2-23-2007

A Compositional Approach to Asynchronous Design Verification with Automated State Space Reduction

Jared Ahrens
University of South Florida

A Compositional Approach to Asynchronous Design Verification with Automated State Space Reduction

by

Jared Ahrens

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science and Engineering
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Hao Zheng, Ph.D.
Srinivas Katkoori, Ph.D.
Dewey Rundus, Ph.D.

Date of Approval:
February 23, 2007

Keywords: Model Checking, Abstraction, Constraint, Autofailure, Formal Verification

ACKNOWLEDGEMENTS

This work has been funded in part by the National Institute for Systems Test and Productivity. This thesis is also based upon work supported by the National Science Foundation under grant No. 0546492. I am thankful for this generous funding. I extend my most sincere thanks to Dr. Zheng for introducing me to such a fascinating topic. He has provided wonderful support, guidance, and funding throughout this thesis. I would also like to thank Dr. Katkoori and Dr. Rundus for being on my committee and providing valuable feedback. I am also grateful to my wife and family for the support they have shown.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

A COMPOSITIONAL APPROACH TO ASYNCHRONOUS DESIGN VERIFICATION WITH AUTOMATED STATE SPACE REDUCTION

Jared Ahrens

ABSTRACT

Model checking is the most effective means of verifying the correctness of asynchronous designs, and state space exploration is central to model checking. Although model checking can achieve very high verification coverage, the high degree of concurrency in asynchronous designs often leads to state explosion during state space exploration. To inhibit this explosion, our approach builds on the ideas of compositional verification. In our approach, a design modeled in a high level description is partitioned into a set of parallel components. Before state space exploration, each component is paired with an over-approximated environment to decouple it from the rest of the design. Then, a global state transition graph is constructed by reducing and incrementally composing component state transition graphs. We take great care during reduction and composition to preserve all failures found during the initial state space exploration of each component. To further reduce complexity, interface constraints are automatically derived for the over-approximated environment of each component. We prove that our approach is conservative in that false positive results are never produced. The effectiveness of our approach is demonstrated by the experimental results of several case studies showing that our approach can verify designs that cannot be handled by traditional flat approaches. The experiments also show that constraints can reduce the size of the global state transition graph and prevent some false failures.

CHAPTER 1

INTRODUCTION

In an asynchronous circuit, there is no global control signal to synchronize the operations of different portions of the circuit. Without global synchronization, different orderings of signal transitions in a circuit may result in different circuit behavior. Therefore, all possible orderings among signal transitions need to be checked to assure correctness. When a circuit displays a high degree of concurrency, the number of all possible orderings of signal transitions can be excessively large, in which case the traditional simulation-based verification approach becomes inadequate. A better approach is model checking which is an exhaustive method of verification that guarantees that a property holds for the all states of a design. Traditional flat model checking attempts to produce a single state transition graph representing the complete design, but this approach quickly fails when applied to both large and small designs containing a high degree of concurrent activities. The high degree of concurrency requires an exponential number of states to represent the design. This is known as *state space explosion*.

To overcome this problem, we develop a framework wherein a reduced global state transition graph is built from the bottom to the top in the design hierarchy. An asynchronous circuit design is modeled as a set of parallel components running concurrently; we express this model as a boolean guarded Petri-net. First, a state transition graph is found for each component. To decouple the component from the rest of design, we use an over-approximated environment to simulate the rest of the design communicating with the component under consideration. The over-approximated environment reproduces all input behavior that the complete design would supply to the component. It is possible that some failures will be found during local state space exploration, and an approach is formalized

to preserve the failures found during this step. Next, the state transition graphs for the components are composed to form the global state transition graph for the complete design. During composition, behavior not observable on the interface of the complete design is abstracted away to contain the size of the intermediate results. The result is a reduced representation of the complete design.

The over-approximated environment may produce extra input behavior which can cause a component to exude extra output behavior. When the component is embedded in the complete design, this extra behavior is not produced. This extra behavior can create extra states and false failures in the state transition graphs. One alternative is a user generated environment. User generated environments may be highly accurate, but they are very difficult to derive. For large designs, they become nearly impossible. To reduce the occurrence of these extra states and false failures, we complement our framework with the automated generation of constraints. During composition, we decouple a component from the rest of the design and add an over-approximated environment to drive its inputs. If the component were embedded in the complete design, another component would drive its inputs. To refine the over-approximation for a particular input, we examine the state transition graph of the component which would drive the input. Then we generate a boolean expression from the state transition graph. The boolean expression, which we refer to as a constraint, describes when an input may occur and is used to restrain the behavior of the over-approximated environment. This results in a more accurate over-approximated environment which produces fewer extra states and false failures.

1.1 Related Work

Compositional reasoning and *abstraction* are essential to verifying large systems. Compositional reasoning, broadly referring to compositional verification or compositional minimization, takes advantage of the given design hierarchy. A general compositional verification method is based on *assume-guarantee* style reasoning, and verifies global properties by verifying local properties of each component in a system [3, 4, 5, 6, 7]. In a compositional

verification framework, each component of a system is considered separately. During verification, assumptions are made about the environment with which the component interacts; then these assumptions need to be discharged later. Assumptions are typically generated by the user. If the component has complex interactions with its environment, it can be difficult to make accurate assumptions. Recent works have attempted to derive assumptions automatically. In [8], an automated approach is described to generate the assumptions for compositional verification. This approach starts with a set of the weakest assumptions for a component, and iteratively refines these assumptions. Although the approach guarantees that the iteration terminates, it is not clear how efficient the approach would be in terms of iterations necessary to generate a set of assumptions to prove the properties. Also, this approach can only handle safety properties. In addition, global specification needs to be broken down to local properties defined on the interfaces of the components, which can be very difficult. [24, 25, 26, 27] also propose methods to automate the generation of assumptions, but these methods are costly because they must first generate false counter-examples.

Abstraction produces the reduced model of a system by abstracting away certain details that are unnecessary when reasoning about the system [14, 15]. Abstraction methods based on Petri-net reduction are described in [1, 2]. These methods simplify Petri-net models of asynchronous circuits either by following the design partitions or as directed by the properties to be verified. The reductions described in these methods attempt to reduce the model before state space exploration. Although these methods are very effective, they are limited to a particular kind of Petri-net. In [9], a compositional minimization method is described where the global minimized state transition system is built by iteratively minimizing and composing the components in finite state system. To contain the size of the intermediate results, user-provided context constraints are required. This may be a problem in that the state space may be large in the first place. The requirement of user-provided context constraints may also be a problem in that the constraints may be over restrictive, thus causing false positive verification results. Similar work is also described in [10, 11, 12, 13]. In [16], a hierarchical approach similar to that in [17] is presented. In this approach, an abstraction

for each module in a system is found and verification is applied to the composition of those abstractions. In [18], a constraint oriented proof methodology is applied to verify infinite systems. Constraints on infinite systems are broken into an infinite number of simple constraints on finite systems, then these constraints are grouped into finite equivalent classes. However, this methodology is not complete in that the reduction of infinite systems is not guaranteed. In [19], a software model checking method utilizing *lazy abstraction* is presented to improve performance by adding information during abstraction refinement only when necessary. It would be interesting to see if this method can be adapted to hardware verification.

1.2 Contributions

This thesis makes two contributions: a new automated compositional verification flow and automated generation of constraints to create a more accurate over-approximated environment for the composition flow.

The first contribution is a new automated verification flow. Previous works require either user provided assumptions or iterative counter example refinement. User provided assumptions are dangerous in that they may be over restrictive and prevent valid behavior from appearing in the model. This could result in false positive verification results. Automated approaches based on counter-example refinement hold some promise but can take longer than the flat approach. Our methodology combines several existing methodologies to reduce and verify large pre-partitioned designs without user intervention. In a later chapter we prove that our method produces no false positives results for a design.

The second and most significant contribution is the automated refinement of the over-approximated environment. During our design flow, we apply an over-approximated environment to each component. The over-approximated environment often supplies additional input to a component. The component can respond to this additional input with additional output. This additional input and output exacerbates state space explosion and increases the resource required to verify the design. The other side-effect of the over-approximated

environment is the production of false failures. To restrain the over-approximated environment, we derive constraints from the state transition graphs of other components. Our constraints can be generated automatically to increase the accuracy of our over-approximated environment. The increased accuracy of the environment contains the size of state space explosion and prevents the creation of many false failures.

1.3 Thesis Overview

This thesis is organized such that each chapter builds on the ideas of the previous chapters. Chapter 2 describes Boolean Guarded Petri-nets and State Transition Graphs, the formalisms we use to represent an asynchronous design. The chapter also describes the relationships we may derive between state transition graphs based on paths and traces. Chapter 3 lays the framework of our automated methodology. The framework includes methods of state space reduction and automatic generation of over-approximated environments. Chapter 4 presents a method of refining the over-approximated environment which Chapter 3 describes. Chapter 5 describes the experimental results using the methods presented in this thesis. Three approaches are compared in this chapter: flat verification of the complete design, automated compositional verification using an unrefined over-approximated environment, and automated compositional verification using a refined over-approximated environment.

CHAPTER 2

BACKGROUND

This chapter presents an overview of Petri-nets which are used to model asynchronous circuits, state transition graphs for verification, and related concepts upon which the later chapters are based.

2.1 Boolean Guarded Petri-nets

Petri-nets are a common modeling formalism for asynchronous designs. There are many different forms of Petri-nets for different applications. This section presents a form of Petri-nets which addresses certain modeling difficulties for asynchronous designs with traditional Petri-nets.

2.1.1 Structural Definitions

A boolean guarded Petri-net (BGPN) is a bipartite directed graph consisting of transitions and places. Its definition is given as follows.

Definition 2.1.1. A Boolean Guarded Petri-net N is a tuple $(W, T, P, F, \mu^0, L, B)$ where

1. W is the set of wires of the asynchronous design being modeled,
2. T is the set of transitions,
3. P is the set of places,
4. F is the flow relation,
5. μ^0 is the initial marking,
6. L is the action labeling function,
7. B is the boolean labeling function.

A visual representation of a BGPN is shown in Figure 2.2. In a BGPN, the transitions in T are represented as the thick bars, and the places in P are represented as the circles. Each transition is preceded and followed by one or more places in P, and each place is preceded and followed by one or more transitions in T. The connections between transitions and places are defined with the flow relations.

Definition 2.1.2. The flow relation of a BGPN N is $F \subseteq (T \times P) \cup (P \times T)$.

For each transition, its preset is the set of places that are connected to the transition, and its postset is the set of places to which the transition is connected. The preset and postset of a place are defined similarly. The preset and postset of both a transition and a place are defined as follows.

Definition 2.1.3. For a transition t in a BGPN N, its preset is $\bullet t = \{p \in P \mid (p,t) \in F\}$, and its postset is $t\bullet = \{p \in P \mid (t,p) \in F\}$. For a place p in a BGPN N, its preset is $\bullet p = \{t \in T \mid (t,p) \in F\}$, and its postset is $p\bullet = \{t \in T \mid (p,t) \in F\}$.

The dots found in some places are tokens. Each place may contain one or more tokens. If a place has a token, it is marked. A set of marked places is a marking of a BGPN. The marking is defined as follows.

Definition 2.1.4. The marking of a BGPN N is $\mu = \{p \in P \mid p \text{ is marked}\}$.

As will be seen later, a marking represents a state of a BGPN and the asynchronous design being modeled. In our method, we use 1-safe BGPNs such that a place can only have at most one token in every μ. μ^0 is the initial marking of a BGPN.

W is a finite set of wires in an asynchronous circuit design. The set W consists of input and output wires which we denote I and O. Each wire $w \in W$ can take one of two actions at any time. $w+$ indicates that the value of w changes from 0 to 1, and $w-$ indicates that the value of w changes from 1 to 0. A rising transition on the wire *ack* is expressed *ack*+. Similarly, *ack*− expresses a falling transition on the wire *ack*. The action labeling function L maps a BGPN transition to an action on a wire, thereby associating BGPN transitions to the dynamic behavior in an asynchronous design. Transitions not associated with any

action are called dummy transitions. For completeness we map dummy transitions to the nil action \$. Dummy transitions do not represent any behavior in the modeled design. They are a modeling construct generated when compiling a design into a BGPN model to hold certain conditions in the design. The labeling function is defined as follows.

Definition 2.1.5. The transition labeling function L of a BGPN N assigns each transition with an action on a wire or a dummy action, $L : T \rightarrow \left\{W \times \{+, -\}\right\} \cup \{\$\}$.

For each transition t, the flow relations from the places in $\bullet t$ to t are labeled with boolean formulas. This makes modeling of asynchronous designs less awkward in situations where the transitions depend not only on other transitions, but also on the values of some wires in a design. The boolean labeling function is defined as follows.

Definition 2.1.6. The boolean labeling function of a BGPN N is $B : (P \times T) \subseteq F \rightarrow b$ where b is a boolean formula defined over W.

Let $b(p, t)$ return the boolean formula labeled for (p, t). Given a transition t, the tuple (p, b, t) denotes an enabling rule of t where $p \in \bullet t$ and $B(p, t) = b$. Given a transition t, `enabling_rules`(t) denotes the set of all enabling rules of t. The enabling rules define the condition when a transition can fire, as will be explained later.

If the boolean labeling function of a BGPN N maps each rule to boolean formula `true`, then the BGPN is converted to a traditional Petri-net. In general, the analysis complexity for BGPNs is higher than that for the traditional Petri-nets. BGPN behavior is defined by both the marking and boolean expressions. However, the structural complexity of BGPNs can be much less than that of the traditional Petri-nets, thus resulting in a large decrease in the analysis complexity for BGPNs. The above point is illustrated by example in Figures 2.1 and 2.2. They show the BGPN and the traditional Petri-net model for an AND-gate described in [20]. Both models are driven by a maximal environment. The traditional Petri-net model requires ten places and seventeen transitions, and its transitions are not required to satisfy any boolean expressions. The BGPN representation only requires six places and six transitions, but it also includes two boolean expressions.

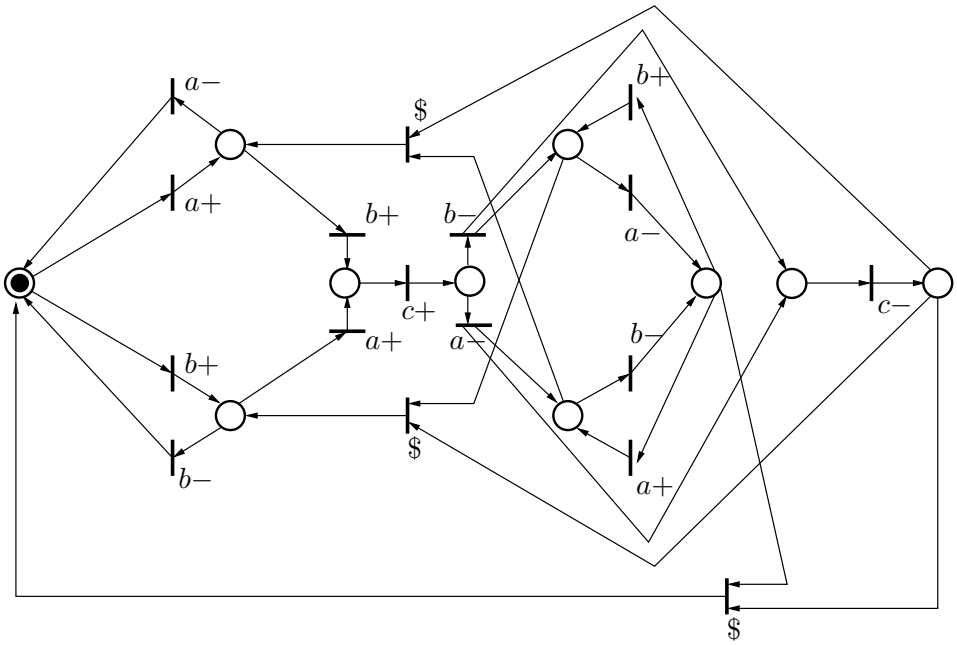

Figure 2.1 Traditional Petri-net modeling an AND-gate

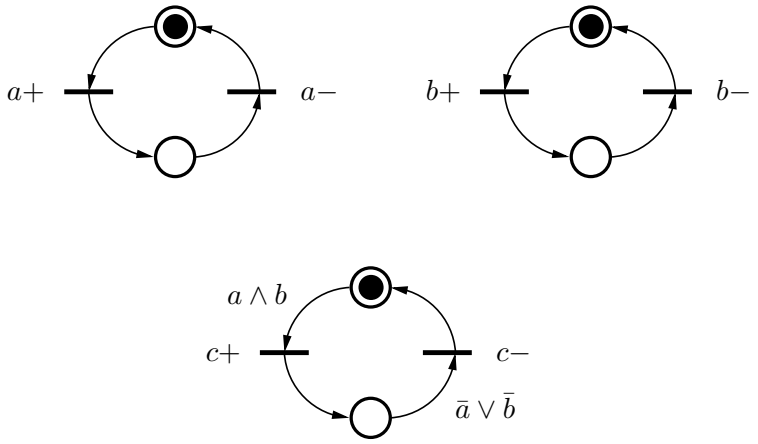

Figure 2.2 The BGPN modeling an AND-gate

2.1.2 Semantics

This section presents the firing semantics of transitions. Since the BGPN transitions in an asynchronous design model are associated with actions on wires, firing the transitions reflects the execution of the associated actions, thus changing the states of the design. Executing an action is known as an event. Without incidents of confusion, events and transition firings are used interchangeably in the following.

The state of a BGPN is the pair (μ, α) where μ is a marking of the BGPN and α is the vector representing the boolean values of the wires in W of the modeled design in μ. Given a state s, the function $\mu(s)$ accesses the μ component of s. Similarly, the function $\alpha(s)$ accesses the α component of s. A transition needs to be enabled before it can fire at a state. For a transition to be enabled, all of its enabling rules must be satisfied. Given a transition t, an enabling rule r of t is satisfied at a state if the place of r is marked and the boolean formula of r is satisfied at the current state. Given a state $s = (\mu, \alpha)$, let $\texttt{eval}(s, b)$ be a function that returns `true` if b evaluates to `true` with α, `false` otherwise.

Definition 2.1.7. A rule (p, b, t) of t is satisfied at a state $s = (\mu, \alpha)$ if $p \in \mu$ and $\texttt{eval}(s, b) =$ `true`.

A set of satisfied enabling rules of a transition t at a state s is denoted as $\texttt{satisfied}(t, s)$. A transition is enabled if all its enabling rules are satisfied.

Definition 2.1.8. A transition t is enabled at a state s if

$$\texttt{enabling\_rules}(t) = \texttt{satisfied}(t, s).$$

The set of transitions that are enabled at a state s is denoted as $\texttt{enabled}(s)$.

Given a set of enabled transitions at a state, we exhaustively fire transitions from the enabled transition set. The firing of a transition changes the marking and causes the action associated with the transition to occur. After firing a transition t, the preset of the transition is removed from the current marking, and the postset is added to the marking. This step is known as the marking update. Given a marking μ, firing t results in a new marking

$\mu' = (\mu - \bullet t) + t\bullet$. Our method requires that the BGPNs are 1-safe such that each place can contain no more than one token at any state. The value of the wire in the state vector is also updated accordingly, depending on the associated action of that transition after a transition firing. Let $s = (\mu, \alpha)$ be a state, and a transition t fired at s. Also, let a be the associated action of t. The state vector α is updated as follows after firing t:

$$\forall w \in W.\ \alpha(w) \quad = \quad \begin{cases} \alpha(w) & \text{if } a \text{ is a dummy action} \\ 1 & \text{if } a = w+ \\ 0 & \text{if } a = w- \end{cases}$$

Updating the marking and state vector of the model constitutes a change in the system's state. Later we will see these events represented as state transitions in a state transition graph.

2.1.3 BGPN Composition

Usually, an asynchronous design consists of a number of components running in parallel. Each component is modeled in a BGPN, and the model for the entire design is the parallel composition of the component BGPNs. This section presents the definition of the parallel composition of BGPNs.

Let $N_1 = (W_1, T_1, P_1, F_1, \mu_1^0, L_1, B_1)$ and $N_2 = (W_2, T_2, P_2, F_2, \mu_2^0, L_2, B_2)$ be two BGPNs where $W_1 = I_1 \cup O_1$, and $W_2 = I_2 \cup O_2$. The parallel composition of N_1 and N_2, referred to as $N_1 \| N_2$, is defined as follows:

Definition 2.1.9. Given two BGPNs N_1 and N_2, if $O_1 \cap O_2 = \emptyset$, the parallel composition of N_1 and N_2, $N = (W, T, P, F, \mu^0, L, B)$, is defined as follows:

1. $W = W_1 \cup W_2$,
2. $T = T_1 \cup T_2$,
3. $P = P_1 \cup P_2$,
4. $F = F_1 \cup F_2$,
5. $\mu^0 = (\mu_1^0, \mu_2^0)$,

6. $L = L_1 \cup L_2$
7. $B = B_1 \cup B_2$

2.2 State Transition Graphs

In the previous section, firing a BGPN transition at a state results in a new state by updating the marking and state vector. By exhaustively firing all enabled transitions at each state, a state transition graph containing all reachable states in a design can be found. This step is often referred to as state space exploration. A state transition graph (STG) is a graph wherein the nodes are states and the arcs are state transitions which are labeled with the BGPN transition firings that cause the state transition. For illustrative purposes we often label the arcs of a state transition graph with the BGPN transition's associated action.

Definition 2.2.1. A state transition graph $\mathcal{G}$ is the tuple $(N, S \cup \{\pi\}, R, s^0)$ where:

1. N is the BGPN from which the STG is derived,
2. S is the set of reachable states,
3. π is the failure state,
4. R is the set of state transitions $\Big(S \times T \times (S \cup \{\pi\})\Big) \cup \Big(\{\pi\} \times T \times \{\pi\}\Big)$,
5. $s^0 \in S$ is the initial state of the STG.

A state transition is the tuple (s, t, s'), (s, t, π), or (π, τ, π). The state transition (s, t, s') indicates that a particular BGPN transition t is fired from state s and changes the state of the system to s'. For the simplicity of presentation, we also use R as a function. Given a state transition (s, t, s'), $(s, t, s') \in R$ iff $R(s, t, s')$ holds. Sometimes we also use $s \xrightarrow{t} s'$ instead of (s, t, s') to indicate a state transition. π is a special state which denotes a failure in the modeled design. This state is used to represent unintended behavior or behavior we wish to prevent. Once the system enters this special state, the behavior produced afterwards is irrelevant, and the system is regarded to remain at this state forever in our method. Let τ represent an arbitrary transition on any wire which is fired from the failure state. Once

a system enters the failure state, all future BGPN transitions firings are represented in the STG using the notation (π, τ, π).

A *path* ρ in a STG $\mathcal{G}$ is an infinite sequence of state transitions $\rho = (s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \cdots)$ such that $s_0 = s^0$ and $R(s_n, t_n, s_{n+1})$ holds for $n = 0, 1, \cdots$. Given the paths $\rho_1 = (s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \pi \xrightarrow{t_2} \pi \xrightarrow{t_3} \pi \cdots)$ and $\rho_2 = (s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \pi \xrightarrow{t_4} \pi \xrightarrow{t_5} \pi \cdots)$, the notation $\rho_{12} = (s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \pi \xrightarrow{\tau} \pi \xrightarrow{\tau} \pi \cdots)$ describes a set of failure paths including ρ_1 and ρ_2.

A *trace* is an infinite sequence of BGPN transition firings. A *trace* $\sigma = (t_0, t_1, \cdots)$ of a STG is valid if a path $(s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \cdots)$ exists. For traces containing the failure state, we use the notation $\sigma = (t_0, t_1, t_2, \cdots, t_n, \tau^*)$. The symbol τ^* indicates an infinite sequence of firings of arbitrary BGPN transitions after reaching the failure state π.

Given a BGPN N, the function $\mathcal{G}(N)$ returns the STG $\mathcal{G}$ by performing state space exploration on N. According to the firing semantics described in Section 2.1.2, each BGPN has an unique STG. Therefore, we also use $\mathcal{P}(N)$ to denote all valid traces of the BGPN N. $\mathcal{P}(N)$ can be derived from the corresponding STG of N by state space exploration from the initial state. Figure 2.3 shows the state space exploration algorithm that is used to produce a STG from a BGPN.

The projection function, $\sigma[W']$ where $W' \subseteq W$, removes all events from a trace $\sigma = (t_0, t_1, t_2, \ldots)$ whose underlying wires are not in W'. More formally, if $\sigma \neq \epsilon$ (i.e., the empty trace) and given the subtrace x where $x \subseteq \sigma$, then

$$\sigma[W'] \quad = \quad \begin{cases} (t_0, x[W']) & \text{if } \exists w \in W' \,.\, t_0 = w \times \{+, -\} \\ (x[W']) & \text{otherwise} \end{cases}$$

If $\sigma = \epsilon$, then $\sigma[W'] = \{\epsilon\}$. This function is extended naturally to a set of traces. Given a STG $\mathcal{G}$, we use $\mathcal{G}[W']$ to denote the projection of $\mathcal{G}$ to W' by applying $\sigma[W']$ to all traces in $\mathcal{G}$. For convenience we will use $W(N)$ to denote the set of wires used to label the transitions of the BGPN N and $W(\mathcal{G})$ to denote the set of wires where $\mathcal{G}$ is defined. Similarly, $W(\sigma)$ denotes the set of wires where the BGPN transitions of σ are defined. The notations $I(N)$ and $O(N)$ are functions returning the input or output wires of a design

represented in BGPN. Similarly, $I(\mathcal{G})$ and $O(\mathcal{G})$ return the input and output wires of a design represented as an STG. For a BGPN transition t, $w(t)$ returns the signal where t is defined. The projection function can also be extended to boolean expressions through existential quantification. The projection of a boolean formula $b[W']$ produces a boolean formula over the wires W'.

Similar to BGPN composition, if a design consists of a set of parallel components, each of which is modeled as an STG, the global STG for the entire design is the parallel composition of the individual ones. The parallel composition of STGs is defined as follows.

Definition 2.2.2. Let $\mathcal{G}_1 = (N_1, S_1, R_1, s_1^0)$ and $\mathcal{G}_2 = (N_2, S_2, R_2, s_2^0)$ be two STGs where $W(N_1) = I_1 \cup O_1$, and $W(N_1) = I_2 \cup O_2$. If $O_1 \cap O_2 = \emptyset$, the parallel composition $\mathcal{G} = \mathcal{G}_1 \| \mathcal{G}_2$ defines $\mathcal{G} = (N, S, R, s^0)$ as follows:

1. $N = N_1 \| N_2$
2. $S \subseteq S_1 \times S_2$
3. $s^0 = (s_1^0, s_2^0)$
4. $R = r_1 \cup r_2 \cup r_3 \cup r_4 \cup r_5 \cup r_6 \cup r_7 \cup r_8$ where for every $s_1 \in S_1$ and $s_2 \in S_2$,
   a. If $w(t) \in W(N_1)$ and $w(t) \notin W(N_2)$
      - $r_1 = \{((s_1, s_2), t, (s_1', s_2) \mid (s_1, t, s_1') \in R_1\}$
      - $r_2 = \{((s_1, s_2), t, \pi) \mid (s_1, t, \pi) \in R_1\}$
   b. If $w(t) \notin W(N_1)$ and $w(t) \in W(N_2)$
      - $r_3 = \{((s_1, s_2), t, (s_1, s_2')) \mid (s_2, t, s_2') \in R_2\}$
      - $r_4 = \{((s_1, s_2), t, \pi) \mid (s_2, t, \pi) \in R_2\}$
   c. If $w(t) \in W(N_1)$ and $w(t) \in W(N_2)$
      - $r_5 = \{((s_1, s_2), t, (s_1', s_2')) \mid (s_1, t, s_1') \in R_1$ and $(s_2, t, s_2') \in R_2\}$
      - $r_6 = \{((s_1, s_2), t, \pi) \mid (s_1, t, \pi) \in R_1$ and $(s_2, t, s_2') \in R_2\}$
      - $r_7 = \{((s_1, s_2), t, \pi) \mid (s_1, t, s_1') \in R_1$ and $(s_2, t, \pi) \in R_2\}$
      - $r_8 = \{((s_1, s_2), t, \pi) \mid (s_1, t, \pi) \in R_1$ and $(s_2, t, \pi) \in R_2\}$

```
find_sg((W, T, P, F, μ⁰, L, B), s⁰)
  T_e = enabled(s⁰)
  push(s⁰, T_e)
  failure = false
  while stack is not empty do
    (s, T_e) = pop()
    s' = s
    t = select(T_e)
    if T_e − t ≠ ∅ then
      push(s, T_e − {t})
    if (μ(s) − •t) ∩ t• ≠ ∅ then              // check safety failure
      failure = true
    μ(s') = (μ(s) − •t) ∪ t•
    if L(t) = w+ then
      if α(s)[w] = 1 then                    // check complement failure
        failure = true
      α(s')[w] = 1
    else if L(t) = w− then
      if α(s)[w] = 0 then                    // check complement failure
        failure = true
      α(s')[w] = 0
    T'_e = enabled(s⁰)
    if T'_e = ∅ and stack is empty then      // check deadlock failure
      failure = true
    else if (T_e − {t}) ⊄ T'_e then          // check disabling failure
      failure = true
    if failure = true then
      R = R ∪ {(s, t, π)}
    else
      if s' ∉ S then
        S = S ∪ {s'}
        if T'_e ≠ ∅ then
          push(s', T'_e)
      R = R ∪ {(s, t, s')}
  return (S, R)
```

Figure 2.3 Algorithm to find the reachable state space with failure preservation

N of the composite STG is the BGPN composition of N_1 and N_2. S of $\mathcal{G}$ is a subset of all possible pairs of states from S_1 and S_2. S consists of those states in $S_1 \times S_2$ which are reachable from s^0. Whether a state in $S_1 \times S_2$ is reachable depends upon the set of state transitions R which is explained below.

Suppose a design consists of two components running in parallel. The synchronization between them is through wires from the outputs of one component to the inputs of another. In this case, both components make a state transition in parallel by changing the values of the common wires between them. Otherwise, if one component makes a transition not visible to another one, the state transition of the entire design follows the component that makes the transition while the state of the other component is regarded to remain the same. In other words, if a component makes an internal state transition, it is reflected in the entire design while the other component is viewed as not changing. All the following cases are considered in the composition definition for R. The first case defines how a global state transition is produced from a state transition visible only to $\mathcal{G}_1$. In addition, when $\mathcal{G}_1$ makes a transition to the failure state, the entire design makes a transition to the failure state regardless of the current state of $\mathcal{G}_2$. The second case is symmetric to case 1 defining how a global state transition is produced from a state transitions only visible to $\mathcal{G}_2$. Similarly, when $\mathcal{G}_2$ makes a transition to the failure state, the entire design makes a transition to the failure state no matter what state of $\mathcal{G}_1$ is. The third case defines global state transitions when both $\mathcal{G}_1$ and $\mathcal{G}_2$ make synchronized transitions. If either one of $\mathcal{G}_1$ or $\mathcal{G}_2$ transitions to the failure state, the entire design transitions to the failure state. It has been proved in [11] that the parallel composition of STGs is commutative and associative. The STG composition algorithm is shown in Figure 2.4.

2.3 Correctness Definitions of Asynchronous Designs

In Section 2.2, a special failure state π is used to denote that the design makes a wrong or unexpected state transition. In this section, we define the conditions under which transition firings cause asynchronous design failure.

In our method, a design is considered correct if none of the following failures are present in a model. There are four types of failures considered in our method: *safety failures*, *complement failures*, *disabling failures*, and *deadlocks*. These failures are defined as follows.

Definition 2.3.1. Let $\rho = (s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \cdots)$ be a path in a STG $\mathcal{G}$ where $R(s_i, t_i, s_{i+1})$ for all $i \geq 0$. Firing t_i causes

1. A *safety failure* if $(\mu(s_i) - {\bullet t_i}) \cap t_i{\bullet} \neq \emptyset$.
2. A *complement failure* if
   a. $t_i = w + \wedge\alpha(s_i)[w] = 1$, or
   b. $t_i = w - \wedge\alpha(s_i)[w] = 0$.
3. A *disabling failure* if $(\texttt{enabled}(s_i) - \{t_i\}) \not\subseteq \texttt{enabled}(s_{i+1})$.
4. A *deadlock* if $\texttt{enabled}(s_{i+1}) = \emptyset$.

Intuitively, a valid trace causes a *safety failure* if after firing a transition, the marking update adds a token to a place that already exists in the marking. The 1-safe requirement of Petri-nets is common for state space exploration algorithms. An unsafe net (i.e., one that is not 1-safe) typically indicates a problem with the underlying design. A valid trace causes a *complement failure* on wire w if there exist two rising (falling) events on w without a falling (rising) event in-between. Complement failures are a common modeling error and usually occur when the set and reset phase of a signal are similar. A *disabling failure* happens if the boolean guard of a satisfied rule becomes disabled before the corresponding enabled transition is fired. It may indicate a violation of hold time requirement of the underlying design. A valid trace causes a *deadlock* if a state is reached where there is no transition enabled. Given a BGPN N, the set of failure traces of N is $\mathcal{F}(N) \subseteq \mathcal{P}(N)$. Given two BGPNs N_1 and N_2, the following property holds:

$$\mathcal{F}(N_1) \subseteq \mathcal{F}(N_2) \;\text{ if }\; \mathcal{P}(N_1) \subseteq \mathcal{P}(N_2) \tag{2.1}$$

A design N is said correct or failure-free if $\mathcal{F}(N) = \emptyset$.

2.4 Conformance Relation

In this section, we describe conformance relation between models of a design and its implication to verification. The purpose of conformance relation is to find a model of smaller size for a design while preserving enough information for sound verification.

First, the definition of conformance is given as follows:

Definition 2.4.1. Given two BGPNs N_1 and N_2 where $W_1 = I_1 \cup O_1$ and $W_2 = I_2 \cup O_2$, N_1 *conforms* to N_2, denoted as $N_1 \preceq N_2$, if $I_1 = I_2$, $O_1 = O_2$, and $\mathcal{P}(N_1 \| N) \subseteq \mathcal{P}(N_2 \| N)$ for all N.

Theorem 2.4.1. Given BGPNs $N_1 \preceq N_2$ and $N_2 \preceq N_3$, then $N_1 \preceq N_3$.

Proof: According to Definition 2.4.1, $N_1 \preceq N_2$ implies $\mathcal{P}(N_1) \subseteq \mathcal{P}(N_2)$, and $N_2 \preceq N_3$ implies $\mathcal{P}(N_2) \subseteq \mathcal{P}(N_3)$. From $\mathcal{P}(N_1) \subseteq \mathcal{P}(N_2) \subseteq \mathcal{P}(N_3)$ we may conclude $\mathcal{P}(N_1) \subseteq \mathcal{P}(N_3)$ making conformance transitive.

∎

From the definition, we see that conformance is defined over the same set of input and output wires. This is sensible because conformance is often applied to models of a design at the different abstraction levels. According to Equation 2.1, if a concrete model conforms to an abstract one, then failures displayed by the concrete model are also displayed by the abstract one. In other words, any failures caught when verifying the abstract model includes those in the concrete one. If no failures are found in the abstract model, we can conclude the unverified concrete model also contains no failures. Therefore, the verification complexity can be greatly reduced by finding an abstract model if abstract model is smaller than the concrete model and the concrete model conforms to the abstract model.

The following lemmas show that conformance is preserved for parallel composition. The proof for Lemma 2.4.1 can be found in [21]

Lemma 2.4.1. Given BGPNs N_1, N_2, and N_3, $N_1 \| N_3 \preceq N_2 \| N_3$ if $N_1 \preceq N_2$.

Lemma 2.4.2. Given BGPNs N_1, N_2, N_3, and N_4, $N_1 \| N_2 \preceq N_3 \| N_4$ if $N_1 \preceq N_3$ and $N_2 \preceq N_4$.

Proof: According to Lemma 2.4.1, $N_1 \| N_2 \preceq N_2 \| N_3$ since $N_1 \preceq N_3$, and $N_2 \| N_3 \preceq N_3 \| N_4$ since $N_2 \preceq N_4$. Given $N_1 \| N_2 \preceq N_2 \| N_3 \preceq N_3 \| N_4$, according to Theorem 2.4.1 we conclude $N_1 \| N_2 \preceq N_3 \| N_4$.

∎

```
compose(G1, G2)
  unexplored = {(s1^0, s2^0)}
  while unexplored ≠ {ε}
    (s1, s2) = select( unexplored )
    explored = explored ∪(s1, s2)
    foreach t1 in G1 where w(t2) ∉ W(G2)
      if R1(s1, t1, s1') then
        R((s1, s2), t, (s1', s2))
        if (s1', s2) ∉ explored then
          unexplored = { unexplored ∪(s1', s2)}
      if R1(s1, t1, π) then
        R((s1, s2), t, π)
    end foreach
    foreach t2 in G2 where w(t2) ∉ W(G1)
      if R2(s2, t, s2') then
        R((s1, s2), t, (s1, s2'))
        if (s1, s2') ∉ explored then
          unexplored = { unexplored ∪(s1, s2')}
      if R2(s2, t, π) then
        R((s1, s2), t, π)
    end foreach
    foreach t1 in G1 and t2 in G2 where w(t1) ∈ D and w(t2) ∈ D
      if R1(s1, t1, s1') and R2(s2, t2, s2') and w(t1) = w(t2) then
        R((s1, s2), t, (s1', s2'))
        if (s1', s2') ∉ explored then
          unexplored = { unexplored ∪(s1', s2')}
      if R1(s1, t1, π) and R2(s2, t2, s2') and w(t1) = w(t2) then
        R((s1, s2), t, π)
      if R1(s1, t1, s1') and R2(s2, t2, π) and w(t1) = w(t2) then
        R((s1, s2), t, π)
      if R1(s1, t1, π) and R2(s2, t2, π) and w(t1) = w(t2) then
        R((s1, s2), t, π)
    end foreach
  end while
```

Figure 2.4 STG composition algorithm

CHAPTER 3

COMPOSITIONAL MINIMIZATION AND VERIFICATION

In this chapter, we describe our compositional approach to verification. The traditional flat approach fails when state space explosion occurs for large design models. Our methodology builds on the previous work of compositional verification. Like traditional compositional verification, we individually examine components of a system and then merge the results to form a global STG. To produce a reduced global STG which is an abstract model of the concrete global STG, the STG for each component is abstracted to reduce complexity. The reduced STGs are then incrementally composed. After each composition, the STG is again abstracted to remove state transitions that have been made internal by composition. When composition is complete, the global STG models the interface behavior of the concrete global STG. Throughout this approach, we preserve failures during composition and abstraction therefore no failure can be missed. We also prove that our compositional approach is sound in that no false positive results can be produced.

3.1 Framework of the Compositional Method

This section presents a general framework of compositional verification. In our method, a circuit is modeled as a set of parallel components formally described using BGPNs. Each BGPN describes a single component. Rather than attack the complexity of the complete system, components are assessed autonomously. Each component BGPN is extracted from the complete system and composed with another BGPN describing an over-approximated environment. The over-approximated environment reproduces all input behavior that would be supplied by the actual environment and perhaps more. State space exploration is then

```
verify(N = N_1 ‖ ··· ‖ N_n)
  find STG G_i for N_i ‖ E_i^approx   (1 ≤ i ≤ n)
   forall 1 ≤ i ≤ n, 1 ≤ j ≤ n, and i ≠ j,
    G = compose(G_i, G_j)
  if π reachable from s^0 of G then
    return "N has an failure"
  else
    return "N is correct"
  end if
```

Figure 3.1 Compositional verification algorithm

performed on each component to obtain an STG. Each STG describes all behavior a component can produce when it is embedded in the complete system. Then the component STGs are incrementally composed to form a global STG describing the complete system. The algorithm for compositional verification is shown in Figure 3.1.

The complete system N is described as the composition $\|_{i=1}^{n} N_i$ where each BGPN N_i describes a component of N. We begin by regarding the rest of the system as $\mathcal{E}_i$, the environment to N_i. For the component N_i, let $\mathcal{E}_i^{actual}$ denote the actual environment $\|_{j=1}^{n} N_j$ where $j \neq i$. Equation 3.1 describes the relationship between N and $N_i \| \mathcal{E}_i^{actual}$ for all i.

$$N \equiv N_i \| \mathcal{E}_i^{actual} \tag{3.1}$$

Often $\mathcal{E}_i^{actual}$ is a very complex BGPN. $\mathcal{E}_i^{actual}$ may contain many wires that are invisible to N_i. The wires invisible to N_i in the BGPN $N_i \| \mathcal{E}_i^{actual}$ may significantly contribute to state space explosion when finding $\mathcal{G}(N_i \| \mathcal{E}_i^{actual})$. To remove these invisible wires and decouple the component from its actual environment, we devise $\mathcal{E}_i^{approx}$ which is an over-approximation of $\mathcal{E}_i^{actual}$.

Definition 3.1.1. Given an arbitrary BGPN N, we say $\mathcal{E}^{approx}$ is an over-approximation of $\mathcal{E}^{actual}$ if $\mathcal{G}(N \| \mathcal{E}^{actual}) \preceq \mathcal{G}(N \| \mathcal{E}^{approx})$.

An over-approximated environment is simpler than the actual environment and exhibits at least as much behavior as the actual environment on the interface wires facilitating

communication between N_i and $\mathcal{E}_i$. For now we omit the procedure to produce such an approximation.

For each component in the complete system N, the algorithm seen in Figure 3.1 composes component N_i with its over-approximated environment $\mathcal{E}_i^{approx}$. State space exploration is then performed on each BGPN $N_i \| \mathcal{E}_i^{approx}$ to form an STG $\mathcal{G}_i$. The component STGs $\mathcal{G}_i$ are then incrementally composed to form the global STG $\mathcal{G}$. Theorem 3.1.1 proves that STG $\mathcal{G}$ produced by the flat approach conforms to the global STG $\mathcal{G}'$ produced by the compositional approach. According to Definition 2.4.1, $\mathcal{P}(\mathcal{G}) \subseteq \mathcal{P}(\mathcal{G}')$ such that all failure traces in $\mathcal{G}$ also appear in $\mathcal{G}'$. Therefore, if the failure state π is reachable from the initial state in the final composition $\mathcal{G}'$, then we cannot say the design is failure free. However, if there is no path from the initial state to the failure state, we can conclude the complete design is failure free.

Theorem 3.1.1. Let $N = \|_{i=1}^{n} N_i$, $\mathcal{G}$ be the STG derived from N, and $\mathcal{G}_i'$ be the STG derived from $N_i \| \mathcal{E}_i^{approx}$ where $1 \leq i \leq n$. The following equation holds:

$$\mathcal{G} \preceq \|_{i=1}^{n} \mathcal{G}_i'$$

Proof: For $1 \leq i \leq n$, $\mathcal{E}_i^{actual}$ is the composition of all the BGPNs of N excluding N_i. Therefore, $N \equiv \|_{i=1}^{n} N_i \equiv N_i \| \mathcal{E}_i^{actual}$. Let $\mathcal{G}_i$ be the STG derived from $N_i \| \mathcal{E}_i^{actual}$ and $\mathcal{G}_i'$ be the STG derived from $N_i \| \mathcal{E}_i^{approx}$. According to Definition 3.1.1, $\mathcal{G}_i \preceq \mathcal{G}_i'$. Similarly for $1 \leq j \leq n$, $\mathcal{G}_j$ is the STG derived from $N_j \| \mathcal{E}_j^{actual}$, and $\mathcal{G}_j'$ is the STG derived from $N_j \| \mathcal{E}_j^{approx}$. Since $N \equiv N_i \| \mathcal{E}_i^{actual} \equiv N_j \| \mathcal{E}_j^{actual}$, $\mathcal{G} \equiv \mathcal{G}_i \equiv \mathcal{G}_j$. Then according to Lemma 2.4.2, $\mathcal{G} \preceq \mathcal{G}_i' \| \mathcal{G}_j'$. The above argument can be repeated for all $1 \leq i,j \leq n$, proving Theorem 3.1.1. ∎

3.2 Abstraction

The compositional approach is chosen when the resources required by the flat approach exceed the available resources. The final STG produced by the algorithm seen in Figure 3.1

is at least as large as the STG produced in the flat approach. If the STG produced by the compositional approach is as large as that of the flat approach, the compositional approach holds no advantage. In order to reduce complexity and remain within the confines of available resources, abstraction is added to the compositional method.

In this section, we describe a method of abstraction that preserves the soundness of compositional verification. Abstraction removes details unnecessary to verification. To ensure the soundness of the abstraction method, traces are never removed from the possible trace set. Rather, traces are shortened by removing the unnecessary events from a trace. Any trace ending in the failure state will still end in the failure state after abstraction. We begin by partitioning the set of wires W of a component into two sets $\mathcal{V}$ and $\mathcal{D}$ where $W = \mathcal{V} \cup \mathcal{D}$. $\mathcal{V}$ is the set of interface wires and $\mathcal{D}$ is the set of internal wires. Given a component's STG $\mathcal{G}_i$, a wire $w \in W$ may be either an interface wire or an internal wire. Interface wires facilitate intercomponent communication. The interface wires of a component can be either inputs denoted I or outputs denoted O. The set of wires $\mathcal{D}$ only provide intracomponent communication. State transitions on these wires are not visible to other components. During abstraction, all state transitions on the wires in $\mathcal{D}$ are removed while state transitions on the wires in $\mathcal{V}$ are preserved. This results in a black box representation of the component where state transitions only occur on input and output wires.

Suppose there exists a state transition (s_i, t', s_j) in an STG $\mathcal{G}$ where $w(t') \in \mathcal{D}$. To abstract this state transition, we simply combine s_i and s_j to form a single merged state s_{ij}. All state transitions that either entered or exited s_i now enter or exit s_{ij}. The result is the same for state transitions entering or exiting s_j. Then the state transition t' in $\mathcal{G}$ is deleted. To preserve failure traces, if s_j is the failure state π, then the state formed by merging s_i and s_j is also the failure state π. By strictly adhering to this rule, failure traces are never lost. An example of abstraction is shown in Figure 3.2.

A side-effect of abstraction is that the possible trace set may be enlarged. For the STG $\mathcal{G}$ seen in Figure 3.3(a), all traces begin with the prefix $(t_0, t_1, t_2 \cdots)$. After abstracting t',

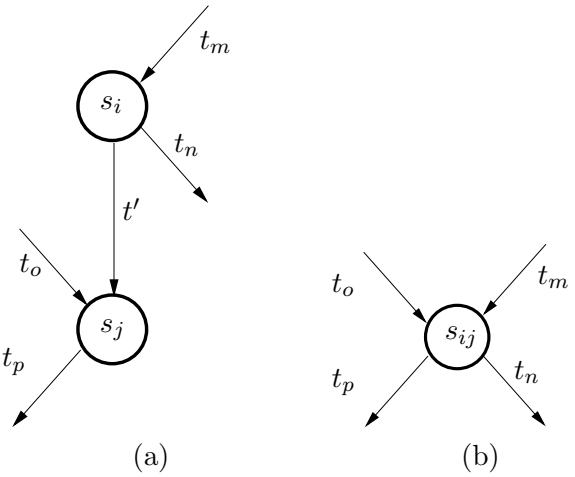

Figure 3.2 (a) STG before abstraction of t' (b) STG after abstraction of t'

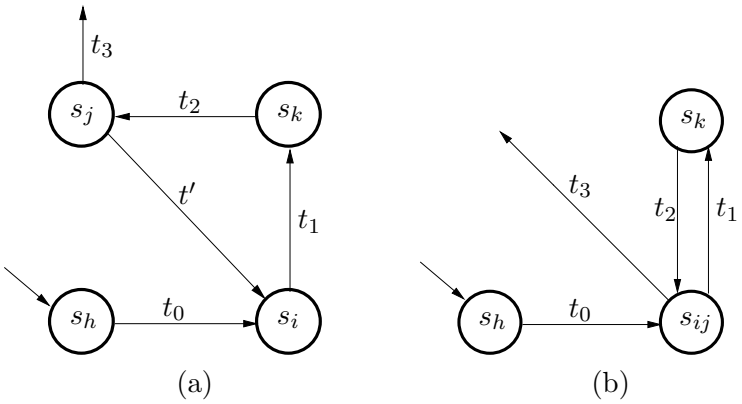

Figure 3.3 (a) STG before producing additional traces (b) STG after producing additional traces

the trace set for $\mathcal{G}$ has been enlarged such that all traces begin with the prefix $(t_0, t_1, t_2, \cdots)$ or $(t_0, t_3, \cdots)$. The abstracted STG containing additional traces is shown in Figure 3.3(b).

Additional failure traces created by abstraction are known as false failures. These false failures do not threaten the correctness of our method, but introducing additional false failures can increase the resource usage required during failure trace refinement. By first abstracting state transitions adjacent to the failure state, we can prevent abstraction from producing some false failures. Figure 3.4(a) shows an STG $\mathcal{G}$ containing two abstractable state transitions t'_3 and t'_4. If we abstract state transition t'_4 before t'_3, The trace $\sigma =$

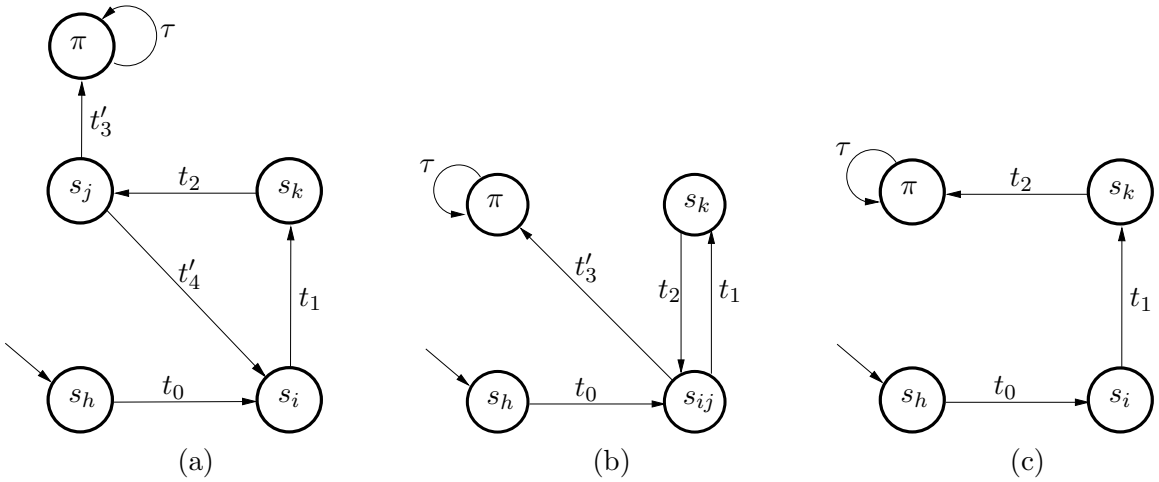

Figure 3.4 (a) STG before abstraction (b) Abstracted STG containing addition failure traces (c) Abstracted STG that does not contain additional failure traces

(t_0, t'_3, τ^*) is added to the trace set. The resulting STG is seen in Figure 3.4(b). However, if we first abstract t'_3, the state transition closest to the failure state, no new traces are added to the STG. The abstracted STG containing no additional traces is shown in Figure 3.4(c). The abstraction algorithm shown in Figure 3.5 attempts to avoid introducing false failures whenever possible by first abstracting internal transitions adjacent to the failure state.

To contain state space explosion during the compositional approach, we augment the original compositional verification algorithm to include abstraction. Before each composition, we abstract the STGs supplied as input to the composition function. Given any substantial number of internal wires, the composition can be significantly smaller than one for which the input STGs were not abstracted. The STG produced by composition with abstraction is usually much less complex than the STG produced by the flat approach. In order to maintain the soundness of the verification algorithm, $\mathcal{G}$ must conform to $\texttt{abstract}(\mathcal{G})$. As previously discussed, abstraction never removes traces from $\mathcal{P}(\mathcal{G})$, but simply shortens existing traces. In some cases, it may add traces to the possible trace set. Fortunately conformance allows this to happen. The following theorem proves that an STG $\mathcal{G}$ conforms to its abstraction $\mathcal{G}'$.

```
abstract(𝒢)
  foreach (s_i, t, π) ∈ R where w(t) ∈ 𝒟(𝒢)
    foreach (s_j, t', s_i) ∈ R
      replace (s_j, t', s_i) with (s_j, t', π)
    foreach (s_i, t', s_j) ∈ R
      replace (s_i, t', s_j) with (π, τ, π)
     S^π = S^π ∪ {s_j}
    delete R(s_i, t, π)
  foreach s_i ∈ S^π
    if unreachable(s_i)
      foreach (s_i, t, s_j) ∈ R
        replace R(s_i, t, s_j) with R(π, τ, π)
       S^π = S^π ∪ {s_j}
  foreach (s_i, t, s_j) ∈ R where w(t) ∈ 𝒟(𝒢)
    delete R(s_i, t, s_j)
    foreach (s_k, t', s_i) ∈ R
      replace (s_k, t', s_i) with (s_k, t', s_ij)
    foreach (s_i, t', s_k) ∈ R
      replace (s_i, t', s_k) with (s_ij, t', s_k)
    foreach (s_k, t', s_j) ∈ R
      replace (s_k, t', s_j) with (s_k, t', s_ij)
    foreach (s_j, t', s_k) ∈ R
      replace (s_j, t', s_k) with (s_ij, t', s_k)
```

Figure 3.5 Internal state transition abstraction algorithm

Theorem 3.2.1. Given an STG $\mathcal{G}$ where $\mathcal{V} = W - \mathcal{D}$, $\mathcal{G}[\mathcal{V}] \preceq \texttt{abstract}(\mathcal{G})$.

Proof: $\mathcal{G}[\mathcal{V}] \preceq \texttt{abstract}(\mathcal{G})$ if for every $\sigma \in \mathcal{P}(\mathcal{G})$ there exists $\sigma' \in \mathcal{P}(\texttt{abstract}(\mathcal{G}))$ such that $\sigma[\mathcal{V}] \equiv \sigma'$. Let $(s_h, t_i, s_i) \in R$, $(s_i, t', s_j) \in R$, and $(s_j, t_j, s_k) \in R$ where where $w(t') \notin \mathcal{V}$. For each $(s_i, t', s_j) \in R$, there are one or more paths in $\mathcal{G}$ containing (s_i, t', s_j). A path in $\mathcal{G}$ containing (s_i, t', s_j) is $\rho = (\cdots s_h \xrightarrow{t_i} s_i \xrightarrow{t'} s_j \xrightarrow{t_j} s_k \cdots)$. The trace corresponding to ρ is $\sigma = (\cdots, t_i, t', t_j, \cdots)$. To abstract (s_i, t', s_j), the states s_i and s_j are merged to form a single state s_{ij}. Then the state transition (s_i, t', s_j) is then deleted from R. This process is repeated for each instance of $(s, t', s') \in R$ in the path ρ where the BGPN transition $t' \notin \mathcal{V}$. As a result, $\texttt{abstract}(\rho) = \rho' = (\cdots s_h \xrightarrow{t_i} s_{ij} \xrightarrow{t_j} s_k \cdots)$. Its corresponding trace is $\sigma' = (\cdots, t_i, t_j, \cdots)$ where all t_i and t_j retain their relative order. If we project the trace σ to the set of wires $\mathcal{V}$, according to the definition of the projection function in Chapter 2, the result $(\cdots, t_i, t_j, \cdots)$ is equivalent to σ'. The same argument is applied for all traces in $\mathcal{G}$. Since $\mathcal{P}(\mathcal{G})[W'] \subseteq \mathcal{P}(\texttt{abstract}(\mathcal{G}))$, by Definition 2.4.1 $\mathcal{G}[\mathcal{V}] \preceq \texttt{abstract}(\mathcal{G})$. ∎

```
autofailure(𝒢)
   foreach (s_j, t, π) ∈ R where w(t) ∉ I(𝒢)
     if s_j = s^0
       return component failure
     replace (s_j, t, π) with (π, τ, π)
     foreach (s_i, t, s_j) ∈ R
       replace (s_i, t, s_j) with (s_i, t, π)
     foreach (s_j, t, s_k) ∈ R
      S^π = S^π ∪ {s_k}
       replace (s_j, t, s_k) with (π, τ, π)
   foreach s_i ∈ S^π
     if unreachable(s_i)
       foreach (s_i, t, s_j) ∈ R
         replace (s_i, t, s_j) with (π, τ, π)
        S^π = S^π ∪ {s_j}
```

Figure 3.6 Algorithm to backward propagate failures

3.3 Failure Backward Propagation

The failure state of an STG may be caused by an output or internal BGPN transition firing. However, in most cases the cause of the failure can be traced back to an input BGPN transition where the environment supplied some input the design could not handle. If an output or internal BGPN transition firing causes a failure, the state from which the BGPN transition fires is also a failure state. We refer to these states as failure states because the environment cannot prevent the failure from occurring. [21] refers to this new failure state as the *autofailure* state. However, [21] presents autofailure as a means of canonicalizing automatons representing asynchronous circuit designs. Like [21], we consider potential failure states to be failure states, though our motive differs. Our method is similar to [25] in that by treating potential failure states as actual failure states, we are able to reduce the size of an STG. In this section, we describe a method of shortening the representation of traces in an STG through autofailure reduction.

When a failure occurs, its autofailure state can be found by traversing each path in the STG backwards from π until an input event is reached. The state transition (s_i, t, π) replaces (s_i, t, s_j) where t is the input event found in the previous step, and all states that were reachable from s_j are marked as failure states. The algorithm shown in Figure 3.6

starts from the failure state π, and checks all incoming paths to π backwards. If, whenever during backward traversal, the initial state is encountered before an input transition is found, that indicates the component fails right from the initial state, and it is reported back to users right away. Otherwise, suppose there is a failure state transition (s_j, t, π) where the BGPN transition t is not an input. The state s_j is marked as a failure state, and the BGPN transition t is represented as τ. Then every state transition (s_i, t, s_j) is updated to (s_i, t, π). This process repeats until only state transitions on inputs enter the failure state. Then every state transition (s_k, t, s_l) that is reachable from the new autofailure state is changed to (π, τ, π).

Figure 3.7(a) is an STG consisting on input and non-input state transitions. The input transitions are denoted as (s, t, s') where $t \in I$. Similarly we use (s, t', s') where $t' \notin I$ to represent an output or internal transition. Autofailure reduction begins by locating a non-input state transition to the failure state. The state transition (s_k, t', π) is found, and the potential failure state s_k is marked as a failure state. Afterward, we convert the state transition (s_k, t', π) to (π, τ, π). By repeating this process we find another non-input state transition (s_i, t', π) entering the failure state. The process is repeated and s_i is marked as failure. Then states reachable from a failure state are marked as failure states. After autofailure, the state transition (π, τ, π) represents all transitions that exit the failure states and potential failure states.

Like abstraction, applying autofailure reduction immediately before each composition safely reduces peak complexity. Autofailure is safe because it does not remove traces from the possible trace set of a design. Chapter 2 introduced the special state transition (π, τ, π) which represents behavior occurring after the failure state with less detail. By applying backwards failure propagation, we reduced the size of state transition graph by representing many BGPN transition firings with the single state transition (π, τ, π). This creates an STG smaller in size. Theorem 3.3.1 proves that the STG $\mathcal{G}$ conforms to the autofailure of $\mathcal{G}$.

Theorem 3.3.1. Given an STG $\mathcal{G}$, $\mathcal{G} \preceq \texttt{autofailure}(\mathcal{G})$.

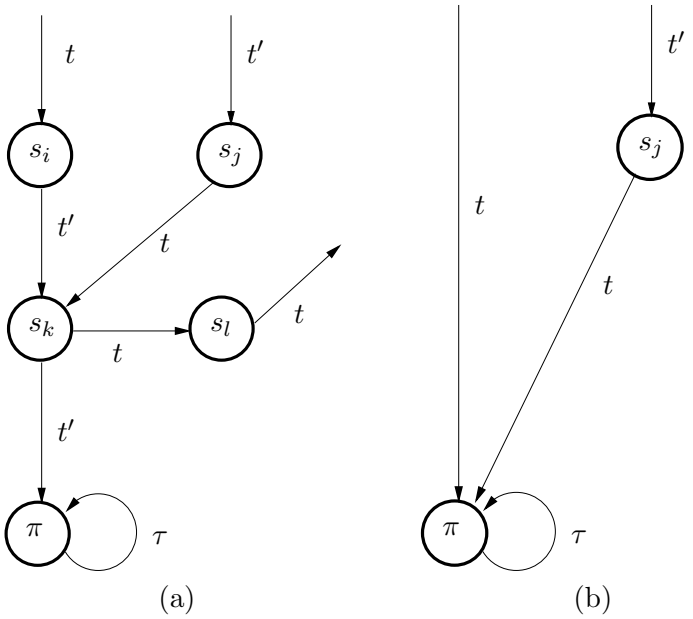

Figure 3.7 (a) STG before autofailure reduction (b) STG after autofailure reduction

Proof: Let $\mathcal{G}$ be an STG representing the firings of BGPN transitions t and t' where $w(t) \in I$ and $w(t') \notin I$. Let ρ be a path in $\mathcal{G}$ such that $\rho = (\cdots \xrightarrow{t} s_i \xrightarrow{t} s_j \xrightarrow{t'} s_k \xrightarrow{t'} \pi \xrightarrow{\tau} \pi \xrightarrow{\tau} \cdots)$. Its corresponding traces is $\sigma = (\cdots, t, t, t', t', \tau^*)$. Performing autofailure on $\mathcal{G}$ relabels the states and state transitions of ρ such that $\texttt{autofailure}(\rho) = \rho'$ and $\rho' = (\cdots \xrightarrow{t} s_i \xrightarrow{t} \pi \xrightarrow{\tau} \pi \xrightarrow{\tau} \pi \xrightarrow{\tau} \pi \xrightarrow{\tau} \cdots)$. σ' is the trace corresponding to ρ' where $\sigma' = (\cdots, t, t, \tau^*)$. After autofailure reduction, the trace $\sigma = (\cdots, t, t, t', t', \tau^*)$ has been reduced to $\sigma' = (\cdots, t, t, \tau^*)$. Recall that τ^* represents an infinite number of arbitrary transition firings in a BGPN N, thus the trace $\sigma' = (\cdots, t, t, \tau^*)$ is equivalent to the trace $\sigma'' = (\cdots, t, t, \tau, \tau, \tau^*)$. In the case of σ'', each τ represents a corresponding t' in σ, thus σ and σ' are equivalent traces differing only in notation. Given that $\mathcal{P}(\mathcal{G}) \subseteq \mathcal{P}(\texttt{autofailure}(\mathcal{G}))$, by Definition 2.4.1 we know $\mathcal{G} \preceq \texttt{autofailure}(\mathcal{G})$. ∎

3.4 Maximal Environment

As discussed above, we wish to decouple a component from the rest of the design by replacing the actual environment with an over-approximated environment. The simplest

approach is to use a *maximal environment*. The concept of the maximal environment was introduced in [5]. The maximal environment for a component may produce any event on any input wire at any moment. Its behavior is larger than any other environment possible for the same component in terms of conformance. The maximal environment is formally described in Definition 3.4.1.

Definition 3.4.1. Given an arbitrary BGPN N, we say $\mathcal{E}^{max}$ is the maximal environment to N if $\mathcal{G}(N\|\mathcal{E}) \preceq \mathcal{G}(N\|\mathcal{E}^{max})$ for all $\mathcal{E}$.

The more input behavior supplied to a design, the more output behavior that design may exude in response to the input. Design N would display all possible behavior if composed with the maximal environment. Therefore, $\mathcal{E}^{max}$ is an over-approximated environment. The maximal environment defines behavior for all input wires of a design. The behavior of each input wire is completely independent to other input wires; how each input wire changes its values is completely non-deterministic. In other words, the behavior of each input wire is unconstrained. Figure 2.2 in the previous chapter shows an AND-gate driven by a maximal environment. We also note that every design has an unique maximal environment.

3.5 Composition with Reduction

In the previous sections we have seen two methods to reduce the complexity of an STG. Both abstraction and autofailure may be combined to minimize the size of the final STG. For now, we choose our approximated environment to be the maximal environment because finding a more accurate approximation is non-trivial. Figure 3.8 shows an augmented verification algorithm that includes abstraction and autofailure. This new algorithm performs reductions on the STGs immediately before composition. Theorem 3.2.1 proves that any STG $\mathcal{G}_i$ conforms to its autofailure reduction $\mathcal{G}_i'$. Also, Theorem 3.3.1 proves that any STG $\mathcal{G}_i'$ conforms to its abstraction $\mathcal{G}_i''$. By using the output of autofailure as the input to abstraction, we can apply Lemma 2.4.1 and conclude $\mathcal{G}_i \preceq \mathcal{G}_i' \preceq \mathcal{G}_i''$.

```
verify(N = N_1 ‖ ··· ‖ N_n)
  find STG G_i for N_i ‖ E_i^max   (1 ≤ i ≤ n)
  ∀1 ≤ i ≤ n, 1 ≤ i ≤ n, and i ≠ j,
    G = compose(autofailure(abstract(G_i)), autofailure(abstract(G_j)))
  if π reachable from s^0 of G then
    return "N has an failure"
  else
    return "N is correct"
```

Figure 3.8 Compositional verification algorithm with autofailure and abstraction

CHAPTER 4

STATE SPACE REFINEMENT WITH CONSTRAINTS

As described in the previous chapter, compositional verification performs state space exploration for each component in the system. For each component, an over-approximated environment is used in place of the actual environment. A maximal environment is the worst case approximation. The input behaviors supplied by the over-approximated environment are a superset of those supplied by the actual environment. Supplying additional input behaviors can cause a design to produce additional output behavior. This extra behavior creates *inessential state space* during state space exploration. Inessential state space describes behavior which cannot occur when the component is embedded within its actual environment. Sometimes this inessential state space is removed when a component's STG is composed with the STG of its actual environment. However, the temporary existence of this inessential state space increases the peak number of states in the intermediate STGs during composition. The most undesirable outcome is for the inessential state space to appear in the final global STG. Larger state space consumes more memory and increases computation time. The inessential state space may also contain false failures. Every failure produced by our method must be verified through counter example refinement. Eliminating false failures can expedite the refinement and verification of true design failures. In this chapter, we present a method of constraint derivation that attempts to reduce these extra states by generating a more accurate approximation of the environment. This method can be used along with the reduction methods described in the previous chapter to further contain the peak size of the intermediate results during composition. We wish to produce a constrained over-approximated environment, $\mathcal{E}_i^{constrained}$ such that $\mathcal{E}^{actual} \preceq \mathcal{E}^{constrained} \preceq \mathcal{E}^{approx}$. We

first give the definition of constraints and how they affect BGPN firing semantics. We then describe how to derive them.

4.1 Constraint Definition

To create a more accurate approximation, we restrict the firings of BGPN transitions when their occurrence would cause an STG to enter inessential state space. In general, a constraint imposes additional logic for a BGPN transition firing that prevents such a transition from being fired unless the imposed constraint is satisfied. In a BGPN, a constraint is associated with each transition. The augmented definition of BGPNs including a boolean constraint mapping function C is given as follows.

Definition 4.1.1. A BGPN N is the tuple $(W, T, P, F, \mu^0, L, B, C)$ where C is the boolean constraint labeling function such that $C : T \rightarrow \mathtt{b}$ where $\mathtt{b}$ is a boolean expression over W, while the other elements in N are defined the same as in Definition 2.1.1.

The addition of the constraint mapping function changes the BGPN firing semantics. First, let $c(t)$ denote the constraint labeling a BGPN transition $t \in T$. Recall that the function $\mathtt{eval}(s, b)$, where $s = (\mu, \alpha)$, returns `true` if b evaluates to `true` with α and `false` otherwise.

Definition 4.1.2. Let N be a BGPN, and s a state of N. Given a BGPN transition $t \in T$, its constraint $c(t)$ is satisfied at a state s if $\mathtt{eval}(s, c(t)) = \mathtt{true}$.

For a BGPN transition to be enabled at a state, all enabling rules must be satisfied at that state; plus, the transition's constraint must also be satisfied at the same state. The modified definition of the BGPN transition enabling condition is given in Definition 4.1.3.

Definition 4.1.3. Let N be a BGPN, and s a state of N. A transition $t \in T$ is enabled at state s if

$$\Big(\mathtt{enabling\_rules}(t) = \mathtt{satisfied}(t, s)\Big) \ \wedge \ \Big(\mathtt{eval}(s, c(t)) = \mathtt{true}\Big).$$

The addition of constraints also requires modification of our failure definitions. The previous definition stated that a BGPN transition firing causes a failure if certain a condition is satisfied. With constraints, failure conditions are restricted and valid only when the constraints are also satisfied. In other words, transition firings are not considered when the constraints are not satisfied, even though these firings would cause failures without considering constraints. The updated failure definitions are given as follows.

Definition 4.1.4. Let $\rho = (s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \cdots)$ be a path in a STG $\mathcal{G}$ where $R(s_i, t_i, s_{i+1})$ for all $i \geq 0$, and $c(t_i)$ a constraint on t_i such that $\texttt{eval}(s_i, c(t_i)) = \texttt{true}$ for all $i \geq 1$. Firing t_i causes

1. A *safety failure* if $(\mu(s_i) - {\bullet t_i}) \cap t_i{\bullet} \neq \emptyset$.
2. A *complement failure* if
   a. $t_i = w+ \wedge \alpha(s_i)[w] = 1$, or
   b. $t_i = w- \wedge \alpha(s_i)[w] = 0$.
3. A *disabling failure* if $(\texttt{enabled}(s_i) - \{t_i\}) \not\subseteq \texttt{enabled}(s_{i+1})$.
4. A *deadlock* if $\texttt{enabled}(s_{i+1}) = \emptyset$.

In a STG $\mathcal{G} = (N, S, R, s^0)$, a BGPN transition $t \in T$ is enabled at a state $s \in S$ if $R(s, t, s')$ holds. According to the definition of constraints described above, a constraint is a condition that must be satisfied in every state where the BGPN transition is enabled. Therefore, $c(t)$ corresponds to a set of state transitions $R_{c(t)} \subseteq R$ such that BGPN transition t is enabled and its constraint $c(t)$ is satisfied at state s for every $(s, t, s') \in R_{c(t)}$. Obviously, the following property holds.

$$c_1(t) \rightarrow c_2(t) \Leftrightarrow R_{c_1(t)} \subseteq R_{c_2(t)} \tag{4.1}$$

where $c_1(t)$ and $c_2(t)$ are two constraints on t. This property states that the behavior in a model regarding t is reduced by imposing a stronger constraint on t, and vice versa. For example, $R_{c_2(t)}$ includes all state transitions $(s, t, s') \in R$ if $c_2(t) = \texttt{true}$, and $R_{c_1(t)} \subseteq R_{c_2(t)}$ for all other $c_1(t)$. Let $C(T')$ be a set of constraints for all BGPN transitions in $T' \subseteq T$.

$C_1(T') \rightarrow C_2(T')$ if $c_1(t) \rightarrow c_2(t)$ for every $t \in T'$. It is easy to see that the following property also holds.

$$C_1(T') \rightarrow C_2(T') \Leftrightarrow R_{C_1(T')} \subseteq R_{C_2(T')} \tag{4.2}$$

where $R_{c(T')} = \bigcup R_{c(t)}$ for all $t \in T'$ and $c(t) \in C(T')$.

Given a BGPN N and a set of constraints $C'(T')$ such that $T' \subseteq T$, we denote the imposition of constraints on N as $N \uparrow C'(T')$. This function updates the constraints of N with $C'(T')$ as shown in Definition 4.1.5.

Definition 4.1.5. Let N be a BGPN and $C'(T')$ be a set of constraints. $N \uparrow C'(T')$ updates $C(T)$ of N as follows.

$$\forall \ t \in T.\ c(t) \quad = \quad \begin{cases} c(t) \wedge c'(t)[W] & \text{if } t \in T' \\ c(t) & \text{otherwise} \end{cases}$$

This definition ensures that constraint update does not weaken the existing constraints in a BGPN. Since imposing constraints on BGPN transitions of a model reduces the possible state transitions caused by firing these BGPN transitions, this reduces the number of traces produced by state space exploration. We can also conclude that one constraint is stronger than the other if one model displays less behavior then the other in terms of possible traces. This conclusion is formalized in Lemma 4.1.1.

Lemma 4.1.1. Let $N = (W, T, P, F, \mu^0, L, B, C)$ be a BGPN, $C_1(T)$ and $C_2(T)$ two constraints on T. $C_1(T) \rightarrow C_2(T)$ if and only if $N \uparrow C_1(T) \preceq N \uparrow C_2(T)$

Proof: First, we prove that $N \uparrow C_1(T) \preceq N \uparrow C_2(T)$ if $C_1(T) \rightarrow C_2(T)$. Let $\rho = (s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots)$ be a path in $\mathcal{G}(N)$ such that it satisfies the conditions

$$\texttt{eval}(s_i, c_1(t_i)) = \texttt{true} \text{ and } R(s_i, t_i, s_{i+1}) \text{ holds for all } i \geq 0$$

where $c_1(t_i) \in C_1(T)$. We can find all paths satisfying the above condition and group them in $\mathcal{P}(N \uparrow C_1(T))$. Since $C_1(T) \rightarrow C_2(T)$, $\rho = (s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots) \in \mathcal{P}(N \uparrow C_1(T))$ also satisfies

the conditions

$$\texttt{eval}(s_i, c_2(t_i) = \texttt{true} \text{ and } R(s_i, t_i, s_{i+1}) \text{ holds for all } i \geq 0$$

where $c_2(t_i) \in C_2(T)$. Similarly, we can group all paths satisfying the above condition in $\mathcal{P}(N \uparrow C_2(T))$. Obviously, $\mathcal{P}(N \uparrow C_1(T)) \subseteq \mathcal{P}(N \uparrow C_2(T))$. Then according to the definition of conformance, we have $N \uparrow C_1(T) \preceq N \uparrow C_2(T)$.

Next, we prove that $C_1(T) \rightarrow C_2(T)$ if $N \uparrow C_1(T) \preceq N \uparrow C_2(T)$. Since $N \uparrow C_1(T) \preceq N \uparrow C_2(T)$, every $\rho = (s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots) \in \mathcal{P}(N \uparrow C_1(T))$ is also in $\mathcal{P}(N \uparrow C_2(T))$. Let $R_{C_1(T)}$ and $R_{C_2(T)}$ be the sets of state transitions extracted from every path in $\mathcal{P}(N \uparrow C_1(T))$ and $\mathcal{P}(N \uparrow C_2(T))$, respectively. And we have $R_{C_1(T)} \subseteq R_{C_2(T)}$. According to Equation 4.2, we have $C_1(T) \rightarrow C_2(T)$. This completes the proof. ∎

Lemma 4.1.2. Let N_1 and N_2 be two BGPNs, $C(T)$ constraints on T. If $N_1 \preceq N_2$, then $N_1 \uparrow C(T) \preceq N_2 \uparrow C(T)$.

Proof: Since $N_1 \preceq N_2$, we have $\mathcal{P}(N_1) \subseteq \mathcal{P}(N_2)$. For every path $\rho = (s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots)$ in $\mathcal{P}(N_1)$ such that for all $i \geq 0$ $c(t_i)$ is satisfied at state s_i on ρ where $c(t_i) \in C(T)$, it must be in $\mathcal{P}(N_2)$ too. Since such a path belongs to $\mathcal{P}(N_1 \uparrow C(T))$, it also belongs to $\mathcal{P}(N_2 \uparrow C(T))$. According to the definition of conformance, we have $N_1 \uparrow C(T) \preceq N_2 \uparrow C(T)$. ∎

Next, we show a lemma that is trivial but useful for proving a theorem later in this chapter.

Lemma 4.1.3. Let N be a BGPN, and $C(T)$ constraints on T such that every $c(t_i) \in C(T)$ is satisfied at state s_i for all $i \geq 0$ on every path $\rho = (s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots)$ in $\mathcal{P}(N)$. Then,

$$N \preceq (N \uparrow C(T)), \text{ and } (N \uparrow C(T)) \preceq N.$$

Proof: For every $\rho = (s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots)$ in $\mathcal{P}(N)$, every $c(t_i) \in C(T)$ is satisfied at state s_i for all $i \geq 0$, therefore, this path belongs to $\mathcal{P}(N \uparrow C(T))$. According to the definition of conformance, $N \preceq N \uparrow C(T)$.

To prove $N \uparrow C(T) \preceq N$, first notice that N is the same as $N \uparrow C'(T)$ where for every $t \in T$, $c'(t_i) \in C'(T)$ is `true`. Obviously, $C(T) \rightarrow C'(T)$. According to Lemma 4.1.1, $N \uparrow C(T) \preceq N$ holds. ∎

4.2 Constraint Derivation

In this section, we first describe how to derive the boolean constraint for a two component design. Next, we describe the more general case where a system consists of many components. We then show a simple example of constraint generation. Finally, we prove that constraint derivation does not invalidate the soundness of our compositional method.

Consider a two component system consisting of N_i and N_j where $I(N_i) \cap O(N_j) \neq \emptyset$. If we wish to perform constrained compositional verification, an over-approximated environment must be applied to each BGPN. Now suppose we wish to refine the over-approximated environment applied to N_i. The first step in constrained compositional verification is to produce the STG $\mathcal{G}(N_j \| \mathcal{E}_j^{approx})$, which we denote as $\mathcal{G}_j$. Now, $\mathcal{G}_j$ is regarded as the environment to N_i. For each wire $w_j \in \Big(I(N_i) \cap O(N_j)\Big)$, we examine $\mathcal{G}_j$ and identify the states where w_j+ and w_j- are enabled to fire. These states are collected and stored in two sets. S^+ is the set of states in the STG where w_j+ is enabled, and S^- is the set of states where w_j- is enabled. The state vectors of S^+ are disjuncted to form the boolean expression $c(w_j+)$. Similarly, the state vectors of S^- are disjuncted to form the boolean expression $c(w_j-)$. At this point $c(w_j+)$ is a boolean expression describing the states in the STG G_j where the BGPN transition w_j+ may be fired. Equally importantly, the negation of $c(w_j+)$ describes states where the BGPN transition w_j+ cannot be fired. $c(w_j-)$ provides a similar stipulation for the BGPN transition w_j-. Repeating this procedure for all $w_j \in \Big(I(N_i) \cap O(N_j)\Big)$ produces the set of constraints C_j, which is applied to BGPN N_i when it is considered. The algorithm for constraint derivation is shown in Figure 4.1.

Constraint derivation is similar for designs containing multiple components. Let us consider a design where the composition $\underset{i=1}{\overset{n}{||}} N_i$ forms the complete system N. From the

```
constrainPN(G, N)
  foreach w ∈ (I(N) ∩ O(G))
    S+ = {s ∈ S | R(s_i, t, s_{i+1})  and  t = w+}
    S− = {s ∈ S | R(s_j, t, s_{j+1})  and  t = w−}
    for all s1 ∈ S+ and s2 ∈ S+
      c(w+) = α(s1) ∨ α(s2)
    for all s1 ∈ S− and s2 ∈ S−
      c(w−) = α(s1) ∨ α(s2)
    C = C ∪ c(w+) ∪ c(w−)
  return (N ↑ C)
```

Figure 4.1 Algorithm to constrain an input of a BGPN

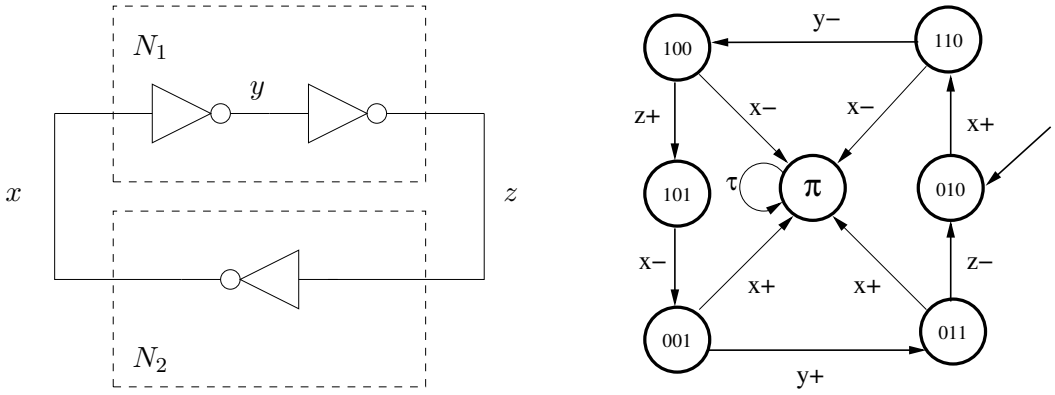

Figure 4.2 (a) Circuit diagram of an inverter composed with a buffer (b) $\mathcal{G}(N_1 \| \mathcal{E}_1^{max})$ where each binary vector corresponds to the wires x, y, and z

set of components we select a BGPN N_i. For N_i, its environment is $\|_{j=1}^{n} N_j$ where $j \neq i$. For each $N_j \| \mathcal{E}_j$ we produce an STG $\mathcal{G}_j$ from which C_j can be derived. We then constrain the environment of N_i by imposing constraints such that $N_i \| \mathcal{E}_i \uparrow C_j$ for all C_j.

As a simple example, Figure 4.2 (a) shows a circuit diagram of an inverter composed with a buffer. For this example, we shall consider the single inverter to be N_2 and the buffer to be N_1. During constrained compositional verification, we apply a maximal environment to both N_1 and N_2. We then perform state space exploration and autofailure reduction on $N_1 \| \mathcal{E}_1^{max}$. $\mathcal{G}(N_1 \| \mathcal{E}_1^{max})$ after reduction is shown in Figure 4.2(b). We now wish to constrain the BGPN $N_2 \| \mathcal{E}_2^{max}$ shown in Figure 4.3(a). z is the only wire in the set $I(N_i) \cap O(N_j)$. We search $\mathcal{G}(N_1 \| \mathcal{E}_1^{max})$ and store each state enabling $z+$ in the set S^+. Similarly, each

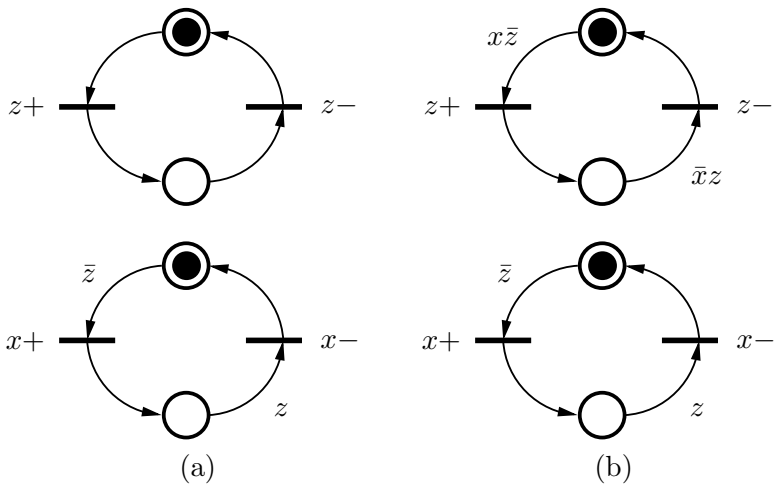

Figure 4.3 (a) $N_1 \| \mathcal{E}_1^{max}$ (b) $(N_2 \| \mathcal{E}_2^{max}) \uparrow C_2$

state enabling $z-$ is stored in S^-. From Figure 4.2(b) we see that S^+ and S^- each contain a single state. Disjuncting the state vectors in the set S^+ produces a boolean expression $c(z+) = x\bar{y}\bar{z}$. $c(z-) = \bar{x}yz$ is produced by disjuncting the state vectors of S^-. In this example, $c(z+)$ and $c(z-)$ form the set C_1. We then impose C_1 on the BGPN $N_2 \| \mathcal{E}_2^{max}$ using the function $(N_2 \| \mathcal{E}_2^{max}) \uparrow C_1$. By Definition 4.1.5, the boolean expression of each constraint is projected to the set of wires $W[N_2 \| \mathcal{E}_2^{max}]$. Then the derived constraints on wire z are conjuncted with existing constraint on z. For the maximal environment, constraints on z are `true`. Constraint imposition updates the constraints for $N_2 \| \mathcal{E}_2^{max}$ such that $c_2(z+) = x\bar{z}$ and $c_2(z-) = \bar{x}z$. The constrained BGPN $(N_2 \| \mathcal{E}_2^{max}) \uparrow C_2$ is shown in Figure 4.3(b). The reduction in inessential state space is shown in Figure 4.4, where the STG in Figure 4.4(a) is derived from $N_2 \| \mathcal{E}_2^{max}$ and the STG in Figure 4.4(b) is derived from $N_2 \| \mathcal{E}_2^{max} \uparrow C_2$.

The following theorem proves that our method of deriving and applying constraints to our compositional verification method maintains soundness

Theorem 4.2.1. Let $N = N_i \| N_j$, $N_j \preceq \mathcal{E}_i^{approx}$, and $N_i \preceq \mathcal{E}_j^{approx}$. Also, let $C(T)$ be the constraints derived for T of N from $\mathcal{G}(N_j \| \mathcal{E}_j^{approx})$. The following equation holds:

$$N_i \| N_j \preceq (N_i \| \mathcal{E}_i^{approx}) \uparrow C(T)$$

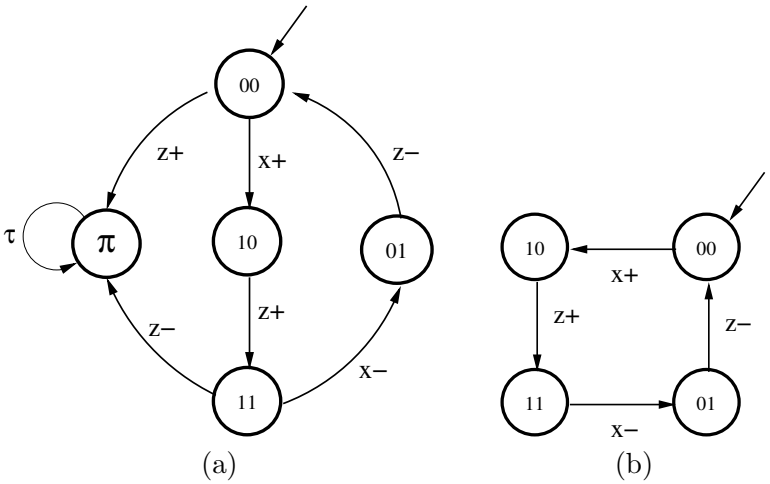

Figure 4.4 (a) $\mathcal{G}(N_1 \| \mathcal{E}_1^{max})$ (b) $\mathcal{G}((N_1 \| \mathcal{E}_1^{max}) \uparrow C_1)$

Proof: Let C_{ij}, C_i, and C_j be constraints derived from $N_i \| N_j$, $N_i \| \mathcal{E}_i^{approx}$, and $\mathcal{E}_j^{approx} \| N_j$, respectively, and $C_j = C(T)$. According to Lemma 4.1.3, we have

$$N_i \| N_j \preceq (N_i \| N_j) \uparrow C_{ij}. \tag{4.3}$$

Since $(N_i \| N_j) \preceq (N_i \| \mathcal{E}_i^{approx})$, according to Lemma 4.1.2, we have

$$(N_i \| N_j) \uparrow C_{ij} \preceq (N_i \| \mathcal{E}_i^{approx}) \uparrow C_{ij} \tag{4.4}$$

Since $(N_i \| N_j) \preceq (N_j \| \mathcal{E}_j^{approx})$, according to Lemma 4.1.1, $C_{ij} \rightarrow C_j$ holds. Next, according to Lemma 4.1.2, we can have the following equation:

$$(N_i \| \mathcal{E}_j^{approx}) \uparrow C_{ij} \preceq (N_i \| \mathcal{E}_j^{approx}) \uparrow C_j \tag{4.5}$$

Combining Equation 4.3, 4.4, 4.5, and $C_j = C(T)$, we can conclude that

$$N_i \| N_j \preceq N_i \| \mathcal{E}_i^{approx} \uparrow C(T)$$

∎

CHAPTER 5

EXPERIMENTAL RESULTS

The methods discussed in previous chapters have been implemented in the verification tool *SoftInspect*. To test the results of our reduction method we performed verification on three asynchronous designs: a FIFO controller, a tree arbiter, and a distributed mutual exclusion circuit. All designs have regular structures, thus simplifying creation of larger designs by replicating the same cells. However, the regularity is not exploited in all experiments.

The FIFO controller chosen is self-timed controlled as described in [22]. Figure 5.1 provides a high level description of a FIFO with n cells. The component connected to the left side of the FIFO is the producer. The right side of the FIFO is connected to the consumer. When the producer wishes to insert data into the FIFO, it first checks to see if there is room for additional data. If additional room exists, the data is added to the FIFO data structure. When the FIFO is not empty, it signals the consumer. The consumer then consumes the data after a period of time.

When the producer wishes to insert data, it makes a request by setting the value of wire *li* to high. The FIFO stores the data supplied by the producer. Then the FIFO acknowledges the producer by setting *lo* to high. After receiving the acknowledgment, the producer changes *li* to low. Once the FIFO is ready to accept additional input, it lowers *lo*. If the FIFO is full, *lo* remains high until the consumer consumes a unit of data. When the

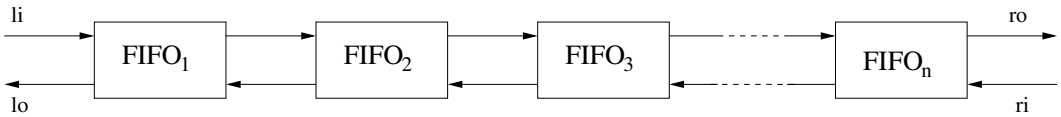

Figure 5.1 FIFO overview

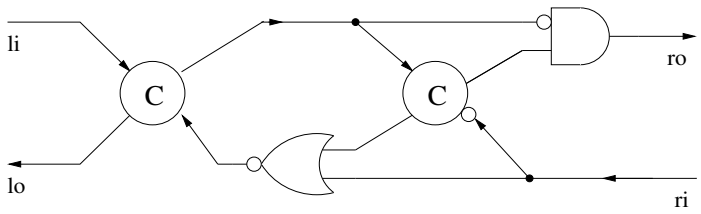

Figure 5.2 The control circuit for a single stage FIFO

Table 5.1 Truth table for the C-element

a	b	c
0	0	0
0	1	c
1	0	c
1	1	1

FIFO is ready to supply the consumer with data, ro is raised to high. Once the data is read, the consumer acknowledges the FIFO by setting ri to high. When the data transaction is completed, and the FIFO lowers its request on ro. The wire ri remains high until the consumer finishes processing the data it read. ri is lowered when ro is low and all data processing is completed. Figure 5.2 shows the control circuit for a single stage of the FIFO without data storage.

The circuit consists of an AND-gate, a NOR-gate, and two C-elements. A C-element is a common component of asynchronous circuit designs. It is represented as a circle with a "C" in the center. The circuit has two inputs and a single output. Its output is low when both inputs are also low. Similarly, its output is high when both inputs are high. The C-element's output retains its previous value for all other input vectors. The truth table for the C-element is shown in Table 5.1.

The second design used in testing our method is the distributed mutual exclusion element in [23]. The distributed mutual exclusion element (DME) is a self-timed circuit which allows multiple devices to use a single shared resource. A master M may request access to the shared resource by communicating with its server. For every master device, there is one server. The servers are connected in a ring, and each server communicates with its

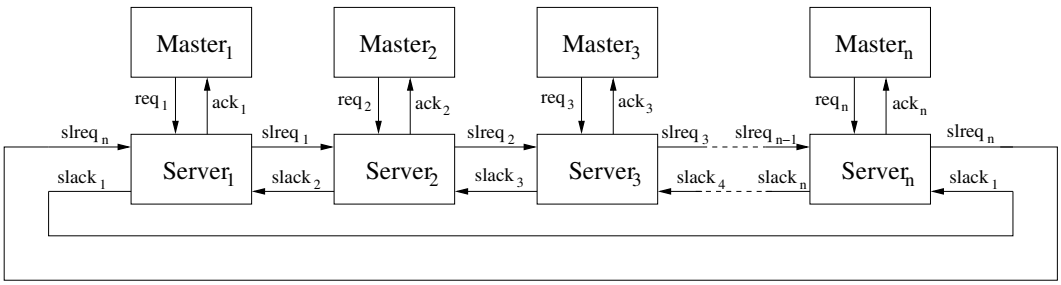

Figure 5.3 DME overview

neighboring servers. At any given time, one server in the ring holds 'privilege'. 'Privilege' gives a server the exclusive right to grant its master access to the resource. Figure 5.3 shows the high level architecture of a design containing n masters and servers. If a server has 'privilege' and its master requests access to the resource, access can be immediately granted. If the server is not 'privileged', it requests privilege from the neighbor to its right. If a server receives a request for privilege from its left-hand neighbor and it is unprivileged, it propagates the request it the server on its right. When the request reaches a server with privilege, privilege is passed to the left. In this manner, requests for privilege are transmitted clockwise in the ring, while privilege is passed counter-clockwise in the ring from one server to another. Should both the left neighbor and the master request access to the resource at the same time, the circuit contains an arbiter which chooses exactly one request to satisfy. Each DME cell also contains an SR-latch. The latch is set to high when a server has privilege. The circuit implementation for a single DME cell is shown in Figure 5.4. Full details of the implementation may be found in [23].

The third circuit chosen to test our method is an arbiter. The arbiter described in [21] allows several devices to share a resource. A single arbiter accepts requests from two users and grants a single user access to a resource at any given time. To allow more users to access the same resource, multiple arbiters may be connected as shown in Figure 5.5. In a multi-celled arbiter, a cell lower in the hierarchy regards another cell higher in the hierarchy as the server. For the experiments shown in this chapter, we connect the arbiters as a complete or nearly complete binary tree.

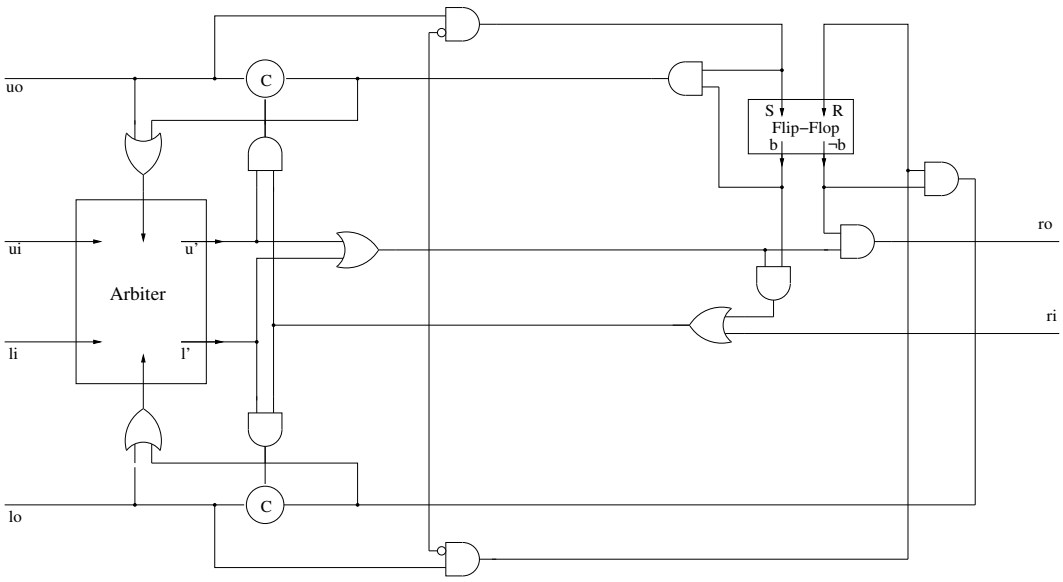

Figure 5.4 DME server circuit implementation

Figure 5.6 shows the circuit implementation for a single arbiter cell. In the arbiter implementation, user requests for the resources are sent to a mutual exclusion element. The mutual exclusion element ensures that the rest of the circuit receives exactly one request for the resource at any time. When a user i makes a request, it sets the wire ur_i to high. The arbiter then requests access to the server by setting sr_i to high and waits. When the server grants the request, it raises sa to high. The arbiter propagates this message to the user by setting ua_i to high. When the user finishes using the resource, it lowers ur_i which causes the arbiter to lower sr. The server then lowers sa. Then then arbiter lowers ua_i, completing the transaction between the user and the server.

Compositional verification requires us to partition the design into components. For simplicity we partition our designs such that each component consists of a single cell. The order in which we chose to compose these components was dictated by the interface of each design. In general, we attempted to minimize the number of interface signals present in a component. To do this we strictly composed component with common interface wires. This allows for early abstraction as interface wires become internal to a component after composition. For the FIFO design, we begin with the cell adjacent to the producer and

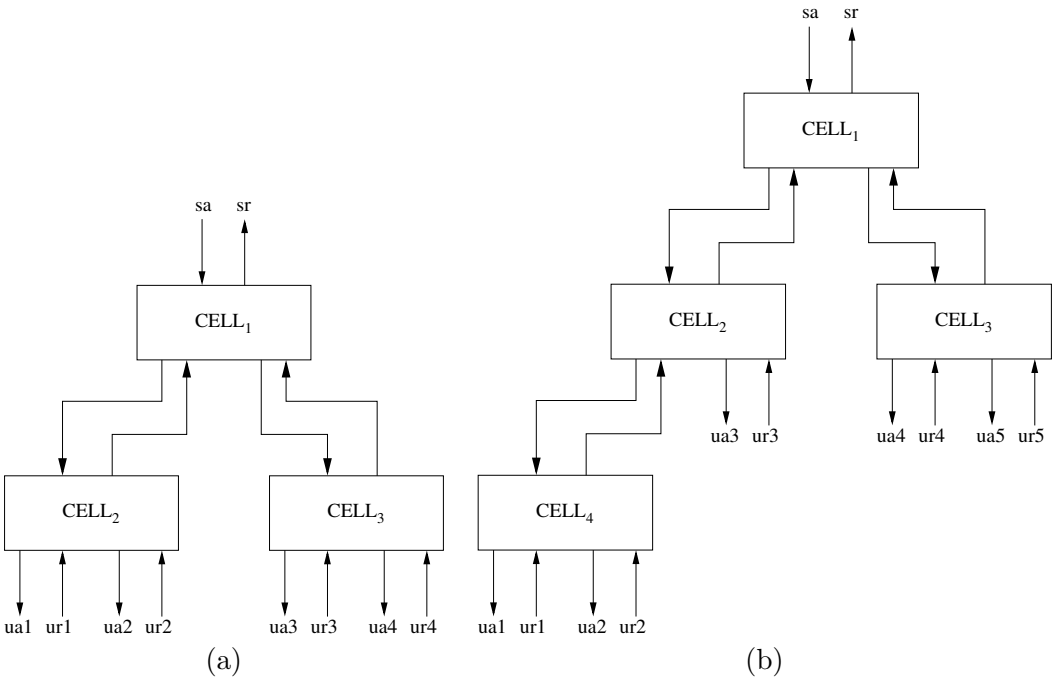

Figure 5.5 (a) Three cell arbiter (b) Four cell arbiter

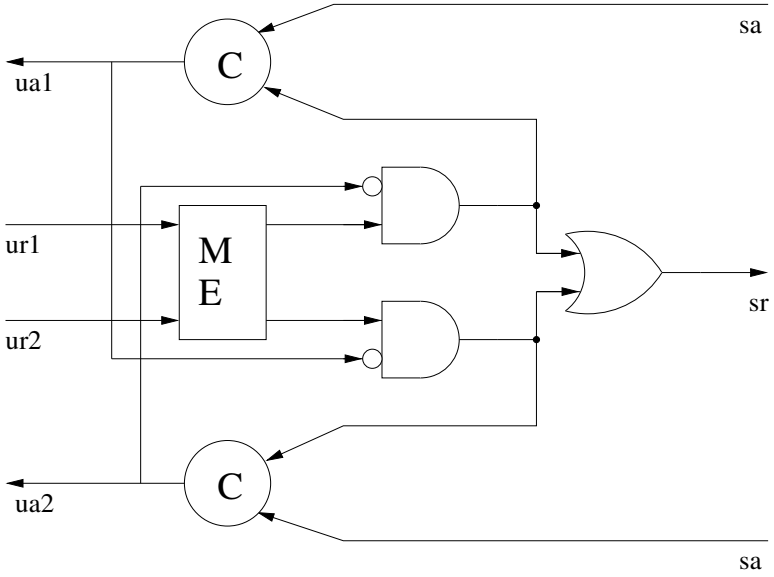

Figure 5.6 Arbiter circuit implementation

iteratively compose each adjacent cell until we reached the consumer. Since the DME design is circular, where we begin began the process of composition has no effect. Optimal composition for the DME design simply requires us to compose components with a common interface and attempt to maintain components of a balanced size. For the arbiter example we begin at the bottom of the tree and incrementally compose components as we move up the tree. Because we form our arbiter trees as complete or nearly complete binary trees, the components of the final composition are fairly balanced in size.

Table 5.2 shows the statistics of each design in BGPN and the verification results from using the flat approach. Each row characterizes a design containing a particular number of cells. Columns three, four, and five describe the sizes of designs in BGPN. $|P|$ is the number of places, $|T|$ is the number of BGPN transitions, and $|W|$ is the total number of wires in a design. Columns six, seven, eight, nine, and ten describe an STG produced by flat state space exploration. $|S|$ is the number of states, and $|R|$ is the number of state transitions. Time is recorded in seconds and memory in megabytes. All time and memory statistics greater than one are rounded. The column labeled π indicates whether or not the failure state is reachable for an STG. From the table, we see that the size of the BGPNs grows linearly while the size of the STGs grow exponentially as the number of cells increases. With regard to the number of cells verified, the FIFO circuit seems to do well compare to the other two designs. However, if we compare the FIFO design to the other designs in terms of places, transitions, and wires, the FIFO design performs the poorest during flat verification. This underscores the unpredictability of state space explosion. We note that all of the designs are shown to be correct during flat verification.

Compositional verification of the same designs without using constraints is shown in Table 5.3. For each component, the maximal environment is used as an over-approximated environment. $|S_F|$ and $|R_F|$ describe the size of the global STG produced by compositional verification. However, the more important metric of this method is the peak number of states and state transitions. $|S_P|$ and $|R_P|$ describe the number of states and state transitions for the largest STG produced during verification. The FIFO experiments produce

Table 5.2 Statistics for designs in BGPN and resources consumed by traditional flat approach

Design	# Cells	Design Statistics			Flat				
		$\|P\|$	$\|T\|$	$\|W\|$	$\|S\|$	$\|R\|$	Time	Mem	π
FIFO	2	20	20	10	116	240	<1	<1	N
	3	28	28	14	644	1724	<1	<1	N
	4	36	36	18	3620	11968	1	3	N
	5	44	44	22	20276	79644	8	21	N
	6	52	52	26	113684	517520	72	152	N
	7	60	60	34	*	*	*	*	N
	8	68	68	38	*	*	*	*	N
	9	76	76	42	*	*	*	*	N
	10	84	84	46	*	*	*	*	N
DME	2	48	60	22	1052	2770	<1	1	N
	3	72	90	33	53094	215847	27	59	N
	4	96	120	44	*	*	*	*	N
	5	120	150	55	*	*	*	*	N
	6	144	180	66	*	*	*	*	N
	7	168	210	77	*	*	*	*	N
	8	192	240	88	*	*	*	*	N
ARB	2	34	36	18	444	1054	<1	<1	N
	3	49	52	26	3756	11600	1	4	N
	4	64	68	34	30164	116776	21	36	N
	5	79	84	42	227472	1041792	254	347	N
	6	94	100	50	*	*	*	*	N
	7	109	116	58	*	*	*	*	N
	8	124	132	66	*	*	*	*	N

$*$ Indicates the design was too large to complete.

Table 5.3 Experimental results for compositional verification without constraints

Design	# Cells	$\|S_F\|$	$\|R_F\|$	$\|S_P\|$	$\|R_P\|$	Time	Mem	π
FIFO	2	43	88	56	160	<1	<1	Y
	3	4	3	63	252	<1	<1	Y
	4	4	3	63	252	<1	<1	Y
	5	4	3	63	252	<1	<1	Y
	6	4	3	63	252	<1	<1	Y
	7	4	3	63	252	<1	1	Y
	8	4	3	63	252	<1	1	Y
	9	4	3	63	252	<1	1	Y
	10	4	3	63	252	<1	1	Y
DME	2	49	86	360	1236	<1	1	N
	3	230	543	995	4206	<1	3	N
	4	823	2444	995	4206	1	5	N
	5	2467	8760	8335	42864	6	20	N
	6	7039	29040	8335	42864	16	38	N
	7	18735	87728	71859	436592	554	189	N
	8	48895	256000	71859	436592	714.49	350	N
ARB	2	159	377	646	2924	<1	1	Y
	3	603	1938	701	3485	<1	2	Y
	4	2254	9163	2254	9163	2	5	Y
	5	4347	20502	4347	20502	10	13	Y
	6	15083	84278	15083	84278	25	31	Y
	7	24843	147466	24843	147466	34	49	Y
	8	63275	421110	63275	421110	350	123	Y

extremely small results because each STG quickly enters the failure state. By using the maximal environment, we have introduced false failures. We know these failures are false because the flat STGs contained no failure states. These failures can also be proved false using failure trace verification as described in [2]. The STGs for the DME designs contain no failures. The resources consumed by compositional verification of DME are significantly less than the flat approach. This is due to the many internal wires which we abstract. The peak size of the arbiter is also much less than the flat STG. Unfortunately, the maximal environment used in the compositional approach also produces false failures in the arbiter examples.

Table 5.4 shows the results of compositional verification with constraint generation. The most apparent differences is the increased STG size for the FIFO designs. In the previous method, the states and transitions that occurred after the false failures were recorded with the single state transition (π, τ, π). By applying constraints, we have removed all of the false failures from the FIFO design, thus increasing the size of the STG representation. The FIFO design exhibits near linear growth under constrained verification and does not requires failure trace verification. Applying constraints to the DME design reduces run-time, memory usage, and the size of the largest intermediate STG. For the DME design, constrained verification produces the same verification results as unconstrained verification, but it does it much more efficiently. This is because constraints eliminate most of the extra behavior cause by the over-approximated environment. The benefits of constraints are not as dramatic for the arbiter design. The constraints remove some false failures, but the final global STG still contains failure. There is little benefit for the arbiter design with respect to peak STG size, but constraints do decrease the run-time significantly by preventing some extra behavior from appearing in the intermediate STGs.

Table 5.4 Experimental results for compositional verification using constraints

Design	# Cells	$\|S_F\|$	$\|R_F\|$	$\|S_P\|$	$\|R_P\|$	Time	Mem	π
FIFO	2	43	80	56	160	<1	<1	N
	3	91	192	91	192	<1	<1	N
	4	139	304	139	304	<1	<1	N
	5	187	416	187	416	<1	1	N
	6	235	528	235	528	<1	1	N
	7	283	640	283	640	<1	2	N
	8	331	752	331	752	<1	2	N
	9	379	864	379	864	<1	3	N
	10	427	976	427	976	<1	4	N
DME	2	49	86	360	1236	<1	1	N
	3	230	543	447	1152	1	2	N
	4	823	2444	775	2700	1	5	N
	5	2467	8760	3935	13872	4	13	N
	6	7039	29040	6815	30208	13	30	N
	7	18735	87728	34383	153424	56	107	N
	8	122688	256000	59540	319700	472	223	N
ARB	2	159	377	646	2924	<1	1	Y
	3	603	1938	701	3485	2	3	Y
	4	2254	9163	2254	9163	3	7	Y
	5	4459	20414	4459	20414	11	14	Y
	6	15511	84492	15511	84492	24	32	Y
	7	25983	146240	25983	146240	28	47	Y
	8	67711	432728	67711	432728	286	123	Y

CHAPTER 6

CONCLUSION

Asynchronous designs contain complex protocols which may hide failures occurring deep in the state space. Traditional approaches like simulation are unlikely to discover deep failures. Because model checking exhaustively explores all the behavior of a design, it either guarantees that a design is correct or produces a counter-example. The guarantee of correctness comes with a heavy price. All approaches to model checking suffer from state space explosion. This thesis has presented two methods to contain state space explosion and efficiently verify asynchronous designs.

6.1 The Compositional Framework

As illustrated in Chapter 5, it is impossible to verify most designs as a single unit. By partitioning a design, our method reduces concurrency which is the driving force of state space explosion. Although approximating an environment may elicit additional behavior from a component, our method scales much better than state space exploration of a flat design. Abstraction is largely responsible for producing state transition graphs of manageable size. The greatest reductions occur when a component contains many concurrent internal behaviors. Although the DME and arbiter examples show exponential growth in state space under the compositional approach, there is a significant reduction in state space size relative to the flat approach. Without constraints, our method fails to prove the correctness of the arbiter and FIFO designs, but its strength is exemplified by correctly verifying four times as many DME cells as the flat approach. An improved version of this method could use failure trace verification and refinement to completely verify the FIFO and arbiter examples. This

improvement would expand abstracted failure traces to create concrete failure traces. We could then use the original BGPN model to verify the concrete trace is a valid failure.

Future works include automated partitioning of a design and optimal composition ordering. For large designs, partitioning a design into appropriately sized components is very time consuming and non-trivial. Though one component may contain many more wires than another, the concurrency of the behavior on those wires dominates state space explosion. When partitioning a design, the user must also consider the degree to which the concurrency is restricted. Often, two wires may change values concurrently, but their transitions may be restricted by some complex protocol. It is difficult to identify the degree of state space explosion without performing state space exploration. Extremely large designs will require an automated partitioning heuristic.

Many of the problems of partitioning are also the problems of optimal composition order. When composing state transition graphs, some orderings produce much higher peak graph sizes than others. The worst possible composition is the state transition graphs of two components that do not communicate over mutual interface wires. The state size of this composition is the cross-product of the states in each component. Usually we attempt to combine components which communicate over some mutual set of interface wires. If these mutual wires are internal to the component after composition, abstraction removes them thereby reducing the size of the state transition graph. When composing components, we must also consider the number of inputs which are driven by a maximal environment. If an inefficient ordering combines several maximal environments, a state transition graph of the maximal environments alone may exceed the available resources. Efficient orderings are essential for verifying large designs. Inspection is a poor approach to ordering and often leads to trial and error. Automating ordering efficiently will remove a large burden from the user.

6.2 Constraints

The most obvious weaknesses in the compositional framework is the creation of false failures. When performing compositional verification without constraints, false failure traces cause the FIFO design to have peculiarly small state transition graphs. These false failures have then been introduced by the maximal environment. By generating constraints we restrict the behavior supplied by the maximal environment and prevent these false failures from occurring. The power of this method is demonstrated by the near linear state space growth of the FIFO example. We also see respectable reductions in the state space size of the DME example. Constraints actually increase the size of the state transition graph for the arbiter example. By removing false failures, fewer state transition firings are represented using the single state transition (π, τ, π) thereby increasing the number of state transitions in the state transition graph. Although the state space size of the arbiter is increased by constraints, designs with a large number of arbiter cells benefit from significant run-time reductions when constraints are applied. This is because the constrained environment reduces the amount of behavior produce by each component thus reducing the time to abstract each component.

Often, the generated constraints restrict the over-approximated environment such that it produces less behavior than the maximal environment but more behavior than the actual environment. This means that we sometimes derive constraints that are not optimal. We can identify two primary causes for this. The first is simply a characteristic of the design. When the output of the state transition graph depends on some internal signal, our method may not be able to identify the a dependency on some input transition in the state transition graph. Suppose the value of an internal register determines output behavior of a state transition graph. Constraints generation will produce a boolean expression representing this, but the expression is not useful to other components because the register's value is not visible on any interface wires. The over-approximated environment is the second cause for sub-optimal constraints. When creating a state transition graph, we apply over-approximated environ-

ment to the component. As previously discussed, the over-approximated environment may introduce extra behavior into the state transition graph. If this state transition graph is then used for constraints generation, the extra states may weaken the derived constraints. Future works should identify starting component least affected by constraints. Another solutions may include iterative refinement of the constraints. By repeatedly applying constraints and deriving new state transitions graphs, we may be able to find some fixed point in terms of constraint strength. Given enough dependency in the communication protocols, we should be able to derive optimal or near-optimal constraints.

REFERENCES

[1] H. Zheng, E. Mercer, and C. Myers, "Modular verification of timed circuits using automatic abstraction," *IEEE Transactions on Computer-Aided Design*, vol. 22, no. 9, pp. 1138–1153, 2003.

[2] H. Zheng, C. Myers, D. Walter, S. Little, and T. Yoneda, "Verification of timed circuits with failure directed abstractions," *IEEE Transactions on Computer-Aided Design*, vol. 25, no. 3, pp. 403–412, 2006.

[3] J. Misra and K. M. Chandy, "Proofs of networks of processes," *IEEE Trans. on Software Eng.*, vol. SE-7, no. 4, pp. 417–426, 1981.

[4] C. Jones, "Tentative steps toward a development for interfering programs," *ACM TOPLAS*, vol. 5, no. 4, pp. 596–619, 1983.

[5] O. Grumberg and D. E. Long, "Model checking and modular verification," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, pp. 843–871, May 1994. [Online]. Available: citeseer.ist.psu.edu/grumberg91model.html.

[6] T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "You assume, we guarantee: Methodology and case studies," in *Proc. Int. Conf. on Computer Aided Verification*. Springer-Verlag, 1998, pp. 440–451.

[7] K. L. Mcmillan, "A methodology for hardware verification using compositional model checking," Cadence Berkeley Labs, Tech. Rep., 1999.

[8] J. M. Jensen, D. Giannakopoulou, and C. S. Pasareanu, "Learning assumptions for compositional verification," in *LNCS*, vol. 2619, 2003, pp. 331–346.

[9] S. Graf and B. Steffen, "Compositional minimization of finite state systems," in *Computer Aided Verification*, 1990, pp. 186–196. [Online]. Available: citeseer.ist.psu.edu/graf91compositional.html.

[10] S. C. Cheung and J. Kramer, "Context constraints for compositional reachability analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 4, pp. 334–377, 1996.

[11] ——, "Checking safety properties using compositional reachability analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 8, no. 1, pp. 49–78, 1999.

[12] J.P. Krimm and L. Mounier, "Compositional state space generation from Lotos programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Brinksma, Ed. Enschede, The Netherlands: Springer Verlag, LNCS 1217, 1997, pp. 239–258.

[13] D. Bustan and O. Grumberg, "Modular minimization of deterministic finite-state machines," in *Proceedings of the 6th International workshop on Formal Methods for Industrial Critical Systems (FMICS'01)*, 2001.

[14] E. Clarke, O. Grumberg, and D. Long, "Model checking and abstraction," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, pp. 1512–1542, 1994.

[15] D. Dams, R. Gerth, and O. Grumberg, "Abstract interpretation of reactive systems," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 2, pp. 253–291, 1997.

[16] H. E. Jensen, K. G. Larsen, and A. Skou, "Scaling up uppaal automatic verification of real-time systems using compositionality and abstraction," in *FTRTFT*, 2000, pp. 19–30. [Online]. Available: citeseer.nj.nec.com/jensen00scaling.html.

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

[18] K. Larsen, B. Steffen, and C. Weise, "A constraint oriented proof methodology," in *Formal Systems Verification*, ser. LNCS, vol. 1169. Springer-Verlag, Nov. 1996, pp. 405–435.

[19] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *The 29th Symposium on Principles of Programming Languages*, Jan. 2002, pp. 58–70.

[20] E. Mercer, "Correctness and reduction in timed circuit analysis," Ph.D. dissertation, University of Utah, 2002.

[21] D. Dill, "Trace theory for automatic hierarchical verification of speed independent circuits," Ph.D. dissertation, Carnegie Mellon University, 1988.

[22] A. J. Martin, Self-timed fifo: An exercise in compiling programs into vlsi circuits, California Institute of Technology, Tech. Rep. 1986.5211- tr-86, 1986.

[23] A. J. Martin, The Design of a Self-timed Circuit for Distributed Mutual Exclusion, California Institute of Technology, Tech. Rep. 1983.5097- tr-83, 1983.

[24] J. Cobleigh, D. Giannakopouluo, and C. Pasareanu. "Learning assumptions for compositional verification." In *Proc. Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. Springer-Verlag, 2003.

[25] D. Giannakopoulou, C. Pasareanu, and H. Barringer. "Assumption generation for software component verification." In *Proceedings of the 17th Int. Conference on Automated Software Engineering*, Sept. 2002.

[26] S. Chaki, E. Clarke, N. Sinha, and P. Thati. "Automated assume-guarantee rea- soning for simulation conformance." In *Proc. International Workshop on Computer Aided Verification*. Springer-Verlag, 2005.

[27] R. Alur, P. Madhusudan, and W. Nam. "Symbolic compositional verification by learning assumptions." In *Proc. International Workshop on Computer Aided Veri- fication.* Springer-Verlag, 2005.