflict. Therefore, a new conflict between a and b needs to be added to the TEL after \$ is abstracted away. The above discussion of Reduction 1 for TEL structures with conflicts is formalized and proved to be safe in the following definition and lemma.

Reduction 6 (Extension of Reduction 1 to TELs with conflicts) *If there exists a sequencing event, \$, in a TEL structure N where there is only one conflict place in its postset, a new TEL structure N' can be derived from N as follows:*

- $E' = E - \{\$\}$,
- $R' = (R - \{r_i, r_j\}) \cup \{r\}$ *where* $r_i = \langle e_i, \$, l_i, u_i \rangle \in \bullet\$$, $r_j = \langle \$, e_j, l_j, u_j \rangle \in \$\bullet$, *and* $r = \langle e_i, e_j, l_i + l_j, u_i + u_j \rangle$,
- *if* \$ *conflicts with* $e \in E$ *and* $\texttt{enabling\_set}(\$) \cap \texttt{enabling\_set}(e) \neq \emptyset$,

$$\#' = \# \cup \{e\#x\} \text{ for all } x \in \texttt{enabled\_set}(\$)$$

- *if* \$ *conflicts with* $e \in E$ *and* $\texttt{enabled\_set}(\$) \cap \texttt{enabled\_set}(e) \neq \emptyset$,

$$\#' = \# \cup \{e\#y\} \text{ for all } y \in \texttt{enabling\_set}(\$)$$

Lemma 5.2.1 *Reduction 6 is a safe transformation.*

Proof: Since only the timing of the enabled set of \$ is changed and preserved in the same way as in Lemma 5.1.1, we only need to show that N and N' produce the same untimed trace to prove this lemma. First consider the TEL N shown in Figure 5.11(a) where the sequencing event does not conflict with other events and the corresponding abstracted TEL N' shown in Figure 5.11(c). N produces four possible untimed traces: $\{ab\$c, ba\$c, ab\$d, ba\$d\}$, while N' also produces four possible untimed traces:

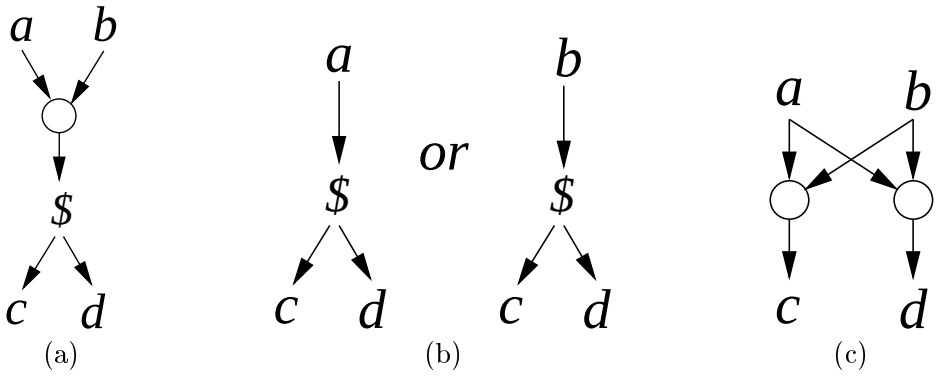

Figure 5.14. An example of sequencing event where its preset has multiple rules but only one conflict place.

$\{abc, bac, abd, bad\}$. It is obvious that these two untimed trace sets are the same after the \$ is deleted. Then, consider the TEL N shown in Figure 5.12(a) where the sequencing event conflicts with e and they are enabled by b. The corresponding abstracted TEL N' shown in Figure 5.12(b). The possible untimed traces produced by N has six traces: $\{ab\$c, ab\$d, ba\$c, ba\$d, bfe, fbe\}$. This maps to the possible untimed traces produced by N' after \$ deleted: $\{abc, abd, bac, bad, bfe, fbe\}$, so N and N' produce the same untimed traces. Now consider the TEL N shown in Figure 5.13(a) where the sequencing event conflicts with b and they enable common events c and d. The corresponding abstracted TEL N' shown in Figure 5.13(b). The possible untimed traces produced by N has six traces: $\{a\$c, a\$d, bc, bd\}$. This maps to the possible untimed traces produced by N' after \$ is deleted: $\{ac, ad, bc, bd\}$, so N and N' produce the same untimed traces. Since the timing of the events enabled by \$ is preserved as proved in Lemma 5.1.1, N and N' produce the same timed traces. According to the definition of safe transformations, Reduction 1 is safe. □

Similar to Reduction 1, Reduction 2 can be extended to TEL structures with conflicts as shown in the following definition and lemma.

Reduction 7 (Extension of Reduction 2 to TELs with conflicts) *If there exists a sequencing event* \$ *in a TEL structure N where there is only one conflict place in* $\bullet\$$*, a new TEL structure N' can be derived from N as follows:*

- $E' = E - \{\$\}$,

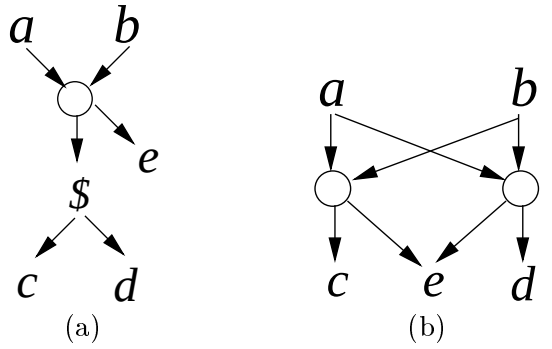

Figure 5.15. An example of sequencing event conflicting with another event.

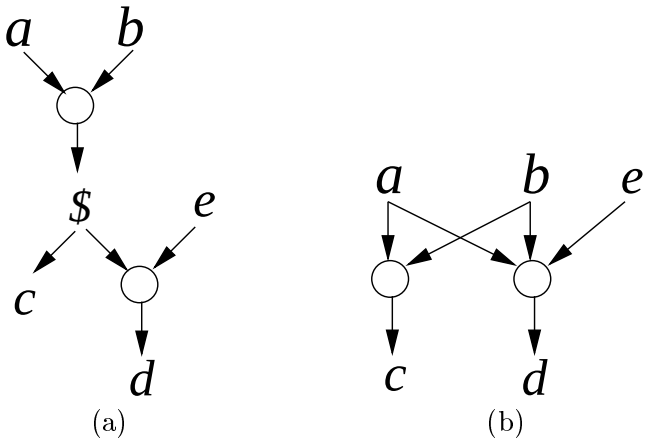

Figure 5.16. An example of sequencing event conflicting with another event.

- $R' = (R - \{r_i, r_j\}) \cup \{r\}$ *where* $r_i = \langle e_i, \$, l_i, u_i \rangle \in \bullet\$$, $r_j = \langle \$, e_j, l_j, u_j \rangle \in \$\bullet$, *and* $r = \langle e_i, e_j, l_i + l_j, u_i + u_j \rangle$,
- *if* \$ *conflicts with* $e \in E$ *and* $\texttt{enabling\_set}(\$) \cap \texttt{enabling\_set}(e) \neq \emptyset$,

$$\#' = \# \cup \{e \# x\} \text{ for all } x \in \texttt{enabled\_set}(\$)$$

- *if* \$ *conflicts with* $e \in E$ *and* $\texttt{enabled\_set}(\$) \cap \texttt{enabled\_set}(e) \neq \emptyset$,

$$\#' = \# \cup \{e \# y\} \text{ for all } y \in \texttt{enabling\_set}(\$)$$

Lemma 5.2.2 *Reduction 7 is a safe transformation.*

Proof: Consider the TEL, N, shown in Figure 5.14(a) where the sequencing event has

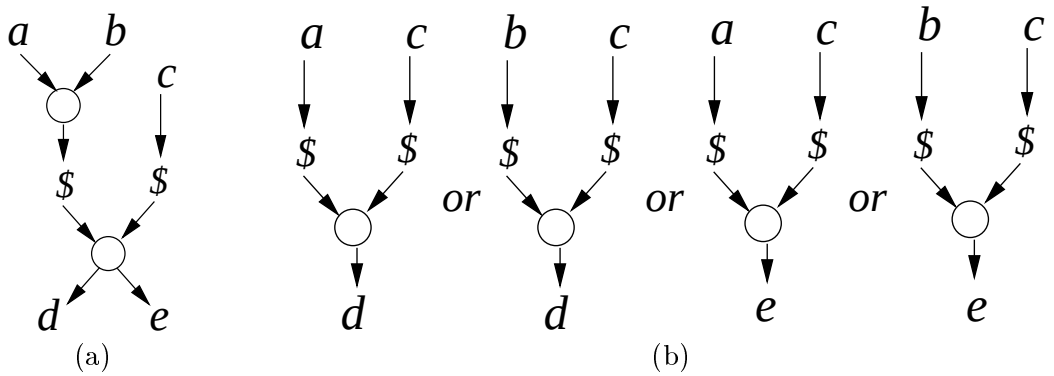

Figure 5.17. Decompose a TEL into TELs where Reduction 8 can be applied.

only one conflict place in its preset. This TEL fragment can be decomposed into two equivalent TELs as shown in Figure 5.14(b). Only one of them is active during one execution cycle. It is obvious that \$ in these two TELs can be abstracted away safely using Reduction 2 in Lemma 5.1.2, and the system produces a superset of timed traces of N. N' is shown in Figure 5.14(c). If \$ conflicts with another event, we only need to show that N and N' produce the same untimed traces to prove this lemma because only the timing of the enabled set of \$ is changed and preserved conservatively. First, consider the TEL N shown in Figure 5.15(a) where \$ conflicts with another event e and they have the same enabling set. N' is shown in Figure 5.15(b) where new conflicts $c\#e$ and $d\#e$ are created. N produces six possible untimed traces: $\{a\$cd, a\$dc, ae, b\$cd, b\$dc, be\}$, while N' also produces six possible untimed traces: $\{acd, adc, ae, bcd, bdc, be\}$. It is obvious that these two untimed trace sets are the same after \$ is deleted. Then, consider the TEL N shown in Figure 5.16(a) where \$ conflicts with e and they both enable d. N' is shown in Figure 5.16(b) where new conflicts $a\#e$ and $b\#e$ are created. The possible untimed traces produced by N has five traces: $\{a\$cd, a\$dc, b\$cd, b\$dc, ed\}$. This maps to the possible untimed traces produced by N' after \$ deleted: $\{acd, adc, bcd, bdc, ed\}$, so N and N' produce the same untimed traces. In both TEL structures shown in Figure 5.15 and Figure 5.16, only the timing of the rules in $\$\bullet$ is changed and preserved conservatively as proven in Lemma 5.1.2, so N' produces a superset of timed traces of N. According to the definition of safe transformations, Reduction 2 is safe. □

Examples shown in Figure 5.17(a) and Figure 5.7(a) have a similar structure, except that the two sequencing events in Figure 5.17(a) are in conflict and there are also conflict

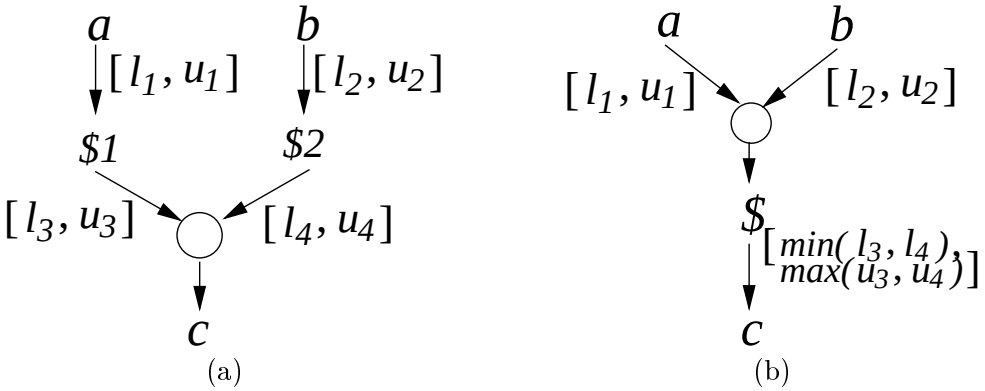

Figure 5.18. An example of Reduction 8.

places in their presets and postsets. The example shown in Figure 5.17(a) can be decomposed into four equivalent TEL structures, each of which has a structure shown in Figure 5.18(a). If the reduction shown in Figure 5.18 is safe, merging the two sequencing events in Figure 5.17(a) is also safe. The above discussion is formalized and proved to be safe in the following definition and lemma.

Reduction 8 (Extension of Reduction 4 to TELs with conflicts) *If there exist two sequencing events* \$1 *and* \$2 *in a TEL structure* N *where* \$1 *conflicts with* \$2, *and* `enabled_set`(\$1) = `enabled_set`(\$2). *A new TEL structure,* N', *can be derived from* N *as follows:*

- $E' = E - \{\$1, \$2\} \cup \{\$\}$,
- *for each* $r_i = \langle e_i, \$1, l_i, u_i\rangle \in {}^\bullet\$1$ *and* $r_j = \langle e_j, \$2, l_j, u_j\rangle \in {}^\bullet\$2$, *they are changed to*

$$r_i' = \langle e_i, \$, l_i, u_i\rangle \quad \textit{and} \quad r_j' = \langle e_j, \$, l_j, u_j\rangle$$

- $R' = (R - \{r_m, r_n\}) \cup \{r\}$ *where* $r_m = \langle \$1, c, l_m, u_m\rangle \in \$1^\bullet$, $r_n = \langle \$2, c, l_n, u_n\rangle \in \$2^\bullet$, *and* $r = \langle \$, c, min(l_m, l_n), max(u_m, u_n)\rangle$

Lemma 5.2.3 *Reduction 8 is a safe transformation.*

Proof: Consider the TEL structure N shown in Figure 5.18(a), it produces two possible untimed traces: $\{a\$1c, b\$2c\}$. The untimed trace set produced by the abstracted TEL structure N' in Figure 5.18(b) has two traces: $\{a\$c, b\$c\}$. It is obvious that N and N' have the same untimed traces after sequencing events are deleted, so the first condition

is satisfied. Next, we must show that the timed traces produced by N' contains all the timed traces produced by N with all sequencing events deleted. Consider a timed trace $x = e_1 e_2 \ldots$ where $e_i = (a, t_a)$, $e_j = (\$1, t_{\$1})$, $e_k = (c, t_c)$ with $i < j < k$. The value of $t_{\$1}$ falls in the following range:

$$t_a + l_1 \leq t_{\$1} \leq t_a + u_1 \tag{5.21}$$

The value of t_c comes from the range:

$$t_a + l_1 + l_3 \leq t_c \leq t_a + u_1 + u_3 \tag{5.22}$$

In the abstracted TEL structure N', suppose a occurs, the value of t_c come from the ranges:

$$t_a + l_1 + min\{l_3, l_4\} \leq t'_c \leq t_a + u_1 + max\{u_3, u_4\}$$

Since $a \geq min\{a, b\}$ and $a \leq max\{a, b\}$, the following two equations are true:

$$t_a + l_1 + min\{l_3, l_4\} \leq t_a + l_1 + l_3$$

$$t_a + u_1 + u_3 \leq t_a + u_1 + max\{u_3, u_4\}$$

The range of values for t'_c in N' is the superset of t_c in N. The same result can be derived if b and \$2 fire in N or b fires in N'. This means that the abstracted TEL structure, N', produces a superset of timed traces of the unabstracted TEL structure, N. The same result can be obtained if the above analysis is applied to sequencing events where the sizes of their presets and postsets are greater than 2. If there exist multiple conflict places in the preset and postset of the sequencing events as shown in Figure 5.17(a), the TEL can be decomposed into equivalent TELs as shown in Figure 5.17(b). Each of them has the structure where the reduction in Figure 5.18 can be applied. Therefore, after merging the sequencing events, the TEL produces a superset of traces of the one shown in Figure 5.17(a). According to the definition of safe transformation, Reduction 8 is safe. □

Similarly, the TEL structure shown in Figure 5.19(a) can also be decomposed into four TEL structures shown in Figure 5.19(b). If the reduction shown in Figure 5.20 is safe, merging the two sequencing events in Figure 5.19(a) is also safe. The above discussion is formalized and proved to be safe in the following definition and lemma.

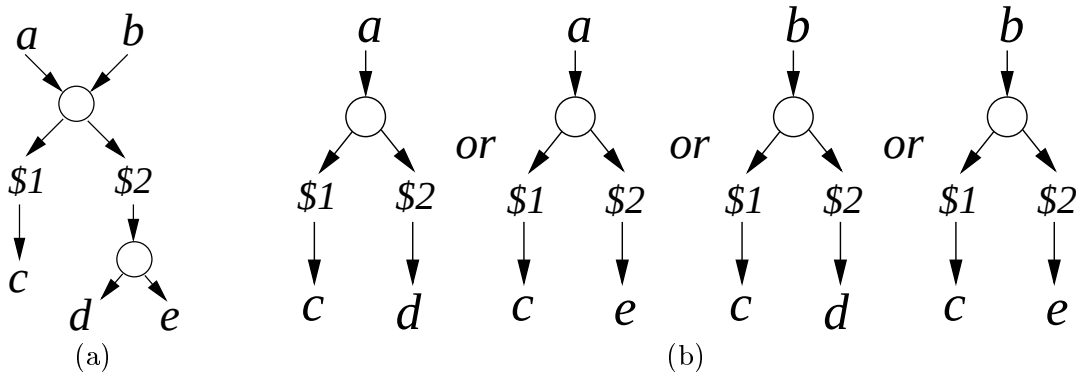

Figure 5.19. Decompose a TEL into TELs where Reduction 9 can be applied.

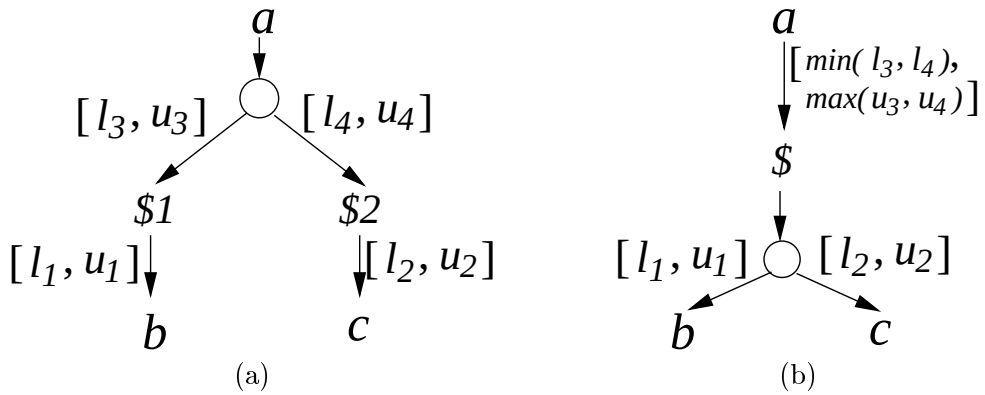

Figure 5.20. An example of Reduction 9.

Reduction 9 (Extension of Reduction 5 to TELs with conflicts) *If there exist two sequencing events* \$1 *and* \$2 *in a TEL structure* N *where* \$1 *and* \$2 *are in conflict, and* `enabling_set`(\$1) = `enabling_set`(\$2). *A new TEL structure* N' *can be derived from* N *as follows:*

- $E' = E - \{\$1, \$2\} \cup \{\$\}$,
- *for each rule* $r_i = \langle \$1, e_i, l_i, u_i \rangle \in \$1\bullet$ *and* $r_j = \langle \$2, e_j, l_j, u_j \rangle \in \$2\bullet$, *they are changed to*

$$r_i' = \langle \$, e_i, l_i, u_i \rangle \quad \textit{and} \quad r_j' = \langle \$, e_j, l_j, u_j \rangle$$

- $R' = (R - \{r_m, r_n\}) \cup \{r\}$ *where* $r_m = \langle a, \$1, l_m, u_m \rangle \in \bullet\$1$, $r_n = \langle a, \$2, l_n, u_n \rangle \in \bullet\$2$, *and* $r = \langle a, \$, min(l_m, l_n), max(u_m, u_n) \rangle$

Lemma 5.2.4 *Reduction 9 is a safe transformation.*

Proof: Consider the TEL structure, N, shown in Figure 5.20(a) and its abstracted counterpart, N', shown in Figure 5.20(b). There are two possible untimed traces produced by N: $\{a\$1b, a\$2c\}$. The untimed trace set produced by N' also has two possible untimed traces: $\{a\$b, a\$c\}$. It is obvious that these two nets have the same untimed traces after all sequencing events in the traces are deleted, so the first condition is satisfied. Next, we must show that the timed traces produced by N' contains all the timed traces produced by N with the sequencing event deleted. Consider a timed trace $x = e_1 e_2 \ldots$ where $e_i = (a, t_a)$, $e_j = (\$1, t_{\$1})$, and $e_k = (b, t_b)$ with $i < j < k$. The value of $t_{\$1}$ falls in the following range:

$$t_a + l_3 \leq t_{\$1} \leq t_a + u_3 \tag{5.23}$$

The value of t_b comes from the range:

$$t_{\$1} + l_1 \leq t_b \leq t_{\$1} + u_1 \tag{5.24}$$

Substituting Equation 5.23 into Equation 5.24 yields:

$$t_a + l_1 + l_3 \leq t_b \leq t_a + u_1 + u_3$$

After abstraction, the value of $t_\$$ comes from the range:

$$t_a + min\{l_3, l_4\} \leq t_\$ \leq t_a + max\{u_3, u_4\}. \tag{5.25}$$

and the value of t_b still comes from the range defined in Equation 5.24 where $t_{\$1}$ is defined in Equation 5.25. Substituting Equation 5.25 into Equation 5.24 yields:

$$t_a + l_1 + min\{l_3, l_4\} \leq t'_b \leq t_a + u_1 + max\{u_3, u_4\} \tag{5.26}$$

Since

$$t_a + l_1 + min\{l_3, l_4\} \leq t_a + l_1 + l_3$$

$$t_a + u_1 + u_3 \leq t_a + u_1 + max\{u_3, u_4\}$$

the range of value for t'_b is a superset of t_b. The same result can be derived if the above analysis is applied to the timed trace that contains events a, \$2, and c. This means that the abstracted TEL structure, N', produces a superset of timed traces in the unabstracted TEL structure, N. If there exist multiple conflict places in the preset and postset of the

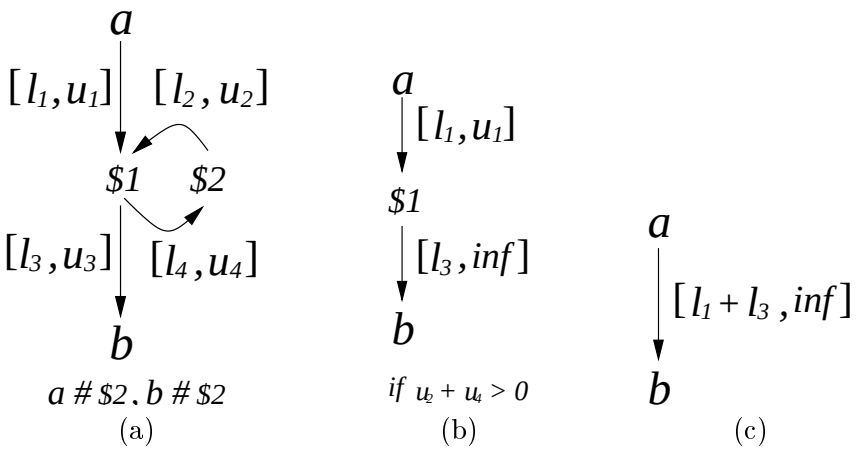

Figure 5.21. An example of Reduction 10.

sequencing events as shown in Figure 5.19(a), the TEL can be decomposed into equivalent TELs as shown in Figure 5.19(b). Each of them has the structure where the reduction in Figure 5.20 can be applied. Therefore, after merging the sequencing events, the abstracted TEL produces a superset of traces of the one shown in Figure 5.19(a). According to the definition of safe transformation, Reduction 9 is safe. □

Another reduction is similar to Reduction 3 where the sequencing event forms a loop as shown in Figure 5.21. In Figure 5.21, \$1 and \$2 form a loop where \$2 is in both the preset and postset of \$1 and \$1 is also in both the preset and postset of \$2. Also, \$2 conflicts with the events in both the enabling set and enabled set of \$1. In this example, \$2 conflicts with a and b. The semantics of this TEL is as follows: after firing a, \$1 can occur since event a and \$2 are in conflict. Next, either b or \$2 can fire. Before firing b, \$2 can fire an infinite number of times. If the upper bound delay along the path $\$1 \rightarrow \$2 \rightarrow \$1$ is greater than 0, this means that b may not fire for an infinite amount of time. After removing \$2 and changing $\texttt{upper}(r)$ to ∞ for all $r \in \$1\bullet$, the system still produces the same set of timed traces. This can be proven by the following lemma.

Reduction 10 (Reduction 10) *If there exist two sequencing event* \$1 *and* \$2 *in a TEL structure* N, \$2 *is in both enabling set and enabled set of* \$1, *and* \$2 *conflicts with the event* $e \in enabling\_set(\$1)$ *and* $e \in enabled\_set(\$1)$. *A new TEL structure,* N', *can be derived from* N *as follows:*

- $R' = R - \{r_1, r_2\}$ *where* $r_1 = \{\$2, \$1, l_1, u_1\} \in \bullet\$1$ *and* $r_2 = \{\$1, \$2, l_2, u_2\} \in \$1\bullet$,

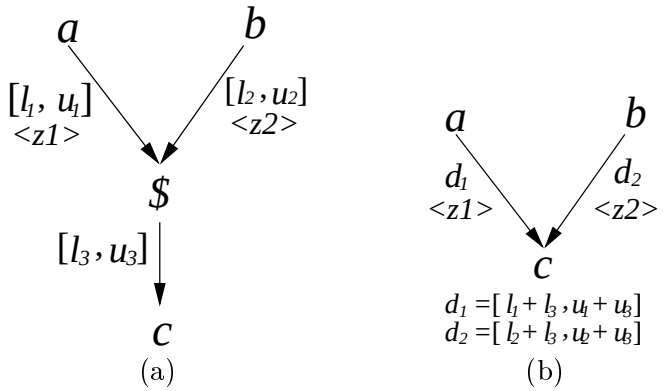

Figure 5.22. An example of Reduction 1 for a TEL with levels.

- *if* $u_1 + u_2 > 0$, $\texttt{upper}(r) = \infty$ *for all* $r \in \bullet\$1$.

Lemma 5.2.5 *Reduction 10 is a safe transformation.*

Proof: Consider the TEL N shown in Figure 5.21(a) and the abstracted TEL N' shown in Figure 5.21(b). There is only one untimed trace produced by N: $\{a\$1(\$2\$1)^*b\}$. $(\$1\$2)^*$ means that the sequence of \$2 and \$1 can occur zero or infinite number of times. After deleting the sequencing events, the untimed trace becomes ab which is what is produced by N'. In N, \$1 can fire once or an infinite number of times before b fires. The first time \$1 fires, $t_a + l_1 \leq t_{\$1} \leq t_a + u_1$. After an infinite number of times of firing \$1, $t_{\$1} = \infty$. Therefore, The value of t_b comes from the following ranges:

$$t_a + l_1 + l_3 \leq t_b \leq \infty \tag{5.27}$$

It is obvious that t_b determined in Equation 5.27 is the same as that obtained from N'. Therefore, N and N' produce the same set of timed traces. According to the definition of safe transformations, this reduction is safe. □

5.3 Dealing With Levels

Reduction 1, 2, 4, and 5 can be extended to TEL structures with levels. However, extension of each reduction to handle levels has different constraints. For Reduction 1 and 2, it is required that all rules in the postset of a sequencing event must have a level of **true**; otherwise, the timing behavior changes. Figure 5.22 shows an example of Reduction 1 for a TEL with levels. Figure 5.23 shows an illegal application of Reduction 1 for a

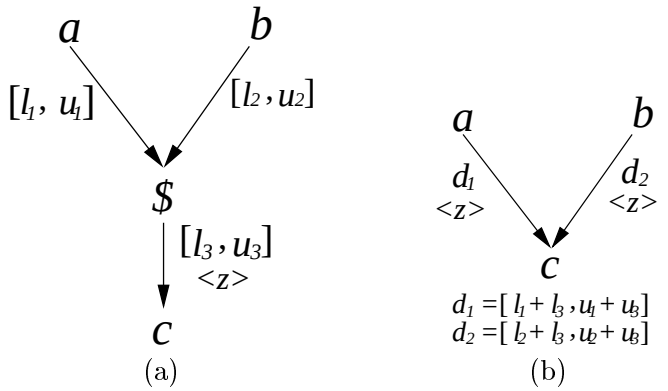

Figure 5.23. Reduction 1 for TEL with levels that causes a change in timing.

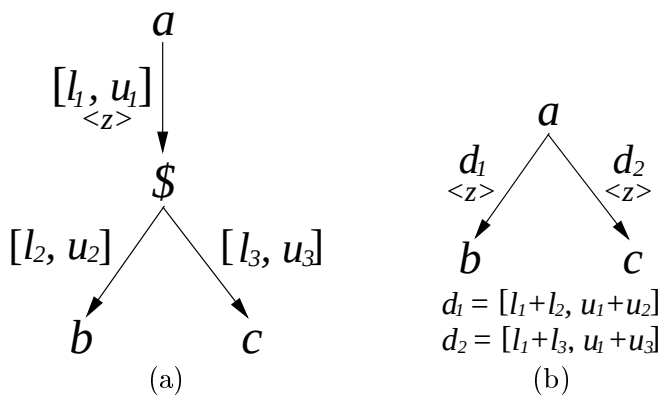

Figure 5.24. An example of Reduction 2 for a TEL with levels.

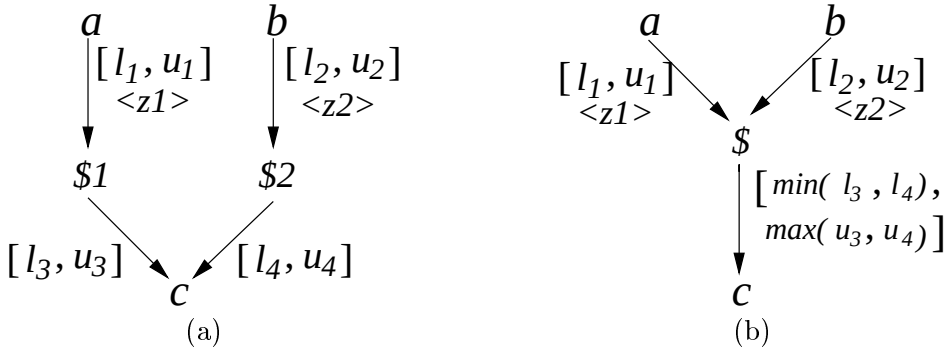

Figure 5.25. An example of Reduction 4 for a TEL with levels.

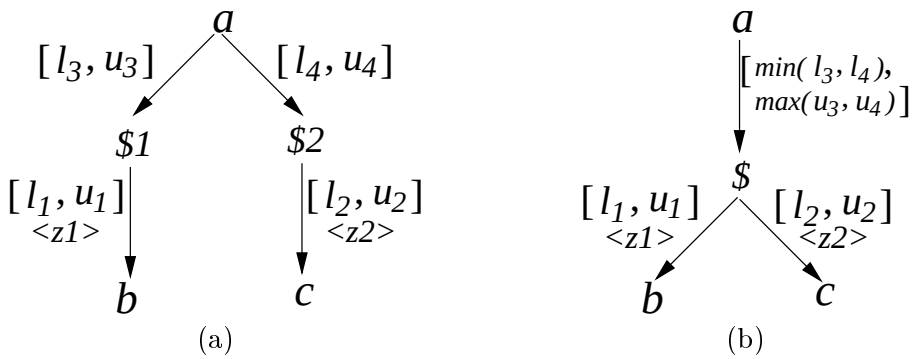

Figure 5.26. An example of Reduction 5 for a TEL with levels.

TEL with levels where the rule in the postset of the sequencing event has a level. In Figure 5.23(a), c fires between l_3 and u_3 time units after the level z evaluates to true, while in Figure 5.23(b), c fires between $max(l_1+l_3, l_2+l_3)$ and $max(u_1+u_3, u_2+u_3)$ time units after the level z evaluates to true. Obviously, timing constraints have changed after the redunction. Figure 5.24 shows Reduction 2 for a TEL with levels. For Reduction 4, it is required that all rules in the postsets of sequencing events must have the same level expression because it is impossible to merge two rules with different levels. For Reduction 5, it is required that all rules in the presets of sequencing events must have the same level expression. Figure 5.25 shows an example of Reduction 4 for a TEL with levels. Figure 5.26 shows an example of Reduction 5 for a TEL with levels.

CHAPTER 6

REMOVING REDUNDANT RULES

A rule puts a constraint on the firing time of an event. If the event is enabled by multiple rules, and the firing time of the event is independent of one of them, then that rule is redundant. Informally, a rule is redundant in a TEL structure if its omission does not change the behavior specified. In other words, given a TEL structure, N, and a rule, r, a new TEL structure, N', constructed by removing r from its rule set has the same timed traces as N. The general approach to determine if a rule is redundant requires finding the minimum and maximum separation times between any two events, however, the calculation of minimum and maximum separation times is an exponential problem. This approach is very undesirable because the purpose of determining and removing redundant rules is to reduce the computational cost of state space exploration. Therefore, an approximate method is necessary. For TEL structures without levels, we can determine if a rule is redundant by analyzing the structure of the net around the rules of interest and the relations of their timing constraints. If the rules have levels, an approximate algorithm is necessary to calculate the worst-case minimum and maximum separation time between every two events in a TEL structure so that this information can be used to determine the redundancy of a rule. This chapter starts with the formal definition of redundant rules. In the second section, a structural analysis method is described to check if a rule is redundant for conflict-free TEL structures without levels. The following section describes how conflicts in the TEL structures affect the above methods. In the last section a method to determine redundancy of rules with levels is described based on the assumption that the minimum and maximum separation time between every two events in a TEL structure is available.

6.1 Definition of Redundant Rules

A redundant rule is defined to be a rule such that removing it from a TEL structure does not change the timed trace set produced, so the first requirement that a redundant

rule must satisfy is that removing it does not reduce the untimed traces specified. Secondly, timing must be preserved exactly for the system to produce the same timed traces. In Chapter 2, a rule is said to be enabled if its enabling event has fired and the boolean expression of its level evaluates to **true**. A rule is satisfied if it is enabled and the timer of the rule exceeds the lower bound of the timing constraint. A rule is expired if it is enabled and the timer of the rule exceeds the upper bound of the timing constraint. If an event is enabled by multiple rules and suppose there is no conflict among the enabling events, the event is enabled to fire when all enabling rules are satisfied. The event is forced to fire before all enabling rules become expired. Note that some may be expired as long as at least one has not expired. Therefore, the rules in the preset of the event determine a range of firing time of the event. The lower bound of this range is decided by the rule which is the last one to become satisfied. The upper bound of this range is decided by the rule which is the last one to become expired. If a rule in the preset of the event is neither the last one becoming satisfied nor the last one becoming expired, that means it does not constrain the timing behavior of the event so it is redundant. Recall in Chapter 2, $\texttt{EFT}(e \leftarrow r_1, \ \cdots, \ r_n)$ and $\texttt{LFT}(e \leftarrow r_1, \ \cdots, \ r_n)$ define the lower and upper bound of the range of firing time of e decided by $r_1, \ \cdots \ r_n$, respectively. Redundant rules are formally defined as follows:

Definition 6.1.1 (Redundant Rules) *Given a TEL structure, N, an event $e \in E$ such that* $\texttt{size}(\bullet e) \geq 2$ *and a rule $r \in \bullet e$, r is redundant if removing r does not reduce the untimed traces produced by N, and the following equations are satisfied:*

$$\texttt{EFT}(e \leftarrow r_1, \ \cdots \ r, \ \cdots, \ r_n) = \texttt{EFT}(e \leftarrow r_1, \ \cdots, \ r_n)$$
$$\texttt{LFT}(e \leftarrow r_1, \ \cdots \ r, \ \cdots, \ r_n) = \texttt{LFT}(e \leftarrow r_1, \ \cdots, \ r_n)$$

From the above discussion, the omission of redundant rules in a TEL does not have any impact on the system behavior. However, the existence of redundant rules increases the computational cost to explore the state space because more rules need to be considered. Therefore, it is highly desirable and necessary to remove redundant rules whenever possible to reduce the cost of state space exploration. Recall the definition of safe transformations that a transformation is safe if the transformed system produces a superset of timed traces of the system before the transformation. According to this definition, it is obvious that removing redundant rules is another kind of safe transformation in that

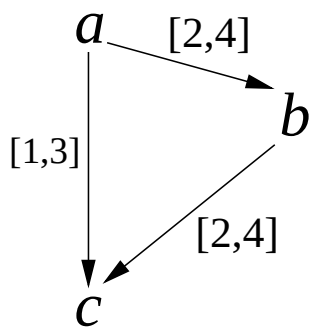

Figure 6.1. A TEL fragment with a redundant rule.

the system produces the same timed traces after the redundant rules are removed.

Figure 6.1 shows a simple example of a redundant rule. The TEL fragment in the example has three rules: $r_1 = \langle a, c, 1, 3\rangle$, $r_2 = \langle b, c, 2, 4\rangle$, and $r_3 = \langle a, b, 2, 4\rangle$. This TEL structure produces only one untimed trace $\{abc\}$. In this TEL fragment, either r_1 or r_2 is potentially redundant because the untimed trace set produced by the TEL after r_1 or r_2 is removed includes abc. Suppose t_a is when event a fires. We have

$$\texttt{EFT}(b \leftarrow r_3) = t_a + 2 \quad \text{and} \quad \texttt{LFT}(b \leftarrow r_3) = t_a + 4$$

and

$$\texttt{EFT}(c \leftarrow r_1) = t_a + 1 \quad \text{and} \quad \texttt{LFT}(c \leftarrow r_1) = t_a + 3$$

From above equations, we can derive the earliest and latest firing time of c determined by r_2:

$$\texttt{EFT}(c \leftarrow r_2) = \texttt{EFT}(b \leftarrow r_3) + 2 \quad \text{and} \quad \texttt{LFT}(c \leftarrow r_2) = \texttt{LFT}(b \leftarrow r_3) + 4$$

Since $\texttt{EFT}(b \leftarrow r_3)$ and $\texttt{LFT}(b \leftarrow r_3)$ are available, the above two equations can be reformulated as follows:

$$\texttt{EFT}(c \leftarrow r_2) = t_a + 4 \quad \text{and} \quad \texttt{LFT}(c \leftarrow r_2) = t_a + 8$$

Since c is enabled by r_1 and r_2, the range of firing time of c determined by r_1 and r_2 is defined as follows:

$$\texttt{EFT}(c \leftarrow r_1,\ r_2) = max(\texttt{EFT}(c \leftarrow r_1),\ \texttt{EFT}(c \leftarrow r_2))$$

$$\texttt{LFT}(c \leftarrow r_1,\ r_2) = max(\texttt{LFT}(c \leftarrow r_1),\ \texttt{LFT}(c \leftarrow r_2))$$

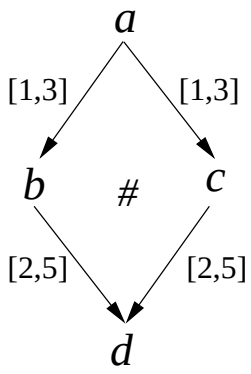

Figure 6.2. An example where removing a rule changes the untimed behavior of a TEL.

Obviously, in this example,

$$\texttt{EFT}(c \leftarrow r_1,\ r_2) = \texttt{EFT}(c \leftarrow r_2) \quad \text{and} \quad \texttt{LFT}(c \leftarrow r_1,\ r_2) = \texttt{LFT}(c \leftarrow r_2)$$

This indicates that the timing behavior of c is solely constrained by rule r_2, which makes rule r_1 redundant. A similar analysis can be applied to r_2, and the result shows it is not redundant.

An example shown in Figure 6.2 illustrates a situation where removing a rule changes the untimed semantics of the system even if timing is preserved exactly. In the example, the event d is enabled by two rules: $r_1 = \langle b, d, 2, 5\rangle$ and $r_2 = \langle c, d, 2, 5\rangle$, and b conflicts with c. If one of r_1 and r_2 fires, d fires sometime between 3 and 8 time units after a has fired. However, removing either one of the them changes the untimed traces produced by the TEL. For example, if r_1 is removed, the TEL does not produce untimed traces containing events b and d. Similar result is obtained if r_2 is removed. Therefore, both r_1 and r_2 cannot be redundant.

6.2 Redundancy Check for Conflict-Free TEL Structures Without Levels

If an event is enabled by multiple rules, according to the definition of redundant rules, it is necessary to know when these rules become satisfied and expired, and the order of when they become satisfied and expired to determine if one of them is redundant. This requires the information of the minimum and maximum separation times between every pair of events in the system. Unfortunately, calaulation of the minimum and maximum separation times has an exponential complexity in the size of the system. This makes the genaral method of removing redundant rules as hard as state space exploration. Since

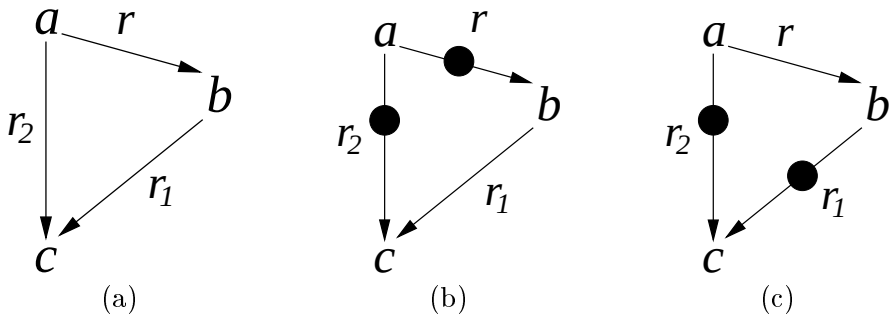

Figure 6.3. The triangle structure for redundancy check.

the information of the minimum and maximum separation times between an arbitrary pair of events is not available until the exponential procedure is done, we have to restrict the general analysis approach to a subset of TEL structures so that calaulation of the minimum and maximum separation times between two events is not required. Instead, such information is derived by a simple structural analysis. This section describes the method to determine redundant rules for TEL structures without conflicts and levels. This method explores the topology of TEL structures to find if there exists a certain structure that the separation time among the events in the enabling set of an event can be obtained easily.

The example shown in Figure 6.1 in the last section points out one kind of structure that can be used to determine redundant rules. This structure can be generalized in Figure 6.3(a). In this triangle structure, an event is enabled by two rules r_1 and r_2, and there is another rule r that connects the enabling events of r_1 and r_2 so that the separation time between enabling events of r_1 and r_2 can be obtained by just checking the timing constraint of r. Since the separation time between enabling events of r_1 and r_2 is available, it can be determined if either r_1 or r_2 is redundant as shown in the following lemma.

Lemma 6.2.1 (Redundancy Check for Rules not in the Initial Marking) *Given A TEL structure, N, that contains an event $e \in E$ and two rules $r_1 \in {}^\bullet e$ and $r_2 \in {}^\bullet e$. If there exists a rule $r \in R$ such that* $\texttt{enabling}(r) = \texttt{enabling}(r_2)$ *and* $\texttt{enabled}(r) = \texttt{enabling}(r_1)$*, r_2 is redundant if the following equations are satisfied:*

$$\texttt{lower}(r_1) + \texttt{lower}(r) \geq \texttt{lower}(r_2) \quad \textit{and} \quad \texttt{upper}(r_1) + \texttt{upper}(r) \geq \texttt{upper}(r_2)$$

Rule r_1 *is redundant if* $\texttt{size}(\bullet b) = 1$ *and the following equations are satisfied:*

$$\texttt{lower}(r_1) + \texttt{lower}(r) \leq \texttt{lower}(r_2) \quad \textit{and} \quad \texttt{upper}(r_1) + \texttt{upper}(r) \leq \texttt{upper}(r_2)$$

To fire c in Figure 6.3(a), a needs to fire first resulting in a marking shown in Figure 6.3(b), then b fires resulting a marking in Figure 6.3(c). If the initial marking is as shown in Figure 6.3(b), the rule r and r_2 are initially enabled. This indicates implicitly that a has fired. Firing c requires b to fire first. In this case, all three rules are involved in firing c, the firing time of c is the same in the first execution cycle as that in the following execution cycles. Therefore, for the purpose of redundancy check, the TEL shown in Figure 6.3(b) is consider the same as that in Figure 6.3(a). However, if the TEL has the initial marking shown in Figure 6.3(c), the firing time of c is different in the first execution cycle where r_1 and r_2 are initially enabled. The reason is that r is not involved in the firing of c in the initial marking. A rule is redundant if it does not affect the firing time of the event enabled by the rule in every execution cycle when it is enabled to fire, so one more step to check if r_1 or r_2 is redundant in the initial marking needs to performed. Besides using Lemma 6.2.1 to decide if either r_1 or r_2 is redundant, it is also necessary to check that either one is also redundant in the initial marking. To make an initially enabled rule r redundant, there must exist another rule that constrains the timing behavior in such a way that it does not change after r is removed. The following lemma gives the conditions to check if a rule in the initial marking is redundant.

Lemma 6.2.2 (Redundancy Check for Rules in the Initial Marking) *Given a TEL structure,* N*, in which there is a* $r \in R$ *and also* $r \in M_0$*, if there exists another* r_x*, and the following equations are true:*

$$\texttt{lower}(r_x) \geq \texttt{lower}(r) \quad \textit{and} \quad \texttt{upper}(r_x) \geq \texttt{upper}(r)$$

then r *is redundant in the initial marking* M_0*.*

This triangle structure only involves three rules. Redundancy check on this kind of TEL structure can be handled very fast, but also limits the applicability of this method. Figure 6.4(a) shows an alternative TEL to those shown in Figure 6.3 based on the same idea. In this kind of TEL structure, there is not a single rule from a to b. Instead, a reaches b through a path. A *path* P from a to b is a sequence of rules so that a can reach b through P by traversing the graph. Lemma 6.2.1 can still be applied to this kind

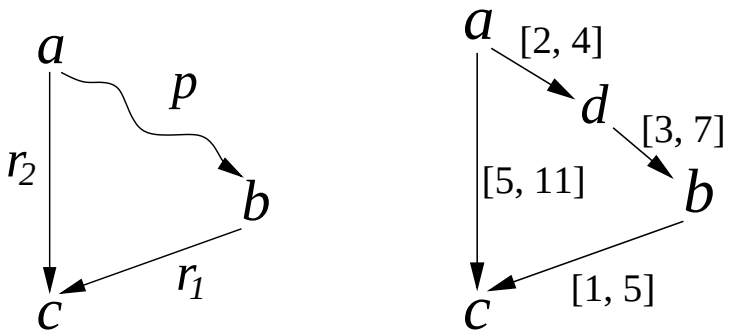

Figure 6.4. Extension of the triangle structure for redundancy check.

of TEL, only $\texttt{lower}(r)$ and $\texttt{upper}(r)$ in Lemma 5.2.5 need to be replaced by $\texttt{lower}(P)$ and $\texttt{upper}(P)$. $\texttt{lower}(P)$ and $\texttt{upper}(P)$ of a path $P = r_1 \rightarrow r_2 \cdots \rightarrow r_n$ are defined as follows:

$$\texttt{lower}(P) = \texttt{lower}(r_1) + \texttt{lower}(r_2) + \cdots + \texttt{lower}(r_n)$$

$$\texttt{upper}(P) = \texttt{upper}(r_1) + \texttt{upper}(r_2) + \cdots + \texttt{upper}(r_n)$$

And also to check if r_1 is redundant, it needs to change the condition that $\texttt{size}(\bullet b) = 1$ in Lemma 5.2.5 to $\texttt{size}(\bullet\texttt{enabled}(r)) = 1$ for all r in P. Figure 6.4(b) shows an example of this kind of TEL structure. In the example, path P from a to b is $(r_3\ r_4)$, where $r_3 = \langle a, d, 2, 4\rangle$ and $r_4 = \langle d, b, 3, 7\rangle$. $\texttt{lower}(P) = 5$ and $\texttt{upper}(P) = 11$. Since $r_1 = \langle b, c, 1, 5\rangle$ and $r_2 = \langle a, c, 5, 11\rangle$, and $\texttt{lower}(r_1)+\texttt{lower}(P) > \texttt{lower}(r_2)$ and $\texttt{upper}(r_1)+\texttt{upper}(P) > \texttt{upper}(r_2)$, according to Lemma 6.2.1, r_2 is redundant. However, redundancy check for this kind of TEL structure requires calculating the transitive closure among all events in a TEL to decide if there is a path between a pair of events. Calculation of the transitive closure has complexity $O(n^3)$ where n is the number of events in the TEL. If n is big, finding the transitive closure is very computationally expensive, therefore the algorithm handling the more general structures should be used in a limited way and preferably when no other transformations work.

6.3 Extending Redundancy Check to Handle Conflicts

The last section describes a method to determine redundant rules in a TEL structure without conflicts. This section extends the method to TELs with conflicts. In a TEL with conflicts, a rule that is identified redundant according to Lemma 6.2.1 and Lemma 6.2.2

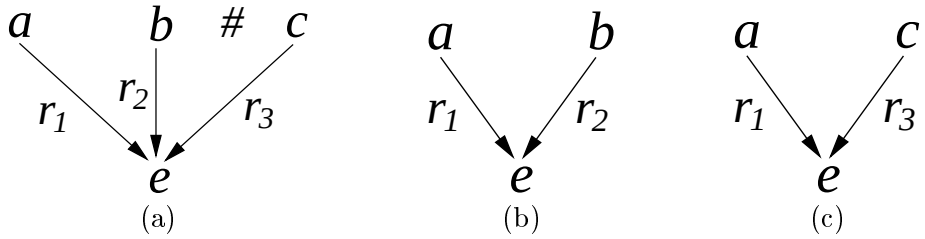

Figure 6.5. Redundancy check with an enabling conflict set.

is called a *potentially* redundant rule. If a potentially redundant rule does not change the untimed semantics of a TEL, it becomes a *truly* redundant rule. It is obvious that a potentially redundant rule is automatically a truly redundant rule if the TEL has no conflicts. There are two groups of conflicts that need to be considered when deciding if a potentially redundant rule r is truly redundant: the conflicts among the events in the enabling set of the event $\texttt{enabled}(r)$ and conflicts involved with $\texttt{enabled}(r)$.

First, consider the conflicts among the events in the enabling set of an event. As shown in Figure 6.2, if all events in the enabling set of another event e conflicts with each other, in other words, there is only one conflict place in the preset of e, none of rules in $\bullet e$ can be redundant because removing any one of them reduces the specified untimed behavior. Figure 6.5(a) shows an example where the TEL, N, has a conflict between b and c. e is enabled by r_1, r_2, and r_3. Since b conflicts with c, either firing of a and b or firing of a and c is required to fire e. N can be decomposed to two equivalent TELs shown in Figure 6.5(b) and Figure 6.5(c). Only one of the two TEL structures is active during an execution cycle. Suppose r_1 is potentially redundant only due to r_2, and it is removed from both the TELs. In the case shown in Figure 6.5(b), the firing time of e does not change. However, in the case shown in Figure 6.5(c), e may fire sooner than specified because it is possible that $\texttt{EFT}(e \leftarrow r_1,\ r_3) \neq \texttt{EFT}(e \leftarrow r_3)$ and $\texttt{LFT}(e \leftarrow r_1,\ r_3) \neq \texttt{LFT}(e \leftarrow r_3)$. Since the firing time of e may change after r_1 is removed, r_1 is not truly redundant. r_1 is truly redundant if it is redundant due to both r_2 and r_3. Let an enabling conflict set of an event e be a set of rules in the preset of e whose enabling events conflict with each other, and an enabled conflict set of an event e be a set of rules in the postset of e whose enabled events conflict with each other. If an enabling conflict set is used to decide the redundancy of another rule, that rule is redundant if it is redundant due to all rules in the enabling conflict set. Now suppose r_2 is potentially redundant due to r_1. However,

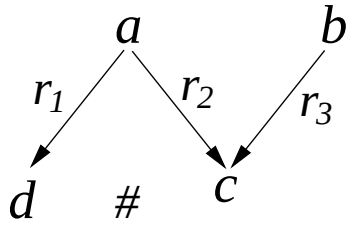

Figure 6.6. Redundancy check with an enabled conflict set.

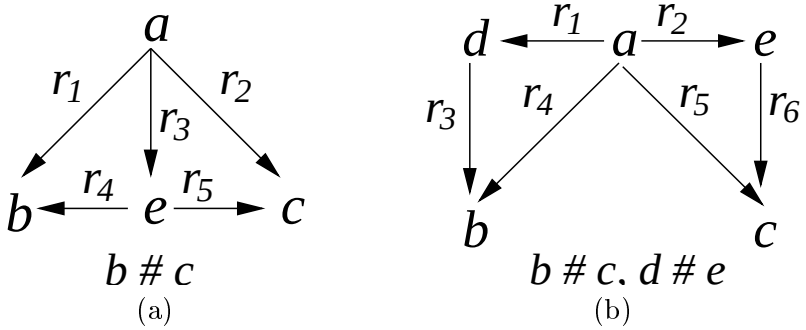

Figure 6.7. Redundancy check for rules in the same postset of an event where their enabled events are in conflict.

removing r_2 changes the semantics of the TEL structure. The reason is as follows: before removing r_2, e fires after a and b, or a and c have fired. After removing r_2, e fires only after a and c have fired. Since b and c are in conflict, this may cause deadlock. If both r_2 and r_3 are redundant and removed, the system produces the same timed traces.

In summary, a rule cannot make another rule in the same enabling conflict set redundant. If a rule in an enabling conflict set makes another rule not in this enabling conflict set redundant, then all rules in the conflict set must make that rule redundant for it to be truly redundant. On the other hand, a rule in an enabling conflict set is truly redundant if all rules in the same conflict set are redundant.

Now consider the situation where the enabled event of a potentially redundant rule conflicts with other events. Figure 6.6 shows an example where the TEL, N, contains a conflict between c and d that are enabled by a. Suppose r_2 is redundant due to r_3. The untimed trace set produced by N is $\{abc, abd, bac, bad, adb\}$. After removing r_2, the untimed trace set produced by N' is $\{abcd, abdc, bacd, badc, adbc, bcad\}$. The change in the untimed semantics is because the conflict between c and d disappears after removing r_2. In N, a choice is made between c and d after a fires that only one of them can fire. In

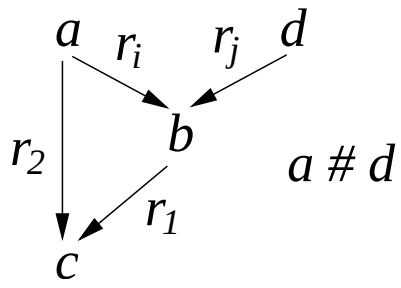

Figure 6.8. The enabling conflicts that affect the conditions for redundancy check.

N', no choice is made after a fires because there is no conflict between c and d. Now both c and d can fire in parallel after a does. Now suppose r_3 is redundant because of r_2. The untimed trace set produced by N is still $\{abc, abd, bac, bad, adb\}$. After removing r_3, the untimed trace set produced by N' is the same. Therefore, r_3 is truly redundant in this case. Based on the above discussion, a potentially redundant rule r is truly redundant if it is not in any enabled conflict set of any events in a TEL. However, in the situation shown in Figure 6.7(a), both r_1 and r_2 are in the enabled conflict set of a, and if they are redundant and removed together, the new TEL structure still produces the same timed traces. This is because the TELs before and after r_1 and r_2 are removed produces the same two untimed traces: $\{aeb,\ aec\}$. And also removing r_1 and r_2 does not change the timing behavior of b and c, so the new TEL produces the same timed traces. Figure 6.7(b) shows a TEL alternative to the one shown in Figure 6.7(a). In this case, if both r_4 and r_5 are redundant and removed together, the behavior described by this TEL does not change. The reason is the same for the TEL shown in Figure 6.7(a). If r_3 and r_6 are redundant based on their own triangle redundancy check, they can be removed together, and it is necessary to create two new conflicts $e\#b$ and $d\#c$ to keep the same untimed traces. Based on the above discussion, it is required that removing a rule does not change the conflict relation of any pair of events.

In Lemma 6.2.1 and Figure 6.3, the conditions using timing constraints of rules r, r_1, and r_2 to decide if r_2 is redundant is based on an assumption that there is no conflict among the enabling events of the rules in the preset of b. If it is not true as shown in Figure 6.8 where a conflicts with d, then checking conditions defined in Lemma 6.2.1 is no longer correct. The reason is explained as follows: since a conflicts with d, the earliest firing time of b has two values shown as follows:

$$\texttt{EFT}(b \leftarrow r_i) = t_a + \texttt{lower}(r_i)$$

$$\texttt{EFT}(b \leftarrow r_j) = t_d + \texttt{lower}(r_j)$$

where t_a and t_d are the earliest firing time of a and d.

$$\texttt{EFT}(c \leftarrow r_1,\ r_2) = max(t_a + \texttt{lower}(r_2),\ t_b + \texttt{lower}(r_1)) \tag{6.1}$$

The earliest firing time of c decided by only r_1 is defined as follows:

$$\texttt{EFT}(c \leftarrow r_1) = t_b + \texttt{lower}(r_1) \tag{6.2}$$

If $t_b = \texttt{EFT}(b \leftarrow r_i)$, then

$$\texttt{EFT}(c \leftarrow r_1) = t_a + \texttt{lower}(r_i) + \texttt{lower}(r_1)$$

$$\texttt{EFT}(c \leftarrow r_1,\ r_2) = max(t_a + \texttt{lower}(r_2),\ t_a + \texttt{lower}(r_i) + \texttt{lower}(r_1))$$

According to Lemma 6.2.1,

$$\texttt{EFT}(c \leftarrow r_1) = \texttt{EFT}(c \leftarrow r_1,\ r_2) \tag{6.3}$$

However, if $t_b = \texttt{EFT}(b \leftarrow r_j)$, then

$$\texttt{EFT}(c \leftarrow r_1) = t_d + \texttt{lower}(r_j) + \texttt{lower}(r_1)$$

$$\texttt{EFT}(c \leftarrow r_1,\ r_2) = max(t_a + \texttt{lower}(r_2),\ t_d + \texttt{lower}(r_j) + \texttt{lower}(r_1))$$

It is possible that

$$\texttt{EFT}(c \leftarrow r_1) \neq \texttt{EFT}(c \leftarrow r_1, r_2)$$

Therefore, it is required that there are no conflicts among the events in the enabling set of b for the checking conditions defined in Lemma 6.2.1 to be correct.

6.4 General Redundancy Check for TEL structures

This section discusses the general method to determine the redundancy of a rule based on an assumption that a conservative estimate of the minimum and maximum separation time between two events is available. We first show two simple redundancy checks for TEL structures with levels that do not require such information. As shown in Figure 6.9(a), rule r and r_1 now have levels z and $z1$, respectively. If r_2 is redundant based on Lemma 6.2.1 ignoring levels, r_2 is still redundant when the levels are taken into consideration. The reason is as follows: if r_2 is checked to be redundant ignoring levels, that means r_1 is the last one to become satisfied and expired. Adding a level into r_1

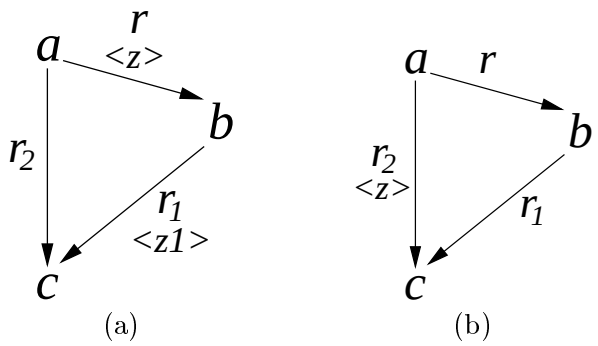

Figure 6.9. Two simple redundancy checks for TEL structures with levels.

delays r_1 to become satisfied and expired so it does not change the redundancy of r_2. Similarly, in the example shown in Figure 6.9(b), if r_1 is redundant based on Lemma 6.2.1 ignoring the level on r_2, r_1 is still redundant when the level on r_2 is considered.

The above two simple cases are applicable only when r_2 has a level but r and r_1 does not, or vice versa. If this is not true, the information of separation time between two events is required. In the first section, if an event is enabled by multiple rules, one of them is redundant if it is neither the last one to become satisfied nor the last one to become expired. This requires knowing when a rule becomes enabled. To make the following discussion easier, we define a reference event e_{ref} and the minimum and maximum separation times between e_{ref} and another event e are represented as t_{e_min} and t_{e_max}, respectively. Therefore, $t_{e\_min} = \texttt{min\_st}(e_{ref}, e)$ and $t_{e\_max} = \texttt{max\_st}(e_{ref}, e)$.

A rule is enabled when the enabling event has fired and the level evaluates to true. As described in Chapter 4, if the boolean expression of a level only consists of a product, for example $z = ab$, z evaluates to true at $t_{z\_min} = max(t_{a\_min}, t_{b\_min})$ and $t_{z\_min} = max(t_{a\_max}, t_{b\_max})$. If the expression only consists of a sum term, for example $z = a+b$, z evaluates to true at $t_{z\_min} = min(t_{a\_min}, t_{b\_min})$ and $t_{z\_max} = max(t_{a\_max}, t_{b\_max})$. The time when an expression with a sum-of-product evaluates to true can be derived similarly. Given a rule $r = \langle e, f, l, u, z \rangle$, the earliest time when it is enabled is $max(t_{e\_min}, t_{z\_min})$, and the latest time when it is enabled is $max(t_{e\_max}, t_{z\_max})$. Therefore, the lower and upper bounds of the range that r becomes satisfied are

$$max(t_{e\_min}, t_{z\_min}) + l \quad \text{and} \quad max(t_{e\_max}, t_{z\_max}) + l$$

Similarly, the lower and upper bounds of the range that r becomes expired are

$$max(t_{e\_min}, t_{z\_min}) + u \quad \text{and} \quad max(t_{e\_max}, t_{z\_max}) + u$$

Suppose an event is enabled by two rules: r_1 and r_2. If the lower bound of the range when r_2 becomes satisfied is larger that the upper bound of the range when r_1 becomes satisfied and the lower bound of the range when r_2 becomes expired is larger than the upper bound of the range when r_1 becomes expired, then r_1 is redundant.

There is a special case when the level only consists of a product. If the level always evaluates to false, the rule with this level would never become enabled, so it is automatically redundant.

CHAPTER 7

EXPERIMENTAL RESULTS

The goal of the automatic abstraction techniques described in this dissertation is to avoid state space explosion in large and complex designs by partitioning designs into blocks with constrained complexity and exploring the state space of each block individually. To reduce the state space, it is necessary to remove the sequencing events and associated rules in a system whenever possible under the constraint that the system behavior is preserved conservatively.

This automatic abstraction technique has been incorporated into the VHDL and THSE compiler [86] frontend of the `ATACS` tool. In this chapter, several examples are presented, and their state space is explored using Bap, an enhanced version of the POSET timing analysis algorithm [55]. Next, the state space of each component in these examples is explored after abstraction. For the purpose of easy comparison between the runtimes of state space exploration of the whole designs and the runtimes of state space exploration of the designs using abstraction, all examples shown in this chapter have a regular structure so that they can be expanded easily. However, it does not necessarily mean that the abstraction technique is limited to only circuits with a regular structure. When a design suffers state explosion, that means that the state space of the design is too large to fit in the memory. Abstraction not only improves the runtime of state exploration, but also the memory usage because the state space of each component can be substantially smaller than the whole design. Besides showing the comparison of runtimes, the comparison of the memory usage is also shown for the whole design and each component.

7.1 Simple FIFOs

The first example is a dataless version of the precharge half buffer (PCHB) from [49]. In one experiment, we explored the state space of the whole PCHBs with different number of stages without using abstraction, and `ATACS` finishes successfully up to 7 stages. In another experiment, we explored the state space of the PCHBs with different number of

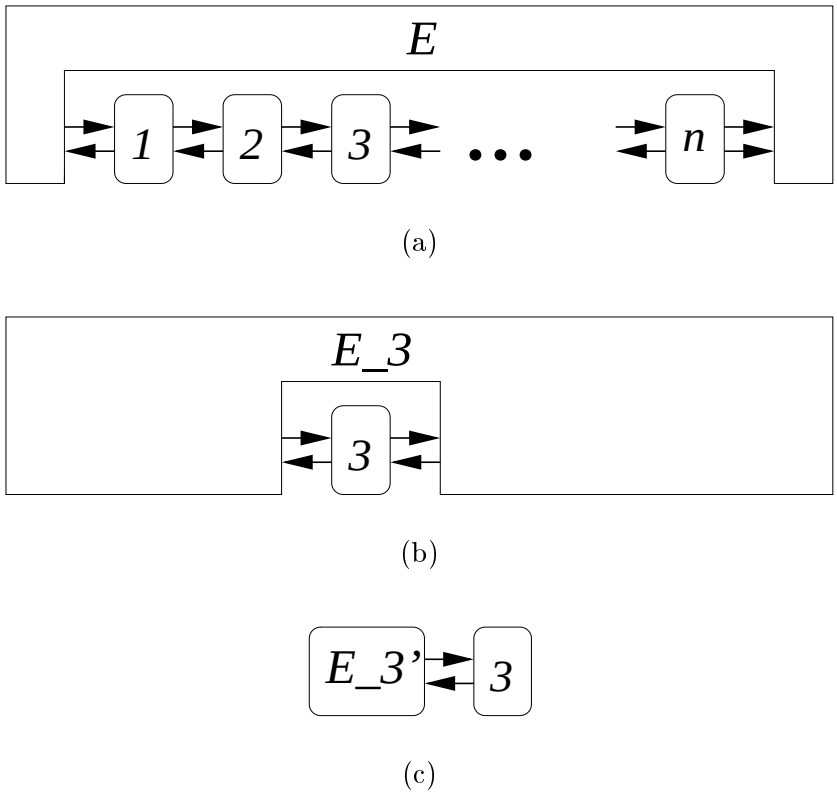

Figure 7.1. Illustration of how modular design works on each stage and how abstraction reduces the complexity of designing each stage.

stages with asbtraction on. In this experiment, a single stage of the PCHB is selected, and the rest of the stages of the PCHB and the environment for the whole PCHB are combined to be the environment for the selected stage. Before the state space of the selected stage is explored, its environment is simplified using the abstraction approach described in this dissertation. This process can be illustrated in the Figure 7.1. The same process is applied to the other stages of the PCHB. Then, the runtimes for the state space exploration of all stages in the PCHB are added together to form the time to finish the whole PCHB. Note that the runtime for each stage also includes the time for abstraction on that stage. Comparative runtimes and memory usages for the state space exploration of 1 through 9 stages are shown in Figure 7.2 and Figure 7.3, respectively. While `ATACS` can only finish PCHBs up to 7 stages on the flat design, it easily finishes 100 stages in about 6.2 minutes with a maximum memory usage of 4 MB with abstraction on.

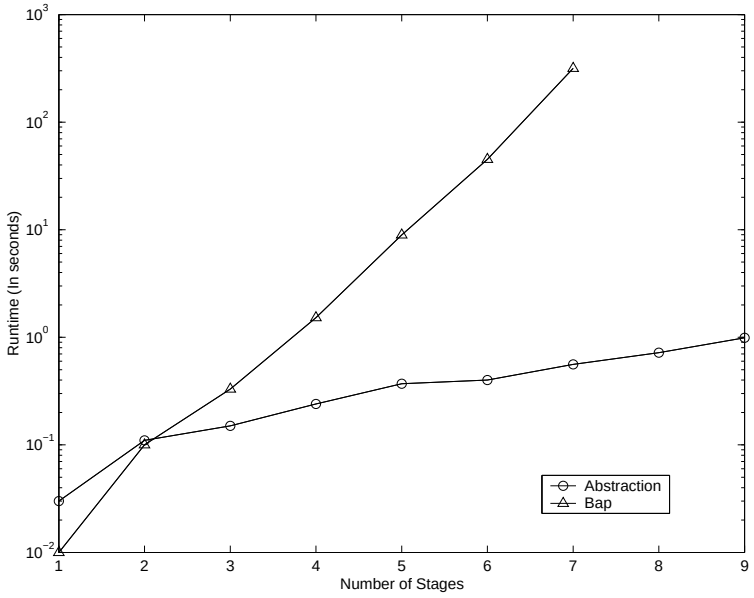

Figure 7.2. Synthesis time for PCHB example.

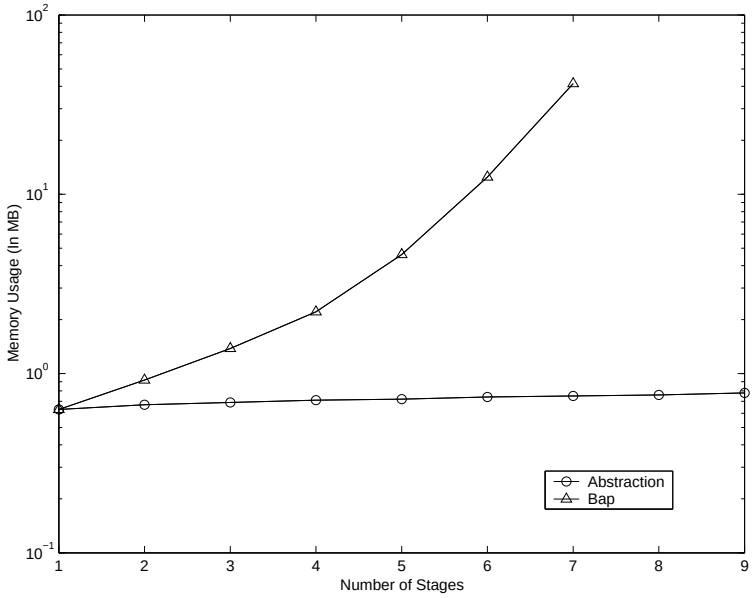

Figure 7.3. Memory usage for PCHB example.

The second example is a multiple stage controller for a self-timed FIFO from Sun Microsystems. In [58], a highly optimized hand designed timed circuit implementation is presented. The purpose of this FIFO is to compare the performance of an asynchronous FIFO with that of a clocked shift register using the same data path. The FIFO uses a pulse-like protocol to advance data along the pipeline. Aggressive timing assumptions

module fifo;
input $fin = \{\langle 180, inf; 180, 260\rangle\}$;
input $seinb = \{\textbf{true},\ \langle 90, 110; 90, inf\rangle\}$;
output $seoutb = \{\textbf{true},\ \langle 90, 110\rangle\}$;
output $fout = \{\langle 90, 110\rangle\}$;
output $eout = \{\textbf{true},\ \langle 90, 110\rangle\}$;
output $foutb = \{\textbf{true},\ \langle 90, 110\rangle\}$;
output $eoutb = \{\langle 90, 110\rangle\}$;
process $seoutb$;
$*[[fin+]; seoutb-; [eout-]; seoutb+; [eout+]]$
endprocess
process $eout$;
$*[[seoutb-]; eoutb+; eout-; [seoutb + \&foutb+]; eoutb-; eout+]$
endprocess
process $fout$;
$*[[eoutb+]; foutb-; fout+; [seinb-]; foutb+; fout-; [seinb+]]$
endprocess
endmodule

Figure 7.4. The timed HSE code for the SUN FIFO.

are made for the FIFO to achieve high performance. In a pipeline using transparent data latches, the movement of a datum from one stage of a pipeline to the next involves two actions: capturing the data value in the latches of the next stage; and unlatching the data latches in the present stage to free them to capture the next datum. Tight control of the timing relationships between these two actions is important if robust and fast circuits are to be achieved. Speed suffers if the unlatching action is too late compared with the latching action. Robustness suffers if the unlatching action takes place before the latching action is complete. The delay requirements of latching and unlatching must be satisfied regardless of the speed at which the pipeline operates.

The operation of the FIFO is very simple: whenever a stage that is *Full* is followed by a stage that is *Empty*, the data in the full stage is moved to the empty stage and the states of both stages are changed correspondingly. Figure 7.5 shows the control circuit for a single stage of the FIFO. When a request comes in ($FIN+$) and the FIFO is empty ($EOUT$ is high), the data is latched (En_bar+ and $En-$). In parallel, the insertion is acknowledged ($SEOUT-$) and the next stage is requested to accept the data ($FOUT+$). When the next stage accepts the data ($SEIN-$), the FIFO is set to be empty ($EOUT+$) and the latch is opened (En_bar- and $En+$). The THSE code for a single stage FIFO is shown in Figure 7.4.

The correctness of this circuit is highly dependent on timing parameters. By using

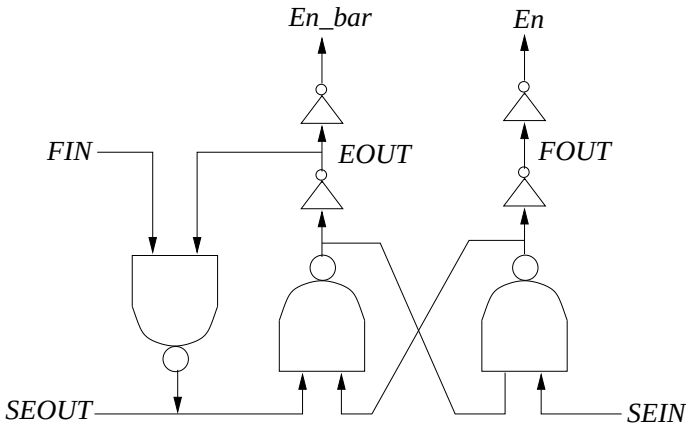

Figure 7.5. The control circuit for a single stage FIFO.

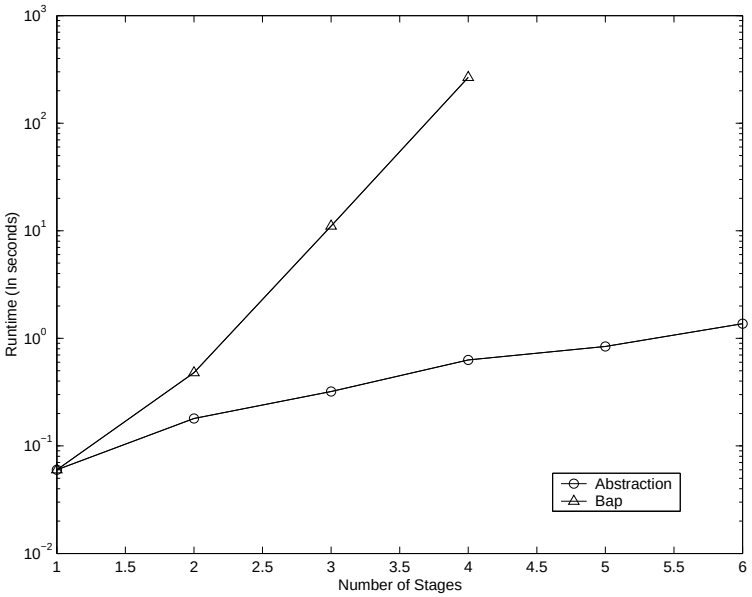

Figure 7.6. Synthesis time for FIFO example.

`ATACS`, the same efficient circuit is derived [77]. We run the same experiments for the FIFO as we did for the PCHB. Without using abstraction, `ATACS` can only finish the FIFO up to 4 stages. For the 5-stage FIFO, we had to kill the process after it ran for over a day. With abstraction on, however, `ATACS` easily proceeds to 100 stages, which takes approximately 31 minutes and 13 MB memory. The direct method [44] can also finish 100 stages, but it takes over 300 minutes. Comparative results upto 6 stages are shown in Figure 7.6.

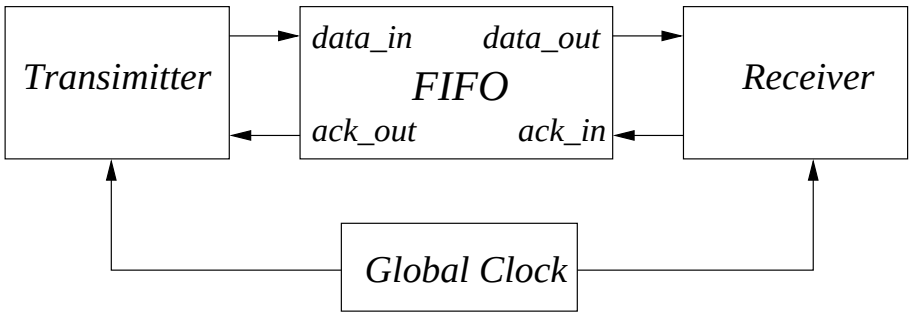

Figure 7.7. The STARI interface.

7.2 STARI: A Communication Circuit

The last example is a STARI communication circuit described in detail in [38]. STARI is a self-timed FIFO that is used to communicate between two synchronous systems that are operating at the same clock frequency, but are out-of-phase due to clock skew. These two systems communicate as though they are part of an ideal synchronous systems (Figure 7.7). During each period of the clock, one value is inserted into the FIFO by the transmitter and one value is removed by the receiver. Because data is inserted and removed at the same rate, no control signals are required to prevent underflow and overflow. However, due to the clock skew, there can be short term fluctuations in the clock rate at the receiver or transmitter and it can appear that one of them is working faster than the other. STARI responds to these fluctuations by building up more data in the FIFO when the transmitter is working faster, and by supplying data from the FIFO when the receiver is working faster.

For correct operation of the STARI, the following two properties need to be verified:

1. Each data value output by the transmitter must be inserted into the FIFO before the next one is output.
2. A new data value must be output by the FIFO before each acknowledgment from the receiver.

To guarantee the second property, it is necessary to initialize the FIFO to be approximately half-full [38]. Intuitively, the longer and faster the FIFO, the more skew it can tolerate. The correctness of the above properties depends on the length of the FIFO, the clock speed, the magnitude of the skew, and the speed of operation of the FIFO stages.

In the STARI circuit, each signal x is represented by the dual-rail code as shown in Table 7.8. The "empty" value is necessary to distinguish between two consecutive data items of the same value and one data value asserted for a long time.

x.t	x.f	x
0	0	E(empty)
0	1	F(false)
1	0	T(true)
1	1	illegal

Figure 7.8. Dual rail coding.

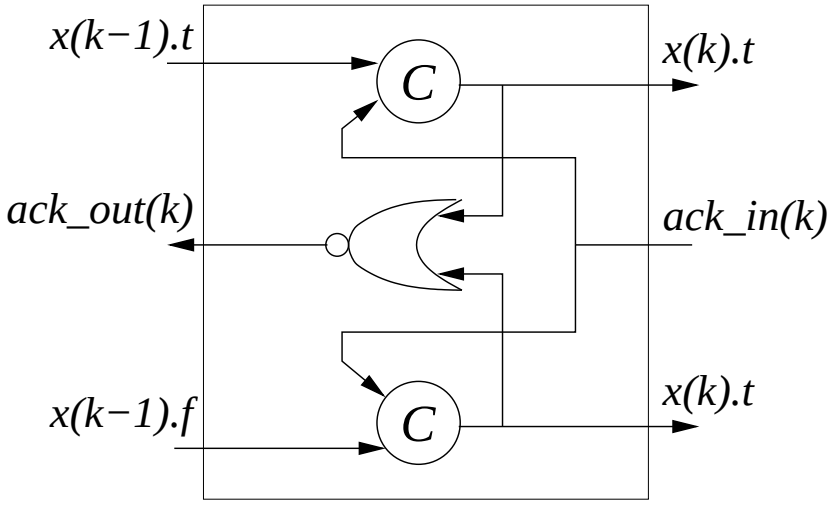

Figure 7.9. Stage k of STARI.

The typical STARI circuit consists of n identical stages, each of which is composed of 2 C-elements and 1 NOR-gate per stage as shown in Figure 7.9. The timed HSE code for a single stage STARI is shown in Figure 2.2. The TEL structure of the first stage of the STARI is shown in Figure 2.12 in chapter 2.

In [75], the authors state that `COSPAN` which uses a region technique for timing verification [3] ran out of memory attempting to verify a 3 stage gate-level version of STARI on a machine with 1 GB of memory. In [7], a flat gate-level design for 10 stages can be verified in 124 MB and 20 minutes using POSET timing. Our automated abstraction method verifies a 14 stage STARI in about 5 minutes with a maximum memory usage of 23 MB of memory for a single stage . Figure 7.10 shows the comparative runtimes for verification using Bap timing [55] with and without abstraction on STARI. Bap is an enhanced version of the POSET timing analysis algorithm. As shown in the chart, Bap can verify STARI for up to 12 stages with a memory usage of 277 MB. In the first few stages, the runtime for verification with abstaction is larger because

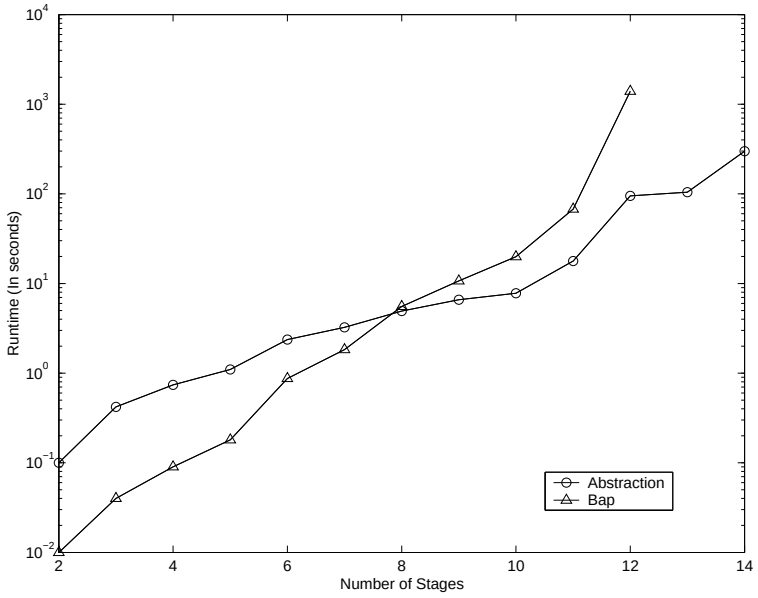

Figure 7.10. Runtimes for STARI verification.

abstraction itself takes time. When the complexity of the design grows, the runtime for flat verification grows much faster. As the designs become more and more complex, the time for abstraction dominates the total synthesis time. However, since abstraction runtime grows polynomially in the size of the specification, the total synthesis time with abstraction grows in an approximately polynomial manner. This is substantially better than the exponential growth in the analysis of flat designs.

As shown in Figure 7.10, the runtime for state space exploration using abstraction is not as good as that for the examples in the first section. The reason is that sequencing events cannot be totally abstracted away in STARI. For example, in the 8 stage STARI, there are 44 sequencing events for each stage before the transformations. After the transformations, 36 sequencing events in stage 1, 40 in stage 4, and 38 in stage 8 are removed. Chapter 5 describes several situations that the sequencing events cannot be removed. For example, if the preset and postset of a sequencing event contain multiple rules of which the enabling events or the enabled events are not in conflict, it cannot be removed as it often causes a safety violation. As designs grow, the number of such sequencing events can become large. This increases the cost of state space exploration dramatically. Sequencing events with $\texttt{size}(\bullet\$) > 1$ and $\texttt{size}(\$\bullet) > 1$ are inevitable. Redundancy check described in Chapter 6 needs to be used in such cases to reduce the size of preset or postset of a sequencing event until some reduction techniques can be

applied. More general redundancy check is needed to expose more sequencing events to be reduced.

This example, as well as the ones shown in the first section, is parameterizable. One may argue the utility of flat synthesis in that for all examples shown in this chapter only one stage needs to be built and copied to create the other stages. However, synthesis of a flat design can lead to a simpler circuit implementation. For STARI, if each stage is built from the circuit shown in Figure 7.9, this design works correctly using the timing parameters shown in Figure 2.12. If a flat design of, for example, an 8 stage STARI, is synthesized, the C-elements in the first 3 stages used to store the data can be reduced to simple buffers. In the last 3 stages, a generalized C-element using one less transistor can be used. Only the middle two stages require full C-elements. 80 literals and 160 transistors are required to implement a 8 stage STARI consisting of the same stages, while the 8 stage STARI synthesized from the flat design requires 56 literals and 136 transistors. Synthesis using our abstraction techniques derives the same results.

CHAPTER 8

CONCLUSIONS

8.1 Summary

State space exploration is required for designing high-performance timed circuits. However, state space explosion limits synthesis and verification methods to small designs. This dissertation described a theoretical framework and techniques to avoid state space explosion encountered in large designs by partitioning a design into blocks with constrained complexity and designing the blocks separately. When designing a block, the rest of the circuit and the system environment together are regarded as the environment that defines the operating condition for the block. Since only the behavior on the interface of the environment for the block is essential, the internal signals of the environment need to be abstracted away. After removing the internal signals in the environment, the total number of states of a block and its abstracted environment can be dramatically reduced, thus significantly reducing the runtime and memory usage for state space exploration.

In this dissertation, we gave an overview of the specification method of timed circuits using hardware description languages such as THSE and a synthesizable subset of VHDL for `ATACS`. These languages include constructs for specifying sequencing, concurrency, choice, and two-sided timing constraints. They also support the structural specification of a circuit. Specifications in these languages are compiled to a graphical representation, TEL structures, to which timing analysis algorithms are applied. The behavioral semantics of TEL structures are defined using timed trace theory. In this dissertation, we have proven, by using timed trace theory, the correctness of a series of theorems that support modular synthesis and verification. We described the concept of safe abstraction that deals with abstracting signals away from levels in TEL structures. It can create extra behavior not specified originally. We also described the concept of safe transformations and defined conditions that safe transformations must satisfy. Then, we described two groups of techniques based on the definition of safe transformations to reduce the number

of events and rules in a TEL structure. The first is safe net reductions that remove the sequencing events and rules in their presets and postsets, and creates new rules that preserve the causal and timing behavior between the enabling sets and enabled sets of the sequencing events. They can remove most sequencing events in a TEL structure. Safe net reductions can create extra behavior not specified originally. The second is removing redundant rules. This technique identifies and removes rules in a TEL structure that have no effect on the behavior. Redundancy check preserves the behavior precisely. These two groups of techniques are used alternatively to achieve the best result in terms of the number of sequening events left and the number of rules in the TEL structure. Safe net reductions are applied first, then redundancy check is applied to reveal more situations that can be simplified by safe net reductions. We have applied the techniques described in this dissertation to several examples including the classic STARI example, and our results show that modular synthesis and verification with abstraction is not only several orders of magnitude faster, but also capable of analyzing several orders of magnitude more complex systems that can be handled previously.

8.2 Future Work

Abstraction is essential when analyzing systems with a huge number of states such as a circuit with a data path. The work presented in this dissertation is only a starting point where there is much work that needs to be done in order to make it practically useful. This section describes the areas that we believe to be the most important research problems that must be addressed.

8.2.1 Specification

In Chapter 5, we discussed that if there exist initially enabled rules in the postset of a sequencing event, Reduction 1 and 2 cannot be applied without creating extra behavior. Also, in Chapter 6, a different redundancy check must be applied if initially enabled rules appear. The reason is that semantics of initially enabled rules are different from the corresponding non-initially enabled rules. In order to make safe reductions and redundancy more applicable, it is necessary to introduce special kinds of rules into TEL structures. These rules are similar to initially enabled rules. However, after firing the enabled events in the first execution cycle, they are removed during the timing analysis.

8.2.2 Calculation of Separation Time Estimates Between Events

In the discussion of safe abstraction in Chapter 4 and redundancy check for TEL structures with levels in Chapter 6, it is assumed that the separation time between events are known. As mentioned, calculating separation time is as hard as state space exploration, and should be avoided. In order to make safe abstraction and redundancy check viable for TEL structures with levels, it is necessary to develop an approximate algorithm to calculate a conservative estimate of the separation time. This algorithm must incur low computational cost. In [63], Myers described such an approximate algorithm to compute a estimate of the minimum and maximum separation time between all events in a cyclic, choice-free graph. It has a polynomial complexity. It is highly desirable to extend this algorithm to apply to TEL structures with arbitrary choice and levels while maintaining the polynomial complexity.

8.2.3 Automatic Partitioning of Designs

The quality of abstraction has several aspects. One of them is that each block after partitioning must have a balanced and constrained complexity. The complexity of state space exploration algorithms is exponential in the size of TEL structures. Extra time is required if abstraction is applied, and time for abstraction for each block grows polynomially in the size of TEL structures. Therefore, the total time for synthesis and verification of a block consists of two portions: time for state space exploration with an exponential complexity in the size of the block and time for abstraction with a polynomial complexity in the size of TEL structures. If the size of the block is too large, the cost of designing a block is close to that of the whole circuit. If the size of each block is too small, then the number of blocks after partitioning is too large, and the overhead for abstraction can grow substantially. Also, the current abstraction technique selects a block in a circuit based on the specified structural information. In other words, a designer chooses a component from a circuit by hand. This process becomes tedious and time-consuming if the circuit consists of many components and levels of hierarchy. To address the above problem, it is necessary to develop an algorithm that takes advantage of the specified structural information to partition a circuit so that each block has a balanced and constrained complexity, so the speed of the design processes using abstraction can be improved in a maximum way.

8.2.4 Refinement Guided Abstraction

Safe abstractions in Chapter 4 and safe net reductions in Chapter 5 create extra behavior that may result in violating states that are unreachable in the original circuit. This may produce a false negative answer for verification. When this situation happens, abstraction has to go backwards to eliminate the extra behavior causing the fake violating states. One method is to run abstraction again without using a safe abstraction or net reduction technique that creates extra behavior. If this is not enough, more such transformations are removed and abstraction needs to be performed again. An alternative is to analyze the error traces and find out if they are caused by removing some sequencing events. If it is true, these sequencing events are restored and timing analysis is performed again.

8.2.5 Structural Analysis for TEL Structures

For safe net reductions described in Chapter 5, it is required that the net after reductions preserves safeness and liveness of the original net. In [31], Commoner described an analysis approach to determine the safeness and liveness of a net based on its structure. However, his approach can only be applied to marked graphs that are a class of Petri-net without choices. In [40], Hack described and proved the sufficient conditions for a free-choice Petri-net to be safe and live. The above approaches either cannot handle choice at all or in a limited way. It is desirable to develop a method to decide the safeness and liveness of a net with arbitrary structure, so more complex safe transformations that preserve safeness and liveness of nets can be developed.

8.2.6 Combining Partial Order Reduction with Abstraction

As introduced in the first chapter, partial order reduction reduces the state space by considering only a subset of all possible interleavings between two concurrent events. This technique is widely used in verification world, but not in synthesis because all behaviors of a circuit need to be considered. In timed circuit design, the internal bebehavior of an environment has no impact on how a circuit is synthesized or verified, therefore, partial order reduction can also be used to simplify the environment besides the abstraction technique described in this dissertation. If these two techniques can be combined together, better results can be obtained.

REFERENCES

[1] Alur, R. *Techniques for Automatic Verification of Real-Time Systems.* PhD thesis, Stanford University, August 1991.

[2] Alur, R., Courcoubetis, C., Dill, D., Halbwachs, N., and Wong-Toi, H. An implementation of three algorithms for timing verification based on automata emptiness. In *Proceedings of the Real-Time Systems Symposium* (1992), IEEE Computer Society Press, pp. 157–166.

[3] Alur, R., and Kurshan, R. P. Timing analysis in cospan. In *Hybrid Systems III* (1996), Springer-Verlag.

[4] Beerel, P. A., Burch, J. R., and Meng, T. H.-Y. Checking combinational equivalence of speed-independent circuits. *Formal Methods in System Design* (Mar. 1998).

[5] Beerel, P. A., Meng, T. H.-Y., and Burch, J. Efficient verification of determinate speed-independent circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)* (Nov. 1993), IEEE Computer Society Press, pp. 261–267.

[6] Belluomini, W. *Algorithms for Synthesis and Verification of Timed Circuits and Systems.* PhD thesis, University of Utah, 1999.

[7] Belluomini, W., and Myers, C. Verification of timed systems using posets. In *International Conference on Computer Aided Verification* (1998), Springer-Verlag.

[8] Belluomini, W., and Myers, C. J. Timed event/level structures. In collection of papers from TAU'97.

[9] Belluomini, W., and Myers, C. J. Timed state space exploration using posets. *IEEE Transactions on Computer-Aided Design 19*, 5 (May 2000), 501–520.

[10] Belluomini, W., and Myers, C. J. Timed circuit verification using tel structures. *IEEE Transactions on Computer-Aided Design 20*, 1 (Jan. 2001), 129–146.

[11] Belluomini, W., Myers, C. J., and Hofstee, H. P. Verification of delayed-reset domino circuits using ATACS. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Apr. 1999), pp. 3–12.

[12] Berkel, K. v. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, vol. 5 of *International Series on Parallel Computation.* Cambridge University Press, 1993.

[13] Berkel, K. v., Kessels, J., Roncken, M., Saeijs, R., and Schalij, F. The

VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)* (1991), pp. 384–389.

[14] Berthelot, G. Checking properties of nets using transformations. In *Lecture Notes in Computer Science, 222* (1986), pp. 19–40.

[15] Berthomieu, B., and Diaz, M. Modeling and verification of time dependent systems using time petri nets. *IEEE Transactions on Software Engineering 17*, 3 (March 1991).

[16] Bochmann, G. V. Finite description of communication protocols. In *Computers Networks 2(4, 5)* (October 1978).

[17] Borriello, G., and Katz, R. H. Synthesis and optimization of interface transducer logic. In *Proceedings IEEE 1987 ICCAD* Digest of Papers (1987), pp. 274–277.

[18] Bozga, M., Maler, O., Pnueli, A., and Yovine, S. Some progress in the symbolic verification of timed automata. In *Proc. International Conference on Computer Aided Verification* (1997).

[19] Brayton, R. K. Vis: A system for verification and synthesis. In *Proc. International Conf. Computer-Aided Design (ICCAD)* (1996), pp. 428–432.

[20] Brown, M. C., Clarke, E. M., Dill, D. L., and Mishra, B. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers C-35(12)* (December 1986), 1035–1044.

[21] Brunvand, E. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. PhD thesis, Carnegie Mellon University, 1991.

[22] Brunvand, E., and Sproull, R. F. Translating concurrent programs into delay-insensitive circuits. In *International Conference on Computer-Aided Design, ICCAD-1989* (1989), IEEE Computer Society Press.

[23] Bryant, R. E. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers C-35(8)* (1987), 677–691.

[24] Burch, J. R. Modeling timing assumptions with trace theory. In *ICCD* (1989).

[25] Burch, J. R. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. PhD thesis, Carnegie Mellon University, 1992.

[26] Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L. J. Symbolic model checking:10^{20} states and beyond. *Information and Computation 98(2)* (1998), 142–170.

[27] Burns, S. M. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, 1991.

[28] Chu, T.-A. *Synthesis of Self-Timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, Massachusetts Institute of Technology, 1987.

[29] Clark, E. M. *Model Checking.* The MIT Press, 2001.

[30] Clark, E. M., and Emerson, E. A. Design and synthesis of synchronization skeletons using branching time temporal logic. In *LNCS 131* (May 1981), Springer-Varlag.

[31] Commoner, F., Holt, A., Even, S., and Pnueli, A. Marked directed graphs. *Journal of Computer Systems and Science Vol. 5* (October 1971), 511–532.

[32] Davis, A., Coates, B., and Stevens, K. The Post Office experience: Designing a large asynchronous chip. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences* (1993), IEEE Computer Science Press, pp. 409–418.

[33] Dill, D. L. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems* (1989).

[34] Dill, D. L. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* MIT Press, 1989.

[35] Dill, D. L. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* ACM Distinguished Dissertations. MIT Press, 1989.

[36] Emerson, E. A. *Branching Time Temporal Logic and the Design of Correct Concurrent Programs.* PhD thesis, Harvard University, 1981.

[37] Godefroid, P. Using partial orders to improve automatic verification methods. In *International Conference on Computer-Aided Verification* (June 1990), pp. 176–185.

[38] Greenstreet, M. R. Stari: Skew tolerant communication. unpublished manuscript, 1997.

[39] Gu, J., and Puri, R. Asynchronous circuit synthesis with boolean satisfiability. In *IEEE Trans. CAD, Vol. 14, No. 8* (1995), pp. 961–973.

[40] Hack, M. Analysis of production schemata by petri nets. Master's thesis, MIT, 1972.

[41] Hofstee, H. P., Dhong, S. H., Meltzer, D., Nowka, K. J., Silberman, J. A., Burns, J. L., Posluszny, S. D., and Takahashi, O. Designing for a gigahertz. *IEEE MICRO* (May-June 1998).

[42] Huffman, D. A. The synthesis of sequential switching circuits. *J. Franklin Institute* (March, April 1954).

[43] Johnsonbaugh, R., and Murata, T. Additional methods for reduction and expansion of marked graphs. In *IEEE TCAS, vol. CAS-28, no. 1* (1981), pp. 1009–1014.

[44] Jung, S. T., and Myers, C. Direct synthesis of timed circuits from free-choice stgs. Tech. rep., University of Utah, 2001. http://www.async.utah.edu.

[45] Kishinevsky, M., Kondratyev, A., Taubin, A., and Varshavsky, V. *Concurrent Hardware: The Theory and Practice of Self-Timed Design.* Series in Parallel Computing. John Wiley & Sons, 1994.

[46] Lavagno, L., Keutzer, K., and Sangiovanni-Vincentelli, A. Synthesis of hazard-free asynchronous circuits with bounded wire delays. *IEEE Transactions on Computer-Aided Design 14*, 1 (January 1995), 61–86.

[47] Lewis, H. R. Finite-state analysis of asynchronous circuits with bounded temporal uncertainty. Tech. rep., Harvard University, July 1989.

[48] Martin, A. J. Programming in VLSI: From communicating processes to delay-insensitive circuits. In *Developments in Concurrency and Communication* (1990), C. A. R. Hoare, Ed., UT Year of Programming Series, Addison-Wesley, pp. 1–64.

[49] Martin, A. J., Lines, A., Manohar, R., and Nystrom, M. The design of an asynchronous mips r3000 microprocessor. In *1997 Michigan Conference on Very Large Scale Integration* (1997), R. B. Brown and A. T. Ishii, Eds., pp. 164–181.

[50] McMillan, K. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proc. International Workshop on Computer Aided Verification* (1992), G. v. Bochman and D. K. Probst, Eds., vol. 663 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 164–177.

[51] McMillan, K., and Dill, D. L. Algorithms for interface timing verification. In *International Conference on Computer Design, ICCD-1992* (1992), IEEE Computer Society Press.

[52] McMillan, K. L. *Symbolic Model Checking: An Approach to the State explosion Probelm.* PhD thesis, Carnegie Mellon University, 1992.

[53] Meng, T. H.-Y., Brodersen, R. W., and Messershmitt, D. G. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer-Aided Design 8*, 11 (November 1989), 1185–1205.

[54] Mercer, E. Performance analysis of timed asynchronous circuits. Master's thesis, University of Utah, 1999.

[55] Mercer, E., Myers, C., and Yoneda, T. Improved poset timing analysis in timed petri nets. Tech. rep., University of Utah, 2001. http://www.async.utah.edu.

[56] Mishra, B., and Clarke, E. M. Hierarchical verification of asynchronous circuits using temporal logic. In *Theoretical Computer Science 38* (1985), pp. 269–291.

[57] Molnar, C. E., Fang, T.-P., and Rosenberger, F. U. Synthesis of delay-insensitive modules. In *1985 Chapel Hill Conference on Very Large Scale Integration* (1985), H. Fuchs, Ed., Computer Science Press, Inc., pp. 67–86.

[58] Molnar, C. E., Jones, I. W., Coates, B., and Lexau, J. A FIFO ring oscillator performance experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Apr. 1997), IEEE Computer Society Press,

pp. 279–289.

[59] Moundanos, D., Abraham, J., and Hoskote, Y. Abstraction techniques for validation coverage analysis and test generation. *IEEE TC 47*, 1 (1998), 2–14.

[60] Muller, D. E., and Bartky, W. S. A theory of asynchronous circuits. In *Proceedings of an International Symposium of the Theory of Switching* (1959), pp. 204–243.

[61] Murata, T. Petri nets: Properties, analysis, and applications. In *Proceedings of the IEEE 77(4)* (1989), pp. 541–580.

[62] Murata, T., and Koh, J. Y. Reduction and expansion of lived and safe marked graphs. In *IEEE TCAS, vol. CAS-27, no. 10* (1980), pp. 68–70.

[63] Myers, C. J. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Stanford University, 1995.

[64] Myers, C. J., and Meng, T. H.-Y. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems 1*, 2 (June 1993), 106–119.

[65] Myers, C. J., Rokicki, T. G., and Meng, T. H.-Y. POSET timing and its application to the synthesis and verification of gate-level timed circuits. *IEEE Transactions on Computer-Aided Design 18*, 6 (June 1999), 769–786.

[66] Namjoshi, K. S., and Kurshan, R. P. Syntactic program transformations for automatic abstraction. In *CAV 2000* (July 2000), vol. 1855 of *lncs*, springer, pp. 435–449.

[67] R.Grosu, R. A., and McDougall, M. Efficient reachability analysis of hierarchical reactive machines. In *12th International Conference on Computer-Aided Verification, LNCS 1855* (2000), pp. 280–295.

[68] Roig, O. *Formal Verification and Testing of Asynchronous Circuits*. PhD thesis, Univsitat Politècnia de Catalunya, May 1997.

[69] Rokicki, T. G. *Representing and Modeling Circuits*. PhD thesis, Stanford University, 1993.

[70] Rokicki, T. G., and Myers, C. J. Automatic verificaton of timed circuits. In *International Conference on Computer-Aided Verification* (1994), Springer-Verlag, pp. 468–480.

[71] Rotem, S., Stevens, K., Ginosar, R., Beerel, P., Myers, C., Yun, K., Kol, R., Dike, C., Roncken, M., and Agapiev, B. RAPPID: An asynchronous instruction length decoder. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Apr. 1999), pp. 60–70.

[72] Subrahmanyam, P. What's in a timing discipline? considerations in the specification and synthesis of systems with interacting asynchronous and synchronous components. In *Hardware Specification, Verification and Synthesis: Mathematical*

Aspects (1990), Springer-Verlag.

[73] Suzuki, I., and Murata, T. *Stepwise refinements for transitions and places.* New York: Springer-Verlag, 1982.

[74] Suzuki, I., and Murata, T. A method for stepwise refinements and abstractions of petri nets. In *Journal Of Computer System Science, 27(1)* (1983), pp. 51–76.

[75] Tasiran, S., and Brayton, R. K. Stari: A case study in compositional and heirarchical timing verification. In *Proc. International Conference on Computer Aided Verification* (1997).

[76] Thacker, R. A. Implicit methods for timed circuit synthesis. Master's thesis, University of Utah, 1998.

[77] Thacker, R. A., Belluomini, W., and Myers, C. J. Timed circuit synthesis using implicit methods. In *Proc. International Conference on VLSI Design* (1999), pp. 181–188.

[78] Valmari, A. A stubborn attack on state explosion. In *International Conference on Computer-Aided Verification* (June 1990), pp. 176–185.

[79] van Berkel, C., and Saeijs, R. Compilation of communicating processes into delay-insensitive circuits. In *International Conference on Computer Design, ICCD-1988* (1988), IEEE Computer Society Press.

[80] Vanbekbergen, P., Goossens, G., and de Man, H. Specification and analysis of timing constraints in signal transition graphs. In *Proceedings of the European Design Automation Conference* (1992).

[81] Varshavsky, V. I., Ed. *Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.

[82] Winskel, G. An introduction to event structures. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. Noordwijkerhout, Norway* (June 1988).

[83] Yoneda, T., and Ryu, H. Timed trace theoretic verification using partial order reduction. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Apr. 1999), pp. 108–121.

[84] Yun, K. Y., Dill, D. L., and Nowick, S. M. Synthesis of 3D asynchronous state machines. In *Proc. International Conf. Computer Design (ICCD)* (Oct. 1992), IEEE Computer Society Press, pp. 346–350.

[85] Zafiropulo, P., West, C. H., Rudin, H., and Brand, D. Towards analyzing and synthesizing protocols. *IEEE Transactions on Communications COM-28(4)* (April 1980).

[86] Zheng, H. Specification and compilation of timed systems. Master's thesis,

University of Utah, 1998.

[87] Zhou, B., and Yoneda, T. Conformance and mirroring for timed asynchronous circuits. In *Proc. of Asia and South Pacific Design Automation Conference* (Feb. 2001), pp. 341–346.