January 2012

Methods and Algorithms for Scalable Verification of Asynchronous Designs

Haiqiong Yao
University of South Florida, yaohaiqiong@gmail.com

Methods and Algorithms for Scalable Verification of Asynchronous Designs

by

Haiqiong Yao

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Date of Approval:
March 23, 2012

Keywords: Formal Method, Model Checking, Compositional Verification, Logic Verification, Abstraction Refinement

ACKNOWLEDGMENTS

I will remember the years in pursuing my PhD degree as a gratifying and learning experience under the support of my major advisor, the faculty and the staff in my department, my parents, and my friends.

I am especially grateful to my advisor, Prof. Hao Zheng, who introduced me to model checking area and gave me priceless guidance and support in my research. His enthusiasm and direction helped me keep on the right track throughout my graduate studies. His trust in me encouraged me to overcome the hardest time in research. Besides the technical knowledge, the most important things he taught me are non-technical. He has always encouraged me to keep curiosity and think creatively, pursue excellence and believe in myself. He has taught me how to face failure when my research ideas turned out fruitless and papers were rejected. He shared happiness for any progress I have made and also my stress while facing a tough job market. He has patiently helped me improve my writing and speaking skills, especially for this dissertation and defense. I am lucky to be one of his students.

I would like to thank my committee members. Prof. Nagarajan Ranganathan broaden my vision on research and has kept an eye on me over the years. Prof. Srinivas Katkoori provided me with great advice on how to be a good graduate student. Prof. Sanjukta Bhanja gave me the guidance on being a woman in computing technology and played a role model for me. Prof. William Stark provided insightful comments on my dissertation and taught me how to balance the work and life.

I would specially like to thank my parents for their love and keen support. They gave me the strength to complete my PhD degree. I also would like to thank Jing Zhang, Wei Yun, Weijian Chen and Zackary Sutphin. I have enjoyed philosophical chats with Zackary.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Concurrent systems are getting more complex with the advent of multi-core processors and the support of concurrent programs. However, errors of concurrent systems are too subtle to detect with the traditional testing and simulation. Model checking is an effective method to verify concurrent systems by exhaustively searching the complete state space exhibited by a system. However, the main challenge for model checking is state explosion, that is the state space of a concurrent system grows exponentially in the number of components of the system. The state space explosion problem prevents model checking from being applied to systems in realistic size.

After decades of intensive research, a large number of methods have been developed to attack this well-known problem. Compositional verification is one of the promising methods that can be scalable to large complex concurrent systems. In compositional verification, the task of verifying an entire system is divided into smaller tasks of verifying each component of the system individually. The correctness of the properties on the entire system can be derived from the results from the local verification on individual components. This method avoids building up the global state space for the entire system, and accordingly alleviates the state space explosion problem. In order to facilitate the application of compositional verification, several issues need to be addressed. The generation of over-approximate and yet accurate environments for components for local verification is a major focus of the automated compositional verification.

This dissertation addresses such issue by proposing two abstraction refinement methods that refine the state space of each component with an over-approximate environment iteratively. The basic idea of these two abstraction refinement methods is to examine the interface interactions among different components and remove the behaviors that are not allowed on the components' interfaces from their corresponding state space. After the extra behaviors introduced by the over-approximate environment are removed by the abstraction refinement methods, the initial coarse environments become more accurate. The difference between these two methods lies in the identification and removal of illegal behaviors generated by the over-approximate environments.

For local properties that can be verified on individual components, compositional reasoning can be scaled to large systems by leveraging the proposed abstraction refinement methods. However, for global properties that cannot be checked locally, the state space of the whole system needs to be constructed. To alleviate the state explosion problem when generating the global state space by composing the local state space of the individual components, this dissertation also proposes several state space reduction techniques to simplify the state space of each component to help the compositional minimization method to generate a much smaller global state space for the entire system. These state space reduction techniques are sound and complete in that they keep all the behaviors on the interface but do not introduce any extra behaviors, therefore, the same verification results derived from the reduced global state space are also valid on the original state space for the entire system.

An automated compositional verification framework integrated with all the abstraction refinement methods and the state space reduction techniques presented in this dissertation has been implemented in an explicit model checker *Platu*. It has been applied to experiments on several non-trivial asynchronous circuit designs to demonstrate its scalability. The experimental results show that our automated compositional verification framework is effective on these examples that are too complex for the monolithic model checking methods to handle.

CHAPTER 1
INTRODUCTION

As increasingly complex and powerful, computing systems are nowadays used in almost all domains of our society, including in safety-critical and mission critical applications, such as nuclear reactors, space shuttle, flight control systems, medical equipment, etc., where reliability is of the utmost importance. Thus ensuring reliability is in high demand for such computing systems. Moreover, to gain high performance and flexibility, systems are designed with multiple computers, processors, and threads as concurrent systems. Due to the intricate sequence of events in the concurrent systems, it is difficult to detect and debug errors that are usually corner cases. Modern verification techniques to ensure system reliability includes formal verification and testing/stimulation. Finite state formal verification refers to mathematical demonstration of the correctness of a finite state system. It constructs a formal model of a system, specifies the desired properties in formalization, and proves or disproves that the properties are satisfied by the model. Since formal verification takes exhaustive examination of all possible behaviors in a system, it has the ability to determine whether a system satisfies the properties with high confidence.

Testing is a more commonly used approach to verifying systems. With respect to testing, an implementation of a system is run for a number of times under the input stimuli. The reasoning is performed based on the observable behaviors of the system to check if the reaction of the system agrees with the expected outputs. Since the number of possible ways that a system can execute is in large size, it is usually infeasible to test all possible behaviors of a system. Therefore, testing can only show the presence of errors, but does not guarantee the correctness of the system. This limitation is exacerbated by concurrent systems for which same test case can produce different results depending on the schedules of the events. Another drawback of testing is lack of information to pinpoint error location. It is hard to identify the actual erroneous computation steps, because numerous computations may be executed from the initial configuration of a test case when the system crashes. Although testing is important to discover errors of the real implementation of a system, errors found at a later stage of the lifecycle of the system development usually cause huge losses.

Theorem proving provides the definitive verification results of a system by mathematical deduction. A system to be verified and the specification to be checked are expressed in some appropriate logic, such as propositional logic, first-order predicate calculus, etc.. The system behavior is represented as axioms. The problem of theorem proving is to derive the specification from the axioms with a set of deduction rules. Since the logic used in theorem proving can be very expressive, theorem proving is able to describe and verify infinite systems under human guidance to direct the strategies of the proof. It requires significant human interaction and expertise in logic reasoning. However, many problems of deduction are undecidable. Even those that are decidable have high computation complexity. Therefore, theorem proving is less acceptable in practice because of few automatic tools and high computation complexity.

However, concurrent systems are difficult to understand and verify because of concurrent behaviors among the components. Although testing can identify significant errors in the sequential systems, it is incapable of detecting concurrency errors since testing is infeasible to achieve a high coverage of all concurrent executions and hard to reproduce errors due to nondeterministic characteristics of the concurrent systems. In contrast, the concurrency errors, such as data race, deadlock and starvation, can be detected with theorem proving. On the other hand, constructing deductive proofs are sufficiently difficult and it may cost weeks or months of the user's time by using a proof checker to complete the verification, which motivates the development of algorithmic methods for reasoning about concurrent systems. Model checking, introduced in the work of Edmund Clarke, Allen Emerson and

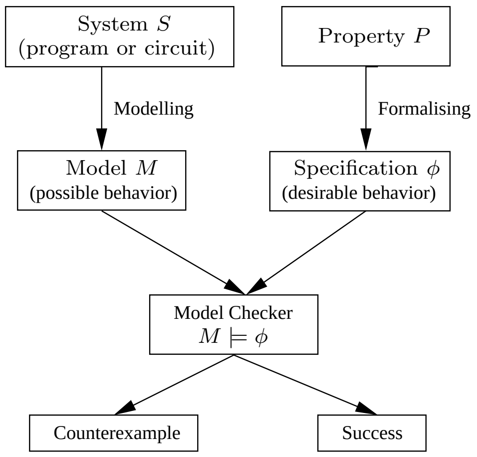


Figure 1.1. The structure of model checking.

Joseph Sifakis [22, 40, 23], is a “push-button” automated method without proofs that supports the verification of the correctness of systems and bug detection in a fast way.

1.1 Model Checking

Model checking is an automatic approach to the execution of algorithms by computer tools, to verify the correctness of finite-state concurrent systems, such as sequential circuit designs, concurrent programs and communication protocols [11, 27]. It requires formal specifications for the properties being verified to represent the desirable behaviors of a system and a finite state model to represent all possible behaviors of the system. Then an effective search procedure is used to explore the model to determine whether the possible behaviors in the model conform to the desired behaviors. When a violation is detected, a counterexample that illustrates how the violation happens is returned for human designers to identify and fix the design faults. Due to the nature of the exhaustive exploration of the model, model checking is not practical for real-size problems because of high space and time complexity.

1.1.1 Model Checking Definition

Model checking takes as inputs a finite state model M representing behaviors of a system S, and a formalization of property P in ϕ, and exhaustively explores all behaviors exhibited in the model M to check if ϕ holds on M, denoted by $M \models \phi$ [23, 21, 58]. Dynamic system behaviors are represented by the changes of states as the system execution progresses over time. The evolution of the system from one state to another is represented by state transitions. The property is formalized as temporal logic formula, automata or state properties , e.g. invariants, assertions, etc. [41, 24]. Since the correctness of a systems is defined by the satisfaction of all properties obtained from the specification, the ultimate goal of model checking is to establish system correctness with the model checking result $M \models \phi$. If the model M reflects the system S faithfully and the specification ϕ represnts the property P authentically, $M \models \phi$ implies S satisfies P. When the specification ϕ is not satisfied on the model M, a counterexample is generated to identify how M violates ϕ. A counterexample is an execution path that starts from the initial state to the state where the property is violated. The model checking structure is shown in Figure 1.1. System model can be extracted from a high-level system description written in some programming languages for software or VHDL/Verilog for hardware, a system's implementation in source code, or a system's executable representation such as machine code or binary code. A preprocessor, model extraction, derives a finite-state model from a program or circuit. The model checking engine, model checker, takes the finite state model of a system and formal representations of the property as inputs, and determines whether the property is satisfied or not. If the property is violated, counterexamples are returned for the designer to debug.

When counterexamples are generated, there may be various causes resulting in violation of the property. One of the reasons might be errors in modeling, that is the model does not reflect the design of the system accurately. The model need to be refined by studying the returned counterexamples. Another reason might be that the property does not reflect the requirements or the property is not formalized correctly. In this case, the property need to be revised to eliminate the property error. If the counterexample analysis shows there is no inconsistency between the model and the system design and no inaccurate formalization for the property, a true design error is exposed.

1.1.2 Strengths and Weaknesses

Model checking has several advantages over testing and theorem proving as shown below.

Automation All users need to do is to provide the high-level description of the system and formal specifications of the property, and then press the “enter” button. Model checking tools take over all the computation of reasoning and calculation without human involvement.

Early benefits Model checking provides an advantage of early check by verifying the correctness of a design in the early stage. Therefore, the bugs can be eliminated before the real implementation of the system is started. Identifying the root causes of bugs in later stages of system development costs significant time and efforts.

Counterexamples for debugging When the specification is violated, model checking can generate counterexamples that show why the specification does not hold. Counterexamples are useful to aid designers in debugging.

Deep bugs detection The exhaustive exploration of the state space of a system contributes to the discovery of corner cases that are easily escaped from the testing.

However, the main challenge for successful applications of model checking on systems in scalable size is the state explosion problem [95]. To prove or disprove a property on a system, a global state space needs to be built up by exhaustively exploring all behaviors exhibited in the system. The state space of a concurrent system is the product of the state space of all components. Therefore, the state space of the concurrent system may exponentially grow in the number of the components. Moreover, the state space can be increased very large or infinite for the large or infinite set of data domains of data types. Due to the limited computing resources, model checking is infeasible if the state space grows very large.

1.1.3 State Explosion Reduction Techniques

Model checking cannot verify large real-size designs because of the inherent state explosion problem. A number of techniques have been developed to attack the state explosion problem. Some of the most important approaches are described as follows.

Partial order reduction focuses on the analysis of the concurrent behaviors of a system [84, 50]. This method identifies independent behaviors such that the executions of them in different ordering lead to the same global state. As a result, the verification result keeps intact from removing certain sequences of such independent behaviors. This method can reduce the state space of concurrent systems significantly.

Counterexample guided abstraction refinement (CEGAR) uses a set of abstraction predicates to build an abstract finite state model for a system [25, 29, 26]. If the abstract model passes verification, the concrete system is concluded to be correct. Otherwise, the abstract model is iteratively refined by adding more relevant details based on the analysis of the spurious counterexamples until the model passes verification, or a counterexample is confirmed to be real.

Symbolic model checking concisely represents the state space with a symbolic encoding, e.g. ordered binary decision diagrams (OBDDs) [69, 16, 15]. This method supports model checker to verify circuits and protocols in large scale by modeling systems with Boolean functions and manipulating in efficient Boolean operations. However, it is not good at verifying concurrent systems.

Although the state explosion problem is alleviated by these methods to some extent, model checking still does not scale large as the system complexity increases. Compositional verification is viewed as one of the most promising approaches to attack the state explosion by applying divide and conquer approach to verification.

1.2 Compositional Verification

Model checking suffers the problem of state explosion due to the fact that the properties are checked on the complete state space through reachability analysis. Compositional verification is an essential method to tackle the state explosion problem in verifying many large systems by leveraging the modular structure of some complicated systems [65, 87]. It can be roughly classified as compositional reasoning and compositional minimization.

Compositional minimization method builds a global state space of the system which is smaller than the complete one generated by the monolithic verification methods using reachability analysis. Moreover, such small global state space still preserves the behaviors with respect to the property being verified [51, 30]. To construct a reduced global state space, in the first step, abstractions of the state space of components with their specific environments are computed. These abstractions hide the details irrelevant to the property and smaller than the state space of components which operate in the context of the complete system. A global state space is generated by composing the abstractions of components in a stepwise manner. This global state space is smaller than the one obtained from the complete system. Then the property is checked on such global state space, which implies the same verification result on the complete system because such global state space is property preserving. The key problem of compositional minimization is to control the size of the intermediate state space.

On the other hand, compositional reasoning method avoids the construction of the global state space for the system [53, 72, 78, 71]. Compositional reasoning divides a system into several components and verifies the properties on the state space of each component in isolation. The correctness of the property on the complete system is then derived from the results of verification of individual components. Therefore, the global state space of the whole design is not necessarily to be built. The effect of the state explosion problem may be alleviated since a single component generally has a much smaller state space than that of the complete system. One of the most frequently advocated compositional reasoning methods is assume-guarantee reasoning [56, 57, 70].

Given a system M made up of components M_1 and M_2 and a property P, to verify $M \models P$, one of the assume-guarantee reasoning rules is shown in Table 1.1 [59, 85]. Often the behavior of a component is dependent on the components with which it inter-

Table 1.1. An assume-guarantee rule.

$$\frac{\begin{array}{ll}\text{premise 1:} & A \parallel M_1 \models P \\ \text{premise 2:} & M_2 \models A\end{array}}{M_1 \parallel M_2 \models P}$$

acts. For a system composed of two components, one component works as the concrete environment of the other component. However, the state space of a component with its concrete environment is same as the complete state space of the entire system. Therefore, it is usually necessary to provide assumptions about the environments in which a component executes. To verify M_1 in isolation, an assumption of the concrete environment, M_2, which captures the expectations that M_1 has about its concrete environment. In this simple assume-guarantee rule, A is an environment assumption for component M_1. $\parallel$ denotes the parallel composition of two components where the behaviors common to their interface are synchronized and the remaining behaviors occur independantly. This rule states that if component M_1 satisfies property P with the assumption A and M_2 satisfies assumption A, then the entire system $M_1 \parallel M_2$ satisfies property P. This method allows a property to be verified locally without the construction of the global state space of the entire system.

There are several issues that make the application of compositional reasoning complicated. First, a system need to be decomposed to several components. Compositional reasoning works effectively for the system where the interface between components are simple [20]. Second, traditionally, environment assumptions are generated manually by the user, which obstacles the application of compositional verification in practice. The key challenge of the automated compositional reasoning is to automatically generate environment assumptions that are compact and precisely capture the expectations that components have from environments.

It is relatively fast to decide if the safety properties of a design hold by compositional reasoning. This is because if any property holds in an individual component under its environment assumption, it holds in the global system. But for the properties defined on the global system such as deadlock and liveness, compositional minimization has the advantage over compositional reasoning.

1.2.1 Environment Problem

Components operate correctly in some specific environments. Therefore, to verify a component individually, it is necessary to generate an environment to define the inputs of the considered component. Model checking is applied on the state space composed by the component and its environment individually. Given a system, the concrete environment of a component is the rest of components in the system. However, the state space generated by composing the component and its concrete environment has the same computation complexity as the verification of the complete system. State explosion is not alleviated if the concrete environment is used to verify the component. Therefore, one of the fundamental problems of composition reasoning is the generation of an approximate environment that captures the expectations that a component makes about its concrete environment but much simpler than the concrete one.

The approximate environment can be modeled by hand or constructed automatically. Traditionally, finding such an environment requires user guidance, which suffers from two severe weaknesses. First, it limited the application of compositional reasoning in practice. Second, the assumptions of environments provided by the users are often error-prone and inaccurate to model the concrete environments. If an environment is too coarse, the extra behaviors increase the chance of producing false counterexamples, which may incur a high computational penalty to distinguish them from the real ones. One key challenge for automated compositional reasoning is to construct compact and accurate approximate environment for each component automatically.

1.2.2 Related Work on Compositional Reasoning

Automated assumption generation Automated environment generation is an active research area of compositional verification over the decades. Rebeca (Reactive Objects Language) is an actor-based modeling language to model open and distributed system, communicating by asynchronous message passing, and also supports the verification by Rebeca Verifier tool [89, 90]. Compositional verification and abstraction techniques are used in Rebeca Verifier tool to reduce the state space and allow the verification of complicated reative systems. Rebeca models, i.e. closed models for components, are translated into languages of existing model checkers, e.g. NuSMV [2] or Spin [3], and then are checked by the corresponding model checkers. For each component, an over-approximate environment is generated which allows all possible interactions on the interface. To reduce the state space of Rebeca models, the environment is reduced from its complete interactions to a set of sent messages and the queue of incoming messages from environment is also reduced.

Bobaru et al. proposed AGAR (Assume-Guarantee Abstraction Refinement) to construct over-approximate assumptions of the environments for components [13]. The assumption is created as an conservative abstraction of the interface behavior of the real environments of a component. Therefore, the premise 2 of the reasoning rule shown in Table 1.1 is satisfied by construction. Only the premise 1 is checked in each iteration. If the premise 1 is valid, the entire system satisfies the properties. Otherwise, the process is similar to CEGAR that spurious counterexamples due to the over-approximate assumption is used to refine the assumption. Our method is similar to AGAR in that the premise 2 is satisfied by the constructed over-approximate environment and only the premise 1 is checked iteratively. However, there are some important differences between AGAR and our methods. In the initial over-approximate environment we create, only the inputs of a component are set to be totally free, while all actions on interface are set to be free in AGAR. The coarser environment brings more extra behaviors to be identified. We apply abstraction refinement methods to refine the over-approximate environments before checking the premise 1, which can lessen the burden of counterexample analysis when a counterexample is returned. However, the refinement of the assumption depends on CEGAR in AGAR.

Recently, learning techniques have been used to solve the verification problems, such as the fixpoint computing, uniform verification of parameterized systems, verifying branchingtime properties by regular model checking, etc. in [66]. Among the learning techniques, Angluin's learning algorithm, L^*, is developed to construct the environment assumptions for the compositional verification. L^* learns an unknown language over alphabet Σ given the oracles that the language holds and produces a deterministic finite automaton (DFA) to accept it [9]. Cobleigh et al. proposed a framework of automated assume-guarantee reasoning for the safety properties by applying L^* to compute assumptions for environments [32, 47]. For a system composed of two components M_1 and M_2, L^* is used to learn an assumption of M_2, A, such that $A \parallel M_1 \models P$ during the verification of the safety property P on the system by assume-guarantee reasoning rules. In the first step, a DFA A is constructed by the strings s such that $s \parallel M_1 \models P$. In the second step, the assumption A is checked by two oracles, i.e. premises of the assume-guarantee reasoning rules. If two premises are satisfied on A, the property P is satisfied on the entire system, i.e. $M_1 \parallel M_2 \models P$. Otherwise, the counterexample might be caused by the imprecise assumption A or by the error of the design that makes $M_1 \parallel M_2$ violate P. In the former case, the counterexample is false and it is returned to L^* to refine A. The refined A is checked by two oracles again in the next iteration. This process is terminated when the system satisfies the property P or when the system violates the property P with the witness of the real counterexample.

The learning framework for automated compositional verification is extended in some ways. In contrast to over-approximate environments, under-approximate environments are computed by learning algorithm. Instead of considering all actions on the interface that the components communicate, under-approximate environments only include the behaviors on the small interface between components [14, 81]. The initial approximate environment only contains the behaviors of a subset of the interface and actions are added to refine the environment by the learning framework until the property is proved or disproved. The learning framework is combined with predicate abstraction [10] to construct an over-approximation (”may” abstraction) and under-approximation (”must” abstraction) for the components with infinite states [45, 88]. L^* is implemented by BDD-based state space exploration to find the appropriate assumptions automatically for assume-guarantee reasoning [8, 76, 77]. This method is adapted to synthesize interfaces for Java classes [5].

Cheung et al. proposed an interface constraint-based approaches such that restrictions from the environment are imposed on the components of a system to remove the behavior that should not take place [19]. Generation of interface constraints based on the analysis of synchronization between components. However, it cannot capture effective interface constraints due to deficiencies in analysis of synchronization between distant components. Alfaro and Henzinger provide interface automata to represent a component and its environment [35, 36, 37]. The component and the environment are refined in an alternating fashion so that the component accepts only input actions generated by the environment, and issues output actions corresponding to these input actions. Refinement of interface automata in the component-based design is similar to refinement of environment assumptions in compositional verification [6, 64]. A similar approach, thread-modular reasoning, is proposed in [55] for multithreaded program verification.

Parizek et al. proposed a technique for automated generation of an artificial environment for a component which derives the inputs of the component in Java code [79, 80]. The runnable program composed of the component and its environment is checked by the Java PathFinder model checker (JPF) [1] to detect concurrency errors, e.g. deadlocks and race conditions. The environment provides inputs for the methods of the component which access shared variables in ordering priority. According to the degree of interaction with the environment via concurrency-related constructs of Java, the methods with high metric are executed earlier to detect concurrency error in less time and memory. This technique is unsound since it does not exhaustively explore the state space. The advantage of this technique is to detect the concurrency bugs in code and avoid the potential of intermediate state explosion due to an over-approximate environment where all possible sequences and interleaving of method invocations of the component are allowed.

Assume-guarantee rules The reasoning rule in Table 1.1 is the simplest one in the use of the two components. When considering multiple components of a system, circular rules and symmetric rules are addressed in [82, 54]. These rules are sound and complete.

Decomposition of design Cobleigh et al. investigated different decompositions of designs and observed that assume-guarantee reasoning is less effective than monolithic verification by exploring more states [31]. This may be caused by the complexity of the interface, or the computation for a deterministic representation of the assumption. Nam et al. proposed to decompose designs with hypergraph partitioning and developed heuristics related to the learning framework to improve the decomposition process [76]. Metzler et al. developed a technique for a two-way decomposition by the analysis of dependence structure of a design and presented a proof that the compositional verification under such decomposition is more efficient than monolithic model checking in [74, 73].

Tools that use compositional verification The Calvin tool incorporates a theorem prover into thread-modular reasoning for the analysis of concurrent Java programs [42]. To verify each thread individually, an environment assumption on shared global variables is created manually to describe the interleaving of other threads. The Mocha tool supports modular verification of shared-memory programs under the requirement specified in alternating-time temporal logic [7]. The environment assumptions are also developed manually in Mocha.

The LTSA tool implements assume-guarantee reasoning in a learning framework to check safety properties represented by labeled transition systems [43, 46, 67]. The LTSA supports a collection of assume-guarantee reasoning rules discussed in [54] to perform compositional verification automatically. Besides the applications at the design level, the LTSA can be used to check the applications at the level of source code with the techniques presented in [49]. JPF is an explicit state model checker for finding bugs of Java byte code. Automated assume-guarantee reasoning is incorporated in JPF to support the verification of safety properties and generation of a permissive interface for a component that preserve every legal behavior of its real environment [44]. The SPLIT compositional verifier implements assertional compositional reasoning by using counterexample guided refinement scheme to automatically verify safety and liveness properties represented by the temporal logic [33, 86].

Compositional verification for formal methods The assumptions generated at the design level can be used to verify the system at the implementation level with assume-guarantee reasoning [49], and also can be used to predicate bad traces by considering different ordering of the independent events during unit testing and system-level testing [48].

1.2.3 Related Work on Compositional Minimization

Compositional minimization method constructs a minimized global state space by composing smaller abstractions of components while preserving semantic equivalence of the whole system with respect to the property being verified. To avoid the state explosion during the process of compositional minimization, several researchers have proposed approaches to containing the size of the intermediate state space.

Clark et al. proposed an approach, interface processes, to obtain abstractions based on the interaction among the components [30, 12]. Consider a system composed of two components M_1 and M_2 and property only refers to the interface between two components. To compute the abstraction of M_1, this approach hides the behaviors of M_1 that are not relevant to the communication with M_2 and then reduces M_1 to a smaller abstraction by equivalence-based reduction technique. In this way, the abstraction of M_1 preserves all behaviors that M_2 can observe through the interface between M_1 and M_2, but the abstraction becomes smaller.Then the abstraction is used to replace M_1 when verifying the property. However, such local minimization do not take into account the interaction between the component and its environment. Therefore, the abstractions may contain behaviors that do no exist in the whole system.

Graf et al. proposed a method to compute abstractions by using user-provided context constraints that impose restrictions on the behaviors of the considered component by communication synchronization with its neighbor components [51, 52]. The component behaviors that should not take place in the whole system are removed during computing the

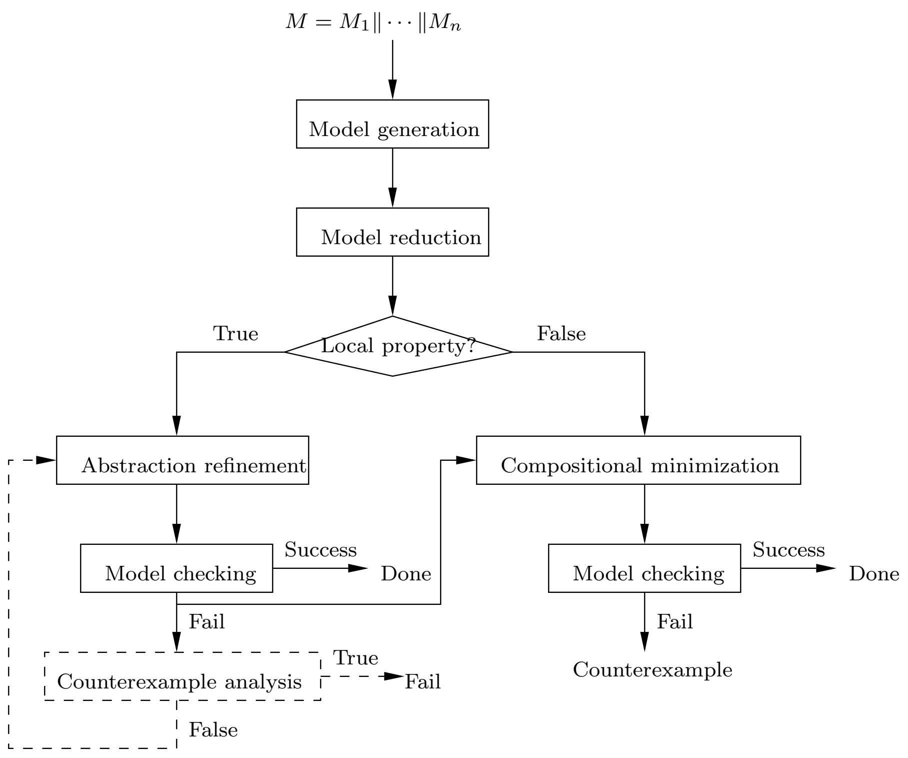


Figure 1.2. The flow of compositional verification.

abstraction by taking account the interface specifications of the environment components. However, these context constraints are not generated automatically. To avoid errors introduced by constraints provided by users, Cheung et al. presented a method to compute context constraints automatically based on the communication among the components [17]. They combined the context constraints derived by algorithms and provided by users. Their technique can detect errors of the user-provided constraints and strengthen the constraints generated automatically [18].

1.3 Overview of the Framework and Organization

The overview of the framework of our methods is shown in Figure 1.2. Compositional verification uses the divide and conquer technique in the model checkin. A system is decomposed into components and these components are verified individually. The correctness of the whole system is then derived from the results of the verification of individual components. In this dissertation, we focus on the issue of environment assumptions and assume that well-formed decomposition of the system is provided.

Compositional verification works effectively for the modular systems. An asynchronous design is composed of several functional components communicating by the interfaces. We choose asynchronous designs as the application area in this dissertation because the concurrency and modularity are two major characteristics of asynchronous designs. Since no global clock exists, asynchronous designs are easy to scale to large size by connecting the components on matching interfaces. However, the sequences of events might be complex because of no restrictions on the timing of the events.

Usually, an asynchronous design is described at the system level by Hardware Description Language (HDLs), e.g. Verilog, VHDL. We assume a heuristic rule for the decomposition is given. We divide an asynchronous design into several components by the provided decomposition rule to obtain the simple interface among components. A finite-state model for each component is constructed from the design specified in high-level description as discussed in Chapter 2. Two model representations and the properties to be checked on asynchronous designs are also presented in Chapter 2.

For local properties that can be verified on individual components, compositional verification can be scaled to large systems by leveraging the proposed abstraction refinement methods. The success of compositional reasoning relies on the discovery of appropriate environment assumptions for each component. Extra behaviors introduced by the over-approximate environments may cause false counterexamples when verifying components individually. Identification and elimination of these false counterexamples incur high computational penalty. Two abstraction refinement methods, *constraint-based refinement* and *synchronization-based refinement*, are proposed in Chapter 4. These abstraction refinement methods aim to refine the state space obtained by a component with its over-approximate environment to be accurate enough for automated compositional reasoning. The purpose of these two abstraction refinement methods is to remove the extra behaviors in the initial coarse approximate environments in an iterative way such that a simple but accurate environment for each component is generated automatically. Since the over-approximate environments contain all possible behaviors of the concrete environments, if the property is satisfied on every components locally, then it is also satisfied on the entire system. However, not all extra behavior can be eliminated from the approximate environments by applying these two methods. The counterexamples returned by the model checking can be caused by the imprecise approximate environments or the bugs of the design. Counterexample analysis is needed to distinguish them. The real counterexamples are reported to the designers and the false ones are returned to the abstraction refinement process to refine the imprecise approximate environments. We have not yet completed the research on the counterexample analysis.

However, for global properties that cannot be checked locally, the state space of the whole system needs to be constructed. To alleviate the intermediate state explosion problem during the generation of the global state space, several state space reduction techniques are proposed in Chapter 3. These state space reduction techniques aim to simplify the state space of components to help the compositional minimization method to generate a much smaller global state space for the entire system. Since the state space reductions preserve all possible behaviors with respect to the property, the proof or disproof of the property on the reduced global state space infers the same verification result on the entire system. The counterexamples returned by the model checking are the real ones. There is no necessary to perform counterexample analysis.

We applied our framework on several non-trivial asynchronous circuit designs to demonstrate the scalability of our methods in Chapter 5. The experimental results show that our automated compositional verification framework is effective on these examples that are too complex for the monolithic model checking methods to handle.

We summarize this dissertation and present directions for future work in Chapter 6.

CHAPTER 2
BACKGROUND

In general, model checking consists of three main components: (1) a finite state model to represent a concurrent system, (2) a formal specification describing the desired properties to be verified, and (3) a state space search algorithm to find the reachable state space for the model where the formal specification is decided. The chapter introduces the related definitions and notations for these model checking components that are necessary for the later chapters.

Two formalisms are used to model concurrent systems in this dissertation, i.e. *Boolean Guarded Petri-nets* and *state graphs*. Petri-nets are commonly used for asynchronous design modeling, and they provide a high-level abstractions to describe the structure and signal transition behavior of a design more easily. On the other hand, state graphs capture the low-level state transition behavior of a design, and are normally used by existing model checking algorithms [28]. Usually, a design is first modeled in Petri-nets, and then its state graph is constructed with a reachability analysis algorithm, which is also described in this chapter.

2.1 Boolean Guarded Petri-nets

Petri-nets are widely used to model and analyze asynchronous designs for its capability to capture both concurrency and non-deterministic choice [63]. They are able to describe a large state space with a compact structure and represent the behavior of asynchronous designs. There are many different forms of Petri-nets for various applications [94]. This section describes a variant of Petri-nets which makes it easier to capture the hierarchical structures as well as signal transition behavior for asynchronous designs [34, 60, 62].

Definition 2.1.1 *A Boolean Guarded Petri-net N is a 8-tuple $(W, T, P, F, \mu^0, \alpha^0, L, B)$ where*

1. *W is a finite set of wires of the asynchronous design being modeled,*
2. *T is a finite set of transitions,*
3. *P is a finite set of places,*
4. *F is the flow relation,*
5. *μ^0 is the initial marking,*
6. *α^0 is the initial state vector,*
7. *L is the action labeling function for transitions,*
8. *B is the Boolean labeling function for transitions.*

W is a set of wires used in an asynchronous circuit design, and it is partitioned into three subsets I, O, and X respectively, where $W = I \cup O \cup X$. Subset I includes input wires whose values are defined externally but accessed in this design. Both subsets O and X include wires defined by this design. However, wires in O are visible to and can be accessed by the external environment, while the wires in X are not visible externally. Each wire in W can take either 0 or 1 representing logical low and high in a digital circuit. For each wire $w \in W$, its value changes when one of two actions happens. $w+$ indicates that the value of w rises from 0 to 1, and $w-$ indicates that its value falls from 1 to 0. Actions are used to model dynamic behavior of a design. For a BGPN, $\mathcal{A} = \mathcal{A}^I \cup \mathcal{A}^O \cup \mathcal{A}^X$ is the set of all actions on W where $\mathcal{A}^I = I \times \{+, -\}$ is the set of actions generated by an environment of a design such that the design can only observe and react, $\mathcal{A}^O = O \times \{+, -\}$ is the set of actions generated by a design responding to its environment, and $\mathcal{A}^X = X \times \{+, -\}$ represents the internal behavior of a system invisible on the interface. The set of visible actions is $\mathcal{A}^I \cup \mathcal{A}^O$ and the set of invisible actions is $\mathcal{A}^X$.

A BGPN consists of two types of vertices: places and transitions. Places represent local states of a system and keep information on system resources and certain conditions necessary to execute transitions. A token assigned to a place indicates that a resource is available or the certain execution condition is satisfied. Transitions are interpreted as signal transitions. A rising and falling signal transitions, i.e. actions, are represented by action $a+$ for the transition of signal a from 0 to 1 and $a-$ for signal a from 1 to 0, respectively. A guard represents conditions for a transition to execute. In BGPN figures, circles represent places, and bullets in some places are tokens. Transitions are represented by boxes with the labeled Boolean guards and actions shown inside.

In a BGPN, 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 relation as follows.

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 a set of places which are connected to the transition through some arcs, and its postset is the set of places to which this transition is connected. The preset and postset of a place are defined similarly. The preset and postset for a transition and a place are formally 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 $p\bullet = \{t \in T \mid (p,t) \in F\}$.*

A marking of a BGPN N is a distribution of tokens to the places in N. A marking represents the global state of the system. If a place has a token, it is marked. A marked place indicates that one of the conditions for a transition leaving that place to be enabled is true. In this dissertation, only 1-safe or 1-bounded BGPNs are considered where each place is allowed to have at most one token. This requirement of BGPNs simplifies the analysis algorithms, and commonly used in asynchronous circuit modeling and analysis [39, 61]. The marking is defined as follows.

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

The initial marking μ^0 is the set of places that are initially marked.

Since a wire can only take either 0 or 1 to represent the logical value, a state vector is a Boolean assignment to all wires in W that represents the state of a BGPN on W at time instant. The definition of state vectors is given as follows.

Definition 2.1.5 *The state vector $\alpha : W \rightarrow \{0, 1\}$ of a BGPN N is a Boolean assignment to W.*

The initial state vector α^0 defines the initial value of the wires in W.

BGPN transitions can be labeled with actions on all wires, thereby associating BGPN transitions with dynamic behaviors of an asynchronous design. The labeled actions from the set of $\mathcal{A}^O \cup \mathcal{A}^X$ represent the behaviors of a design responding to the environment. The labeled actions on $\mathcal{A}^I$ represent the stimuli that a design receives from the environment. The action labeling function L is defined as follows.

Definition 2.1.6 *The action labeling function $L : T \rightarrow \mathcal{A}$ of BGPN N assigns each transition with an action on a wire.*

Each transition is also labeled with a Boolean function, *guard*, to predicate when the transitions can happen. Using Boolean guards makes modeling of asynchronous designs less awkward and much simplified in situations where the transitions depend not only on other transitions but also on the values of wires in a design [4, 61]. It also allows design structures to be captured more easily. The Boolean labeling function is defined as follows.

Definition 2.1.7 *The Boolean labeling function $B : T \rightarrow \mathcal{B}$ of BGPN N assigns each transition with a Boolean function $\mathcal{B} : \{0, 1\}^{|W|} \rightarrow \{0, 1\}$ defined over W.*

An example is given to illustrate modeling of a simple asynchronous circuit in BGPN. Figure 2.1(a) shows a simple asynchronous circuit. The component labeled with "C" is a C-element whose output is high when both inputs are high, low when both inputs are low,

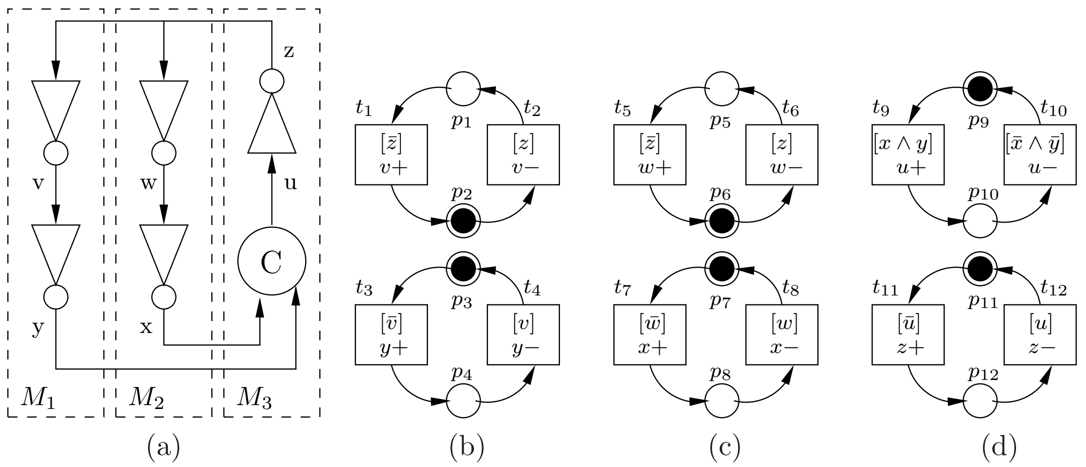


Figure 2.1. (a) A simple asynchronous circuit with three components, M_1, M_2, and M_3, (b)- (d) The BGPNs for component M_1, M_2, and M_3.

or remains unchanged otherwise. This circuit is partitioned into three components, M_1, M_2 and M_3, whose BGPN models are shown in Figure 2.1(b) - (d). The BGPN model is build on the gate-level netlist. For example, the BGPN of component M_1 is composed of BGPNs for two gates. Consider the BGPN model of the inverter with input wire z and output wire v. The rising action $v+$ occurs under the conditions that current output value of v is 0 and the input value of z is 0. Therefore, the transition t_1 is labeled with action $v+$ and the guard $\bar{z}$. Similarly, the transition t_2 is labeled with falling action $v-$ and the guard z for the conditions that t_2 happens. The place p_2 is marked with a token, which means t_2 is a candidate of enabled transitions. t_2 is to be enabled if the guard z is valued to be 1. Suppose the initial marking of the whole design $\mu^0 = \{p_2, p_3, p_6, p_7, p_9, p_{11}\}$ and the initial state vector $\alpha^0 = \{(v, 1), (y, 0), (w, 1), (x, 0), (u, 0), (z, 0)\}$. In the initial state, transitions $t_2, t_3, t_6, t_7, t_9, t_{11}$ are the candidates to enable since the places of their presets are marked. But only t_{11} is enabled because its guard is satisfied by α^0.

Next, the transition firing semantics are described to show how the dynamic behavior of an asynchronous design is represented by a BGPN. As described above, a BGPN transition is labeled with an action. Firing such a transition, or executing it, causes the labeled action to happen, thus leading to a change of the state of the design. First, the state of BGPNs needs to be defined. Given a BGPN N, a state s of N is a pair (μ, α) where μ is a marking of the N and α is a state vector of the values of all wires in W of N. Given a state $s = (\mu, \alpha)$, μ_s is used to refer to the μ component of s. Similarly, α_s is used to refer to the α component of s. Given two states $s = (\mu, \alpha)$ and $s' = (\mu', \alpha')$, $s = s'$ if $\mu = \mu'$ and $\alpha = \alpha'$.

In a state, firing a transition can lead to a new state. A transition needs to be enabled before it can fire in a state. For a transition to be enabled in a state, all places in its preset need to be marked, and its Boolean guard is satisfied by the state vector of the current state.

Definition 2.1.8 *A transition t is enabled in a state $s = (\mu, \alpha)$ if $\bullet t \subseteq \mu$ and $\alpha \models B(t)$.*

The set of transitions that are enabled in a state s is denoted as *enabled*(s). A transition is disabled in a state if it is not enabled.

An enabled transition may or may not fire right away in a state. The firing of a transition changes the current state and generates a new state. Let $s \xrightarrow{t} s'$ represent the state transition when firing transition t where $s = (\mu, \alpha)$ and $s' = (\mu', \alpha')$. When a transition t is fired, one token is removed from each place of the preset of transition t and one token is added to each place of the postset of transition t. Thus the marking μ' of the new state is defined as

$$\mu' = (\mu - \bullet t) + t\bullet .$$

This step is known as the marking update. The value of the wires in the state vector of the current state is also updated accordingly, depending on the labeled action of the fired transition. Let $a = L(t)$ be the labeled action of transition t. The new state vector α' is updated from α as follows after t is fired. Therefore, a BGPN moves from one state to another by firing one of the enabled transitions.

$$\forall w \in W,\ \alpha'(w) \quad = \quad \begin{cases} 1 & \text{if } a = w+ \\ 0 & \text{if } a = w- \\ \alpha(w) & \text{otherwise} \end{cases}$$

2.1.1 Parallel Composition of BGPN

An asynchronous design is usually composed of a number of components represented by BGPNs. The parallel composition of two components creates a composite component which is the asynchronous products of these two composed components. In the composite component, both composed components are synchronized with events on wires of the common interface. The BGPN of the whole design is obtained by applying parallel composition on all component BGPNs.

Let $N_1 = (W_1, T_1, P_1, F_1, \mu_1^0, \alpha_1^0, L_1, B_1)$ and $N_2 = (W_2, T_2, P_2, F_2, \mu_2^0, \alpha_2^0, L_2, B_2)$ be two BGPNs where $W_1 = I_1 \cup O_1 \cup X_1$ and $W_2 = I_2 \cup O_2 \cup X_2$. If two transitions can happen concurrently, they can happen in arbitrary ordering. The parallel composition of BGPN captures the concurrency of two BGPNs. The definition of parallel composition of N_1 and N_2, denoted as $N_1 \| N_2$, is shown in the following.

Definition 2.1.9 *Given two BGPNs N_1 and N_2, the parallel composition of N_1 and N_2 is a BGPN $N = (W, T, P, F, \mu^0, \alpha^0, L, B)$ 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. $\mu^0 = \mu_1^0 \cup \mu_2^0$,
6. $\alpha^0 = \alpha_1^0 \cup \alpha_2^0$,
7. $L = L_1 \cup L_2$,
8. $B = B_1 \cup B_2$.

After the parallel composition, the input, output, and internal wires of N is defined as follows.

1. $I = (I_1 - O_2) \cup (I_2 - O_1)$
2. $O = O_1 \cup O_2$
3. $X = X_1 \cup X_2$

If $I = \emptyset$, N is referred to as *closed*.

2.2 State Graphs

Even though BGPNs are able to represent the behavior of concurrent designs in a compact way, analysis and verification of the properties of these systems have traditionally been performed on state based models. State graph is a finite state model to represent the state transition behavior of concurrent systems where the actual verification is applied. A state graph can be constructed by the exhaustive analysis of the token flow of the underlying BGPN. This section introduces basic notations and definitions of state graphs.

A state graph is a vertex and edge-labeled digraph. Vertices represent the BGPN states as defined in the previous section. Edges represent state transitions, each labeled with an action. The definition of state graphs is given as follows.

Definition 2.2.1 *A state graph (SG) G is a 4-tuple $(N, S, init, R)$ where*

1. *N is a BGPN,*
2. *S is a finite set of states,*
3. *$init \in S$ is the initial state, and $init = (\mu^0, \alpha^0)$,*
4. *$R \subseteq S \times \mathcal{A} \times S$ is a finite set of state transitions where $\mathcal{A}$ is the set of actions of N.*

A state of S is a state generated by firing an enabled BGPN transition of the underlying BGPN N. A state s is represented by a pair of marking and state vector, denoted by $s = (\mu, \alpha)$ as shown in Section 2.1. The set of states S includes a special state π which denotes the *failure state* of a SG G, and represents violations of some prescribed properties.

How a system behaves does not matter after it enters the failure state. Therefore, for every $a \in \mathcal{A}$, there is a $(\pi, a, \pi) \in R$. This section uses $(s_1, a, s_2) \in R$ or $R(s_1, a, s_2)$ to denote that (s_1, a, s_2) is a state transition of a SG G. We assume that the state transition set R is *total* such that every state has some successor. A SG G is *deterministic* if for all states $s, s', s'' \in S$ and all actions $a \in \mathcal{A}$, $R(s, a, s')$ and $R(s, a, s'')$ hold, then $s' = s''$. Otherwise, G is non-deterministic. Non-deterministic SGs are allowed in the dissertation.

An example of a state graph is shown in Figure 2.2 for the asynchronous circuit described in Figure 2.1(a). The SG is constructed by following the reachability analysis of the underlying BGPNs shown in Figure 2.1(b)-(d). The set of wires of the BGPNs are $W = \{v, y, w, x, u, z\}$. The initial marking of the BGPNs is $\mu^0 = \{p_2, p_3, p_6, p_7, p_9, p_{11}\}$. The initial state vector of the BGPNs is $\alpha^0 = \{(v, 1), (y, 0), (w, 1), (x, 0), (u, 0), (z, 0)\}$. At the initial state, only the transition labeled with the action $z+$ is enabled. After such transition is fired, the token is moved from p_{11} to p_{12} and a new state s_1 is generated, where $\mu_{s_1} = \{p_2, p_3, p_6, p_7, p_9, p_{12}\}$ and $\alpha_{s_1} = \{(v, 1), (y, 0), (w, 1), (x, 0), (u, 0), (z, 0)\}$. According to the token flow analysis of the underlying BGPNs, the reachable states from the initial states are explicitly enumerated in Table 2.1.

A *path* ρ of G is an infinite sequence of alternating states and actions $(s_0, a_0, s_1, a_1, s_2, \cdots)$ such that $s_0 = init$, $s_i \in S$, $a_i \in \mathcal{A}$, and $(s_i, a_i, s_{i+1}) \in R$ for all $i \geq 0$. A path is *autonomous* if all actions on that path are in $\mathcal{A}^O \cup \mathcal{A}^X$. An autonomous path executes irrespective of input actions. A path is *visible* if it does not contain any action on $\mathcal{A}^X$. Given a SG G, the set of all paths starting from the initial state *init* is the language of G, denoted as $\mathcal{L}(G)$. A subpath is defined as a fragment of a path such that $\hat{\rho} = (s_i, a_i, s_{i+1}, a_{i+1}, \cdots, s_{i+j})$ for $i, j \geq 0$. The *trace* of path ρ, denoted by $\sigma(\rho)$, is the sequence of actions $(a_0, a_1, \cdots)$. Two traces $\sigma = (a_0, a_1, \cdots)$ and $\sigma' = (a'_0, a'_1, \cdots)$ are equivalent, denoted by $\sigma = \sigma'$, iff $\forall i \leq 0, a_i = a'_i$. A state $s' \in S$ is *reachable from* a state $s \in S$ if there exists a subpath $\hat{\rho} = (s_0, a_0, s_1, a_1, s_2, \cdots, s_n)$ such that $s_0 = s$ and $s_n = s'$. A state s is reachable in G if s is reachable from the initial state *init*.

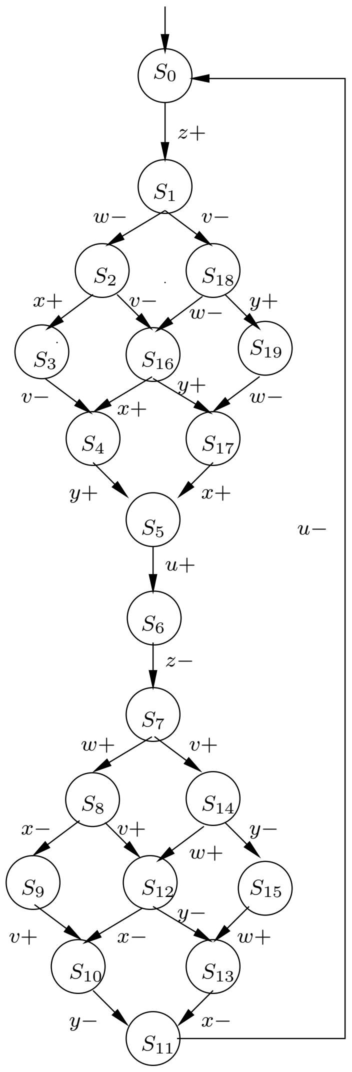


Figure 2.2. The whole SG of the asynchronous circuit shown in Figure 2.1(a).

Table 2.1. State definition of the SG in Figure 2.2.

State	Marking	State vector
s_0	$\{p_2, p_3, p_6, p_7, p_9, p_{11}\}$	$\{(v,1),(y,0),(w,1),(x,0),(u,0),(z,0)\}$
s_1	$\{p_2, p_3, p_6, p_7, p_9, p_{12}\}$	$\{(v,1),(y,0),(w,1),(x,0),(u,0),(z,1)\}$
s_2	$\{p_2, p_3, p_5, p_7, p_9, p_{12}\}$	$\{(v,1),(y,0),(w,0),(x,0),(u,0),(z,1)\}$
s_{18}	$\{p_1, p_3, p_6, p_7, p_9, p_{12}\}$	$\{(v,0),(y,0),(w,1),(x,0),(u,0),(z,1)\}$
s_3	$\{p_2, p_3, p_5, p_8, p_9, p_{12}\}$	$\{(v,1),(y,0),(w,0),(x,1),(u,0),(z,1)\}$
s_{16}	$\{p_1, p_3, p_5, p_7, p_9, p_{12}\}$	$\{(v,0),(y,0),(w,0),(x,0),(u,0),(z,1)\}$
s_{19}	$\{p_1, p_4, p_6, p_7, p_9, p_{12}\}$	$\{(v,0),(y,1),(w,1),(x,0),(u,0),(z,1)\}$
s_4	$\{p_1, p_3, p_5, p_8, p_9, p_{12}\}$	$\{(v,0),(y,0),(w,0),(x,1),(u,0),(z,1)\}$
s_{17}	$\{p_1, p_4, p_5, p_7, p_9, p_{12}\}$	$\{(v,0),(y,1),(w,0),(x,0),(u,0),(z,1)\}$
s_5	$\{p_1, p_4, p_5, p_8, p_9, p_{12}\}$	$\{(v,0),(y,1),(w,0),(x,1),(u,0),(z,1)\}$
s_6	$\{p_1, p_4, p_5, p_8, p_{10}, p_{12}\}$	$\{(v,0),(y,1),(w,0),(x,1),(u,1),(z,1)\}$
s_7	$\{p_1, p_4, p_5, p_8, p_{10}, p_{11}\}$	$\{(v,0),(y,1),(w,0),(x,1),(u,1),(z,0)\}$
s_8	$\{p_1, p_4, p_6, p_8, p_{10}, p_{11}\}$	$\{(v,0),(y,1),(w,1),(x,1),(u,1),(z,0)\}$
s_{14}	$\{p_2, p_4, p_5, p_8, p_{10}, p_{11}\}$	$\{(v,1),(y,1),(w,0),(x,1),(u,1),(z,0)\}$
s_9	$\{p_1, p_4, p_6, p_7, p_{10}, p_{11}\}$	$\{(v,0),(y,1),(w,1),(x,0),(u,1),(z,0)\}$
s_{12}	$\{p_2, p_4, p_6, p_8, p_{10}, p_{11}\}$	$\{(v,1),(y,1),(w,1),(x,1),(u,1),(z,0)\}$
s_{15}	$\{p_2, p_3, p_5, p_8, p_{10}, p_{11}\}$	$\{(v,1),(y,0),(w,0),(x,1),(u,1),(z,0)\}$
s_{10}	$\{p_2, p_4, p_6, p_7, p_{10}, p_{11}\}$	$\{(v,1),(y,1),(w,1),(x,0),(u,1),(z,0)\}$
s_{13}	$\{p_2, p_3, p_6, p_8, p_{10}, p_{11}\}$	$\{(v,1),(y,0),(w,1),(x,1),(u,1),(z,0)\}$
s_{11}	$\{p_2, p_3, p_6, p_7, p_{10}, p_{11}\}$	$\{(v,1),(y,0),(w,1),(x,0),(u,1),(z,0)\}$

During verification, sometimes not all actions are necessary, and some of them can be removed to simplify complexity. Trace projection function that removes all unnecessary actions from traces is defined as follows.

Definition 2.2.2 *Let $\sigma = (a_0, a_1, \cdots)$ be a trace. Its projection onto a subset of actions $\mathcal{A}' \subseteq \mathcal{A}$, denoted by $\sigma[\mathcal{A}']$, is a sequence of actions by removing from σ all the actions $a \notin \mathcal{A}'$, as shown below.*

$$\sigma[\mathcal{A}'] = \begin{cases} \sigma' & \textit{if } a_0 \notin \mathcal{A}', \\ a_0 \circ \sigma' & \textit{otherwise.} \end{cases}$$

where $\sigma' = (a_1, a_2, ...)[\mathcal{A}']$, and $\circ$ is an operator that concatenates an action to a trace.

The concept of the path projection is based on the trace projection. Given a path, its *projection* onto a set of actions is defined as follows.

Definition 2.2.3 *Let $\rho = (s_0, a_0, s_1, a_1, \cdots)$ be a path. Its projection onto a subset of actions $\mathcal{A}' \subseteq \mathcal{A}$, denoted by $\rho[\mathcal{A}']$, is a path such that*

$$\rho[\mathcal{A}'] \quad = \quad \begin{cases} \rho' & \text{if } a_0 \notin \mathcal{A}', \\ (s_0, a_0) \circ \rho' & \text{otherwise.} \end{cases}$$

where $\rho' = (s_1, a_1, s_2, ...)[\mathcal{A}']$, and $\circ$ is the concatenation operator.

Based on the above definitions of projection functions, $\sigma(\rho[\mathcal{A}']) = \sigma(\rho)[\mathcal{A}']$. Given two paths, their equivalence is defined by the equivalence of their traces as follows.

Definition 2.2.4 *Let $\rho = (s_0, a_0, s_1, a_1, \cdots)$ and $\rho' = (s'_0, a'_0, s'_1, a'_1, \cdots)$ be two paths of a SG G. ρ and ρ' are equivalent, denoted as $\rho \sim \rho'$, iff $\sigma(\rho) = \sigma(\rho')$.*

Definition 2.2.5 *Let ρ and ρ' be two paths of a SG G, and $\mathcal{A}' \subseteq \mathcal{A}^I \cup \mathcal{A}^O$ be a subset of observable actions.*

ρ and ρ' are observably equivalent with respect to $\mathcal{A}'$, denoted as $\rho[\mathcal{A}'] \sim \rho'[\mathcal{A}']$, iff

$$\sigma(\rho[\mathcal{A}']) \sim \sigma(\rho'[\mathcal{A}'])$$

Failure state π can be used to represent various undesirable behavior that a system is not expected to produce. A system is regarded as being correct if π is not reachable in its SG. A path is referred to as a *failure path* if a SG contains the failure state π reachable via such path. The set of all failure traces of a SG G is denoted as $\mathcal{F}(G)$. Obviously, $\mathcal{F}(G) \subseteq \mathcal{L}(G)$. A system is correct if $\mathcal{F}(G) = \emptyset$.

Given a failure trace $\rho = (s_0, a_0, \cdots, s_i, a_i, \pi, \cdots)$, its non-failure prefix is $(s_0, a_0, \cdots, s_i, a_i)$. If another trace ρ' has the same non-failure prefix of ρ, ρ' is also regarded as a failure trace. This is because when a design reaches the state s_i, executing action a_i may or may not lead to the failure. In verification, such possibility is as bad as the certainty for the failure to be generated. In this case, ρ and ρ' are called *failure equivalent*.

Definition 2.2.6 *Given two paths $\rho = (s_0, \cdots, s_i, a_i, \pi, \cdots)$ and $\rho' = (s'_0, \cdots, s'_i, a'_i, \cdots)$, ρ and ρ' are failure equivalent, denoted as $\rho \sim_F \rho'$ iff*

$$\forall 0 \le h \le i, a_h = a'_h$$

2.2.1 Parallel Composition of State Graphs

Similar to BGPN modeling, a well structured design consists of a number of components, each of which can be represented by a state graph. The SG of the entire design can be constructed by finding the SGs for the individual components first, and then composing them together.

Definition 2.2.7 *Let G_1 and G_2*

$$G_1 = (N_1, S_1, init_1, R_1)$$

$$G_2 = (N_2, S_2, init_2, R_2)$$

be two SGs. The parallel composition of G_1 and G_2 is defined as

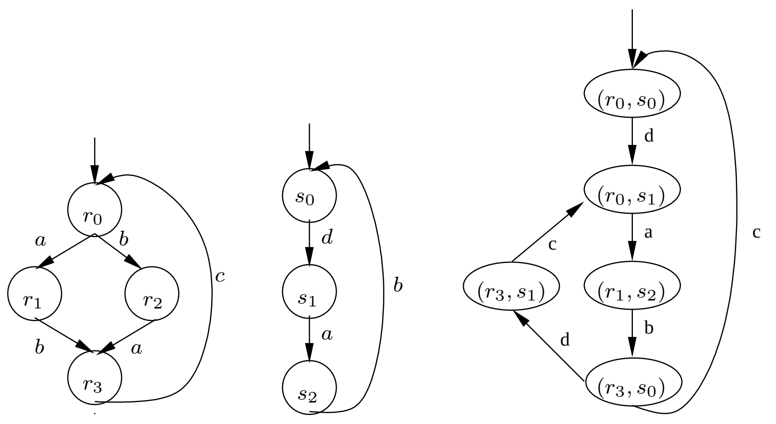
$$G_1 \parallel G_2 = (N_1 \parallel N_2, S, (init_1, init_2), R)$$

where

1. *$S = \{(s_1, s_2) \mid s_1 \in S_1 \wedge s_2 \in S_2\}$ such that $(s_1 = \pi \Rightarrow s_2 = \pi) \vee (s_2 = \pi \Rightarrow s_1 = \pi)$.*
2. *$R \subseteq S \times \mathcal{A} \times S$ such that $\forall s_1 \in S_1, \forall s_2 \in S_2,\ (s_1, s_2) \in S,\ s_1 \neq \pi,\ s_2 \neq \pi$, and one of the following conditions holds.*

   (a) $\forall a \in \mathcal{A}_1 - \mathcal{A}_2,\ R_1(s_1, a, s'_1)$ and

$$\begin{cases} s'_1 \neq \pi & \Rightarrow & R((s_1, s_2), a, (s'_1, s_2)) \\ s'_1 = \pi & \Rightarrow & R((s_1, s_2), a, (\pi, \pi)) \end{cases}$$



(a) (b) (c)

Figure 2.3. (a) The SG G_1 where $\mathcal{A}_1 = \{a, b, c\}$. (b) The SG G_2 where $\mathcal{A}_2 = \{a, b, d\}$. (c) The composite SG after applying the parallel composition on G_1 and G_2.

(b) $\forall a \in \mathcal{A}_2 - \mathcal{A}_1,\ R_2(s_2, a, s_2')$ *and*

$$\begin{cases} s_2' \neq \pi & \Rightarrow \quad R((s_1, s_2), a, (s_1, s_2')) \\ s_2' = \pi & \Rightarrow \quad R((s_1, s_2), a, (\pi, \pi)) \end{cases}$$

(c) $\forall a \in \mathcal{A}_1 \cap \mathcal{A}_2,\ R_1(s_1, a, s_1') \wedge R_2(s_2, a, s_2')$ *and*

$$\begin{cases} s_1' \neq \pi \wedge s_2' \neq \pi & \Rightarrow \quad R((s_1, s_2), a, (s_1', s_2')) \\ s_1' = \pi \vee s_2' = \pi & \Rightarrow \quad R((s_1, s_2), a, (\pi, \pi)) \end{cases}$$

Similarly, R *also includes* $((\pi, \pi), a, (\pi, \pi))$ *for all* $a \in \mathcal{A}_1 \cup \mathcal{A}_2$*.*

When several components execute concurrently, they synchronize on the shared actions, and proceed independently on their invisible actions. An error occurs if either component state is the failure state. If either individual SG makes a state transition to the failure state, there is a corresponding state transition to the failure state in the composite SG. The behavior of the composite SG captures the interaction between the two individual SGs. It has been shown that parallel composition of SGs is commutative and associative in [20].

We illustrate how parallel composition is applied to two SGs as shown in Figure 2.3. Given SGs G_1 with $\mathcal{A} = \{a, b, c\}$ and G_2 with $\mathcal{A}_2 = \{a, b, d\}$ shown in Figure 2.3(a) and (b), respectively, the composite SG is shown in Figure 2.3(c). The common interface of G_1 and G_2 is the set $\{a, b\}$ on which G_1 and G_2 are synchronized by Definition 2.2.7 3(c). In terms of G_1's invisible action c and G_2's invisible action d, they are enabled independently by Definition 2.2.7 3(a) and (b).

2.2.2 Abstraction Relation

Given a design, it can be modeled in different ways with different levels of details. The relation among these models can be characterized by the behavior they represent. This relation, *abstraction relation*, is defined as follows.

Definition 2.2.8 *Given two SGs G and G', G' is an abstraction of G, denoted as $G \preceq G'$, iff the following conditions hold:*

1. *$\mathcal{A}' = \mathcal{A}$ where $\mathcal{A}'$ and $\mathcal{A}$ are the sets of actions of N' and N, respectively.*

2. *$\forall \rho \in \mathcal{L}(G), \exists \rho' \in \mathcal{L}(G'), \rho \sim \rho'$ or $\rho \sim_F \rho'$.*

Intuitively, the abstraction relation states that any path in G has a corresponding equivalent path in G'. Moreover, for any failure path in G, there exists an equivalent failure path in G'. In other words, the language accepted by G is also accepted by G'. Therefore, given G and G', they satisfy the following property:

$$G \preceq G' \wedge \mathcal{F}(G') = \emptyset \quad \Rightarrow \quad \mathcal{F}(G) = \emptyset. \tag{2.1}$$

This property states that G is correct if G' is correct.

Let G and G' be two SGs, and $\mathcal{A}$ be the action set of G. The following property states that a component's behaviors become more restricted if it is composed with another component. It is useful when proving a theorem in Chapter 4

$$\forall \rho \in \mathcal{L}(G \| G'), \exists \rho' \in \mathcal{L}(G), \rho[\mathcal{A}] \sim \rho' \text{ or } \rho[\mathcal{A}] \sim_F \rho'. \tag{2.2}$$

The equivalence relation, which represnets that two SGs contain the same set of equivalent paths, is defined as follows.

Definition 2.2.9 *Let G and G' be SGs. G is equivalent to G', denoted as $G \equiv G'$, if and only if $G \preceq G'$ and $G' \preceq G$.*

Therefore, if $G \equiv G'$, the following property holds.

$$\mathcal{F}(G) = \emptyset \quad \Leftrightarrow \quad \mathcal{F}(G') = \emptyset. \tag{2.3}$$

Intuitively, the same conclusion can be drawn for two equivalent SGs.

2.3 Correctness Properties

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

In this dissertation, a design is considered to be correct if none of the following failures occur during searching the state space of the BGPN of the design: *safety failures*, *disabling failures* and *deadlock*, which are defined below.

Definition 2.3.1 *Let s_i be a state of a BGPN N, and $t_i \in enabled(s_i)$ be a transition fired in s_i.*

1. *A safety failure occurs in s_i if $(\mu_{s_i} - \bullet t_i) \cap t_i\bullet \neq \emptyset$,*

2. *A disabling failure occurs in s_{i+1} if the following conditions are satisfied:*

   (a) *$t_i \neq t_j$, $t_j \in enabled(s_i)$ but $t_j \notin enabled(s_{i+1})$ where s_{i+1} is the state after firing transition t_i at s_i.*

   (b) *$\bullet t_i \cap \bullet t_j = \emptyset$.*

3. *A deadlock failure occurs in s_i if $enabled(s_i) = \emptyset$.*

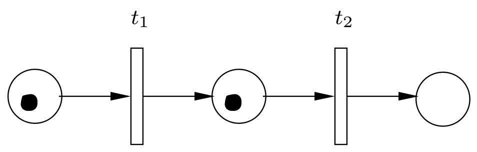


Figure 2.4. A Petri-net structure violates the 1-safety property.

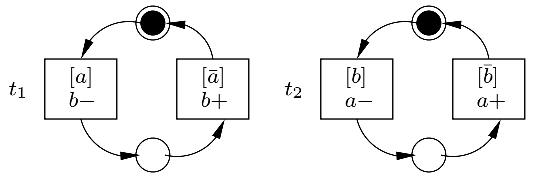


Figure 2.5. An example of disabling failure.

The safety property checks if there is only one token in a place at a time in a design [94, 91]. A design is safe when every place holds at most one token in any state. Without the restriction of 1-safe property, the ordering of the transitions occurring cannot be distinguished. For example, the presets of transitions t_1 and t_2 are marked as shown in Figure 2.4. Both transitions t_1 and t_2 are enable to fire. We cannot distinguish whether t_1 will fire first or t_2 will fire first. The requirement of 1-safe property eliminates the non-determinism of the firing ordering among several enabled transitions.

A *disabling failure* is to check whether the persistence property of a design is satisfied or not. A design is persistent if, for any two enabled transitions, firing one of them will not disable the other one [83]. A disabling failure happens in a state s when any of enabled transitions becomes disabled after another enabled transition is fired. It captures a violation of concurrency among transitions. For example, given an initial state s_0 where $\alpha(s_0) = \{(a, 1), (b, 1)\}$, both t_1 and t_2 are enabled as shown in Figure 2.5. However, a disabling failure occurs as firing either t_1 or t_2 disables the other transition. Typically, a disabling error indicates there may be glitches in the underlying asynchronous circuits.

Checking these three types of failures can be reduced to checking some invariants on each state during state space search, and in general absence of these failures is considered as safety properties are satisfied on a design. This dissertation focuses on the verification of safety properties, and the liveness property verification is left for future work.

2.4 Reachability Analysis

Reachability analysis is a procedure to build state graphs (SGs) for the underlying concurrent system represented by BGPNs. The BGPN moves from one state, i.e. a pair of marking and state vector, to another by firing one of the enabled transition. Each reachable state is explicitly enumerated and kept as a state of the SG. The action of the enabled transitions and the current state and the new generated state consist of a state transition of the SG.

Given a BGPN N, the algorithm `findSG` shown in Algorithm 1 returns the SG G by performing depth-first state space search (DFS) on N. According to the firing semantics described in section 2.1, each BGPN has an unique SG. Function $fire(s, t)$ implements the transition firing semantics such that a new state s' is obtained by firing the BGPN transition t enabled at state s in a BGPN. Given $s \xrightarrow{t} s'$, function $CheckFailure(s, t, s')$ check if firing t would cause a safety violation or a disabling error as described in section 2.3.

The algorithm `findSG` takes a BGPN N as a design description. It starts from initial state s_0 for execution (line 2) and pushes the tuple of *init* and *enabled*(*init*) onto stack (line 3). If the stack is not empty, repeat the iterations to find the new states. A transition t enabled at the current state is picked and the tuple of current state and remaining enabled transitions is pushed onto the stack (line 6 - line 9). When a transition is executed, a new state s' is created (line 10). If s' is the failure state π, a state transition (s, a, π) is added to the SG (line 12 - 13). Otherwise, a new state transition is added to the state transition set (line 20). Then, we check if the state s' is explored before or not. If s' has not been encountered during executing other transitions before, s' is added to the state set S (line 16). To prepare for the next step of DFS exploration, we collect the enabled state transitions in the state s', into T'_{en} (line 17). If T'_{en} is not empty, s' and its enabled transitions are pushed onto the stack (line 18 - 19). If the enabled set of s' is empty, a deadlock happens,

Algorithm 1: findSG (N)

```
1  $init = (\mu^0, \alpha^0)$ ;
2  $S = \{init\}, T = \emptyset, T_{en} = enabled(init)$;
3  $push(init, T_{en})$;
4  $failure = false$ ;
5  while $stack\ is\ not\ empty$ do
6      $(s, T_{en}) = pop()$ ;
7      $t = select(T_{en})$ ;
8      if $T_{en} - t \neq \emptyset$ then
9          $push(s, T_{en} - \{t\})$ ;
10     $s' = fire(s, t)$ ;
11     $failure = checkFailure(s, t, s')$ ;
12     if $failure = true$ then
13         $T = T \cup \{(s, L(t), \pi)\}$ ;
14     else
15         if $s' \notin S$ then
16             $S = S \cup \{s'\}$ ;
17             $T'_{en} = enabled(s')$ ;
18             if $T'_{en} \neq \emptyset$ then
19                 $push(s', T'_{en})$ ;
20             else
21                 $s' = \pi$;
22         $T = T \cup \{(s, L(t), s')\}$ ;
23 return $(N, S, init, T)$ ;
```

and s' is converted to the failure state π (line 21). Then, the state transition from firing transition t is added into the state transition set (line 22). When the while loop in the function terminates, the found SG is returned.

CHAPTER 3

STATE SPACE REDUCTIONS

The key problem in compositional reasoning is to verify each component individually with its appropriate environment. Since the real environment for a component is hard to obtain, a typical solution is to create an approximate environment for the component such that it captures all possible behaviors and interleavings of the actual environment. Such approximate environment introduces a large amount of impossible behavior, which may cause the reachable state space of a component grows too large and increase the complexity of the verification process. To improve the efficiency of the verification of each component, this chapter presents three techniques to reduce state space without affecting verification results. The first technique, *autofailure reduction*, trims failure paths by pinpointing the real reason indicating which input actions from the environments lead to the failure state. This technique simplifies the parts of state graphs near the failure state, but does not bring reduction to the rest parts of the state graphs. Taking into account the interaction among the components, the invisible behaviors of a component are irrelevant to the interface specifications between the considered component and its neighbors. To hide these unnecessary details, the second technique, *transition-based reduction*, removes all invisible state transitions from the state graph while keeping all visible behavior without introducing any extra behavior. But the visible state transitions added may introduce nondeterminism into the state graphs and create equivalent paths. Extra computation is needed to remove such equivalent paths. To remove as much as possible observably equivalent paths and avoid generating any extra paths, the third technique, *region-based reductions* removes observably equivalent paths of a state graph. This technique does not aggressively remove all invisible state transitions as the transition-based reduction.

These three reduction techniques are effective in reducing the state graphs of components with the approximate environments and thus improve the efficiency of the abstraction refinement and verification in the next stage as discussed in Chapter 4. The reduced state graphs keep all observable paths of the original ones and remove unnecessary state transitions with the respect of the observable behavior of SGs. Thus the verification results based on the simplified SGs are sound and complete.

3.1 Autofailure Reduction

The autofailure reduction is based on the following observation. The failure state of a design may be entered by an action on an output or an invisible action. However, the real reason of the failure can be traced back to an input action. This is because if an environment produces an input action that a system cannot handle, then the failure happens immediately or through a sequence of invisible or output actions, and the environment cannot prevent it from eventually happening. This is referred to as *autofailure manifestation* in [38]. However, autofailure manifestation is only used to canonicalize trace structures for hierarchical verification. It is adopted in our method as a technique to reduce SGs.

Let $\rho = (s_0, t_0, s_1, t_1, s_2, \cdots, \pi)$ be a failure path in G. Recall that a path is autonomous if all actions on that path are in $\mathcal{A}^O \cup \{\zeta\}$. An autonomous path is independent of input actions. If a failure path of a SG is autonomous, the failure is inherent in the SG, and occurs no matter how the environment behaves. The autofailure reduction reduces a SG containing an autonomous failure path starting from the initial state *init* to the one consisting of only a single failure state. If ρ is not autonomous, the autofailure reduction searches for the largest index i such that action a_i is an input action, and $(s_{i+1}, a_{i+1}, s_{i+2}, \cdots, \pi)$ is an autonomous subpath of ρ. All states on that autonomous subpath, referred to as *autofailure states*, are removed, and s_{i+1} is converted to the failure state π. Notice that the removed state transitions on the autonomous subpath may be on the output actions.

Consider the design shown in Figure 2.1(a). Suppose some approximate environments obtained for the components of the design, which generate the inputs for the components

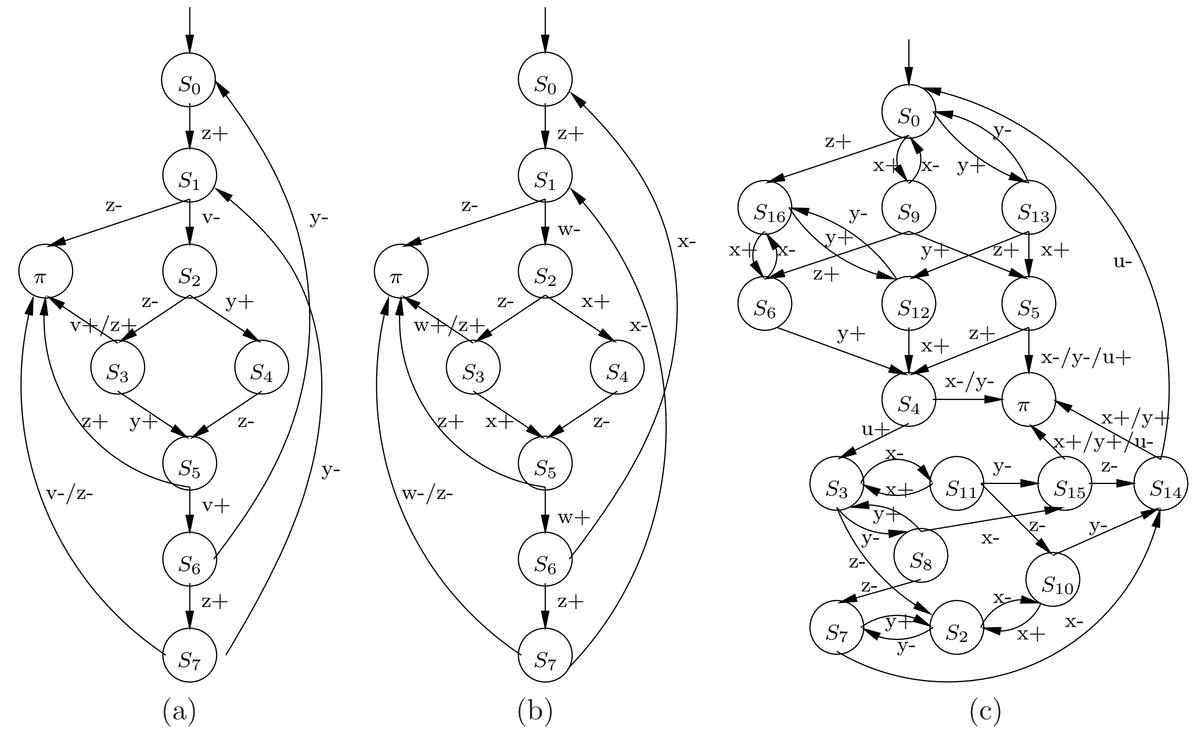


Figure 3.1. (a) - (c) are the SGs generated by components M_1, M_2 and M_3 with their approximate environment for the design shown in Figure 2.1(a), respectively.

at any time. The SGs constructed for the components operating with their approximate environments are shown in Figure 3.1(a) - (c), respectively. For example, the SG G_3 in Figure 3.1(c) is generated from the component M_3 in Figure 2.1(a) where the inputs of M_3 are set to completely free. The state transition $t_1 = (s_{15}, u-, \pi)$ is on an invisible action $u-$. Both incoming state transitions $t_2 = (s_8, x-, s_{15})$ and $t_3 = (s_{11}, y-, s_{15})$ are on input actions $x-$ and $y-$, respectively. This indicates that $x-$ and $y-$ happen too early before M_3 is ready. The autofailure reduction removes t_1, and changes t_2 and t_3 to $(s_8, x-, \pi)$ and $(s_{11}, y-, \pi)$, respectively. The operation is also applied to $(s_5, u+, \pi)$ similarly. After these operations, s_5 and s_{15} become the failure state. Thus original state transitions starting from s_5 and s_{15}, respectively, become unreachable, and are removed. After applying the autofailure reduction to the SGs shown in Figure 3.1(a)-(c), the reduced SGs are shown in Figure 3.2(a)-(c), respectively. Note that all state transitions entering the failure state are on input actions after applying the autofailure reduction.

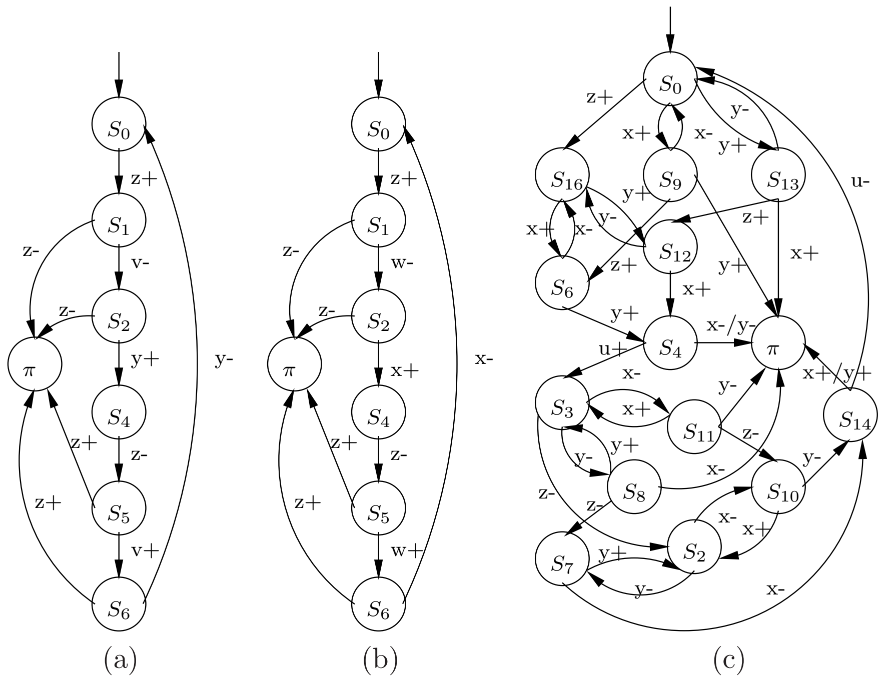


Figure 3.2. (a) - (c) show the SGs after applying the autofailure reduction on SGs in Figure 3.1(a) - (c), respectively.

3.1.1 Basic Algorithm

Algorithm 2 illustrates the `autofailure` algorithm. Given a state graph G, algorithm `autofailure` explores each path ending in the failure state π backward. On each path, it iteratively removes the last state transition of the path whose action is an invisible action or output action, and converts the start state of such state transitions to the failure state (line 5 - 7). During the backward examination, whenever a state transition is encountered such that it starts from the initial state and is on a non-input action, `autofailure` returns a message to report the SG G contains a failure (line 3 - 4). In this case, a component is definitely involved in a failure no matter what input behavior is generated by its environment. The process of backward removal of state transitions is repeated until a state transition encountered is on an input action, where the end state of the state transition is converted to the failure state (line 9 - 10). When a state is converted to the failure state, all its original outgoing state transitions become unreachable from the initial state

Algorithm 2: `autofailure`(G)

```
1  $s_2 = \pi$;
2  foreach $(s_1, a_1, s_2) \in R \wedge s_1 \neq \pi$ do
3      if $s_1 = init \wedge a_1 \notin A^I$ then
4          return "$G$ has a failure";
5      if $a_1 \notin A^I$ then
6          delete $(s_1, a_1, s_2)$ ;
7          $s_2 = s_1$;
8      else
9          replace $(s_1, a_1, s_2)$ with $(s_1, a_1, \pi)$;
10         $s_2 = \pi$;
11 Remove unreachable states and transitions from $G$;
```

and thus are removed from G. As a result, some state transitions on invisible or output actions leading to the failure state and those originating from the newly converted failure states are removed. Therefore, all failure paths in the reduced SG G' are caused by the state transitions on input actions, which explains why a path leads to the failure state.

The autofailure reduction locates the real reason for the failure state on input actions produced by an environment. This technique only shortens some failure paths and does not affect other paths in a SG. Lemma 3.1.1 shows that the autofailure reduction preserves all paths of a SG in terms of observable and failure equivalence.

Lemma 3.1.1 *Given a SG G,* $G \equiv$ `autofailure`(G).

Proof: If no failure trace exists in G, the procedure of the autofailure reduction does nothing. Therefore, $G \equiv$ `autofailure`(G).

Next, we consider G that contains failure paths. Let $\rho = (s_0, t_0, s_1, t_1, s_2, \cdots, \pi)$ be a failure path in G. Suppose ρ becomes $\rho' = (s_0, a_0, \cdots, s_i, a_i, \pi)$ after applying the autofailure reduction. According to Definition 2.2.6 in section 2.2.2, $\rho \sim_F \rho'$.

The above discussion indicates that every failure trace in G is reduced to an equivalent failure in `autofailure`(G). For each non-failure trace in G, it either has a corresponding equivalent failure trace in `autofailure`(G), or simply exists in `autofailure`(G) if it does not have the prefix of any failure trace in `autofailure`(G). ∎

3.1.2 On-the-fly Reduction

Since the actual environment for a component is difficult to obtain, the typical solution is to create an approximate environment that preserves all possible behaviors of the actual environment. This approximate environment may introduce a large number of extra states and state transitions during reachability analysis, causing the size of the SGs to explode. Integrating the autofailure reduction with reachability analysis can prevent unnecessary state transitions from being generated in the first place, since these state transitions would be removed by the autofailure reduction later in the normal way. Thus, the on-the-fly reduction is an improvement for the normal reachability analysis to reduce the complexity of the generated SGs for the components in the initial step. During the reachability analysis, when a state transition (s, a, π) on a non-input action a is enabled in s, it is not necessary to explore other state transitions enabled in s because s is to be converted to π according to the autofailure reduction. Other state transitions that have been generated from s are removed since they become unreachable after s is converted to π. Therefore, combining the autofailure reduction on-the-fly with reachability analysis removes unnecessary state transitions as early as possible to control the size of the initial SGs effectively.

Compared with the algorithm, findSG, shown in Algorithm 1 in Section 2.4, the on-the-fly reduction integrates the autofailure reduction by initializing a state set F composed of π and replaces line 11 and line 12 with the backward examination of the autofailure reduction. The modifications of Algorithm 1 are shown as follows.

1. Add $F = \{\pi\}$ to line 1.

2. Replace line 11 and line 12 with the following statements.
   line $11'$: If$(failure = true \vee s' \in F) \wedge a \notin \mathcal{A}^I$
   line $12'$: $F = \texttt{backtrack}(F, R, s)$

In Algorithm 1, a state transition (s, a, s') is a potential new state transition of the created SG G. We examine if the state transition (s, a, s') satisfies the condition to trigger the autofailure reduction in line $11'$. If s' is the failure state π or in the state set F of which all

Algorithm 3: `backtrack` (F, R, s)

```
1  $s_2 = s, F = F \cup \{s\}$;
2  foreach $(s_1, a_1, s_2) \in R \wedge s_2 \in F$ do
3      if $s_1 = init \wedge a_1 \notin A^I$ then
4          return "$G$ has a failure";
5      if $a_1 \notin A^I$ then
6          delete $(s_1, a_1, s_2)$ ;
7          $s_2 = s_1$;
8          $F = F \cup \{s_1\}$;
9      else
10         replace $(s_1, a_1, s_2)$ with $(s_1, a_1, \pi)$;
11         $s_2 = \pi$;
12 return $F$;
```

states are detected by the autofailure reduction as the failure state, and the executed action a is not an input action, the embedded autofailure reduction, algorithm `backtrack`(F, R, s), as to be described below, is invoked to detect and remove state transitions by the autofailure reduction in line $12'$.

The algorithm `backtrack`(F, R, s) is shown in Algorithm 3. It takes the set of autofailure states F, the state transition set R and a state s which enables an non-input action to the failure state. The Algorithm 3, `backtrack`(F, R, s), is similar to the Algorithm 2, `auofailure`(G). However `backtrack`(F, R, s) starts from the state transitions ending in the state s and F contains s by initialization (line 1), while `auofailure`(G) starts from the state transitions ending in the failure state π. All states of the set F are considered as the failure state. During the backward examination, the set of autofailure states F is augmented by adding the newly detected the failure state (line 8). In the end, F is returned where all states in F are converted to the failure state.

3.2 Transition-based Reduction

The autofailure reduction shortens the failure paths, but in general does not remove state transitions in a large amount. Recall that given a failure path $\rho = (s_0, a_0, \cdots, s_i, a_i, \pi)$, the non-failure prefix of ρ is $(s_0, a_0, \cdots, s_i, a_i)$. The autofailure reduction does not reduce the non-failure prefix of a path when a_i is an input action. Therefore, a large amount of extra behavior can still exist in a state graph. Notice that the invisible state transitions of a state graph do not affect the behavior on the interface when this state graph interacts with the other ones. In this section, another reduction technique, transition-based reduction, is proposed to remove all invisible state transitions of each path while preserving all observable behavior of the original SG. The basic idea of the transition-based reduction is to compress a sequence of invisible state transitions into a single state transition. Let $(s_{i-1}, a_{i-1}, s_i, \zeta, s_{i+1}, \zeta, \cdots, s_{j-1}, \zeta, s_j, a_j, s_{j+1})$ be a subpath of a path in a SG G, where ζ denotes the invisible actions and other actions on the subpath are visible ones. The sequence of state transitions $(s_i, \zeta, s_{i+1}, \zeta, \cdots, s_{j-1}, \zeta, s_j, a_j, s_{j+1})$ is compressed to state transition (s_i, a_j, s_{j+1}). Therefore, the original subpath is reduced to a shorter one $(s_{i-1}, a_{i-1}, s_i, a_j, s_{j+1})$.

However, the transition-based reduction may increase the complexity of the reduced state graph in the following two cases.

1. Case 1.

   When compressing a sequence of invisible state transitions on a path to a state transition, the last visible state transition (s_j, a_j, s_{j+1}) is added to the outgoing transition set of the state s_i as (s_i, a_j, s_{j+1}). To keep all visible behavior, the last visible state transition is added to the outgoing state transition set of each state on the subpath being compressed, where an incoming state transition of such state is on a visible action. In general, the reduced SG becomes simpler since all invisible state transitions are removed from the original SG. However, in some cases, the complexity of the reduced SG may be increased reversely as illustrated by the example shown in Figure 3.4.

2. Case 2.

   Nondeterminism may be introduced into a SG after the reduction. Consider two subpaths $(s_i, \zeta, \cdots, s_{j-1}, \zeta, s_j, a_j, s_{j+1})$ and $(s_i, \zeta, \cdots, s_{k-1}, \zeta, s_k, a_k, s_{k+1})$. They are

Algorithm 4: `tranRedue` (G)

```
1  $T = \emptyset, V = \emptyset$;
2  foreach $(s_2, a_2, s_1) \in R \wedge (a_2 \neq \zeta) \wedge s_2 \neq \pi$ do
3      foreach $(s_3, a_3, s_2) \in R \wedge s_3 \neq \pi$ do
4          if $s_2 = init \vee a_3 \neq \zeta$ then
5              $T = T \cup \{ (s_2, a_2, s_1)\}$;
6              $V = V \cup \{s_2, s_1\}$;
7          if $a_3 = \zeta$ then
8              $s_2 = s_3$;
9  replace $R$ with $T$;
10 replace $S$ with $V$;
11 Remove unreachable states and state transitions from $G$;
```

reduced to (s_i, a_j, s_{j+1}) and (s_i, a_k, s_{k+1}), respectively. This causes nondeterminism even though the original SG is deterministic. The nondeterministic state transitions may be eliminated if s_{j+1} or s_{k+1} is redundant as described later in this section.

Algorithm 4 shows the algorithm for the transition-based reduction. Given a SG G, `tranReduce` stores all visible state transitions and their states of the reduced SG in T and V, respectively. The reduced SG G' is the one to be returned. The algorithm searches backwards from each visible state transition (line 2 - 3), and bypasses all the invisible state transitions along a path (line 7 - 8) until another visible state transition is found or the initial state is reached (line 4). Under either of these two situations, a new transition is created to replace the sequences of invisible state transitions and it is added into T (line 5). As a result, the last visible state transition is added to the outgoing state transition set of the state which is the initial state or one of the states whose incoming state transitions are on visible actions. Then, the start state and end state of this newly added state transition are inserted into V (line 6). During the backward search, the invisible state transitions and states with all incoming state transitions on invisible actions are not added into T and V, since after the invisible state transitions are removed, all states whose incoming state transitions are on invisible actions become unreachable from the initial state. After all state transitions have been handled, R and S of G are replaced with T and V, respectively. Therefore, a new reduced SG G' is created.

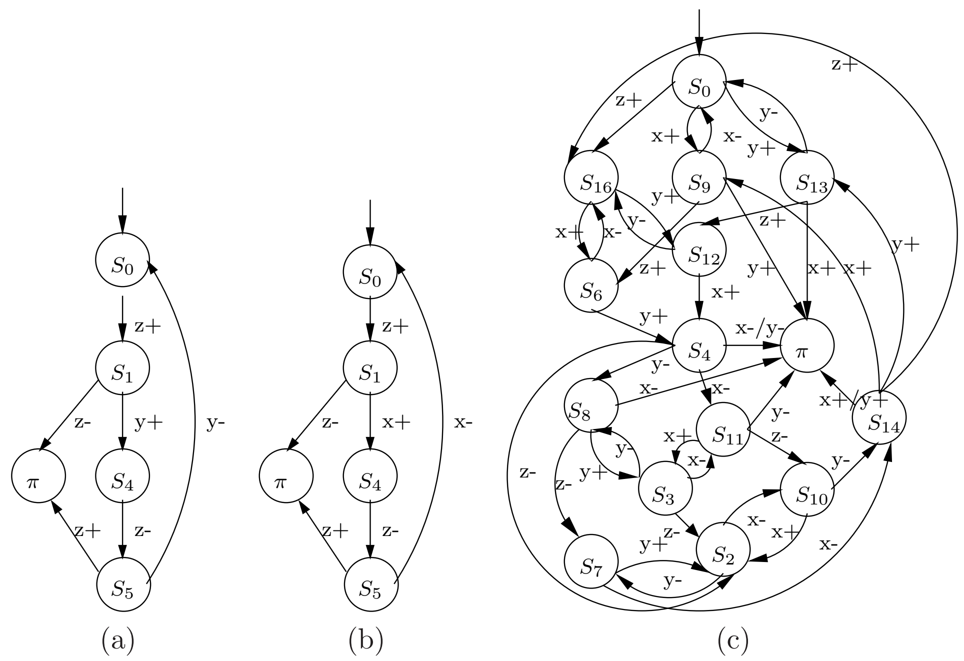


Figure 3.3. (a) - (c) show the SGs after applying the transition-based reduction on SGs in Figure 3.2(a) - (c), respectively.

As an example, Algorithm 4 is applied to remove the invisible state transitions in G_1, G_2, and G_3 shown in Figure 3.2, which have been reduced by applying the autofailure reduction on the initial SGs. The process of the transition-based reduction being applied on G_1, G_2 and G_3 is illustrated as follows.

1. Consider the transition-based reduction for the SG G_1.

   For a path to be compressed, e.g. $(s_{i-1}, a_{i-1}, s_i, \zeta, s_{i+1}, \zeta, \cdots, s_{j-1}, \zeta, s_j, a_j, s_{j+1})$, we start backward examination from the visible state transition (s_j, a_j, s_{j+1}). Thus we need apply the backward examination on each visible state transition of G_1 to get all the paths to be compressed. For example, we start backward examination from the state transition $(s_2, y+, s_4)$. After bypassing the invisible state transition $(s_1, v-, s_2)$, the backward examination stops at the state s_1 since s_1 contains an visible incoming state transition $(s_0, z+, s_1)$. According to the transition-based reduction, the subpath $(s_1, v-, s_2, y+, s_4)$ is compressed to a visible state transition $(s_1, y+, s_4)$. Consequently, $(s_1, y+, s_4)$ is added into G_1. Similarly, considering backward examination from the state transition $(s_2, z-, \pi)$, the subpath $(s_1, v-, s_2, z-, \pi)$ is to be compressed to $(s_1, z-, \pi)$. Since the state transition $(s_1, z-, \pi)$ already exists in G_1, it is not necessary to add this state transition to G_1. After all outgoing state transitions at state s_2 are considered, $(s_1, v-, s_2)$ is removed from G_1, and thus s_2 becomes unreachable from the initial state. So all outgoing state transitions from s_2 are unreachable from the initial state as well, and they are also removed from G_1. Thus the state transitions $(s_2, y+, s_4)$ and $(s_2, z-, \pi)$ are deleted. Similarly, consider the subpaths $(s_5, v+, s_6, y-, s_0)$ and $(s_5, v+, s_6, z+, \pi)$ starting from the state transitions $(s_6, y-, s_0)$ and $(s_6, z+, \pi)$, respectively. The state transition $(s_5, y-, s_0)$ is added to G_1 and $(s_5, z+, \pi)$ is not necessary to added since $(s_5, z+, \pi)$ already exists. Then the invisible state transition $(s_5, v+, s_6)$ is removed and the state s_6 becomes unreachable from the initial states. So all the outgoing state transitions at s_6 are removed since they are unreachable from the initial state. Thus the state transitions $(s_6, y-, s_0)$ and $(s_6, z+, \pi)$ are deleted from G_1. After all invisible state transitions and unreachable state transitions are removed from G_1, the reduced SG is shown in Figure 3.3(a).

2. Similar to handling G_1, G_2 is also reduced to a SG shown in Figure 3.3(b).

3. Consider the transition-based reduction for the SG G_3.

   The subpaths $(s_4, u+, s_3, z-, s_2)$, $(s_4, u+, s_3, y-, s_8)$ and $(s_4, u+, s_3, x-, s_{11})$ are to be compress to $(s_4, z-, s_2), (s_4, y-, s_8)$ and $(s_4, x-, s_{11})$, respectively. Therefore, $(s_4, z-, s_2), (s_4, y-, s_8)$ and $(s_4, x-, s_{11})$ are added to G_3 and $(s_4, u+, s_3)$ is removed. Notice that the reduced SG, shown in Figure 3.3(c), turns out to be a nondeterministic SG because the state transition $(s_4, x-, \pi)$ and newly added $(s_4, x-, s_{11})$ are chosen nondetermisticly in the state s_4, while the original G_3 before applying the transition-based reduction is a deterministic SG.

We give an example to illustrate how the complexity of the reduced SGs can increase by the transition-based reduction. Figure 3.4(a) shows a partial SG containing the invisible action ζ. For example, to remove the state transition (s_2, ζ, s_3), during the backward search,

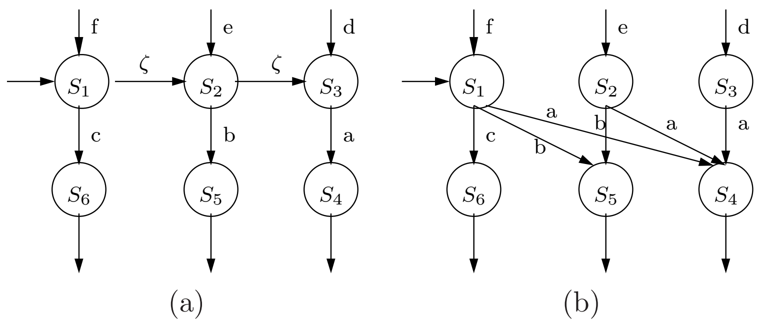


Figure 3.4. (a) A partial SG. (b) The reduced SG after applying the transition-based reduction on the SG in (a).

new state transitions (s_2, a, s_4) and (s_1, a, s_4) are inserted to the SG, since there are visible state transitions entering state s_2 and s_1, respectively. Similarly, to remove the invisible state transition (s_1, ζ, s_2), the new state transition (s_1, b, s_5) is added to the SG. Thus, the reduced SG, shown in Figure 3.4(b) contains one more state transition than the original SG in Figure 3.4(a). In some cases, the number of state transitions in the reduced SG can increase significantly.

The transition-based reduction removes all invisible state transitions, but does not introduce new observable paths. The following lemma asserts that the reduced SG generated by `tranReduce`(G) is observably equivalent to the original SG G.

Lemma 3.2.1 *Given a SG G, $G \equiv$ `tranReduce`(G).*

Proof: It is straightforward to see that for every path ρ in G, there exists a path ρ' in `abstract`(G) such that $\rho \sim \rho'$. This satisfies the conditions of the abstraction relation, and completes the proof. ∎

3.2.1 Equivalent State Removal

Recall that the procedure for the transition-based reduction potentially introduces nondeterminism. A nondeterministic SG can be determinized with some well-known but very expensive algorithms [19]. Even though nondeterminism does not affect the soundness of the verification results in our framework, it increases the complexity of the verification.

Therefore, we propose a light-weight algorithm instead that targets on removing redundant state transitions and states due to nondeterminism introduced by the transition-based reduction.

Let $incoming(s_i)$ be the set of state transitions (s_{i-1}, a_{i-1}, s_i) such that $R(s_{i-1}, a_{i-1}, s_i)$ holds, and $outgoing(s_i)$ be the set of state transitions (s_i, a_i, s_{i+1}) such that $R(s_i, a_i, s_{i+1})$ holds.

Definition 3.2.1 *Let G be a SG, and the states $s_{i-1}, s_i, s_j \in S$ such that $s_i \neq \pi$ and $s_i \neq init$. We say that s_i is equivalent to s_j, denoted as $s_i \equiv s_j$, if for each state transition $(s_{i-1}, a_{i-1}, s_i) \in incoming(s_i)$, there exists a state transition $(s_{i-1}, a_{i-1}, s_j) \in incoming(s_j)$ and vice versa.*

When the state s_i is equivalent to the state s_j, all prefix of the paths containing s_i, such as $(s_0, \cdots, s_{i-1}, a_{i-1}, s_i)$, are also the prefix of the paths passing the state s_j, and vice versa. After adding the outgoing state transitions of s_i to s_j, all path containing s_i are equivalent to the corresponding paths containing s_j. Thus the equivalent state s_i and its incoming and outgoing transitions can be removed safely. The two steps of the equivalent state removal are shown as follows.

1. For each $(s_i, a_i, s_{i+1}) \in outgoing(s_i)$, add (s_j, a_i, s_{i+1}) into R.
2. Remove all state transitions in $incoming(s_i)$ and $outgoing(s_i)$.

Figure 3.5 illustrates the removal of the equivalent states of a partial SG. In Figure 3.5(a), action a and b are nondeterminstically enabled at the state s_1 and s_2, respectively. According to Definition 3.2.1, s_4 is equivalent to s_5, i.e. $s_4 \equiv s_5$. By adding the outgoing transition of s_4 to s_5, a new state transition (s_5, g, s_6) is added to the SG. Thus, we have the subpath (f, s_1, a, s_4, g, s_6) is equivalent to the subpath (f, s_1, a, s_5, g, s_6). Similarly, the subpath (e, s_2, b, s_4, g, s_6) is equivalent to the subpath (e, s_2, b, s_5, g, s_6). After removing the equivalent state s_4 and its incoming and outgoing state transitions, the reduced SG is shown

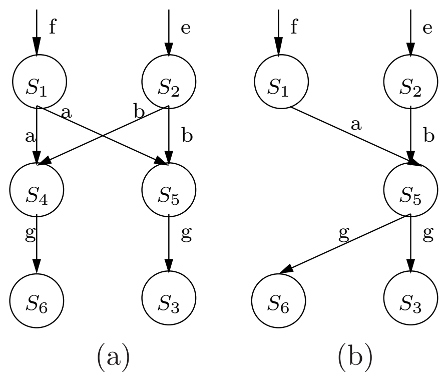


Figure 3.5. (a) A partial SG. (b) The reduced SG after applying the equivalent state removal on the SG in (a).

in Figure 3.5(b). Obviously, removing equivalent states always results in decreasing the number of states and state transitions of a SG.

If the failure state is involved in nondeterminism, equivalent state transitions are identified based on the following understanding: if there is a possibility that an action in a state may cause a failure, it is always regarded as causing a failure. Therefore, no failure behavior is missed in the process of the verification, and the soundness of the verification is ensured. The failure equivalent state transitions are formalized in the following definition.

Definition 3.2.2 *Given two state transitions* (s_i, a_i, s_{i+1}) *and* (s_i, a_i, π) *of a SG,* (s_i, a_i, s_{i+1}) *is failure equivalent to* (s_i, a_i, π).

The failure equivalent transitions do not have any impact on the behavior represented by a SG, and can safely be removed. Therefore, to some extent, the nondeterminism due to the failure equivalent transitions are removed, and the SG is simplified.

The algorithm `rmRed`(G) to remove the equivalent states and equivalent paths is shown in Algorithm 5. It first examines the failure equivalent transitions and remove the equivalent ones from the SG (line1 - 3). Then it searchs the state transition set for the pairs of the equivalent states, $s_i \equiv s_j$ according to the Definition 3.2.1 (line 4 - 6). When the pair of equivalent states $s_i \equiv s_j$ is detected, for each outgoing state transition of s_i, a new state transition is created and added to the outgoing state transition set of s_j (line 7 - 8). Then

Algorithm 5: `rmRed` (G)

```
1  foreach $(s_i, a_i, \pi) \in R$ do
2      if $(s_i, a_i, s_{i+1}) \in R$ then
3          $R = R - \{(s_i, a_i, s_{i+1})\}$ ;
4  foreach $s_i \in S$ do
5      foreach $s_j \in S$ do
6          if $s_i \equiv s_j$ then
7              foreach $(s_i, a_i, s_{i+1}) \in outgoing(s_i)$ do
8                  $R = R \cup \{(s_j, a_i, s_{i+1})\}$;
9              $R = R - \{incoming(s_i) \cup outgoing(s_i)\}$;
10             $S = S - \{s_o\}$;
11 Remove unreachable states and state transitions from $G$;
```

all the incoming state transitions and outgoing state transitions of s_i are removed from the state transition set (line 9), and the equivalent state s_i is removed from the state set (line 10). In the end, all state and state transitions unreachable from the initial state are removed from the SG (line 11). In the end, the reduced SG is returned.

We illustrate how the algorithm **`rmRed`** is applied to the SGs shown in Figure 3.3 after applying the transition-based reduction. G_1 and G_2 shown in Figure 3.3(a) and (b), respectively, cannot be reduced further by the equivalent state removal since both SGs are deterministic. However, with the respect of G_3 shown in Figure 3.3(c), nondeterminism is detected in the state s_4 and s_{14}. We observe that the state transition $(s_4, x-, s_{11})$ is failure equivalent to $(s_4, x-, \pi)$, and $(s_4, y-, s_8)$ is failure equivalent to $(s_4, y-, \pi)$ as well. Thus, the state transitions $(s_4, x-, s_{11})$ and $(s_4, y-, s_8)$ are removed from G_3 safely. Similarly, for the failure equivalent transitions in the state s_{14}, the state transitions $(s_{14}, x+, s_9)$ and $(s_{14}, y+, s_{13})$ are removed from G_3. Consequently, s_3 , s_8 and s_{11} become unreachable from the initial state. After removing these three unreachable states and their outgoing state transitions, the reduced SG is shown in Figure 3.6.

The equivalent state removal is a conservative reduction technique. It removes the equivalent states and paths and does not introduce any extra paths. Therefore, nondeterminism introduced by the transition-based reduction can be eliminated partially. The following lemma states that the reduced SG generated by **`rmRed`** is equivalent to the original SG.

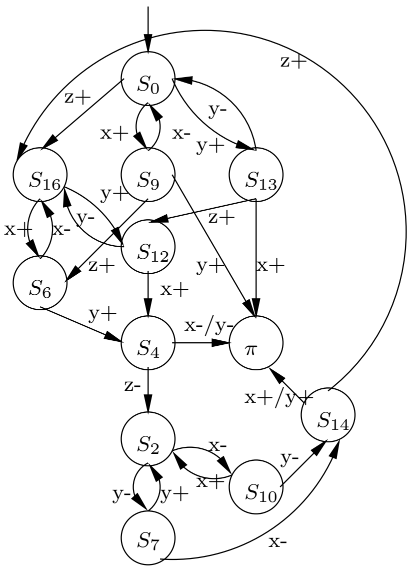


Figure 3.6. The reduced SG after applying the equivalent state removal on the SG G_3 in Figure 3.3(c).

Lemma 3.2.2 *Given a SG G, $G \equiv$ rmRed(G).*

Proof: We consider two cases.

1. Case 1. $s'_i \equiv s_i$ where s'_i and s_i are not the failure state. For any path $\rho = (\cdots, s_{i-1}, a_{i-1}, s_i, a_i, \cdots)$ in G, there exists a path $\rho' = (\cdots, s_{i-1}, a_{i-1}, s'_i, a_i, \cdots)$ in `rmRed`(G) such that $\rho \sim \rho'$.

2. Case 2. For any path $\rho = (\cdots, s_i, a_i, s_{i+1}, \cdots)$ in G and $(s_i, a_i, \pi) \in R$, there exists $\rho' = (\cdots, s_i, a_i, \pi, \cdots)$ in in `rmRed`(G) such that $\rho \sim_F \rho'$.

From case 1 and 2, it is concluded that $G \equiv$ `rmRed`(G). ∎

3.3 Region-based Reduction

The transition-based reduction keeps all observable paths while removing all invisible state transitions of a SG. To guarantee all observable paths preserved in the reduced SGs, a large amount of new visible state transitions may need to be inserted. Therefore, the size of the reduced SGs may be increased and nondeterminism may be introduced. The complementary technique for the transition-based reduction, the equivalent state removal, removes the observably equivalent paths caused by the nondeterminism to some extent. However the conditions to detect equivalent states are restricted. Thus the reduction by the equivalent state removal is limited. To simplify the state spaces and guarantee the correctness of the verification, reduction techniques that can remove as many as possible the observably equivalent paths are needed. In this section, a region-based reduction is proposed to detect a set of observably paths from the perspective of the paths instead of the pairs of states. In addition, we present a complementary technique for the region-based reduction, case-based simplifications, to remove more observably equivalent paths that cannot be identified by the region-based reduction. With combination of these two techniques, a lot of observably equivalent paths can be removed from SGs. The reduced SGs can significantly simplify the verification. However, with the purpose of keeping all possible observable behavior of the original SGs, not all the invisible state transitions can be removed from the state graphs.

Intuitively, a *region* is a subgraph such that some paths that are constructed using the state transitions in the region have an observably equivalent path constructed by using some other state transitions in the same region. The purpose of the region-based reduction is to identify and remove as many state transitions in the region as possible such that only one representative of a set of observably equivalent paths passing through that region is preserved. The formal definition of a region is given as follows. First, a set of *concurrent internal* transitions of a state graph is defined by which a region can be formed directly. Recall that the set of actions enabled at a state $s_i \in S$ is denoted as $enb(s_i) = \{a \mid (s_i, a, s_{i+1}) \in R\}$. Let $outgoing(s_i)$ be the set of state transitions leaving a state s_i, i.e. $\{(s_i, a, s_{i+1}) \in R\}$.

Definition 3.3.1 *Let G be a SG, and $\zeta \in \mathcal{A}^X$ be an invisible action. A set of concurrent invisible transitions, $CIT_\zeta \subseteq \{(s_i, \zeta, s_{i+1}) \in R\}$ is defined such that the following condition hold. For each $(s_i, \zeta, s_{i+1}) \in CIT_\zeta$,*

$$\forall_{(s_i,a_i,s_j)\in outgoing(s_i)}, a_i \neq \zeta \Rightarrow$$
$$\exists_{(s_{i+1},a_i,s_{j+1})\in outgoing(s_{i+1})}, ((s_j, \zeta, s_{j+1}) \in CIT_\zeta \vee s_j = s_{j+1}).$$

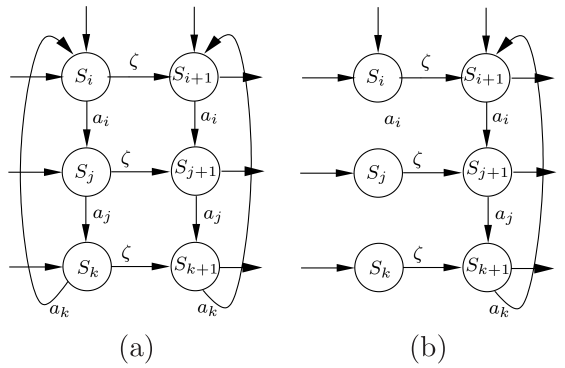


Figure 3.7. (a) An example of a region which contains all six states and the state transitions in *outgoing*(s_i), *outgoing*(s_j), and *outgoing*(s_k), (b) the region in (a) after the removing the state transitions in *outgoing*(s_i), *outgoing*(s_j) and *outgoing*(s_k) other than those on the invisible action ζ.

In the above definition, the first condition specifies that for each concurrent invisible transition (s_i, ζ, s_{i+1}), there is no disabling between any actions in *enb*(s_i) allowed. In other words, the enabling of all actions in s_i other than ζ is independent of ζ. To explain the second condition, it is convenient to define *concurrent successor* transitions. Given a state transition (s_i, ζ, s_{i+1}), state transition (s_j, ζ, s_{j+1}) is a concurrent successor of (s_i, ζ, s_{i+1}) if there are $(s_i, a_i, s_j) \in outgoing(s_i)$ and $(s_{i+1}, a_i, s_{j+1}) \in outgoing(s_{i+1})$. The second condition requires that for an invisible transition $(s_i, \zeta, s_{i+1}) \in CIT_\zeta$, all of its concurrent successors must also be in CIT_ζ or the future behavior from s_i and s_{i+1} converges. These two conditions together indicate that concurrent invisible transitions do not distinguish the observable behaviors.

Given a set of concurrent invisible transitions, a region can be formed as defined below.

Definition 3.3.2 *Let G be a SG, and $CIT_\zeta \subseteq \{(s_i, \zeta, s_{i+1}) \in R\}$ be a set of concurrent invisible transitions in G. A region RG_ζ is (S_ζ, R_ζ) where*

1. $S_\zeta = \{s_i, s_{i+1} \mid (s_i, \zeta, s_{i+1}) \in CIT_\zeta\}$,

2. $R_\zeta = \{outgoing(s_i) \mid (s_i, \zeta, s_{i+1}) \in CIT_\zeta\}$.

The first example on the region is shown in Figure 3.7(a). In Figure 3.7(a), the region contains all six states, and the state transitions in *outgoing*(s_i), *outgoing*(s_j), and

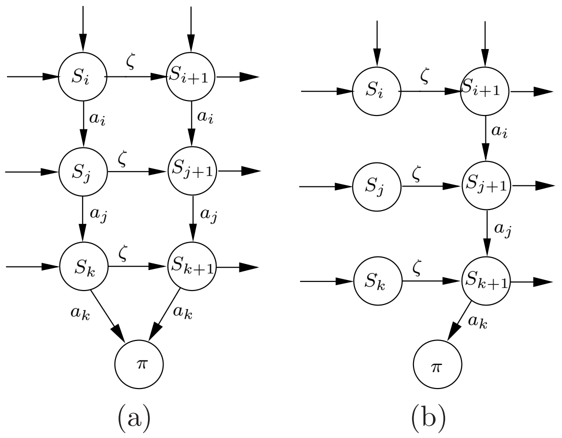


Figure 3.8. (a) Another example of a region where paths $(\ldots, s_i, a_i, s_j, a_j, \ldots)$ and $(\ldots, s_{i+1}, a_i, s_{j+1}, a_j, \ldots)$ converge to the failure state π, (b) the region in (a) after the reduction.

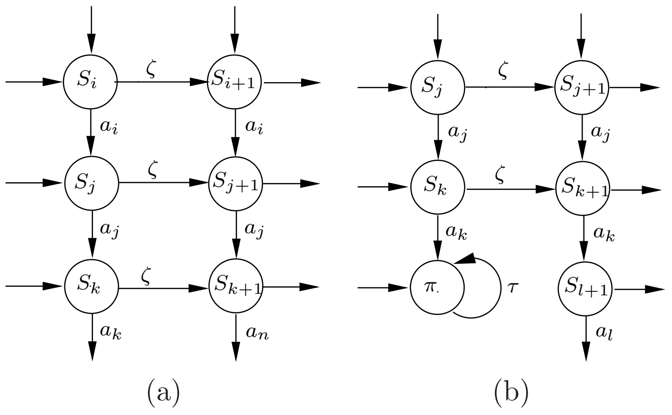


Figure 3.9. Examples of invisible state transitions that cannot form a region. (a) State s_k and s_{k+1} have different outgoing transitions, (b) State s_k has a transition going to the failure state while s_{k+1} does not.

outgoing(s_k). Figure 3.7(b) shows the reduced partial SG by removing the observably equivalent paths by the region-based reduction described in the following.

The second example on the region is shown in Figure 3.8(a). From states s_k and s_{k+1}, two state transitions on a_k converge to the failure state π. Figure 3.8(b) shows the reduced partial SG after applying the region-based reduction.

In contrast, SGs in Figure 3.9(a) and Figure 3.9(b) show examples where a region cannot be formed. In Figure 3.9(a), action a_k is enabled in state s_k, but not in s_{k+1}. Neither the state transition (s_k, ζ, s_{k+1}) nor the state transition (s_j, ζ, s_{j+1}) can be in CIT_ζ since the condition in Definition 3.3.1 cannot be satisfied. Subsequently, (s_i, ζ, s_{i+1}) cannot be in CIT_ζ either on the similar basis. In Figure 3.9(b), transition (s_k, ζ, s_{k+1}) also violates the condition in Definition 3.3.1 because it does not have its concurrent successor in CIT_ζ and thus the state transitions from s_k and s_{k+1} do not converge to the same state. Similarly, (s_j, ζ, s_{j+1}) cannot be in CIT_ζ either, therefore no region can be formed.

Consider an invisible state transition $(s_i, \zeta, s_{i+1}) \in R_\zeta$ of a region RG_ζ, every path that includes a state transition in *outgoing*(s_i) but excluding (s_i, ζ, s_{i+1}) has a different but observably equivalent path involving a corresponding state transition in $out(s_{i+1})$. Either one path can be safely removed. After a region in a SG is identified, we choose to keep the paths containing (s_i, ζ, s_{i+1}) and remove their observably equivalent ones. Therefore,s all transitions in *outgoing*(s_i) excluding (s_i, ζ, s_{i+1}) can be removed from the SG without changing the observable behavior of the SG. The region in Figure 3.7(a) and 3.8(a) are reduced as shown in Figure 3.7(b) and 3.8(b), respectively.

The region-based reduction removes the observably equivalent paths and keeps all visible behaviors of the original state graph. The following lemma shows that the new SG resulting from the region-based reduction is observably equivalent to the original one.

Lemma 3.3.1 *Let G be a SG, and RG_ζ be a region in G. Also let G' denote the SG G after removing all state transitions in outgoing$(s_i) \backslash (s_i, \zeta, s_{i+1})$ for each (s_i, ζ, s_{i+1}) in RG_ζ. Then,*

$$G' \equiv G$$

Proof: To prove the equivalence, one needs to show that for every path in G that disappears in G', there is a corresponding equivalent path in G'. Let RG_ζ be a region. According to condition in Definition 3.3.1, the following condition holds for each $(s_i, \zeta, s_{i+1}) \in R_\zeta$ of RG_ζ,

$$\forall (s_i, a_i, s_j) \in outgoing(s_i) \text{ where } a_i \neq \zeta \Rightarrow \exists (s_{i+1}, a_i, s_{j+1}) \in outgoing(s_{i+1})$$

This means that for every path in G, $\rho = (\ldots, s_i, a_i, s_j, \ldots)$, such that $(s_i, a_i, s_j) \in outgoing(s_i)$, there is a corresponding path $\rho' = (\ldots, s_i, \zeta, s_{i+1}, a_i, s_{j+1}, \ldots)$ such that $(s_{i+1}, a_i, s_{j+1}) \in outgoing(s_{i+1})$, and $\rho[\mathcal{A}'] \sim \rho'[\mathcal{A}']$ where $\mathcal{A}' = \mathcal{A}^I \cup \mathcal{A}^O$. This indicates that removing state transition (s_i, a_i, s_j) such that $a_i \neq \zeta$ from the region does not reduce any observably equivalent paths in G. And obviously, this reduction does not introduce any extra paths either. Therefore, $G \equiv G'$. ∎

For each invisible action ζ, the state transitions on ζ can be partitioned into a number of sets such that some of them form regions as shown in Figure 3.7(a) and Figure 3.8(a), while the other ones do not form regions as shown in Figure 3.9(a) and (b). Finding regions can be implemented directly by following Definition 3.3.1 and 3.3.2. However, this may not always be efficient if the number of state transitions that cannot form regions is large. An alternative approach is to first search for the state transitions on ζ that cannot form regions. Thereafter, the remaining state transitions on ζ form one or more regions.

Based on the region definition, the set of invisible state transitions shown in Figure 3.9 does not form a region due to the state transition (s_k, ζ, s_{k+1}), which is referred to as *essential*. In general, an essential state transition (s_k, ζ, s_{k+1}) satisfies one of the following conditions:

1. $enb(s_k) \backslash \zeta \not\subseteq enb(s_{k+1})$,
2. $\exists_{(s_k, a_k, \pi) \in outgoing(s_k)}, a_k \neq \zeta \wedge R(s_{k+1}, a_k, s_{l+1}) \wedge s_{l+1} \neq \pi$.

A state transition is essential because whether taking such a state transition or not may distinguish the observable behaviors. For such an essential state transition (s_k, ζ, s_{k+1}), all state transitions in $outgoing(s_k)$ need to be preserved; otherwise if paths containing the

Algorithm 6: `RegionRd` (G)

```
1 foreach $\zeta \in \mathcal{A}^X$ do
2     $Essential_\zeta = \{(s_i, \zeta, s_{i+1}) \in R \mid (s_i, \zeta, s_{i+1})$ is essential$\}$;
3     $Keep_\zeta = FindKeep(Essential_\zeta)$ ;
4     foreach $(s_i, \zeta, s_{i+1}) \notin Essential_\zeta$ do
5         foreach $(s_i, a_i, s_j) \in outgoing(s_i) \wedge a_i \neq \zeta$ do
6             if $(s_i, a_i, s_j) \notin Keep_\zeta$ then
7                 $R = R \backslash (s_i, a_i, s_{i+1})$;
8 return $G$ ;
```

state transitions in *outgoing*(s_k) were removed, there would be no corresponding observably equivalent paths containing the state transitions in *outgoing*(s_{k+1}).

Similarly, given an essential state transition (s_k, ζ, s_{k+1}), if there exists another state transition (s_j, ζ, s_{j+1}) such that (s_k, ζ, s_{k+1}) is a concurrent successor resulting from executing a_j in s_j and s_{j+1} by the state transitions (s_j, a_j, s_k) and (s_{j+1}, a_j, s_{k+1}), respectively, then the state transition $(s_j, a_j, s_k) \in$ *outgoing*(s_j) must be preserved. Otherwise, if (s_j, a_j, s_k) were removed, the paths containing the essential state transition (s_k, ζ, s_{k+1}), would be lost and result in loss of behavior. Since for the state transition (s_j, ζ, s_{j+1}), all the outgoing state transitions of s_j, i.e. *outgoing*(s_j), needs to be preserved similar to the essential state transition (s_k, ζ, s_{k+1}), the state transition (s_j, ζ, s_{j+1}) should be regarded as essential too. Then the above steps are performed again based on (s_j, ζ, s_{j+1}) to find more essential state transitions that should be preserved. As an example, in Figure 3.9(a), the state transitions (s_i, ζ, s_{i+1}), (s_j, ζ, s_{j+1}), and (s_k, ζ, s_{k+1}) are essential, and the state transitions in *outgoing*(s_i), *outgoing*(s_j) and *outgoing*(s_k) need to be preserved.

After every essential state transition on an ζ is considered, a set of state transitions, denoted by $Keep_\zeta$, that must be preserved can also be found. Then, for each non-essential state transition, i.e. $(s_i, \zeta, s_{i+1}) \notin Essential_\zeta$, the set of state transitions that construct the observably equivalent paths to be removed is denoted as *Remove* $=$ *outgoing*$(s_i) - \{(s_i, \zeta, s_{i+1})\} - Keep_\zeta$. All the state transitions in the set of *Remove* can be removed safely.

Algorithm 7: `FindKeep` (E_ζ)

```
1 foreach $(s_k, \zeta, s_{k+1}) \in E_\zeta$ do
2     $Keep_\zeta = Keep_\zeta \cup outgoing(s_k)$;
3     if $\exists_{(s_j,\zeta,s_{j+1}) \in R}, (s_j, a_j, s_k) \in R \wedge (s_{j+1}, a_j, s_{k+1}) \in R$ then
4         $E_\zeta = E_\zeta \cup \{(s_j, \zeta, s_{j+1})\}$;
5         $Keep_\zeta = Keep_\zeta \cup outgoing(s_j)$;
6 return $Keep_\zeta$;
```

The above reduction is implemented in the algorithm `RegionRd` as shown in Algorithm 6. It takes a SG G, and performs region reduction for each invisible action ζ. For an invisible action ζ, `RegionRd` algorithm first finds all state transitions on ζ that must be preserved with respect to the two conditions for the essential state transitions (line 1 - 2). Then it collects the set of state transitions to be preserved for $Essential_\zeta$ by the invocation of the algorithm `FindKeep` (line 3). For an invisible action ζ, the set of state transitions on ζ that cannot form regions is held in $Essential_\zeta$. Therefore, for a region RG_ζ involved in ζ, $R_\zeta = \{(s_i, \zeta, s_{i+1}) \in R\} - \{Essential_\zeta\}$. To present the paths containing the state transitions of $Essential_\zeta$ from removing, all the state transitions of $Keep_\zeta$ must be kept. Thus, we have $Essential_\zeta \subseteq Keep_\zeta$. For each state transition of a region, i.e.$(s_i, \zeta, s_{i+1}) \notin Essential_\zeta$ (line 4), any path containing s_i has a corresponding observably equivalent path containing s_{i+1}. As observed above, all state transitions of $Keep_\zeta$ must be kept and all the state transitions of the set *Remove* are removed from the SG (line5 - 7). In the end, the reduced SG is retuned (line 8).

Algorithm 7 illustrates the `FindKeep` algorithm. It takes a set of essential state transitions on the invisible action ζ, E_ζ. For each state transition $(s_k, \zeta, s_{k+1}) \in E_\zeta$, all state transitions in $outgoing(s_k)$ are kept in $Keep_\zeta$ (line 2). Then we search for another essential state transition (s_j, ζ, s_{j+1}) such that (s_k, ζ, s_{k+1}) is a concurrent successor of (s_j, ζ, s_{j+1}) connected by the state transitions (s_j, a_j, s_k) and (s_{j+1}, a_j, s_{k+1}) (line 3). When such essential state transition is found, it is added to E_ζ (line 4), and all state transitions in $outgoing(s_j)$ are added to $Keep_\zeta$ (line 5). The set of state transition that must be kept for E_ζ is returned (line 6).

3.3.1 Case-based Simplifications

The region-based reduction detects a set of state transitions such that for each state transition of the region, every path passing through the start state of such state transition has an observably equivalent path passing through the end state of such state transition. Then only one representative of the observably equivalent paths needs to be preserved. To preserve all observable behavior of the original SG, the essential state transitions and all outgoing state transitions of the start state of an essential state transition must be kept in the reduced SG. Therefore, there are still some invisible state transitions remained in the reduced SG. Removing these invisible transitions reduces the size of SGs, but does not change the observably equivalent behavior of SGs if certain conditions are met. This section presents the techniques to determine observably equivalent paths that cannot be identified by the region-based reduction and provide more reduction to the SGs.

1. Case 1.

   This case involves an invisible transition $(s_i, \zeta, s_{i+1}) \in R$ such that there exists an $(s_{i+1}, a_i, \pi) \in R$ for an outgoing state transition $(s_i, a_i, s_j) \in \mathit{outgoing}(s_i)$ where s_j is either π or not. Then for any path $\rho_1 = (\cdots, s_i, a_i, s_j, \cdots)$ in G, there is another path $\rho_2 = (\cdots, s_i, \zeta, s_{i+1}, a_i, \pi)$ such that $\rho_1[\mathcal{A}'] \sim_F \rho_2[\mathcal{A}']$ where $\mathcal{A}' = \mathcal{A}^I \cup \mathcal{A}^O$. Figure 3.10 shows two examples where case 1 simplification can be applied.

   After this case in a SG is identified, the path ρ_1 is redundant, and it can be removed from G by deleting (s_i, a_i, s_j) whether s_j is a failure state or not. After reduction, the SGs in Figure 3.10(a) and (b) are reduced to ones as shown in Figure 3.11(a) and (b), respectively. It may seem that Case 1 simplification is a special case of the region-based reduction. However, Case 1 simplification can still be applied if the condition in the Definition 3.3.1 does not hold in state s_i. As shown in Figure 3.10(a) and (b), Case 1 is identified, but the region-based reduction cannot be applied, since (s_i, ζ, s_{i+1}) is an essential state transition due to $a_k \in \mathit{enb}(s_i)$ and $a_k \notin \mathit{enb}(s_{i+1})$.

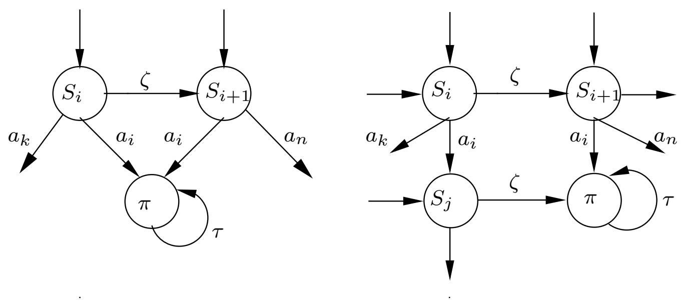


Figure 3.10. Two examples where case 1 simplification can be applied. (a) Paths $(\ldots, s_i, a_i, \pi)$ and $(\ldots, s_i, \zeta, s_{i+1}, a_i, \pi)$ are observably equivalent, (b) $(\ldots, s_i, a_i, s_j, \zeta, \pi)$ and $(\ldots, s_i, \zeta, s_{i+1}, a_i, \pi)$ are observably equivalent.

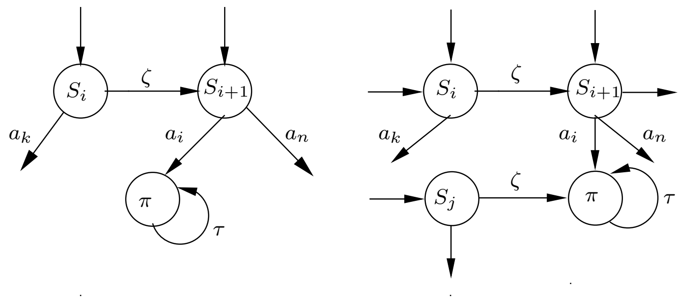


Figure 3.11. The corresponding reduced SGs for the examples shown in Figure 3.10 after removing the state transitions on a_i from s_i to eliminate the equivalent paths.

The following lemma shows that the new SG resulting from Case 1 simplification is observably equivalent to the original one.

Lemma 3.3.2 *Let G be a SG and G' is the SG after reduction using Case 1 simplification. Then, $G' \equiv G$.*

Proof: It is easy to see that for every path in G $\rho = (\cdots, s_i, a_i, s_j, \cdots)$, there is a corresponding path in G, $\rho' = (\cdots, s_i, \zeta, s_{i+1}, a_i, \pi)$, such that $\rho[\mathcal{A}'] \sim_F \rho'[\mathcal{A}']$ where $\mathcal{A}' = \mathcal{A}^I \cup \mathcal{A}^O$. Case 1 simplification removes the state transition (s_i, a_i, s_j) from G, thus eliminating path ρ. Path ρ' and all the other paths that do not include (s_i, a_i, s_j) still exist in G' after the reduction. Therefore, $G' \equiv G$. ∎

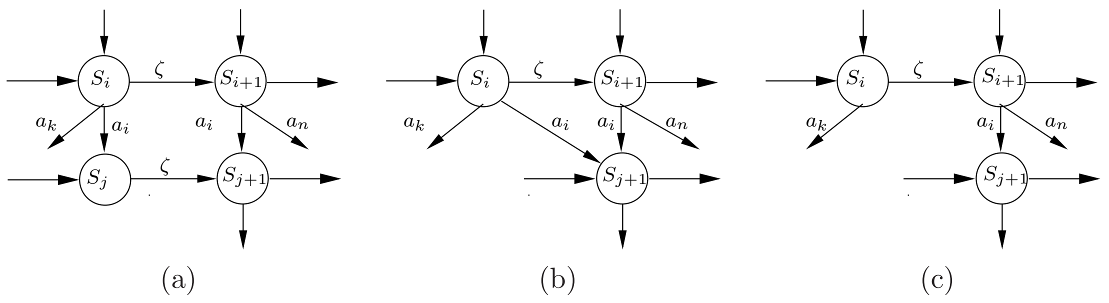


Figure 3.12. Illustration of Case 2 simplification. The observably equivalent paths $(\ldots, s_j, \zeta, s_{j+1}, \ldots)$ are removed. State s_j is also removed since it has only one outgoing state transition.

2. Case 2.

   This reduction deals with state transition (s_j, ζ, s_{j+1}) in the case such that (s_j, ζ, s_{j+1}) is the only state transition in *outgoing*(s_j) as shown in Figure 3.12(a). This case can be handled in two steps as follows. For every state transition entering the state s_j, e.g. (s_i, a_i, s_j), a new state transition (s_i, a_i, s_{j+1}) is created. Therefore, for any path passing through (s_i, a_i, s_j), e.g. $\rho = (\cdots, s_i, a_i, s_j, \zeta, s_{j+1}, \cdots)$, there exists another path passing through the newly added state transition (s_i, a_i, s_{j+1}), e.g. $\rho' = (\cdots, s_i, a_i, s_{j+1}, \cdots)$. Apparently, $\rho[\mathcal{A}'] \sim \rho'[\mathcal{A}']$. The path ρ can be removed safely by deleting (s_j, ζ, s_{j+1}) and all state transitions entering the state s_j. Figure 3.12(b) shows the SG after the first step. In the second step, if there exist state transitions (s_i, ζ, s_{i+1}) and (s_{i+1}, a_i, s_{j+1}) in the SG, then (s_i, a_i, s_{j+1}) is removed. During the second step, any path passing through (s_i, a_i, s_{j+1}) is observably equivalent to the corresponding path containing the subpath $(s_i, \zeta, a_i, s_{j+1})$. Thus, the former path is removed safely by deleting (s_i, a_i, s_{j+1}). The final simplified SG is shown in Figure 3.12(c). Similarly, when the condition in the Definition 3.3.1 does not hold in s_i as shown in Figure 3.12(a), the SG can still be simplified by Case 2 simplification but not the region-based reduction.

   The following lemma shows that the reduced SG resulting from Case 2 simplification is observably equivalent to the original one.

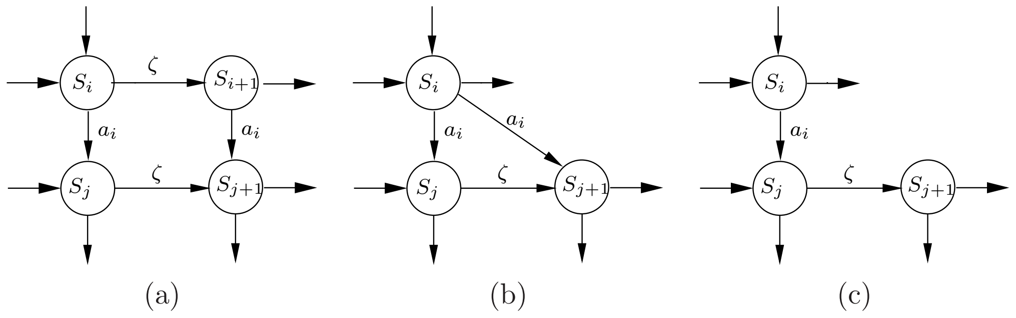


Figure 3.13. Illustration of Case 3 simplification. The observably equivalent paths $(\ldots, s_i, \zeta, s_{i+1}, \ldots)$ are removed. State s_{i+1} is also removed since it has only one incoming state transition.

Lemma 3.3.3 *Let G and G' be SGs, and G' is obtained from G by Case 2 simplification. Then, $G' \equiv G$.*

Proof: For every path $\rho = (\ldots s_i, a_i, s_j, \zeta, s_{j+1}, \ldots)$ in G, there is a corresponding path $\rho' = (\ldots s_i, \zeta, s_{i+1}, a_i, s_{j+1}, \ldots)$ in G' such that $\rho[\mathcal{A}'] \sim \rho'[\mathcal{A}']$ where $\mathcal{A}' = \mathcal{A}^I \cup \mathcal{A}^O$. All the other paths in G are preserved in G'. Therefore, $G' \equiv G$. ∎

3. Case 3.

This reduction deals with state transition (s_i, ζ, s_{i+1}) where there is only one state transition going into s_{i+1}, as shown in Figure 3.13(a). Similar to Case 2 simplification, this case is handled in two steps. In the first step, a new state transition (s_i, a_i, s_{j+1}) is created for each state transition (s_{i+1}, a_i, s_{j+1}) in *outgoing*(s_{i+1}). And then, all state transitions entering and leaving s_{i+1} are removed. The SG after this step is shown in Figure 3.13(b). In the next step, if there exist state transitions (s_i, a_i, s_j) and (s_i, a_i, s_{j+1}) such that there exists another (s_j, ζ, s_{j+1}), then (s_i, a_i, s_{j+1}) is redundant and removed. The final simplified SG is shown in Figure 3.13(c).

Similar to Case 2 simplification, Case 3 simplification can be applied in some cases where the region-based reduction is invalid since the condition of the Definition 3.3.1 does not hold. The following lemma shows that the reduced SG resulting from Case 3 simplification is observably equivalent to the original one.

Lemma 3.3.4 *Let G and G' be two SGs, and G' is obtained from G by Case 3 simplification. Then, $G' \equiv G$.*

Proof: For every path $\rho = (\ldots, s_i, \zeta, s_{i+1}, a_i, s_{j+1}, \ldots)$ in G, there is a corresponding path $\rho' = (\ldots, s_i, a_i, s_j, \zeta, s_{j+1}, \ldots)$ in G' such that $\rho[\mathcal{A}'] \sim \rho'[\mathcal{A}']$ where $\mathcal{A}' = \mathcal{A}^I \cup \mathcal{A}^O$. All the other paths in G are preserved in G'. Therefore, $G' \equiv G$ holds. ∎

3.3.2 Overall Reduction

All the reductions discussed in the previous sections remove observably equivalent paths while not introducing new paths. After applying one reduction, more situations may be exposed for further reduction. Therefore, they are used iteratively until a SG cannot be reduced anymore. This is captured in the algorithm `Reduce` shown in Algorithm 8. In the algorithm `Reduce`, $|R|$ denotes the number of state transitions in R, and the algorithm $Case_iRd(G), i = 1, 2, 3$ implements the simplifications for the Case 1, Case 2 and Case 3, respectively, as described in the previous section. The following theorem shows that using the region-based reduction and case-based simplifications together produces a reduced SG that is observably equivalent to the original one.

Theorem 3.3.1 *Let G be a SG. Then, $G \equiv Reduce(G)$.*

Proof: In Algorithm 8, initially, G_0 is the same as G, thus $G_0 \equiv G$. Then, in each iteration, $G_1 \equiv G_0$ according to Lemma 3.3.1, $G_2 \equiv G_1$ according to Lemma 3.3.2, $G_3 \equiv G_2$ according to Lemma 3.3.3, $G_4 \equiv G_3$ according to Lemma 3.3.4, therefore we can conclude that $G_0 \equiv G_4$ in every iteration. Since $G_0 \equiv G$ in the initial iteration, it is easy to see that $G_4 \equiv G$ in every iteration. ∎

3.4 Comparison between Abstraction and Reduction

The efficient state space reductions are key to success of compositional verification since the state graph for each individual component in a design created with an approximate environment may explode because of lots of extra behavior introduced by the approximate

Algorithm 8: `Reduce` (G)

```
1  $G_0 = G$;
2  repeat
3      $size = |R_0|$;
4      $G_1 = RegionRd(G_0)$;
5      $G_2 = Case1Rd(G_1)$;
6      $G_3 = Case2Rd(G_2)$;
7      $G_4 = Case3Rd(G_3)$;
8      if $|R_4| == size$ then
9          $G_0 = G_4$;
10 until $|R_4| == size$ ;
11 return $G_4$;
```

environment. The large state graphs result in deterioration in the performance of verification, therefore it is necessary to reduce the complexity of state graphs for efficiency. There are two different approaches to reduce SGs: *abstraction* and *reduction*. The abstraction approach intends to reduce state graphs by removing all invisible state transitions. However, the reduced state graphs after applying abstraction often include extra behavior that does not exist in the original ones. In contrast, the reduction approach described in this chapter simplifies state graphs but does not introduce any extra behavior.

A state-based abstraction method is discribed in [99], which is illustrated by an example shown in Figure 3.14. This abstraction removes every invisible state transition $(s_i, \zeta, s_j) \in R$ from an SG, and merges s_i and s_j to form a single state s_{ij} as shown in Figure 3.14(b). All state transitions entering s_i and s_j now enter s_{ij}, and all state transitions leaving s_i or s_j now leave s_{ij}. To preserve failure traces, if s_j is the failure state π, then the merged state s_{ij} is also the failure state. This abstraction can remove all invisible state transitions from an SG. The state-based abstraction is efficient by simply merging an invisible transition to a single state. However, it often introduces a lot of extra behavior including failures. In Figure 3.14(b), there is a path $\rho = (\ldots, a_k, s_{ij}, a_j, \ldots)$ that does not exist in the original SG in Figure 3.14(a). This extra path may cause a false failure in the reduced SG.

Unlike the abstraction approach, the reduction aims at simplifying state graphs but does not introduce any extra behavior. To pinpoint the real reason for the failure caused by

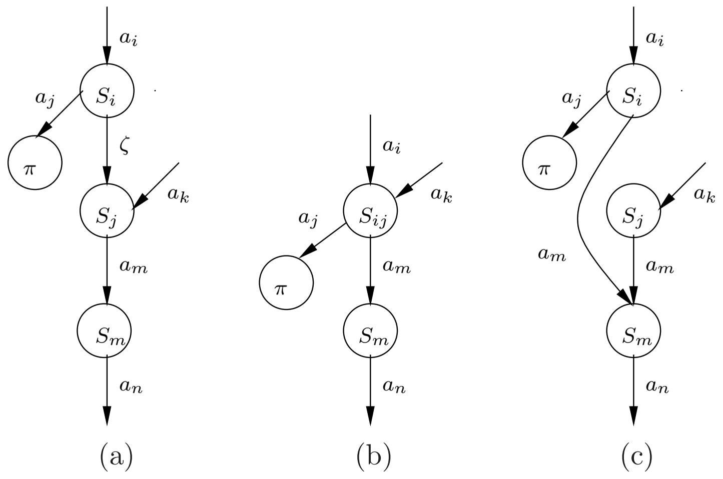


Figure 3.14. Illustration of the state-based abstraction where states connected by an invisible transitions are merged. (a) A partial SG. (b) The SG after applying the state-based abstraction. (c) The SG after applying the transition-based reduction.

an environment, the autofailure reduction shortens the failure paths by removing invisible and output state transitions leading to the failure state. As shown in this chapter, the verification result is not affected by the autofailure reduction. Additionally, the autofailure reduction can be integrated with reachability analysis to further improve efficiency. On the other hand, the autofailure reduction can only remove state transitions that lead to the failure state, therefore there can still be a large number of invisible state transitions left in state graphs after applying the autofailure reduction.

Similar to the state-based abstraction, the transitions-based reduction removes all invisible state transitions aggressively. On the other hand, it preserves all observable behavior of the original SGs by adding new visible state transitions into the reduced SGs whenever necessary. In some cases, the number of visible transitions that need to be added becomes too large, and they make the reduced SGs nondeterministic. The reduced SGs usually include a lot of equivalent behavior and therefore redundant states and transitions that are only necessary to preserve the equivalent behavior. To remove the observably equivalent behavior in the reduced SGs, extra computation is needed to identify and remove equivalent states and the related state transitions. However, the reduction by the equivalent state removal can be modest as the conditions defining the equivalent states are not very general.

Thus, the complexity of the reduced SGs after the transition-based reduction may become higher due to the equivalent behavior that remains. Both the state-based abstraction and the transition-based reduction remove all invisible state transitions from SGs. However, the state-based abstraction aggressively merge the states connected by an invisible state transition, and extra behaviors can often be introduced in the reduced SGs. In contrast, the transition-based reduction adds necessary visible state transitions into the reduced SGs to preserve all observable behavior of the original SGs, even though some of the preserved behavior may already have an observably equivalent ones included in the reduced SGs. The difference between the state-based abstraction and the transition-based reduction is illustrated in Figure 3.14(c) which is the reduced SG by the transition-based reduction applied to the one shown in Figure 3.14(a).

To preserve all observable behavior in a SG similar to the transition-based reduction but to avoid the problem related to it, an alternative technique, the region-based reduction, is developed. This technique removes certain state transitions, visible or invisible, so as to reduce observably equivalent behavior from SGs. Since it only removes state transitions and subsequently unreachable states, the complexity of the reduced SGs consistently decreases. However, to ensure all the observable behavior of the original SGs to be preserved, state transitions that can be removed are limited. To further simplify SGs, the case-based simplification is developed to complement the region-based reduction. This simplification technique is applied to a number of specific cases involving invisible state transitions typically resulting from the region-based reduction.

In summary, the three reduction techniques described in this chapter can significantly simplify the complexity of SGs, and preserve the observable behavior of the original SGs. The reduced SGs often make the process of the abstraction refinement and verification as discussed in Chapter 4 much more efficient.

CHAPTER 4
ABSTRACTION REFINEMENT

Compositional reasoning method is essential to address the state-explosion problem by verifying the individual components without considering the whole system. Effective compositional verification requires finding a simple but accurate over-approximate environment for each component such that each individual component in a system can be checked in isolation. If such approximate environment is not sufficiently accurate, a large number of false counterexamples may be produced by the extra behavior, which do not exist in the real environment. Distinguishing the false counter-examples from the real ones incurs high computational penalty. Traditionally, such environments are obtained by hand, which is very error prone and tedious. This chapter presents two abstraction refinement methods that can refine the state space of each component with an over-approximate environment to be accurate enough for successful verification in a fully automated way.

4.1 Introduction

Given a system $M = M_1 \parallel M_2 \parallel \cdots \parallel M_n$, compositional model checking presented in this dissertation converts the problem of the global verification of M to several subproblems of the local verification of the individual components M_i of the system, and derives the verification result for the entire system from the results of verifying individual components by the following reasoning rule.

$$\begin{array}{rrl} 1: & \mathcal{F}(G'_i) = \emptyset & \text{for } 1 \leq i \leq n \\ 2: & G_i \preceq G'_i & \text{for } 1 \leq i \leq n \\ \hline & \multicolumn{2}{r}{\mathcal{F}(G_1 \parallel \cdots \parallel G_n) = \emptyset} \end{array}$$

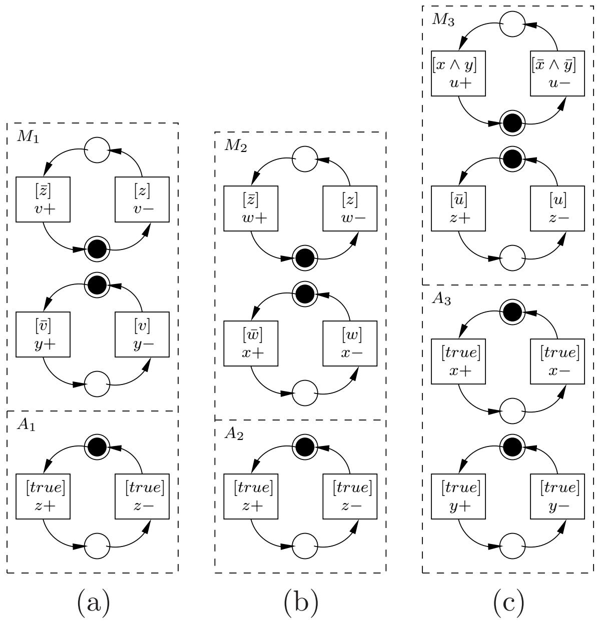


Figure 4.1. (a) - (c) The BGPNs of the components M_1, M_2, and M_3 in Figure 2.1(a) and their completely free environments A_1, A_2, A_3, respectively.

where G_i and G'_i are the SGs generated from $M_i \parallel \mathcal{E}_i$ and $M_i \parallel A_i$ with the exact environment $\mathcal{E}_i$ and an approximate environment A_i for M_i, respectively, using some state space exploration algorithm. This ensures that M_i is failure free in the entire system if it is failure free in A_i. If this is true for every component, the entire system is guaranteed to be failure free.

We illustrate the generation of the state space of a component with an over-approximate environment by the design shown in Figure 2.1(a). Suppose the over-approximate environments for the components are very coarse such that the inputs of the components are set to be totally free as shown in Figure 4.1(a)-(c), respectively. All possible behavior of the concrete environment of the component are included in such coarse over-approximate environment. In addition, some extra behaviors that do not exist in the concrete environment are also included.

The BGPNs of component M_1, M_2, and M_3 in Figure 2.1(a) and their own approximate environments are shown in Figure 4.1(a)-(c), respectively. In an approximate environment where inputs are set to be totally free, meaning they can change to high or low in any

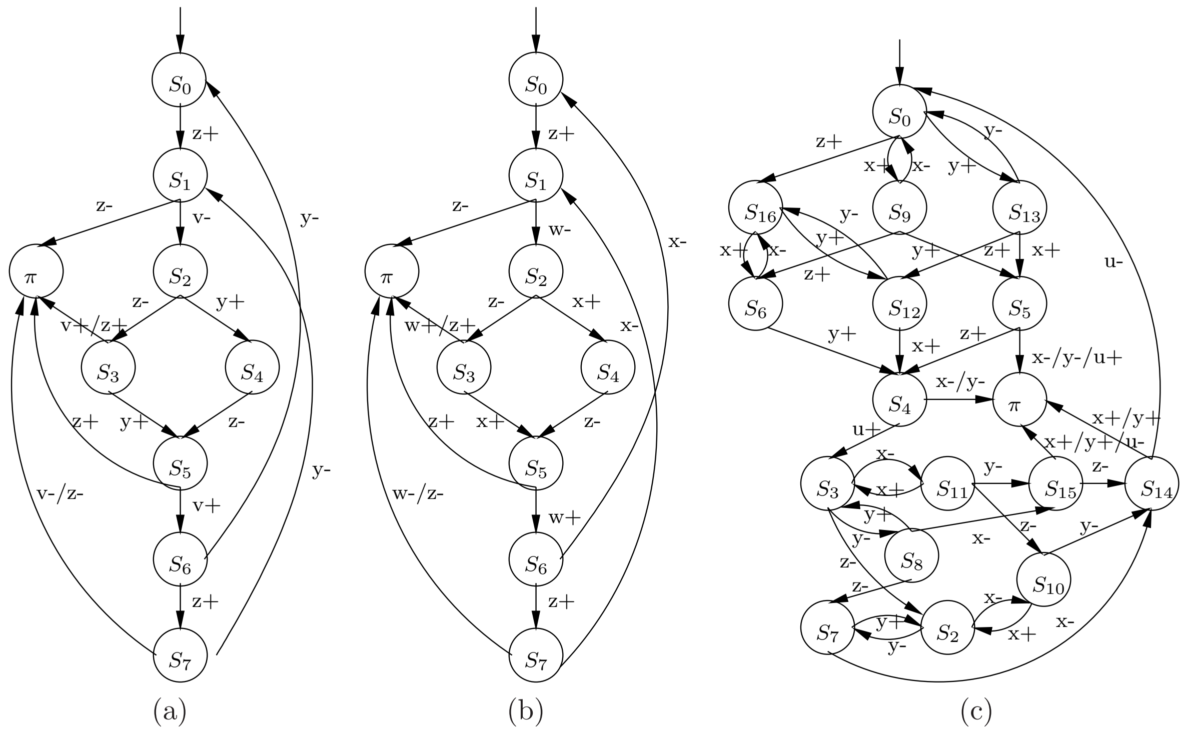


Figure 4.2. (a) - (c) The SGs generated by components with their approximate environments, as shown in Figure 4.1(a)-(c), respectively.

state. Each component is composed with its approximate environment to form a closed system. Figure 4.2(a), (b) and (c) show the SGs generated by Algorithm 1 from such closed system shown in n Figure 4.1(a)-(c) corresponding to the components M_1, M_2, and M_3 , respectively.

The success of compositional reasoning relies on discovery of appropriate environment assumptions for each component. Extra behaviors introduced by the over-approximate environments may cause false counterexamples when verifying components individually. Identification and elimination of these false counterexamples incur high computational penalty. Two abstraction refinement methods, *constraint-based refinement* and *synchronization-based refinement*, are developed to refine the state space of a component with its over-approximate environment to be accurate enough for automated compositional reasoning. The purpose of these two abstraction refinement methods is to remove the extra behavior in the initially coarse approximate environments iteratively in order to generate a simple but

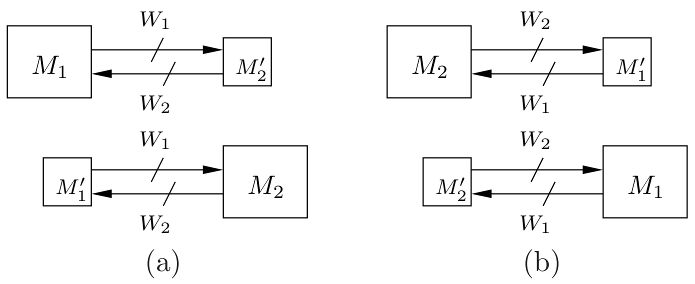


Figure 4.3. Abstraction refinement for compositional verification. Block M_1' and M_2', abstractions of components M_1 and $M2$, are the environments for component M_2 and M_1, respectively.

accurate environment for each component automatically. With these two abstraction refinement methods, large amount of extra behavior can be removed from the initial approximate environments, which facilitates the verification of each component individually. However, not all extra behavior can be eliminated from the approximate environments obtained by applying these two methods.

Compared to the learning-based compositional model checking [32, 76], the abstraction refinement methods presented in this chapter has several significant differences. First, the interface behavior of a component is refined by iteratively examining the interactions between its neighbors and itself, rather than relying on local counter-example analysis. Second, verification based on these methods does not require complex reasoning rules, and circular structures in a design can be handled without difficulty. Third and probably more importantly, there are no assumptions generated in these methods, therefore there is no need for the assumption discharging step. Despite the differences, this method and the learning-based methods can be combined to achieve better results.

4.2 Constraint-based Refinement

4.2.1 Overview

With the approach of constraint-based refinement, interface constraints for each component are extracted and used to refine its state space for better compositional verification in a fully automated way. For two interactive components, the inputs and outputs of one component are the outputs and inputs of another component individually. We illustrate the idea of constraint-based refinement with a system composed of two components, as shown in Figure 4.3. The inputs and outputs of M_1, w_2 and w_1, are the outputs and inputs of M_2, respectively. During verification, the inputs of M_1 and M_2 are driven by some environment abstraction M'_2 and M'_1, instead of M_2 and M_1 themselves. After finding the state space of M_1 and M_2, constraints on their outputs, w_1 and w_2, are derived. Since w_1 and w_2 are the inputs to M_2 and M_1, respectively, these derived constraints are used to restrict the behavior on w_1 and w_2 defined by M'_1 and M'_2, respectively. With respect to a system, the outputs depend on inputs. Therefore, more restricted constraints on the outputs may be derived from these components after their input behavior is constrained. If so, the newly derived output constraints are used in the next iteration to restrict the input behavior of the neighboring components again. This process repeats until the output constraints from both components can no longer be strengthened. Although the idea is illustrated using an example with two components, it naturally applies to systems with an arbitrary number of components.

Combining this method with several state space reduction techniques presented in the previous chapter may help to extract stronger interface constraints, thus enabling the refinement to be more effective. They may also reduce the intermediate state space significantly to allow more flexible system partitioning by lowering the peak space requirement of the largest component in a system. In addition, these reduction techniques do not produce extra behavior compared to traditional abstraction approaches. Therefore, the only source of false failures is the over-approximate environment used for verifying each component. This is highly desirable because it requires less computation to confirm the found failures.

This method is sound as long as an over-approximate environment is found for each component at the beginning of the verification process, and it has less restrictions on system partitioning. Later in this section, this method is proved to be sound by showing that the refined state space of each component is still an abstraction of the exact one. Unfortunately, this method is not complete in that false counter-examples may still exist if the over-approximate environment is not completely refined. However, the chances of finding false counter-examples are significantly reduced when interfaces of the components are refined to be more accurate. Even though false counter-examples may still show up after refinement, it would be much easier for users to refine the derived interface constraints further by hand because a substantial amount of unnecessary information has been removed.

4.2.2 Notations and Definitions

An action a is enabled in a state s if there is a state s' such that $R(s, a, s')$ holds. Recall that states of a SG come from the states of the underlying BGPN. Given a BGPN N, the state s of N is a pair (μ, α) where μ is a marking of the N and α is a state vector of the values of all wires in W of N. A state vector is a Boolean assignment to all wires in W that represents the state of a BGPN on W.

Function $conj(s)$ returns a Boolean conjunction over the wires corresponding to the state vector of state s if it is not the failure state shown as follows.

$$conj(s) = \bigwedge_{w \in W} P_w \text{ where } s \neq \pi \text{ and } P_w \;=\; \begin{cases} w & \text{if } \alpha(w) \equiv 1 \\ \bar{w} & \text{if } \alpha(w) \equiv 0 \end{cases} \qquad (4.1)$$

An action is enabled in s if $conj(s)$ evaluates to true. This definition relates each enabled action with a Boolean formula. Therefore, we can characterize the enabling conditions of actions with Boolean formulas, denoted as *constraints*, which are defined as follows.

Definition 4.2.1 *Let $G = (init, \mathcal{A}, S, R)$ be a SG. Let $f : \{0,1\}^{|W|} \rightarrow \{0,1\}$ be a Boolean function defined over W. A constraint $\mathcal{C} = \{(a, f) \mid a \in \mathcal{A}\}$ of G is a set of pairs of actions of G and their assigned Boolean functions.*

The rest of this chapter uses $\mathcal{C}(a)$ to denote the reference to f corresponding to a such that $(a, f) \in \mathcal{C}$. Additionally, if $\mathcal{C}_1$ and $\mathcal{C}_2$ are defined on the same set of $\mathcal{A}$, $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$ is used to denote $\forall a \in \mathcal{A}, \mathcal{C}_1(a) \Rightarrow \mathcal{C}_2(a)$.

This section assumes that the constraints are defined for all actions of SGs to simplify the presentation. A constraint for actions may be provided by users, or derived automatically as shown in this section. When a constraint is imposed on actions, it may restrict how actions are enabled, therefore causing some state transitions to become invalid.

Definition 4.2.2 *A state transition $(s, a, s') \in R$ such that $s \neq \pi$ is valid with respect to a constraint $\mathcal{C}$ iff $conj(s) \Rightarrow \mathcal{C}(a)$ holds.*

By the above definition, a constraint $\mathcal{C}$ of a SG G on an action a corresponds to a set of valid state transitions defined as follows.

$$R_{\mathcal{C}(a)} = \{(s, a, s') \in R \mid conj(s) \Rightarrow \mathcal{C}(a) \wedge s \neq \pi\}$$

It can be seen that $R_{\mathcal{C}(a)}$ becomes smaller if a stronger constraint $\mathcal{C}$ on a is imposed. Intuitively, a stronger constraint implies that the enabling conditions for actions become more restricted, and more state transitions may not be valid anymore. This observation is reflected in the following property.

$$\forall a \in \mathcal{A}, \Big((\mathcal{C}_1(a) \Rightarrow \mathcal{C}_2(a)) \Leftrightarrow (R_{\mathcal{C}_1(a)} \subseteq R_{\mathcal{C}_2(a)})\Big) \tag{4.2}$$

where $\mathcal{C}_1$ and $\mathcal{C}_2$ are two different constraints. This property states that the behavior in a SG regarding an action a is reduced when a stronger constraint is imposed on a, and vice versa. For example, $R_{\mathcal{C}_2(a)}$ includes all state transitions $(s, a, s') \in R$ in a SG if $\mathcal{C}_2(a) = \mathbf{1}$, and $R_{\mathcal{C}_1(a)} \subseteq R_{\mathcal{C}_2(a)}$ for all other $\mathcal{C}_1(a)$. This example illustrates that $\mathbf{1}$ is the weakest constraint for any action of a SG, and the SG remains the same with such a constraint.

According to the above discussion, a reduced SG results from applying a stronger constraint.

Definition 4.2.3 *Let $G = (init, \mathcal{A}, S, R)$ be a SG, and $\mathcal{C}$ be a constraint on $\mathcal{A}$. Applying $\mathcal{C}$ to G, denoted as $\langle\mathcal{C}\rangle G$, results in a new SG $G' = (init', \mathcal{A}', S', R')$ such that*

1. $init' = init$,

2. $\mathcal{A}' = \mathcal{A}$,

3. $S' = S$,

4. $R' = \bigcup_{\forall a \in \mathcal{A}} \Big(R_{\mathcal{C}(a)} \cup \{(\pi, a, \pi)\} \Big)$.

By the definition of the constraints and abstraction relation, a constraint $\mathcal{C}_1$ is stronger than another constraint $\mathcal{C}_2$ if and only if one SG imposed with $\mathcal{C}_1$ accepts a subset of language of a SG imposed with $\mathcal{C}_2$. This is formulated in the following lemma.

Lemma 4.2.1 *Let $G = (init, \mathcal{A}, S, R)$ be a SG, and $\mathcal{C}_1$ and $\mathcal{C}_2$ two constraints on $\mathcal{A}$. Then, the following property holds.*

$$(\mathcal{C}_1 \Rightarrow \mathcal{C}_2) \quad \Leftrightarrow \quad (\langle \mathcal{C}_1 \rangle G \preceq \langle \mathcal{C}_2 \rangle G)$$

Proof: Let $G_1 = \langle \mathcal{C}_1 \rangle G$, $G_2 = \langle \mathcal{C}_2 \rangle G$. Therefore,

$$R_1 = \bigcup_{\forall a \in \mathcal{A}} \Big(R_{\mathcal{C}_1(a)} \cup \{(\pi, a, \pi)\} \Big)$$

$$R_2 = \bigcup_{\forall a \in \mathcal{A}} \Big(R_{\mathcal{C}_2(a)} \cup \{(\pi, a, \pi)\} \Big)$$

First, according to (4.2), $\forall a \in \mathcal{A}.\ R_{\mathcal{C}_1}(a) \subseteq R_{\mathcal{C}_2}(a)$ holds on account of $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$. Hence, $G_1 \preceq G_2$ holds.

Next, for every path $\rho_1 \in \mathcal{L}(G_1)$, there exists a path $\rho_2 \in \mathcal{L}(G_2)$ such that $\rho_1 \sim \rho_2$. ρ_1 consists of the state transitions from R_1 and ρ_2 from R_2. This implies $R_1 \subseteq R_2$. Thus, for $\forall a \in \mathcal{A}.R_{\mathcal{C}_1(a)} \subseteq R_{\mathcal{C}_2(a)}$, which leads to $\forall a \in \mathcal{A}.\mathcal{C}_1(a) \Rightarrow \mathcal{C}_2(a)$ by (4.2). Hence, $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$ holds. ∎

The following lemma states that the abstraction relation between two SGs is preserved when the same constraint is applied to both of them.

Lemma 4.2.2 *Let G_1 and G_2 be two SGs with the same $\mathcal{A}$, and $\mathcal{C}$ a constraint on $\mathcal{A}$. The following property holds*

$$(G_1 \preceq G_2) \quad \Rightarrow \quad (\langle \mathcal{C} \rangle G_1 \preceq \langle \mathcal{C} \rangle G_2)$$

Proof: Since $G_1 \preceq G_2$, for every path $\rho_1 = (s_0, a_0, s_1, \cdots)$ in $\mathcal{L}(G_1)$, there exists a path $\rho_2 \in \mathcal{L}(G_2)$ such that $\rho_1 \sim \rho_2$. If all state transitions (s_i, a_i, s_{i+1}) for $0 \leq i$ on ρ_1 are valid with respect to $\mathcal{C}$, they are also valid in G_2 with respect to $\mathcal{C}$. In other words, a path that is valid in $\langle \mathcal{C} \rangle G_1$ is also valid in $\langle \mathcal{C} \rangle G_2$. ∎

As seen above, a constraint corresponds to a set of state transitions of a SG. Therefore, the constraint of a given SG can also be extracted. This is defined as follows.

Definition 4.2.4 *Let $G = (init, \mathcal{A}, S, R)$ be a SG. The constraint $\mathcal{C}$ extracted from G, denoted by $G\langle \mathcal{C} \rangle$, satisfies*

$$\forall a \in \mathcal{A}, \quad \left(\mathcal{C}(a) = \bigvee_{R(s,a,s') \wedge s \neq \pi} conj(s) \right)$$

where $\bigvee_{R(s,a,s') \wedge s \neq \pi} conj(s)$ is the disjunction of $conj(s)$ for all state transitions $(s, a, s') \in R$ such that s is not the failure state.

Let G_1 and G_2 be two SGs such that $G_1 \preceq G_2$. According to the definition of the abstraction relation, the behavior of G_1 is more restricted than that of G_2. This implies that the enabling condition of an action is more restricted in G_1 than in G_2. This indicates that a stronger constraint may be derived from the more refined SG.

Lemma 4.2.3 *Let G_1 and G_2 be two SGs, and $\mathcal{C}_1$ and $\mathcal{C}_2$ two constraints derived by $G_1\langle \mathcal{C}_1 \rangle$ and $G_2\langle \mathcal{C}_2 \rangle$, respectively. Then the following property holds.*

$$(G_1 \preceq G_2) \quad \Rightarrow \quad (\mathcal{C}_1 \Rightarrow \mathcal{C}_2)$$

Proof: Since $G_1 \preceq G_2$, for every path $\rho_1 \in \mathcal{L}(G_1)$, there exists a path $\rho_2 \in \mathcal{L}(G_2)$ such that $\rho_1 \sim \rho_2$. Therefore, for every $a \in \mathcal{A}$, if it is enabled on path ρ_1, it is also enabled on path ρ_2. It is possible that G_2 may have some path that does not exist in G_1. This implies that an action may be enabled on some path in G_2 but not enabled in G_1. To summarize, any action, if enabled in G_1, is also enabled in G_2, but this is not true in the other direction. This is equivalent to $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$. ∎

4.2.3 Abstraction Refinement with Constraints

The previous section shows how constraints help refine SGs by removing invalid state transitions. However, manually generating such constraints may be too expensive. This section describes an abstraction refinement algorithm that makes the SGs obtained from the system components with approximate environment more accurate. This algorithm is fully automated, and iteratively generates more accurate yet conservative interface constraints to refine the SGs as long as the initially selected environments for the components are abstractions of the exact ones.

To simplify the discussion, consider a system of two components, $G = G_1 \parallel G_2$, such that $\mathcal{A}_1^I = \mathcal{A}_2^O$ and $\mathcal{A}_1^O = \mathcal{A}_2^I$. In the sequel, the input and output constraints refer to those on input and output actions of a component, respectively. A shared action between G_1 and G_2 is in $\mathcal{A}_1 \cap \mathcal{A}_2$. If a shared action a is in $\mathcal{A}_1^O \cap \mathcal{A}_2^I$, then the output constraints on a derived from G_1 can be used as input constraints to reduce G_2 by pruning the invalid state transitions on a. The case where a is in $\mathcal{A}_2^O \cap \mathcal{A}_1^I$ is handled similarly.

The essence of the abstraction refinement lies in the alternating refinement on G_1 and G_2 with the interface constraints. When refining a SG, the output constraints derived from other SGs are applied to the considered SG where the invalid state transitions on input actions are removed. The output constraints are extracted from the reduced SGs, and then served as input constraints for other SGs in the next iteration. Let $\mathcal{C}_1^i$ and $\mathcal{C}_2^i$ be the output constraints extracted from G_1^i and G_2^i in the ith iteration, respectively. The iterative process of the abstraction refinement is illustrated as follows.

$$\begin{array}{lll} \text{iteration } 0: & \langle \mathcal{C}_2^0 \rangle G_1^0 \langle \mathcal{C}_1^1 \rangle, & \langle \mathcal{C}_1^0 \rangle G_2^0 \langle \mathcal{C}_2^1 \rangle \\ \text{iteration } 1: & \langle \mathcal{C}_2^1 \rangle G_1^1 \langle \mathcal{C}_1^2 \rangle, & \langle \mathcal{C}_1^1 \rangle G_2^1 \langle \mathcal{C}_2^2 \rangle \\ & \cdots & \\ \text{iteration } l: & \langle \mathcal{C}_2^l \rangle G_1^l \langle \mathcal{C}_1^{l+1} \rangle, & \langle \mathcal{C}_1^l \rangle G_2^l \langle \mathcal{C}_2^{l+1} \rangle \end{array}$$

where $\langle \mathcal{C}_2^i \rangle G_1^i \langle \mathcal{C}_1^{i+1} \rangle$ specifies that input constraint $\mathcal{C}_2^i$ is applied to G_1^i, and output constraints $\mathcal{C}_1^{i+1}$ are derived from $\langle \mathcal{C}_2^i \rangle G_1^i$. Let $G_1^{i+1} = \langle \mathcal{C}_2^i \rangle G_1^i$ and $\mathcal{C}_1^0 = \{(a, \mathbf{1}) \mid \forall a,\ a \in \mathcal{A}_1\}$ to denote the initial constraints for an SG. Since $G_1^i = \langle \mathcal{C}_1^0 \rangle G_1^i$ and $\mathcal{C}_2^i \Rightarrow \mathcal{C}_1^0$, we have $G_1^{i+1} \preceq G_i^i$ by Lemma 4.2.1. The enabling condition of the output actions of G_1^{i+1} may become more restricted after applying $\mathcal{C}_2^i$, therefore $\mathcal{C}_1^{i+1} \Rightarrow \mathcal{C}_1^i$ by Lemma 4.2.3. The stronger constraint $\mathcal{C}_1^{i+1}$ extracted from the reduced G_1^{i+1} is used as the input constraint for G_2^{i+1}. The same reasoning applies to G_2^i. The above process terminates in the lth iteration when the extracted output constraints of all components are stable, e.g. $\mathcal{C}_1^l = \mathcal{C}_1^{l+1}$ and $\mathcal{C}_2^l = \mathcal{C}_2^{l+1}$. This implies that G_1^l and G_2^l cannot be reduced anymore.

Theorem 4.2.1 below proves the soundness of the interface refinement process. It shows that this compositional verification method combined with the described refinement process is sound in that the refined SGs are still abstractions of the exact SGs after refinement.

To prove Theorem 4.2.1, the exact SGs need to be defined. Intuitively, the SG of a component is exact if its behavior is exactly the same when it is embedded in a larger system. The formal definition of exact SGs is shown as follows.

Definition 4.2.5 *Let G_1 and G_2 be two SGs. G_1 is exact within $G = G_1 \parallel G_2$ if the following condition holds.*

$$\forall (s_1, a, s_1') \in R_1, \exists s_2, s_2' \in S_2, ((s_1, s_2), a, (s_1', s_2')) \in R.$$

From Definition 4.2.5, the following property holds for a SG $G = G_1 \parallel G_2 \parallel \cdots \parallel G_n$.

$$G[\mathcal{A}_i] = \{G_0, \ldots, G_n\}[\mathcal{A}_i] \preceq \{G_i, G_j\}[\mathcal{A}_i] \tag{4.3}$$

Theorem 4.2.1 *Let G_1 and G_2 be exact within $G_1 \parallel G_2$. If G_1', and G_2' are SGs such that*

$$G_1 \preceq G_1' \text{ and } G_2 \preceq G_2'$$

the following property holds.

$$G_1 \preceq \langle \mathcal{C}_2' \rangle G_1' \text{ and } G_2 \preceq \langle \mathcal{C}_1' \rangle G_2'$$

where $\mathcal{C}_i'$ is obtained by $G_i'\langle \mathcal{C}_i' \rangle$ for $i = 1, 2$.

Proof: According to Lemma 4.2.2,

$$\langle \mathcal{C}_2' \rangle G_1 \preceq \langle \mathcal{C}_2' \rangle G_1' \text{ and } \langle \mathcal{C}_1' \rangle G_2 \preceq \langle \mathcal{C}_1' \rangle G_2'$$

Let $\mathcal{C}_i$ be the constraints obtained by $G_i\langle \mathcal{C}_i \rangle$ for $i = 1, 2$. According to Lemma 4.2.3, $\forall a \in \mathcal{A}_i^O,\ \mathcal{C}_i \Rightarrow \mathcal{C}_i'$ for $i = 1, 2$. Again, according to Lemma 4.2.1,

$$\langle \mathcal{C}_2 \rangle G_1 \preceq \langle \mathcal{C}_2' \rangle G_1 \text{ and } \langle \mathcal{C}_1 \rangle G_2 \preceq \langle \mathcal{C}_1' \rangle G_2$$

Combining the results in the above steps, we have

$$\langle \mathcal{C}_2 \rangle G_1 \preceq \langle \mathcal{C}_2' \rangle G_1' \text{ and } \langle \mathcal{C}_1 \rangle G_2 \preceq \langle \mathcal{C}_1' \rangle G_2'$$

According to (4.2.5), $\langle \mathcal{C}_2 \rangle G_1 = G_1$ and $\langle \mathcal{C}_1 \rangle G_2 = G_2$. Therefore, $G_1 \preceq \langle \mathcal{C}_2' \rangle G_1'$ and $G_2 \preceq \langle \mathcal{C}_1' \rangle G_2'$. This completes the proof. ∎

4.2.4 Algorithms

Algorithm 9 illustrates the `refine` algorithm which implements the abstraction refinement process presented above. It takes as arguments a set of SGs G_i, each of which is generated from a component in a system with an over-approximate environment, and a set of initial constraints $\mathcal{C}_i$ on the outputs of each component. The algorithm first merges these

Algorithm 9: `refine`($\{G_1, \cdots, G_n\}, \{\mathcal{C}_1, \cdots, \mathcal{C}_n\}$)

```
1 $\mathcal{C}' = \mathcal{C}_1 \cup \ldots \cup \mathcal{C}_n$;
2 $\mathcal{C} = \emptyset$;
3 while $\mathcal{C} \neq \mathcal{C}'$ do
4     $\mathcal{C} = \mathcal{C}'$;
5     $\mathcal{C}' = \emptyset$;
6     foreach $G_i,\ 0 \leq i \leq n$ do
7         apply($G_i, \mathcal{C}$);
8         $\mathcal{C}_i$ = extract($G_i$);
9         $\mathcal{C}' = \mathcal{C}' \cup \mathcal{C}_i$;
```

constraints into a single set, and then iteratively applies the constraint set to reduce each SG and extracts new output constraints from the reduced SGs until the constraint set does not change anymore. At this point, all state transitions in every SG are valid with respect to the constraints extracted from their neighbors, therefore no further reduction is possible. The initial constraints may be provided by users or obtained from high level representations. These constraints may be very abstract at the beginning, and may possibly be set to **1** for all actions by default if nothing is known about the input interface of a component. However, more restricted initial constraints help reduce the number of iterations. Algorithms `apply` and `extract` follow the Definitions 4.2.3 and 4.2.4, and are described in more detail later in this section.

Next, the complexity of the above algorithm in terms of the number of iterations needed to find the stable constraints is considered. Assume that the size of a SG G_i, $|G_i|$, is measured by the number of state transitions in R_i of G_i. Suppose the number of components in a system is n and $|G_i| \leq m$ for all $1 \leq i \leq n$. In theory, the number of iterations needed to find the stable constraints is $O(mn)$. This complexity can be understood as follows. Consider the extreme case where exactly one state transition of exactly one SG is removed in each iteration. And suppose that all state transitions in G_i can be removed. Obviously, the process stops when the state transition set R_i of every G_i is reduced to be empty. Therefore, the maximal number of iterations necessary for termination is $O(mn)$. Although this complexity seems very high, in practice the total number of iterations is not that large

Algorithm 10: `apply`$(G_i, \mathcal{C})$

```
1 foreach $(s, a, s') \in R_i \wedge s \neq \pi \wedge a \in \mathcal{A}_i^I$ do
2     if $conj(s) \Rightarrow \mathcal{C}(a)$ does not hold then
3         Delete $(s, a, s')$ from $R_i$;
4 Remove unreachable states and transitions from $G_i$;
```

because many state transitions can be eliminated from multiple components in a single iteration as shown by the experimental results.

In the above discussion, the application of a constraint to and extraction of a constraint from a SG are represented as $\langle\mathcal{C}\rangle G\langle\mathcal{C}'\rangle$. Next, the algorithms are shown on how to reduce SGs by applying a constraint on a SG, i.e. $\langle\mathcal{C}\rangle G$, and extract a constraint from a SG, i.e. $G\langle\mathcal{C}'\rangle$.

Given a SG G and a constraint $\mathcal{C}$, the objective is to apply $\mathcal{C}$ on G to remove the invalid state transitions in G. A state transition $(s, a, s') \in R$ of G such that $s \neq \pi$ is invalid if $conj(s) \not\Rightarrow \mathcal{C}(a)$. The removal of the state transitions may render some states unreachable in G when all of their incoming state transitions are eliminated. In the last step, all unreachable states and their outgoing state transitions are also removed. Algorithm 10 shows the procedure to reduce G with $\mathcal{C}$.

The constraint $\mathcal{C}$ is applied only on the input actions as shown in Algorithm 10. In general, the constraint provided to algorithm `apply` can be on either input or output actions. For example, when one describes a system, a constraint may be used to elaborate the system description additionally. This constraint can be created for any actions. However, when algorithm `apply` is used for a SG in the above abstraction refinement framework, only the part of the constraint extracted from other SGs for the input actions of the SG under consideration is necessary. The part of the constraint for the output actions of this SG would not reduce this SG because it is extracted from itself. Therefore, only the state transitions labeled with input actions of a SG may be removed with respect to constraint $\mathcal{C}$ when algorithm `apply` is invoked. As a side effect, some other state transitions, when become unreachable due to the removed state transitions on input actions, are also removed.

Algorithm 11: $\texttt{extract}(G_i)$

```
1 foreach $a \in \mathcal{A}_i^O$ do
2     $f = 0$ ;
3     Add $(a, f)$ into $\mathcal{C}_i$;
4 foreach $(s, a, s') \in R_i$ and $s \neq \pi$ and $a \in \mathcal{A}_i^O$ do
5     Project $conj(s_i)$ on $I \cup O$ ;
6     Replace $(a, f) \in \mathcal{C}_i$ with $(a, f \vee conj(s))$ ;
7 return $\mathcal{C}_i$;
```

Algorithm 11 illustrates the extraction of constraints. Each component updates its behavior on its output actions, while its input actions are defined by the environment. Therefore, given a SG of a component, only the constraint for non-input actions are extracted. Since the behavior on invisible action ζ of a SG is invisible to other SGs, the constraint for the invisible actions is meaningless to other components. It is unnecessary to generate the constraints on internal actions. Furthermore, the constraints need to be projected to the set of visible wires $I \cup O$ to hide invisible details (line 5). The constraints for the output actions are extracted by Definition 4.2.1 and equation 4.1 (line 6).

4.2.5 Example

For the components M_1, M_2 and M_3 of the circuit shown in Figure 2.1(a), their SGs after applying the autofailure reduction, which is described in the previous section, are shown in Figure 4.4(a)-(c), respectively. The process of invalid state transitions being removed from SGs by the constraint-based refinement method is illustrated as follows.

First, the constraints for the output actions of the SG G_1 shown in Figure 4.4(a) are extracted. For a state $s_i \in S_1$ of G_1, where $s_i \neq \pi$, $conj(s_i)$ having being projected on the visible wires is listed as follows.

$$conj(s_0) = \neg z \wedge \neg y, conj(s_1) = z \wedge y, conj(s_2) = z \wedge \neg y$$
$$conj(s_4) = z \wedge y, conj(s_5) = \neg z \wedge y, conj(s_6) = \neg z \wedge y$$

The output of G_1 is y, which is the input of G_3. The constraints for actions on y are constructed by disjoining $conj(s)$ for all $(s, a, s') \in R$ where action a corresponding to wire y

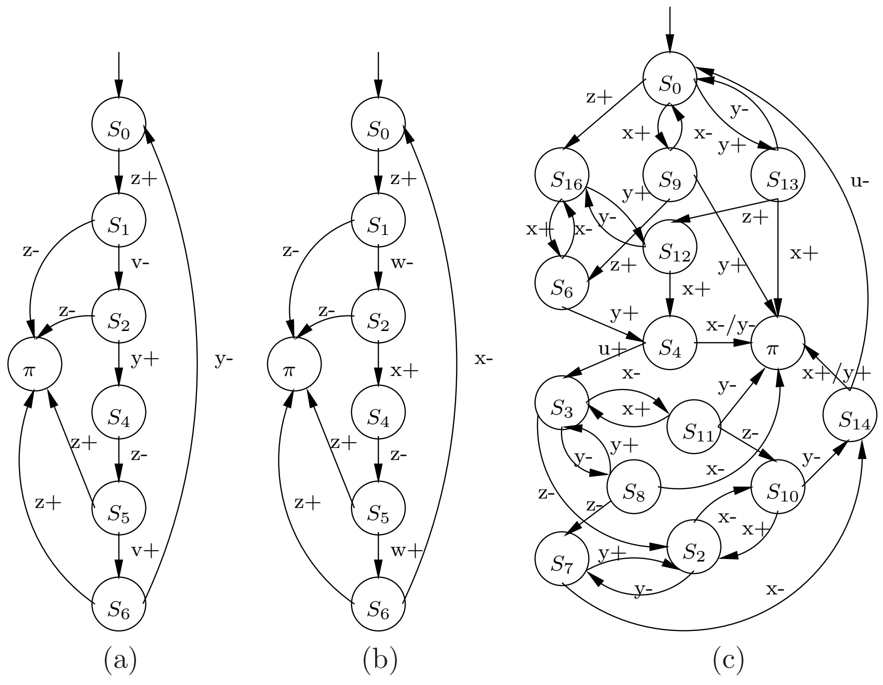


Figure 4.4. (a) - (c) The SGs for component M_1, M_2, and M_3 after applying the autofailure reduction as shown in Figure 3.2(a) - (c), respectively.

and $s \neq \pi$. Since action $y+$ is enabled in state s_2, the constraint for $y+$ is $z \wedge \neg y$. Similarly, the constraint for $y-$ is $\neg z \wedge y$. The output constraints C_1^0 are shown as follows.

$$\mathcal{C}_1^0(y+) = conj(s_2) = z \wedge \neg y$$
$$\mathcal{C}_1^0(y-) = conj(s_6) = \neg z \wedge y$$

Second, the constraints for the output actions of the SG G_2 shown in Figure 4.4(b) are extracted. For a state $s_i \in S_2$ of G_2, where $s_i \neq \pi$, $conj(s_i)$ having being projected on the visible wires is listed as follows.

$$s_0 : \neg z \wedge \neg x, s_1 : z \wedge x, s_2 : z \wedge \neg x$$
$$s_4 : z \wedge x, s_5 : \neg z \wedge x, s_6 : \neg z \wedge x$$

The output of G_2 is x, which is the input of G_3. Similarly, the output constraints C_2^0 are shown as follows.

$$\mathcal{C}_2^0(x+) = conj(s_2) = z \wedge \neg x$$
$$\mathcal{C}_2^0(x-) = conj(s_6) = \neg z \wedge$$

Third, the constraints for the input actions of the SG G_3 shown in Figure 4.4(c) are applied to reduce G_3. For a state $s_i \in S_3$ of G_3, where $s_i \neq \pi$, $conj(s_i)$ having being projected on the visible wires is listed as follows.

$$s_0 : \neg x \wedge \neg y \wedge \neg z, s_2 : x \wedge y \wedge \neg z, s_3 : x \wedge y \wedge z$$

$$s_4 : x \wedge y \wedge z, s_6 : x \wedge \neg y \wedge z, s_7 : x \wedge \neg y \wedge \neg z$$

$$s_8 : x \wedge \neg y \wedge z, s_9 : x \wedge \neg y \wedge \neg z, s_{10} : \neg x \wedge y \wedge \neg z$$

$$s_{11} : \neg x \wedge y \wedge z, s_{12} : \neg x \wedge y \wedge z, s_{13} : \neg x \wedge y \wedge \neg z$$

$$s_{14} : \neg x \wedge \neg y \wedge \neg z, s_{16} : \neg x \wedge \neg y \wedge z$$

Since the output y of G_1 and the output x of G_2 are the inputs of G_3, the output constraints generated from G_1 and G_2 are the input constraints for G_3 being used to reduce G_3. We apply the input constraints $C_1^0(y+), C_1^0(y-), C_2^0(x+), C_2^0(x-)$ to G_3 to remove invalid state transitions, the state vector of starting states of which violates the input constraints. The process of applying input constraints to G_3 in Figure 4.4(c) is shown in the following.

1. Applying $C_1^0(y+) = z \wedge \neg y$. The constraint is invalid at state s_0, s_7, s_9, s_{14} where input action $y+$ is enabled. Thus, the following state transitions are removed from G_3.

$$(s_0, y + s_{13}), (s_7, y+, s_2), (s_9, y+, s_6), (s_{14}, y+, \pi).$$

2. Applying $C_1^0(y-) = \neg z \wedge y$. The constraint is invalid at state s_3, s_4, s_{11}, s_{12} where input action $y-$ is enabled. Thus, the following state transitions are removed from G_3.

$$(s_3, y-, s_8), (s_4, y-, \pi), (s_{11}, y-, \pi), (s_{12}, y-, \pi).$$

3. Applying $C_2^0(x+) = z \wedge \neg x$. The constraint is invalid at state $s_0, s_{10}, s_{13}, s_{14}$ where input action $x+$ is enabled. Thus, the following state transitions are removed from G_3.

$$(s_0, x+, s_9), (s_{10}, x+, s_2), (s_{13}, x+, \pi), (s_{14}, x+, \pi).$$

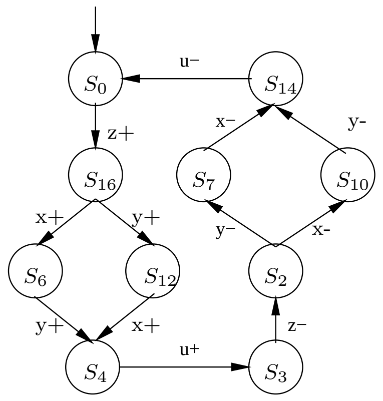


Figure 4.5. The reduced G_3^1 after applying constraints derived from G_1 and G_2 shown in Figure 4.4 (a) and (b) on G_3 in Figure 4.4(c).

4. Applying $C_2^0(x-) = \neg z \wedge w \wedge x$. The constraint is invalid at state s_3, s_4, s_6, s_8 where input action $x-$ is enabled. Thus, the following state transitions are removed from G_3.

$$(s_3, x-, s_{11}), (s_4, x-, \pi), (s_6, x-, s_{16}), (s_8, x-, \pi).$$

States s_8, s_9, s_{10} and s_{11} becomes unreachable after removing invalid state transitions. After removing the unreachable states and their outgoing state transitions, the reduced SG G_3^1 is shown in the Figure 4.5.

Forth, the constraints for the output actions of the SG C_3^1 shown in Figure 4.5 are extracted as follows.

$$\mathcal{C}_3^1(z+) = conj(s_0) = \neg z \wedge \neg x \wedge \neg y$$
$$\mathcal{C}_3^1(z-) = conj(s_3) = z \wedge x \wedge y$$

The output of G_3 is z, which is the input of G_1 and G_2.

Fifth, the constraints for the input actions of the SGs G_1 and in G_2 shown in Figure 4.4(a) and (b), respectively, are applied. The output constraints $\mathcal{C}_3^1$ are the input constraint for the SGs G_1 and G_2, which become more restricted due to the generation from the reduced SG G_3^1. After applying the stronger input constraints to G_1 and G_2, respectively, more extra behavior from G_1 and G_2 are removed as follows.

1. Applying $C_3^1(z+) = \neg z \wedge \neg x \wedge \neg y$ on G_1. The constraint is invalid at state s_5, s_6 where input action $z+$ is enabled. Thus, the following state transitions are removed from G_1.
$$(s_5, z+, \pi), (s_6, z+, \pi).$$

2. Applying $C_3^1(z-) = z \wedge x \wedge y$ on G_1. The constraint is invalid at state s_1, s_2 where input action $z+$ is enabled. Thus, the following state transitions are removed from G_1.
$$(s_1, z-, \pi), (s_2, z-, \pi).$$

3. Applying $C_3^1(z+) = \neg z \wedge \neg x \wedge \neg y$ on G_2. The constraint is invalid at state s_5, s_6 where input action $z+$ is enabled. Thus, the following state transitions are removed from G_2.
$$(s_5, z+, \pi), (s_6, z+, \pi).$$

4. Applying $C_3^1(z-) = z \wedge x \wedge y$ on G_2. The constraint is invalid at state s_1, s_2 where input action $z+$ is enabled. Thus, the following state transitions are removed from G_2.
$$(s_1, z-, \pi), (s_2, z-, \pi).$$

The reduced SGs G_1^1 and G_2^1 are shown in the Figure 4.6(a) and (b), respectively.

Sixth, in the second iteration of constraints generation, the output constraints $\mathcal{C}_1^1$ of G_1^1 and $\mathcal{C}_2^1$ of G_2^1 are extracted as follows.

$$\mathcal{C}_1^1(y+) = conj(s_2) = z \wedge \neg y$$
$$\mathcal{C}_1^1(y-) = conj(s_6) = \neg z \wedge y$$
$$\mathcal{C}_2^1(x+) = conj(s_2) = z \wedge \neg x$$
$$\mathcal{C}_2^1(x-) = conj(s_6) = \neg z \wedge x$$

When applying the output constraints $\mathcal{C}_1^1$ and $\mathcal{C}_2^1$ to G_3^1 shown in the Figure 4.6(c), no more state transitions can be removed from G_3^1. This implies that G_3^1 is stable such

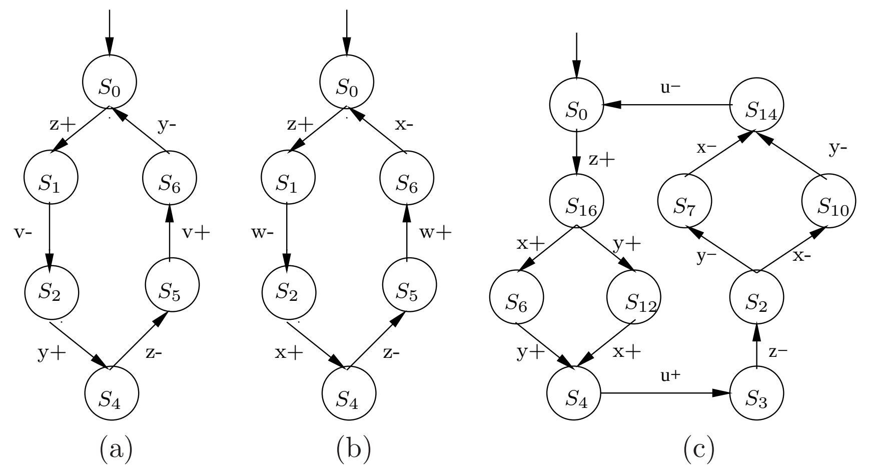


Figure 4.6. SGs of G_1^1, G_2^1 and G_3^1 in (a), (b), (c) are the results of applying the constraint-based refinement on the SGs G_1, G_2 and G_3 in Figure 4.4(a), (b), (c), respectively.

that no more restricted output constraints can be extracted from G_3^1. Accordingly, G_1^1 and G_2^1 are in stable status too. Thus, the process of the constraint-based refinement terminates. The final refinement results of G_1, G_2 and G_3 are shown in the Figure 4.6(a), (b) and (c), respectively. The fact that these SGs are failure free implies that all failure paths introduced by the approximate environments are eliminated by the constraint-based refinement eventually. The verification of the components can be applied on the reduced SGs G_1^1, G_2^1 and G_3^1, respectively.

4.3 Synchronization-based Refinement

4.3.1 Motivation

With the constraint-based abstraction refinement, extra behaviors introduced by the approximate environments for components can be removed by iteratively extracting output constraints from components and applying to their interactive neighboring components. Since stronger output constraints can be generated from reduced SGs, the SGs can be reduced increasingly by eliminating extra behaviors until all SGs reach the stable status. In this method, constraints extraction and application are employed on individual components. Thus the computation complexity of the constraints extraction and application

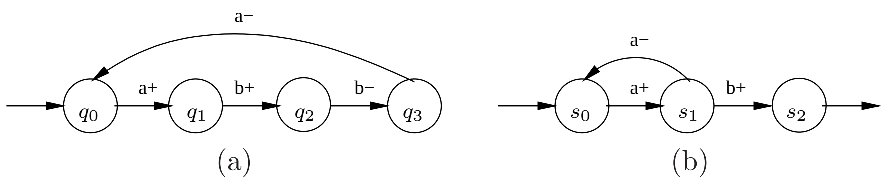


Figure 4.7. (a) and (b) are two partial SGs G_1 and G_2 of components M_1 and M_2 generated with some approximate environments, respectively.

is $O(m)$ given m being the number of state transitions of a component. However, the constraint-based interface refinement can not capture the sequencing information of extra input behaviors. Therefore, the constraint-based refinement is unable to remove all additional state transitions.

For example, given two interactive components M_1 and M_2, where an input and output action of M_1 is b and a, respectively, and M_2 is a and b. The partial SGs of M_1 and M_2 with their approximate environments are shown in the Figure 4.7(a) and (b), respectively. Suppose $(s_0, a+, s_1, a-, s_2)$ in G_2 is an extra path introduced by the approximate environment of M_2. We generate the output constraints of G_1 and apply them to G_2 in order to remove this extra path from G_2. The following shows $conj(s)$ for each non-failure state of the G_1 and G_2 in Figure 4.7 (a) and (b), respectively.

$$G_1 : q_0 : \neg a \wedge \neg b, q_1 : a \wedge \neg b, q_2 : a \wedge b, q_3 : a \wedge \neg b$$
$$G_2 : s_0 : \neg a \wedge \neg b, s_1 : a \wedge \neg b, s_2 : a \wedge b$$

The output constraints of G_1 are $\mathcal{C}(a-) = conj(q_3) = a \wedge \neg b$. When applying $\mathcal{C}(a-)$ to G_2, we have $conj(s_1) \Rightarrow conj(a-)$. The extra state transition $(s_1, a-, s_2)$ can not be removed from G_2, since $conj(a-)$ does not capture the fact that actions $b+$ and $b-$ are executed between $a+$ and $a-$.

We proposed another abstraction refinement method, synchronization-based refinement, to capture the execution sequences of the actions. With synchronization-based refinement, the SGs of components obtained with over-approximate environments are refined iteratively such that behaviors not allowed on the interface among components are removed. This abstraction refinement method is based on the observation that the interface behaviors are allowed if they are synchronized among the components connected by that interface. Consider a system $M = M_0 \parallel \ldots \parallel M_n$ whose behavior is captured in a SG $G = G_1 \parallel \ldots \parallel G_n$. When composing two SGs $G_i \parallel G_j$ of components M_i and M_j, only the synchronized behaviors in G_i and G_j appear in the composition, while the unsynchronized ones are removed. This method borrows the idea of parallel composition and refines G_i and G_j by checking their synchronized interface behaviors with respect to those allowed by $G_i \parallel G_j$. The key to this method is to identify and remove the unsynchronized behaviors in each component without actually constructing the composition.

Traditional parallel composition is defined on two components, therefore synchronization-based refinement can only refine two components at a time. This is not sufficient in practice since a component may interact with multiple neighbors, which themselves may have interactions with each other. The interface interactions among the neighbors may have an effect on how the component interacts with its neighbors. If these interactions are not considered, extra behaviors in the component may not be removed causing a large number of false counter-examples. Therefore, the above refinement approach is enhanced by extending synchronization to more than two components at a time to take inter-dependencies among components into account for stronger refinement.

Ideally, the best result can be obtained if synchronization is applied to the whole SG G of all components in one step. However, the complexity of this approach may be as high as that of $G_0 \parallel \ldots \parallel G_n$. Therefore, it can only be applied locally to a subset of components at a time where they share some common interfaces. In each iteration of the refinement process, components in each subset are refined together. It is possible that components M_i or M_j exists in different subsets, and they can be further refined with other components after they are refined with each other. Therefore, the above step needs to be repeated to achieve stronger refinement results. This method guarantees to terminate when the violating behaviors in each component are eliminated by refinement, or the components cannot be refined further.

4.3.2 Local Synchronization Detection

In a concurrent system, communicating components are synchronized on shared actions. The parallel composition $G = G_1 \parallel G_2$ captures concurrent executions of G_1 and G_2 with synchronization on the alphabet $\mathcal{A}_1 \cap \mathcal{A}_2$. Unsynchronized behavior, which G_2 does not allow G_1 to take place, or vice versa, can be eliminated from the composite SG G. Using the notion of languages of state graphs, the parallel composition $\parallel$ satisfies

$$\mathcal{L}(G)[\mathcal{A}_1] \subseteq \mathcal{L}(G_1) \text{ and } \mathcal{L}(G)[\mathcal{A}_2] \subseteq \mathcal{L}(G_2)$$

The illegal paths such as $\mathcal{L}(G_1) - \mathcal{L}(G)[\mathcal{A}_1]$ and $\mathcal{L}(G_2) - \mathcal{L}(G)[\mathcal{A}_2]$ in G_1 and G_2, respectively, are removed during parallel composition. In this method, the state transitions that appear on paths in $\mathcal{L}(G)[\mathcal{A}_i]$ where $i = 1, 2$ are referred to as *synchronized* transitions.

Definition 4.3.1 *Given two SGs G_1 and G_2, a state transition (s_i, a, s'_i) in M_i where $i = 1, 2$ is synchronized if there exists a corresponding transition $((s_1, s_2), a, (s'_1, s'_2))$ in $G_1 \parallel G_2$. Otherwise, the transition is not synchronized.*

According to the above definition and parallel composition, a transition on an invisible action is automatically synchronized. A transition on a visible action is synchronized if there is a corresponding transition in another component on the same visible action.

The goal of refinement is to remove from G_i the unsynchronized transitions. Based on the above discussion, the synchronized transitions can be identified during parallel composition. Given G_1, G_2 and $G = G_1 \parallel G_2$, the idea of local synchronization detection is to refine G_1 and G_2 by removing the unsynchronized transitions identified during parallel composition. The above procedure is denoted by $sync(G_1, G_2)$, which returns two SGs G'_1 and G'_2 such that the following conditions hold for G'_1.

1. $S'_1 = \{s_1 \in S_1 \mid (s_1, s_2) \in S\}$

2. $R'_1 = \{(s_1, a, s'_1) \in R_1 \mid ((s_1, s_2), a, (s'_2, s'_2)) \in R\}$

The similar condition holds for G'_2 too. In the above definition, S and R are the set of states and state transitions of $G_1 \parallel G_2$. It is clear that $G'_i \preceq G_i$ since G'_i includes states reachable only in the composition and the unsynchronized state transitions in G_i are removed from the resultant SG G'_i, i.e. $\mathcal{L}(G'_i) \subseteq \mathcal{L}(G_i), i = 1, 2$. This indicates that function *sync*() can be viewed as a monotonic function in that applying it each time makes the SGs more restricted.

The concept of parallel composition is not new. However, using it as an approach to refinement is novel in this work. Although the composite SG is used in the above description to make presentation clear, that full composite SG is never generated during synchronization to save time and memory. This point becomes clear in Section 4.3.5.

4.3.3 Abstraction Refinement with Synchronization

In the following discussion, given a system composed of n components $M = M_0 \parallel \ldots \parallel M_n$, the parallel composition $G = G_0 \parallel \ldots \parallel G_n$ is also denoted as $G = \{G_0, \ldots, G_n\}$. When applying *sync*() to every pair of components $\{G_i, G_j\} \subseteq G$ and $\mathcal{A}_{ij} = \mathcal{A}_i \cap \mathcal{A}_j \neq \emptyset$, it reduces G_i and G_j with respect to $\mathcal{A}_{ij}$. Additionally, this reduction can probably render some other transitions in G_i on $\mathcal{A}_i - \mathcal{A}_{ij}$ to become unreachable, causing further reduction on behaviors on the interfaces of G_i with components other than G_j. The same argument also applies to G_j. Since each component may share interfaces with multiple other components, to maximally refine each component, *sync*() needs to be applied iteratively until all components cannot be reduced further. The above discussion is formulated as shown in the following framework.

1. $\forall 0 \leq i, j \leq n \wedge i \neq j,\ (G_i^1, G_j^1) = sync(G_i, G_j)$
2. $\forall 0 \leq i, j \leq n \wedge i \neq j,\ (G_i^2, G_j^2) = sync(G_i^1, G_j^1)$

   $\ldots$

l. $\forall 0 \leq i, j \leq n \wedge i \neq j,\ (G_i^l, G_j^l) = sync(G_i^{l-1}, G_j^{1-1})$

In each iteration, function $sync()$ refines G_i^k and G_j^k such that $\left(\mathcal{A}_i^O \cap \mathcal{A}_j^I\right) \cup \left(\mathcal{A}_i^I \cap \mathcal{A}_j^O\right) \neq \emptyset$ to be G_i^{k+1} and G_j^{k+1}, respectively. Since function $sync()$ is monotonic,

$$\forall 0 \leq i, j \leq n \wedge i \neq j,\ G_i^{k+1} \preceq G_i^k \text{ and } G_j^{k+1} \preceq G_j^k.$$

If G_i also has an interface with another component G_h and $sync(G_i, G_h)$ is completed before $sync(G_i, G_j)$, G_i may become more restricted after $sync(G_i, G_j)$, but this change is not available to G_h until the next iteration. This indicates that through iterations the effect of refinement of one component propagates to all other components gradually. And naturally, the above framework also works well for systems with circular structures, which are difficult for assume-guarantee reasoning based approaches.

The process performed by the above iterative refinement framework terminates when either of the following two conditions is satisfied:

1. $\forall 0 \leq i \leq n,\ \mathcal{F}(G_i) = \emptyset$.

2. $\forall 0 \leq i \leq n,\ G_i^l \equiv G_i^{l-1}$.

The first condition says that it is unnecessary to continue refinement if all components are failure free because this implies that the complete system is failure free too. The second condition indicates that continuing the process does not result in any further refinement if all components remain the same after being refined. Since function $sync()$ monotonically refines components, it is guaranteed that the iterative refinement process eventually terminates. At that point, any counter-examples found in any component are returned to determine if they are real. Identifying real counter-examples itself is very critical because it may become the bottleneck of the whole verification flow.

The following theorem shows that given $G = \{G_0, \ldots, G_n\}$, no valid behaviors in G_i and G_j for $0 \leq i, j \leq n$ are removed by $sync(G_i, G_j)$. The valid behaviors in G_i are also kept in the reduced SGs after refinement. This ensures soundness of the verification results when function $sync()$ is used.

Theorem 4.3.1 *Let G_i for $0 \leq i \leq n$ be SGs, and $G = \{G_0, \ldots, G_n\}$. Also let G'_i and G'_j be two SGs such that $(G'_i, G'_j) = sync(G_i, G_j)$ for all $0 \leq i, j \leq n$ and $i \neq j$. The following condition holds for G'_i and G'_j.*

$$G[\mathcal{A}_i] \preceq G'_i \textit{ and } G[\mathcal{A}_j] \preceq G'_j$$

Proof: According to property 2.2, the following condition holds.

$$G[\mathcal{A}_i] = \{G_0, \ldots, G_n\}[\mathcal{A}_i] \preceq \{G_i, G_j\}[\mathcal{A}_i] \tag{4.4}$$

Note that the principle of function *sync*() follows parallel composition. According to the definition of parallel composition, for every state transition $((s_i, s_j), a, (s'_i, s'_j))$ in $\{G_i, G_j\}$, there is a (s_i, a, s'_i) in G'_i. This implies that for every path ρ_{ij} in $\{G_i, G_j\}$, there exists a path ρ_i in G'_i such that $\rho_{ij}[\mathcal{A}_i] = \rho_i$. This means that the following condition holds

$$\{G_i, G_j\}[\mathcal{A}_i] \preceq G'_i \tag{4.5}$$

Therefore, combining (4.4) and (4.5) leads to

$$G[\mathcal{A}_i] \preceq G'_i$$

The same argument applies to G'_j, too. ∎

4.3.4 Example

We illustrate the limitation of the constraint-based refinement and improvement of synchronization-based refinement by an example as shown in the Figure 4.2. The SGs for three components with approximate environments where the inputs of the components are set to be completely free are shown again in Figure 4.8(a) - (c), respectively.

First, we present the limitation of the constraint-based refinement for this example. For G_1, the output wire of G_1 is y and the input wire is z. For a state $s_i \in S_1$ of G_1 in

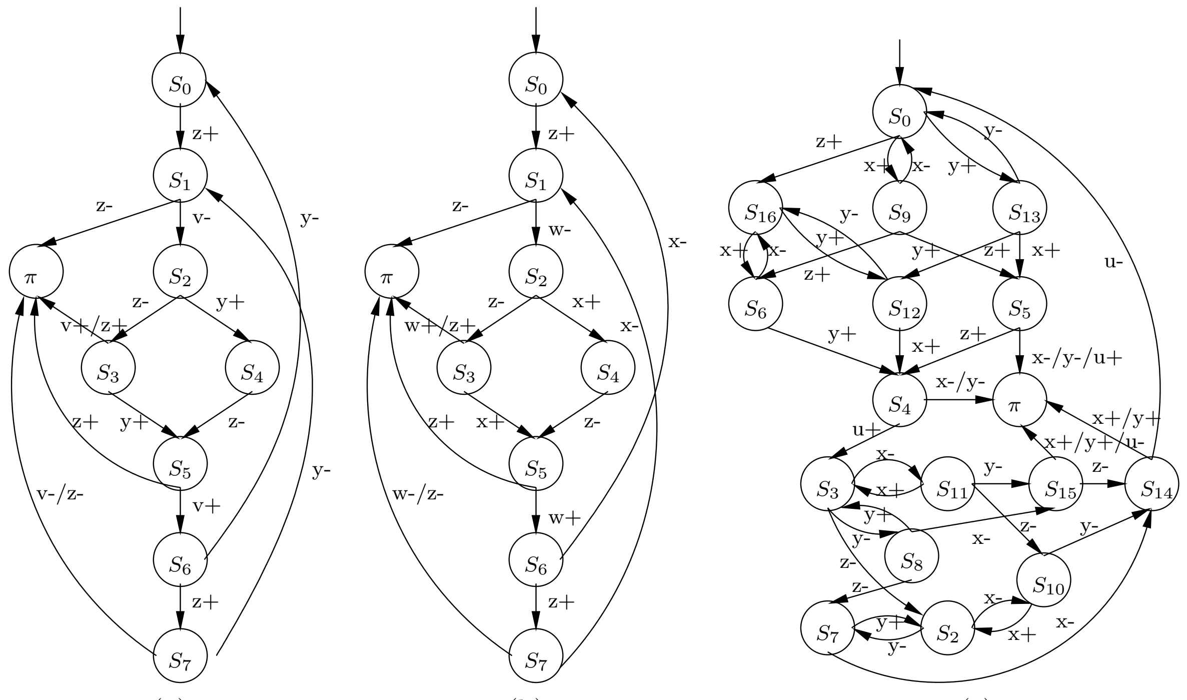

(a) (b) (c)

Figure 4.8. (a) - (c) The SGs G_1, G_2, and G_3 for the components M_1, M_2, M_3 in Figure 2.1(a) where the inputs of the components are set to be completely free, and the autofailure reduction is not applied.

Figure 4.8(a), where $s_i \neq \pi$, $conj(s_i)$ having being projected on the visible wires is listed as follows.

$$s_0 : \neg z \wedge \neg y, s_1 : z \wedge y, s_2 : z \wedge \neg y$$

$$s_3 : \neg z \wedge \neg y, s_4 : z \wedge y, s_5 : \neg z \wedge y$$

$$s_6 : \neg z \wedge y, s_7 : z \wedge y$$

The output constraints C_1^0 generate from G_1 are shown as follows.

$$\mathcal{C}_1^0(y+) = conj(s_2) \vee conj(s_3) = \neg y$$

$$\mathcal{C}_1^0(y-) = conj(s_6) \vee conj(s_7) = y$$

When applying $\mathcal{C}_1^0(y+)$ and $\mathcal{C}_1^0(y-)$ to G_3, no state transition is removed. The input constraint $\mathcal{C}_1^0(y+)$ is satisfied by $conj(s)$ where the action $y+$ is enabled at states

$s_0, s_6, s_7, s_8, s_9, s_{14}, s_{15}, s_{16}$. Similarly, the input constraint $\mathcal{C}_1^0(y-)$ is satisfied by $conj(s)$ where the action $y-$ is enabled at states $s_2, s_3, s_4, s_5, s_{10}, s_{11}, s_{12}, s_{13}$.

For G_2, the output wire of G_2 is x and the input wire is z. For a state $s_i \in S_2$ of G_2 in Figure 4.8(b), where $s_i \neq \pi$, $conj(s_i)$ having being projected on the visible wires is listed as follows.

$$s_0 : \neg z \wedge \neg x, s_1 : z \wedge x, s_2 : z \wedge \neg x$$
$$s_3 : \neg z \wedge \neg x, s_4 : z \wedge x, s_5 : \neg z \wedge x$$
$$s_6 : \neg z \wedge x, s_7 : z \wedge x$$

The output constraints C_2^0 generate from G_2 are shown as follows.

$$\mathcal{C}_2^0(x+) = conj(s_2) \vee conj(s_3) = \neg x$$
$$\mathcal{C}_2^0(x-) = conj(s_6) \vee conj(s_7) = \wedge x$$

When applying $\mathcal{C}_2^0(x+)$ and $\mathcal{C}_2^0(x-)$ to G_3, no state transition is removed. The input constraint $\mathcal{C}_2^0(x+)$ is satisfied by $conj(s)$ where the action $x+$ is enabled at states $s_0, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}, s_{15}, s_{16}$. Similarly, the input constraint $\mathcal{C}_2^0(x-)$ is satisfied by $conj(s)$ where the action $x-$ is enabled at states $s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9$.

For G_3, the output wire of G_3 is z and the input wires are x and y. For a state $s_i \in S_3$ of G_3 in Figure 4.8(c), where $s_i \neq \pi$, $conj(s_i)$ having being projected on the visible wires is listed as follows.

$$s_0 : \neg x \wedge \neg y \wedge \neg z, s_2 : x \wedge y \wedge \neg z, s_3 : x \wedge y \wedge z$$
$$s_4 : x \wedge y \wedge z, s_5 : x \wedge y \wedge \neg z, s_6 : x \wedge \neg y \wedge z$$
$$s_7 : x \wedge \neg y \wedge \neg z, s_8 : x \wedge \neg y \wedge z, s_9 : x \wedge \neg y \wedge \neg z$$
$$s_{10} : \neg x \wedge y \wedge \neg z, s_{11} : \neg x \wedge y \wedge z, s_{12} : \neg x \wedge y \wedge z$$
$$s_{13} : \neg x \wedge y \wedge \neg z, s_{14} : \neg x \wedge \neg y \wedge \neg z, s_{15} : \neg x \wedge \neg y \wedge z$$
$$s_{16} : \neg x \wedge \neg y \wedge z$$

The action $z+$ is enabled in the states s_0, s_5, s_9, s_{13}. The action $z-$ is enabled in the states s_3, s_8, s_{11}, s_{15}. Therefore, the output constraints C_3^0 generate from G_3 are shown as follows.

Table 4.1. Synchronization of G_1 and G_3.

State pair	unsync in G_1	unsync in G_3
(s_0, s_0)		$(s_0, y+, s_{13})$
(s_1, s_{16})	$(s_1, z-, \pi)$	
(s_2, s_{16})	$(s_2, z-, s_3)$	
(s_4, s_{12})		$(s_{12}, y-, s_{16})$
(s_4, s_4)		$(s_4, y-, \pi)$
(s_4, s_3)		$(s_3, y-, s_8)$
(s_5, s_2)	$(s_5, z+, \pi)$	
(s_6, s_2)	$(s_6, z+, s_7)$	
(s_0, s_7)		$(s_7, y+, s_2)$
(s_0, s_{14})		$(s_{14}, y+, s_\pi)$

$$\mathcal{C}_3^0(z+) = conj(s_0) \vee conj(s_5) \vee conj(s_9) \vee conj(s_{13}) = \neg z$$

$$\mathcal{C}_3^0(z-) = conj(s_3) \vee conj(s_8) \vee conj(s_{11}) \vee conj(s_{15}) = z$$

When applying $\mathcal{C}_3^0(z+)$ and $\mathcal{C}_3^0(z-)$ to G_1, no state transition is removed. The input constraint $\mathcal{C}_3^0(z+)$ is satisfied by $conj(s)$ where the action $z+$ is enabled at states s_0, s_3, s_5, s_6. The input constraint $\mathcal{C}_3^0(z-)$ is satisfied by $conj(s)$ where the action $z-$ is enabled at states s_1, s_2, s_4, s_7. Similarly, When applying $\mathcal{C}_3^0(z+)$ and $\mathcal{C}_3^0(z-)$ to G_2, no state transition is removed.

Therefore, no extra state transitions introduced by the approximate environments can be removed from G_1, G_2 and G_3 with the constraint-based refinement. Then, we show how these extra state transitions are eliminated by the synchronization-based refinement.

1. $sync(G_1, G_3)$.

   G_1 and G_3 are synchronized on the common interface z and y. The unsynchronized state transitions of G_1 and G_3 at each composite state pair are shown in Table 4.1. After removing the unsynchronized state transitions and unreachable state transitions from G_1 and G_3, the reduced G_1^1 and G_3^1 are shown in Figure 4.9 (a) and (b), respectively.

2. $sync(G_2, G_3^1)$.

Table 4.2. Synchronization of G_2 and G_3^1.

State pair	unsync in G_2	unsync in G_3^1
(s_0, s_0)		$(s_0, x+, s_9)$
(s_1, s_{16})	$(s_1, z-, \pi)$	
(s_2, s_{16})	$(s_2, z-, s_3)$	
(s_4, s_6)		$(s_6, x-, s_{16})$
(s_4, s_4)		$(s_4, x-, \pi)$
(s_4, s_3)		$(s_3, x-, s_{11})$
(s_5, s_2)	$(s_5, z+, \pi)$	
(s_6, s_2)	$(s_6, z+, s_7)$	
(s_0, s_{10})		$(s_{10}, x+, s_2)$
(s_0, s_{14})		$(s_{14}, x+, s_\pi)$

G_2 and G_3^1 are synchronized on the common interface z and x. The unsynchronized state transitions of G_1 and G_3^1 at each composite state pair are shown in Table 4.2. After removing the unsynchronized state transitions and unreachable state transitions from G_2 and G_3^1, the reduced G_2^1 and G_3^2 are shown in Figure 4.10(a) and (b), respectively.

As shown in the example in Figure 4.8 where the inputs of the components are set to be completely free, the constraint-based refinement is unable to capture the execution sequences of the actions, and thus can not lead to any reduction for the components. However, the synchronization-based refinement overcomes the limitation of the constraint-based refinement by synchronizing the common behaviors between two interactive components. The final SGs reduced by the synchronization-based refinement are shown in Figure 4.9(a) and Figure 4.10 for G_1, G_2 and G_3, respectively. Noted that the final SGs obtained by applying the synchronization-based refinement are same to the ones, as shown in Figure 4.6, by applying the constraint-based refinement on SGs, as shown in Figure 4.4, after being reduced by the autofaiure reducation on SGs in Figure 4.8. The reason that the autofailure reduction can facilitate the constraint-based refinement is that it is able to remove some extra states and state transitions from SGs where stronger constraints can be generated.

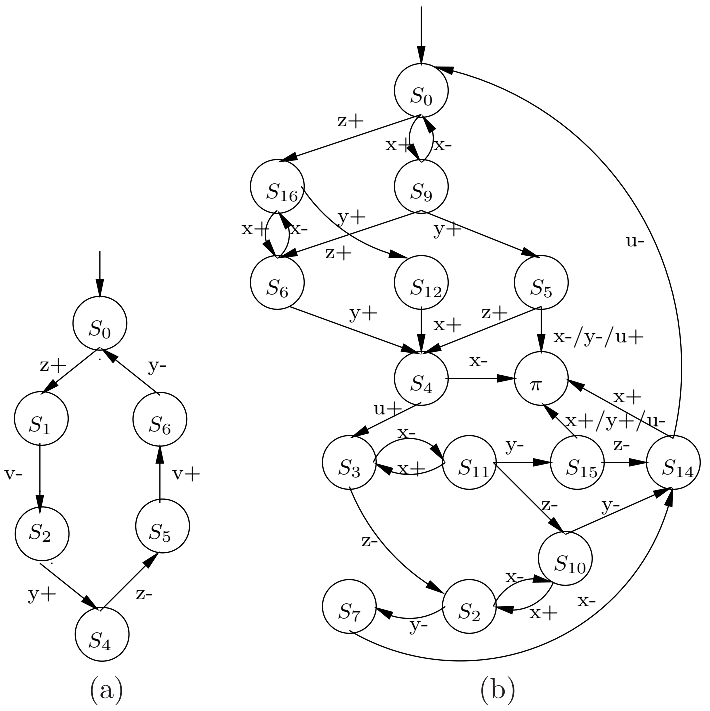


Figure 4.9. Reduced SGs G_1^1 and G_3^1 after applying the synchronization-based refinement on the SGs G_1 and G_3 in Figure 4.8(a) and (c), respectively.

4.3.5 Simultaneous Multi-Synchronization

Function $sync(M_i, M_j)$ works fine if the components of a system have simple interactions. However, this is not always true in practice, and the function may not result in good refinement if there exist inter-dependencies among the components. Consider the example shown in Figure 4.11, which shows three components in a system and their partial SGs are shown in Figure 4.12. Among them, M_1 shares actions a and d with M_2 and M_3, respectively, and M_2 and M_3 also share an action e. According to M_2 and M_3 in the figure, action d occurs before action a, therefore path (s_0, a, s_1, d, s_2) in G_1 is invalid. However, applying $sync(G_i, G_j)$ for $i, j = 1, 2, 3$ and $i \neq j$ cannot remove such a path from G_1. The portions of SGs after applying $sync(G_1, G_2), sync(G_1, G_3)$ and $sync(G_2, G_3)$ are shown in Figure 4.13(a), (b), (c), respectively. All the state transitions of G_1, G_2 and G_3 are synchronized on their common interfaces. This is because the ordering of actions a and d is determined by action e of G_3, which is invisible to G_1 when applying $sync(G_1, G_2)$

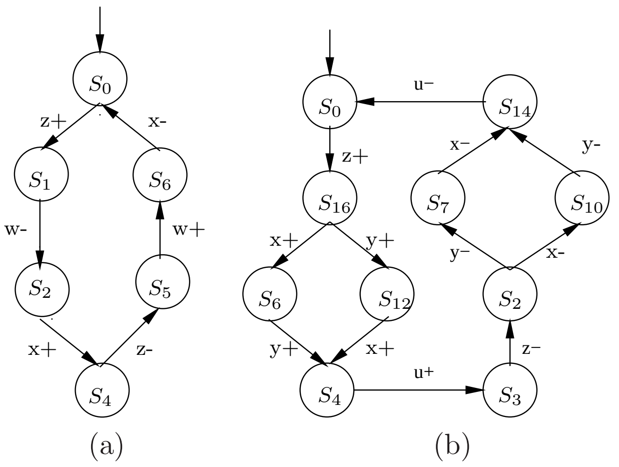


Figure 4.10. Reduced SGs G_2^1 and G_3^2 after applying the synchronization-based refinement on the SGs G_2 in Figure 4.8(a) and G_3^1 in Figure 4.9(b), respectively.

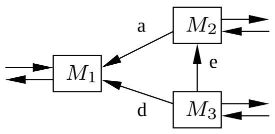


Figure 4.11. A system consists of three components M_1, M_2 and M_3. There is inter-dependency on e between M_2 and M_3.

and $sync(G_1, G_3)$, respectively, therefore the correlation between a and d due to e is lost. This example illustrates the impact of the inter-dependencies among components on the refinement results.

To address this problem and take the component inter-dependencies into account, we extend function $sync()$ to synchronize multiple components simultaneously. For convenience, the function that synchronizes two components is denoted as $sync_2()$, while the function that synchronizes multiple components is denoted as $sync_n()$, which is also referred to as multi-synchronization. Function $sync_n()$ works similarly to $sync_2()$, but the key is that more actions become visible on the interfaces and participate in the synchronization process at the same time. In the following, we first illustrate the idea of this function by applying it to G_1, G_2, and G_3 in Figure 4.12. Starting from $t_0 = (p_0, q_0, s_0)$, transitions (q_0, d, q_1) in G_3 and (s_0, d, s_3) in G_1 are synchronized, which result in a new composite state $t_1 = (p_0, q_1, s_3)$. In this new state, (p_0, e, p_1) in G_2 and (q_1, e, q_2) in G_3 are synchronized,

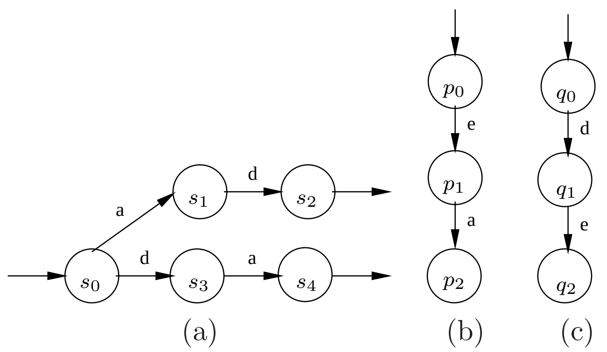


Figure 4.12. (a) - (c) The portions of SGs for M_1, M_2 and M_3 in Figure 4.11, respectively.

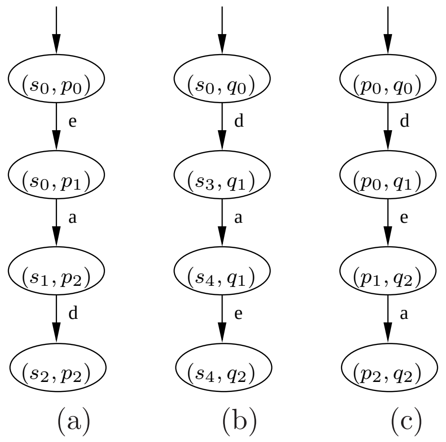


Figure 4.13. (a) - (c) are the potions of SGs after applying $sync(G_1, G_2), sync(G_1, G_3)$ and $sync(G_2, G_3)$, respectively.

thus resulting another new composite state $t_2 = (p_1, q_2, s_3)$. From this state, (p_1, a, p_2) in G_2 and (s_3, a, s_4) in G_1 are synchronized resulting in $t_3 = (p_2, q_2, s_4)$. In the above synchronization process, as shown in Figure 4.14, only the path $(t_0, d, t_1, e, t_2, a, t_3)$ is explored. After the synchronization is done, transition (s_0, a, s_1) and (s_1, d, s_2) and states s_1 and s_2 in G_1 are removed, and the reduced G_1 is shown in Figure 4.15(a). All the state transitions in G_2 and G_3 are synchronized in the multi-synchronization process and kept in the final SGs shown in Figure 4.15(b) and (c). This example shows that making more actions visible for synchronization leads to better refinement results.

From the above description, $sync_2()$ is a special case of $sync_n()$, both of which are denoted as $sync()$ in the rest of this chapter. Algorithm 12 shows a pseudo procedure for

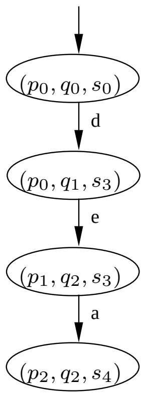


Figure 4.14. The portion of the composite SG by applying the multi-syncronization on SGs G_1, G_2, G_3.

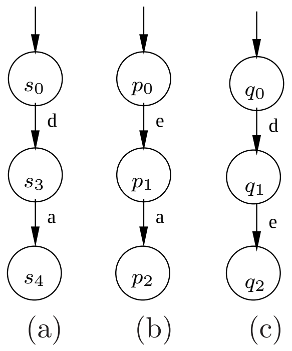


Figure 4.15. (a) - (c) are the potions of reduced SGs of G_1, G_2 and G_3 after applying the multi-synchronization, respectively.

the multi-synchronization function *sync*. This procedure takes a set of k component SGs. First, the shared actions between each pair of components are found, and together they are grouped into a set $\mathcal{A}$, which is the set of all visible actions in this set of components. Next, a tuple of component initial states are pushed onto a stack. Then, the procedure iterates until the stack is empty. In each iteration, the component states in the tuple on the top of stack is popped. For each pair of popped component states, all their outgoing transitions in these states are considered. If the actions a_h and a_j of transitions (s_h, a_h, s'_h) and (s_j, a_j, s'_j) in different components matches and they are in $\mathcal{A}$, these transitions are synchronized, and added into R'_h and R'_j, respectively. If actions are not the same but not in $\mathcal{A}$, they are automatically synchronized and also added into R'_h and R'_j, respectively. In

Algorithm 12: `sync` $(\{G_i, G_{i+1}, \ldots, G_{i+k}\})$

```
1  $\mathcal{A}_{hj} = \mathcal{A}^I_h \cap \mathcal{A}^O_j$ for $0 \le h, j \le k,$ and $i \ne j$;
2  $\mathcal{A} = \bigcup_{h,j=0}^{k} \mathcal{A}_{hj}$ for $h \ne j$;
3  Push $(init_i, init_{i+1}, \ldots, init_{i+k})$ onto stack;
4  while stack is not empty do
5      Let $(s_i, s_{i+1}, \ldots, s_{i+k})$ be the top of stack;
6      Pop stack;
7      foreach pair of $s_h$ and $s_j$ such that $h \ne j$ do
8          $q = (s_i, s_{i+1}, \ldots, s_{i+k})$;
9          foreach $(s_h, a_h, s'_h) \in outgoing(s_h)$ do
10             foreach $(s_j, a_j, s'_j) \in outgoing(s_j)$ do
11                 if $a_h = a_j \wedge a_h \in \mathcal{A} \wedge a_j \in \mathcal{A}$ then
12                     Add $(s_h, a_h, s'_h)$ into $R'_h$;
13                     Add $(s_j, a_j, s'_j)$ into $R'_j$;
14                     Replace $s_h$ in $q$ with $s'_h$;
15                     Replace $s_j$ in $q$ with $s'_j$;
16                 else if $a_h \notin \mathcal{A}$ then
17                     Add $(s_h, a_h, s'_h)$ into $R'_h$;
18                     Replace $s_h$ in $q$ with $s'_h$;
19                 else if $a_j \notin \mathcal{A}$ then
20                     Add $(s_j, a_j, s'_j)$ into $R'_j$;
21                     Replace $s_j$ in $q$ with $s'_j$;
22         Push $q$ onto stack;
23 foreach $0 \le h \le k$ do
24     $R_h = R'_h$;
25     Remove unreachable states from $S_h$;
```

each case, a new tuple of component states is created, and pushed onto stack for future synchronization. At the end of synchronization, R'_h for $0 \le h \le k$ stores all synchronized transitions of component G_h, and it replaces the state transition set in the original SG. And finally, there may be some states to become unreachable with respect to the new but smaller transition set, and they need to be removed.

Algorithm 12 is similar to the definition of parallel composition. However, the ultimate goal of this algorithm is to identify the synchronized transitions for each component efficiently, therefore the complete composite SG is not generated during synchronization as in parallel composition. In the algorithm, function *sync*() does not store the reachable state transitions of the composite SG. Instead, the transitions in the individual components are marked synchronized corresponding to the reachable transitions found during synchronization. On the other hand, the reachable composite states (state tuples in the algorithm) found during synchronization are stored to guarantee the termination of the function. The time complexity of function *sync*() is close to that of parallel composition since conceptually the function needs to explore all composite states reachable in the composite SG. The space complexity of that function can be much lower than that of parallel composition since the complete composite SG is not constructed and the memory usage is reduced by not storing the potentially very large number of composite state transitions.

4.3.6 Determining k

The abstraction refinement framework described in the previous section can be more effective when multi-synchronization function *sync*() is used. Ideally, if function $sync_n()$ is applied to all components in a system simultaneously, then each component can be maximally refined. Even though the space complexity of function $sync(\{G_0, \dots, G_n\})$ is lower than that of parallel composition of $\{G_0, \dots, G_n\}$, all reachable composite states still need to be generated, and this number can be very large in practice. Therefore, the memory requirement may be prohibitively high if the number n of components for *sync*() is too large. This put a limit on the number of components and the size of each component that can be taken by *sync*(). To control the size when calling *sync*(), only a subset of all components in a system are handled at a time. Given a design $G = \{G_0, \dots, G_n\}$ and a number k, it is divided into subsets Q_0, ..., and Q_l such that

1. $\forall 0 \le i \le l,\ |Q_i| \le k$.
2. $Q_0 \cup \ldots \cup Q_l = G$.
3. The following condition holds for any Q_i.

$$\forall G_h \in Q_i, \exists G_j \in Q_i,\ \mathcal{A}_h^O \cap \mathcal{A}_j^I \neq \emptyset \vee \mathcal{A}_h^I \cap \mathcal{A}_j^O \neq \emptyset$$

Note that it is possible for a component to be in different subsets if it shares common interfaces with multiple components which cannot be grouped together. The last condition above requires that the system structures need to be examined for partitioning. Each component in a subset needs to share interfaces with at least another component. Otherwise, such a component is not grouped in that subset to avoid unnecessary work.

In our method, k is first set to 2, and all components are grouped into subsets, each of which contains two components. Then verification with the iterative refinement process presented in the previous section is applied. If one or more components have failures, k is increased to 3 and the refinement is repeated. This sequence of the alternating steps continues until either of the termination conditions of the abstraction refinement process is satisfied, or k becomes too large for $sync()$. However, the experimental results show that $k = 2$ or 3 is usually enough, and is no larger than 4 for very finely partitioned systems with complex interactions among components.

To control the complexity during synchronization, users can also set an upper bound on k such that refinement stops when this upper bound is reached. When verification on a refinement fails and k is smaller than the upper bound, it is incremented by 1 indicating that one more component can be added into each subset. To avoid redundant work, a component is added into a subset if one of the following cases exists. The first case is as shown in Figure 4.11 such that a component is added if it shares common interface with 2 or more other components in the same subset. This makes sure that the inter-dependencies among components are taken into consideration. When k becomes larger, this condition is strengthened by requiring such a component have common interfaces with more components in the same subset for it to be added. In the second case, given two subsets Q_1 and Q_2 such that a component $G_j \in Q_1 \cap Q_2$. After refinement on these subsets separately, if G_j still has failures, this implies that G_j is a control point in a design whose correctness requires coordination of other components in Q_1 and Q_2. Therefore, a new subset Q_3 is created as follows.

$$Q_3 = \{G_i \mid G_i \in Q_1 \cup Q_2 \wedge \mathcal{A}_i^O \cap \mathcal{A}_j^I \neq \emptyset\} \cup \{G_j\}$$

4.4 Summary

Constraint-based refinement iteratively refines each component in a design by extracting the output constraints of a component and applying these output constraints to its neighboring components as input constraints to remove invalid input, and consequently some output, state transitions. When a SG is reduced, the output constraints of the SGs may become stronger, which may cause more input transitions in its neighboring components to become invalid and be eliminated. As the neighboring components become more reduced, the output constraints of these neighboring components may again become stronger. Among three state space reduction techniques presented in the previous chapter, the autofailure reduction can remove some output state transitions to pinpoint the real causes for the failure states, therefore allowing stronger output constraints to be produced. However, other two state space reduction, e.g. transition-based reduction and region-based reduction, which remove the observably equivalent state transitions, cannot lead to the stronger output constraints. Therefore, applying the autofailure reduction facilitates the constraint-based refinement by extracting stronger output constraints. The process of extraction of output constraints and application of input constraints occurs on a single component. Therefore, the time complexity of the constraint-based refinement is linear to the number of state transitions of the biggest component of a system.

Synchronization-based refinement iteratively refines each component in a design by examining its interface interactions with its neighbors, and the behaviors not synchronized with its neighbors are removed. This method is enhanced by synchronizing multiple components simultaneously so that inter-dependencies among different components are considered. The idea of this method comes from the observation that the interface behaviors are allowed if they are synchronized among the components connected by that interface. When composing two components, the unsynchronized behaviors between these two components are removed from these two components individually. This method takes advantage of traditional parallel composition by identifying and removing the unsynchronized behaviors from each component rather than actually constructing a composite component. By synchronizing multiple components simultaneously, the effect of the inter-dependencies among different components are also considered, thus leading to stronger refinement for the components. The main contributions of this method are a local synchronization detection method for component refinement based on parallel composition, and an interface refinement method for efficient compositional model checking where each component is refined iteratively and incrementally by checking synchronization among components sharing common interfaces.

CHAPTER 5

EXPERIMENTAL RESULTS AND DISCUSSION

We have developed an explicit model checker, *Platu*, which can perform non-compositional and compositional verification. The abstraction refinement methods described in Chapter 4 and the compositional minimization with state space reduction techniques discussed in Chapter 3 are implemented in Platu. Experiments have been performed on several non-trivial asynchronous circuit designs to demonstrate the scalability of verification using the method presented in this dissertation.

5.1 Examples and Environment Setup

In our method, asynchronous designs are specified in a high level description. To verify a design, all components in that high level description are translated into BGPNs and then converted to SGs. To ensure soundness, an over-approximate environment for each component needs to be found to simulate the interface behaviors between the component and the rest of the design. The SGs generated this way are abstractions of the concrete ones. The *maximal environment* [53] is used for each component in all experiments. In the maximal environment for a component, all inputs of the component are totally free. Therefore, the maximal environment defines all possible behaviors on inputs of a component and captures the values of interleaved inputs in all orderings. The reason for choosing the maximal environment for each component is to generate the most general abstract SGs to experiment the effectiveness of the described refinement method. In practice, more restricted and accurate environment is highly desirable since it yields smaller and more concrete SGs and thus making the described refinement methods more efficient.

The first three designs are a self-timed FIFO [68], a tree arbiter of multiple cells [38], and a distributed mutual exclusion element consisting of a ring of DME cells [38]. Despite all these designs having regular structures to be scaled easily, the regularity is not exploited in our method, and all the components are treated as black boxes. The fourth example is a tag unit (TU) circuit in the Intel's RAPPID asynchronous instruction length decoder [92]. This example is an unoptimized version of the actual circuit used in RAPPID with higher complexity, which is more interesting for experimenting our methods. These four examples are failure free. The last example is a pipeline controller (PC) for an asynchronous processor TITAC2 [97]. The PC example contain deadlock. All the examples are too large for the non-compositional approaches.

In the experiments, DME, arbiter, and FIFO examples are partitioned according to their natural structures. In other words, each cell is a component. For the tag unit circuit, TU, it is partitioned into three components, where the middle five blocks form a component, and gates on the sides of the component in the middle form the other two. The pipeline controller, PC, is partitioned into ten component, each of which contains five gates.

All experiments were performed on a Linux workstation with a Intel Pentium-D dual-core CPU and 1 GB memory. In the following tables, the column $\#Cells$ is the number of cells in a design, and the column $|W|$ is the total number of wires in each design, which is a rough estimate of the design complexity. All SGs of all components are kept in memory for simplicity. The Column $Mem(MB)$ shows the total peak memory usage for all components during verification, calculated in MBs. The column $Time(Sec.)$ is the total runtime taken for verifying all components, calculated in seconds.

5.2 Local Verification with Constraint-based Refinement

In the first experiment, all examples are verified with the compositional method combined with the constraint-based abstract refinement and two state space reduction techniques, the autofailure reduction and transition-based reduction, presented in Chapter 3. The results are shown in Table 5.1. In the table, the column $\#Iter$ shows the number of iterations required to complete the constraint-based refinement. The last column $\#\pi$ shows the number of components in a design that have the failure state after the verification.

In these experiments, selective composition is performed if SGs of some components are found to have failures. In selective composition, some or all of the SGs with failures are composed as allowed by the available memory. This results in some state transitions becoming invisible and leads to reductions on the composed SGs. Then, the refinement is applied again. The reduced composite SGs in turn may result in reductions on other SGs. The number of iterations in Table 5.1 is the total number of iterations for the refinement before and after selective composition. In the experiments, selective composition is needed for ARB and PC. For PC, the original 10 components all contain failures. Selective composition merges these components into 5 larger ones, and one of them is failure-free.

From Table 5.1, we obtain the following observations. First, for scalable FIFO, ARB, and DME examples, memory and runtime usages grow polynomially as the number of components in the designs increases. Second, although the refined SG for each component is still an abstraction of the exact one after the abstract refinement, all examples except PC are shown to be failure-free.

Even though constraint-based refinement may not eliminate failures for each component in PC completely, the number of components containing failures and the number of failure traces in those components are reduced significantly by the refinement and reductions, therefore making distinguishing false counterexamples easier. In the table, we only show the size of the largest SG encountered during verification. This is because it is the largest component of a design that determines the success or failure of verification. With respect to compositional reasoning, only one component needs to stay in memory at a time.

Table 5.1. Experimental results with constraint-based refinement + reduction + composition.

Design	$\#Cells$	$\|A\|$	Mem	$Time$	$\|S\|^{peak}$	$\|R\|^{peak}$	$\|S\|$	$\|R\|$	$\#Iter$	$\#\pi$
FIFO	100	804	30	18	57	153	15	21	3	0
	200	1604	80	41	57	153	15	21	3	0
	400	3204	237	102	57	153	15	21	3	0
	600	4804	471	184	57	153	15	21	3	0
	800	6404	781	290	57	153	15	21	3	0
DME	20	440	35	43	361	9497	33	61	4	0
	50	1100	88	113	361	9497	33	61	4	0
	100	2200	191	249	361	9497	33	61	4	0
	200	4400	446	600	361	9497	33	61	4	0
	300	6600	771	1044	361	9497	33	61	4	0
ARB	7	132	3	2	290	758	38	68	3	0
	15	244	7	6	385	1407	104	340	5	0
	31	500	33	47	1737	9636	689	3833	7	0
	63	1012	262	988	15684	134438	5018	48342	8	0
TU	3	96	117	103	3697	280392	2112	122648	2	0
PC	10	100	23	47	1409	12736	679	4882	6	4

To fully appreciate the power of the state space reduction techniques, including the autofailure and transition-based reduction, the difference between the state-based abstraction [4] and these two reduction techniques, and the impact of selective composition on the verification results, we run three more experiments. All these experiments use the constraint-based refinement method without using selective SG composition as in the previous experiment. The results are shown in Table 5.2. In the first experiment labeled as $E1$, all examples are verified with only the constraint-based refinement. In the second experiment labeled as $E2$, all examples are verified with the constraint-based refinement, the autofailure reduction and the state-based abstraction. In the third experiment labeled as $E3$, two reduction techniques, the autofailure reduction and transition-based reduction, and the constraint-based refinement are used for all examples. This experiment is similar to the one used for Table 5.1 except that selective composition is not applied. Comparing the results in these two tables, the runtime and memory usage for all examples as shown in Table 5.2 are generally less, much less in some cases, than those shown in Table 5.1. The main reason is the SG composition, which causes the size blowup if used for some examples. However, without selective composition, the verification results become worse in terms of the number of components with failures. For example, 31 out of 63 components in the ARB have failures under $E3$ without using the SG composition, while none has failures if selective composition is applied as shown in Table 5.1.

Table 5.2. Comparison of three different experiments.

		$E1$			$E2$			$E3$		
Design	$\#Cells$	Mem	$Time$	$\#\pi$	Mem	$Time$	$\#\pi$	Mem	$Time$	$\#\pi$
FIFO	100	30	11	0	29	11	96	30	18	0
	200	80	28	0	78	26	196	80	41	0
	400	236	74	0	232	66	396	237	102	0
	600	470	140	0	470	140	596	471	184	0
	800	780	227	0	772	201	796	781	290	0
DME	20	14	13	0	13	9	0	35	43	0
	50	40	37	0	38	27	0	88	113	0
	100	97	90	0	94	69	0	191	249	0
	200	264	249	0	258	202	0	446	600	0
	300	502	474	0	492	402	0	771	1044	0
ARB	7	3	1	6	2	1	2	3	2	0
	15	8	3	15	6	4	3	7	5	7
	31	18	8	30	15	9	12	17	13	15
	63	42	22	59	36	23	30	40	33	31
TU	3	26	28	0	11	10	0	117	103	0
PC	5	13	10	5	8	10	4	19	50	5

Comparing the results in columns $E1$ and $E3$, whether applying reductions or not does not make much difference in these experiments except for the TU. In this case, memory blows up because the transition-based reduction creates a much large number of state transitions to preserve all possible behaviors of the components in TU and avoid introducing extra paths. The increased state transitions may be removed by performing redundancy removal or the autofailure reduction for other examples rather than TU. This example illustrates the negative effect of the transition-based reduction technique. On the positive side, using the new abstraction causes less failures introduced. Comparing results of FIFO in columns under $E2$ and $E3$, no component has failures when the transition-based reduction is used, while the number of components with failures increases as the number of cells increases when applying the state-based abstraction. However, for ARB and PC, the number of components with failures under $E2$ is actually less than that under $E3$. This is because of the aggressive reduction feature of the state-based abstraction where larger state space may be trimmed as long as the failure traces are preserved. Trimming a larger state space, including the valid portion, helps to produce stronger output constraints, which then cause the other components to be more reduced. However, the extra behavior may be introduced by the state-based abstraction. This is why the memory and runtime are also less under $E2$. Using the state-based abstraction always preserves failures in at least one component, but its aggressiveness in reducing state space may cause some components to lose their failure traces.

5.3 Local Verification with Synchronization-based Refinement

In Table 5.3, columns under $Method\ 1$ show the results from Table 5.1 by the constraint-based refinement method, while the results obtained by using the synchronization-based refinement method are shown in columns under $Method\ 2$.

Table 5.3. Experimental results and comparison with the constraint-based refinement.

Design	$\#Cells$	$\lvert\mathcal{A}\rvert$	$Method$ 1				$Method$ 2				
			Mem	Time	#Iter	$\#\pi$	Mem	Time	#Iter	$\#\pi$	k
FIFO	100	804	30	18	3	0	30	9.46	1	0	2
	200	1604	80	41	3	0	80	27	1	0	2
	400	3204	237	102	3	0	237	84	1	0	2
	600	4804	471	184	3	0	470	174	1	0	2
	800	6404	781	290	3	0	780	301	1	0	2
DME	20	440	35	43	4	0	14	14	1	0	2
	50	1100	88	113	4	0	40	39	1	0	2
	100	2200	191	249	4	0	97	96	1	0	2
	200	4400	446	600	4	0	264	257	1	0	2
	300	6600	771	1044	4	0	502	487	1	0	2
ARB	7	132	3	2	3	0	3	1.18	1	0	2
	15	244	7	6	5	0	6	2.99	1	0	2
	31	500	33	47	7	0	16	7.70	1	0	2
	63	1012	262	988	8	0	39	25	2	0	2
TU	3	96	117	103	2	0	10	3.9	1	0	2
PC	10	100	23	47	6	4	13	30	4	0	4

In Table 5.3, the column $|A|$ show the size of $\mathcal{A}$ of each design. The four columns under *Method* 1 show the peak memory, the total runtime, the total number of iterations needed to finish verifying each design, and the number of components containing the failures at the end, respectively. There are five columns under *Method* 2. The first four columns have the same meanings as those under *Method* 1. The last one shows the largest k needed to finish verifying each design successfully [102]. The results shown under *Method* 2 are obtained with on-the-fly autofailure.

From the results under *Method* 2 in Table 5.3, the following observations can be obtained. First of all, for scalable FIFO, ARB and DME examples, memory and runtime usages grow polynomially similar to the constraint-based refinement as the number of components in the designs increases.

Second, all examples are verified failure free by the synchronization-based refinement method, even though the results are obtained from the still over-approximate environments after refinement. Third, the peak memory is usually reached during synchronization. However, the memory usage is much smaller than that of the parallel composition if the components were composed. As observed in the experiments, the memory usage required is highest at the beginning of the refinement process. This is because all component SGs are generated with the their maximal environments, and these SGs contain a lot of extra states and state transitions, which make the complexity of synchronization higher. This indicates the negative effect of using the maximal environment for each component as the initial approximation. As indicated above, memory consumption may decrease if the initial approximate environment chosen for each component is more restricted than the maximal environment. On the other hand, these results show the effectiveness of this method to refine very coarse environment description to be accurate enough to finish verification successfully.

It is also interesting to notice that the number of iterations for the refinement process to terminate is pretty small for all examples except PC. In these experiments, k is first set to 2 for all examples. This is enough for all examples other than PC to show failure free. This is understandable in that all the designs other than PC have well defined and relatively

Table 5.4. Results from the synchronization-based refinement without any reduction and with only the static autofailure reduction (AFR).

		No Reduction		Static AFR		on-the-fly AFR	
Design	$\#Cells$	Mem	Time	Mem	Time	Mem	Time
ARB	7	12	19	3	2.3	3	1.18
	15	19	62	7	6.9	6	2.99
	31	33	175	17	27	16	7.70
TU	3	73	57	73	62	10	3.9
PC	10	16	41	14	25	13	30

simple communication interactions on the interface. For PC, k has to be increased to 4 to refine it to be failure free. This is because that the interactions among the components in PC are much more complex, and there are rich inter-dependencies among the components.

Also note that with respect to compositional reasoning, only one component needs to stay in memory at a time. However, we keep SGs of all components in memory for simplicity, and the memory usages in columns under Mem show the total peak memory usage for all component during verification.

Compared with the results by the constraint-based refinement in Column $Method$ 1, the synchronization-based refinement method is much more effective. This method shows that PC contains no failure while $Method$ 1 does not succeed. Additionally, for all examples, this method requires much less memory and runtime. It shows the effectiveness of the synchronization-based refinement to remove impossible behavior introduced by over-approximate environment for each component.

In all experiments, the maximal environment is used to find the SG for each component because we assume no knowledge of its interface. However, it causes a large number of extra states and state transitions to be introduced, including ones leading to the failure state. As described before, the failure traces can be trimmed using autofailure reduction to reduce the size of SGs. To show the effectiveness of on-the-fly autofailure, two sets of experiments are performed where no reduction is used during the whole refinement process, and only autofailure reduction is used after the initial SGs are generated with the maximal environment. The results are shown in Table 5.4.

Comparing the results in Table 5.4 and those under $Method$ 2 in Table 5.3, the runtime results are much worse without any reduction at all during refinement. For ARB 31, it takes 175 seconds for the whole verification process to finish while it takes only less than 8 seconds for the refinement process in $Method$ 2 in Table 5.3 to succeed. This is because that all possible behaviors within the over-approximate environements where all inputs are totally free are generated, and it takes much more time for function *sync* to determine and remove behaviors that are not allowed by the interface among a component and its neighbors. The results of applying autofailure reduction on the constructed SGs obtained by components with their approximate environments are not as bad the ones shown in columns under Static AFR in Table 5.4, but the runtime is still higher across all examples because more states and state transitions are generated in the first place compared to using on-the-fly autofailure reduction and more time is needed to perform the autofailure reduction afterward. These results show the effectiveness of on-the-fly autofailure reduction and the fundamental necessity of controlling the size of the initially generated SGs.

If counterexamples still exist after refinement, they are checked on the whole design as in [100], which can be very expensive. From the above experimental results, it shows that the described abstraction refinement method can very effectively eliminate the extra behaviors, including those leading to counterexamples. This helps to avoid high computation penalty for confirming the false counterexamples.

5.4 Compositional Minimization with State Space Reductions

The set of test cases is appended by a circuit implementation of a memory management unit (MMU) from [75]. The MMU example is also too large for the non-compositional approaches to handle. We divide the MMU example by following the structure provided in [75] such that each component defines one output that is used by other components.

Since the autofailure reduction preserves all observable behavior of SGs and prevents from introducing extra behavior, the on-the-fly autofailure reduction is applied in all the following experiments to simplify the SGs composed by the components and their initial approximate environments. We focus on comparison of the effectiveness of the state-based abstraction, the transition-based reduction and the region-based reduction in three experiments as follows.

In Table 5.5, $|S|$ and $|R|$ are the numbers of states and state transitions of the largest SG encountered during the whole course of compositional minimiziation. π shows whether the final SG has a failure state. The largest SGs are recorded because their sizes in general determine whether the whole process of compositional minimization can be finished or not, therefore their sizes need to be carefully controlled. For examples which use too much memory, the corresponding entries are filled with $-$. The results by the state-based abstraction, the transitions-based reduction, and the region-based reduction are shown under Method 1, Method 2, and Method 3, respectively.

From Table 5.5, it can be seen that state-based abstraction is very efficient, and can finish all the examples in a small amount of time. This is because the state-based abstraction does not follow the restrictions imposed on the transition-based and region-based reduction, and it can remove all invisible transitions aggressively. On the other hand, all examples except design TU have false failures introduced in the final reduced SG after the state-based abstraction, which often introduces a lot of extra traces including failures as pointed out earlier. This is the significant negative outcome of the state-based abstraction as it can be very expensive to identify whether these failures are real. The transition-based reduction can only finish on a FIFO with only 100 cells or less. For larger designs, the intermediate SGs get too large for the whole composition to finish. As an example, for DME, after the component SGs are abstracted using the transition-based reduction, the number of states in these SGs remains almost the same while the number of transitions increases dramatically. When composing the first three cells of a DME design, the number of states and transitions continues to increase with no sign of termination after the composition process runs for over 10 minutes. With the region-based reduction technique, all examples can be finished compared with the transition-based reduction, but they all take significantly more time than the state-based abstraction does. On the other hand, all examples except PC are proved to be failure free.

The component SGs generated with the maximal environment often have a large number of extra states and transitions in the SGs, which increases the complexity of the composition and the size of the intermediate and final SGs. In the second experiment, the synchronization-based refinement [101] is applied to the component SGs to simplify their complexity before the composition. This refinement identifies and removes states and state transitions from a component that are not allowed by its neighboring components. Since the state-based abstraction often introduces a lot of extra traces, which may then cause a large number of false failures, it is not considered in this experiment as the region-based reduction does not generate extra traces. The results are shown in Table 5.6. By comparing Table 5.6 and the results for Method 2 and 3 in Table 5.5, it can be seen that runtime is reduced significantly for DME examples for Method 3. This reduction is because extra state transitions may cause a subgraph of a SG not to be able to form a region as shown in Figure 3.9, and by removing these extra state transitions more regions can be exposed, therefore more reductions can be obtained. This is why applying the SG refinement generally decreases runtime and memory usage. On the other hand, the SG refinement itself is quite expensive, and applying it may also increase runtime slightly in some cases, for example PC. The refinement helps a little for Method 2 in that it can finish a few more examples, while the majority of the examples still cannot be handled. This is due to the nature of the transition-based abstraction such that it can introduce a large number of redundant paths into the reduced SGs therefore causing the size of the intermediate SGs to blow up during the composition process. From this experiment, it can be seen that removing the extra states and transitions early can have big positive impact on the performance mainly for Method 3 by the region-based reduction become more effective after the extra states and transitions are removed.

Table 5.5. Comparison of different abstraction/reduction methods.

Design	$\#M$	$\|\mathcal{A}\|$	Method 1 [99]					Method 2 [96]					Method 3				
			Time	Mem	$\|S\|$	$\|R\|$	π	Time	Mem	$\|S\|$	$\|R\|$	π	Time	Mem	$\|S\|$	$\|R\|$	π
DME	6	132	3.7	3	329	1100	Y	–	–	–	–	–	19.8	4	585	2154	N
	7	154	3.9	3	329	1100	Y	–	–	–	–	–	51.7	6	1329	5340	N
	8	176	4.5	4	329	1100	Y	–	–	–	–	–	83.5	8	1329	5340	N
	9	198	5.1	4	329	1100	Y	–	–	–	–	–	170	9	1881	7580	N
	10	220	5.8	5	329	1100	Y	–	–	–	–	–	226	11	1881	7590	N
ARB	11	180	2.3	2	264	490	Y	–	–	–	–	–	5.1	2	264	490	N
	13	216	2.8	3	350	1066	Y	–	–	–	–	–	6.2	3	350	1066	N
	15	244	3.6	4	113	1071	Y	–	–	–	–	–	7.6	4	350	1066	N
	31	500	10	11	409	1271	Y	–	–	–	–	–	18.9	11	409	1271	N
	63	1012	36.3	19	409	1271	Y	–	–	–	–	–	44.3	29	409	1271	N
FIFO	10	84	1.1	1	41	120	Y	2.4	1	145	295	N	1.8	1	51	118	N
	50	404	4.6	6	51	116	Y	7.1	8	51	118	N	7.2	8.8	51	118	N
	100	804	12.7	18	57	162	Y	17.8	23	51	118	N	21.6	23	51	118	N
TU	3	96	6.6	15	3601	19100	N	–	–	–	–	–	167	14	3601	19100	N
PC	5	100	7.3	6	1473	8112	Y	–	–	–	–	–	27.8	21	6777	21260	Y
MMU	11	118	144	289	53162	447410	Y	–	–	–	–	–	982	230	53618	536568	N

– indicates that the example times out.

As pointed out earlier, the largest SGs encountered in any step of the compositional minimization process determines the success or failure of the entire process. In all experiments, the SGs initially extracted from the high-level descriptions are often very large due to the maximal environment where the inputs are totally free. Using abstract environment for components to generate the initial SGs may cause a lot of extra states and state transitions that can lead to a significant increase in the overall complexity of the whole compositional minimization process by making the reductions less effective. Additionally, composing the larger SGs also becomes more difficult. In another experiment, the reduction method is applied to examples with regular structures where larger partitions are used, and the results are shown in Table 5.7. In this experiment, DME with six cells, ARB with eleven cells, and FIFO with ten cells are selected. DME is partitioned into three components, each of which has two cells. ARB is partitioned into four components, one of which has two cells while each of the other three has three cells. FIFO is partitioned into five components, each of which has two cells. From Table 5.7, it can be seen that both runtime and memory usage increase significantly, and the size of the largest SGs has also gone up dramatically. In fact, the largest SGs are generated when they are extracted from the high level descriptions. This indicates that large partitions combined with the maximal environment can really cause excessive increase in complexity. Also in Table 5.5, even though the sizes of MMU and PC are close, verifying MMU takes significantly more time and memory. This is because the components in PC have simple and limited interfaces, while each component in MMU generally has many more inputs. Therefore, the abstract environments result in much larger SGs compared to PC in every step of the entire compositional minimization process. This indicates the need to have stronger initial constraints on inputs to avoid generating large SGs from the beginning thus making the whole process more efficient.

Table 5.6. Results from using refinement for method 2 and 3.

Design	$\#M$	Method 2 with refinement					Method 3 with refinement				
		Time	Mem	$\|S\|$	$\|R\|$	π	Time	Mem	$\|S\|$	$\|R\|$	π
DME	6	179	107	18567	223100	N	9.2	3	329	1100	N
	7	–	–	–	–	–	11.2	3	329	1100	N
	8	–	–	–	–	–	12.5	4	329	1100	N
	9	–	–	–	–	–	14.9	4	329	1100	N
	10	–	–	–	–	–	17.7	5	329	1100	N
ARB	11	50.3	22	2574	15123	N	4.6	2	264	490	N
	13	236	129	20631	190799	N	5.7	3	350	1066	N
	15	–	–	–	–	–	7.1	4	350	1066	N
	31	–	–	–	–	–	18.3	11	409	1271	N
	63	–	–	–	–	–	43.3	28	409	1271	N
FIFO	10	3	1	51	118	N	1.6	1	51	118	N
	50	9.1	8	51	118	N	9.6	8	51	118	N
	100	21.1	21	51	118	N	24.7	21	51	118	N
TU	3	–	–	–	–	–	167	14	3601	19100	N
PC	5	–	–	–	–	–	36.2	11	1473	8112	Y
MMU	12	–	–	–	–	–	819	230	53618	536568	N

– indicates that the example times out.

Table 5.7. Results from using larger partitions.

Design	$\#M$	Time	Mem	$\|S\|$	$\|R\|$	π
DME-6	3	101	29	7379	30257	N
ARB-11	4	45.8	16	3247	15611	N
FIFO-10	5	11.2	6	475	1435	N

CHAPTER 6
CONCLUSIONS AND FUTURE WORK

Model checking has become a very important alternative to simulation for verifying complex concurrent systems. However, the state space explosion problem limits it to small designs, a serious barrier which prevents its widespread acceptance. On the other hand, large complex systems are often naturally structured, and allow the divide-and-conquer approaches to be used to attack the complexity. Following this thought, compositional verification breaks a system into components, verify each component locally, and compose the results of local verification to form the conclusion for the whole system.

While compositional verification in general is effective at attacking state explosion in model checking, finding accurate yet simple environments for system components poses a big challenge for compositional verification to be practical. In the past, this step usually requires nontrivial manual effort, which is often very tedious and can easily introduce errors into the verification process. In recent years, there have been successes in using machine learning algorithms for computing environment assumptions automatically, and automated compositional reasoning becomes feasible to a certain extent. However, compositional verification based on learning algorithms often requires complex reasoning rules, and its efficiency can be sensitive to system structures, and learning algorithms sometimes generate environment assumptions that are too complex for model checking to succeed.

This dissertation addresses the problem of finding a suitable component environment for a component by proposing two abstraction refinement methods where an initial very coarse component environment can be gradually refined into a smaller and more precise one. These methods are fully automated, and they examine the interface interactions among different components and remove the state transitions not allowed on the components' interfaces from their corresponding environments. Unlike the approaches based on the learning algorithms, the abstraction refinement methods developed in this dissertation refine the over-approximate component environments monotonically.

For local properties that are defined on individual components, compositional verification with the above abstraction refinement methods can be very efficient and scaled up to very large systems. However, if the properties to be verified such as deadlock freedom or liveness properties are global, they have to be checked on the state space of the whole system. Because of the state explosion problem, generating the full state space needs to be avoided. For this purpose, several new state space reduction techniques are developed to support compositional minimization. These reductions, compared to the previous techniques, are complete and sound in that the verification results drawn from the final reduced state space are the same as those drawn from the full state space of the whole system. This brings a huge benefit as any counter-example found in the reduced state space is real, therefore saving the step of checking the truth of the found counter-examples, which can be very costly. With these reductions, iteratively component state spaces are reduced, and then composed until all components are composed. With the careful selection of the orderings of how components are composed, the complexity of the intermediate results during compositional minimization can be contained, thus making the entire verification process be highly efficient.

Although the compositional verification framework integrated with all the reduction techniques and abstraction refinement methods developed in this dissertation has been used successfully to verify a number of large asynchronous circuit designs, there are a few of open problems that need to be addressed in the future. First, automated techniques for decomposing systems for efficient compositional verification need to be investigated. Currently, our framework assumes that a given system is well structured and does not require further decomposition. However, system decomposition has significant impact on the performance of the compositional verification. It is highly desirable that a system is decomposed into a set of components of similar complexity and with simple and constrained interfaces. This would put the complexity of the initial state space abstraction for each component under control, and also reduce the complexity of the intermediate results during the compositional minimization process. An initial approach has been proposed by Nam and Alur [76] based on using hyper-graph partitioning algorithms.

Second, the front-end parser needs to be improved so that the framework developed in this dissertation can be applied to different examples than asynchronous designs. The current parser only supports Boolean variables and operations. New data types and language constructs such as integers, arrays, and arithmetic and relational operations need to be added into the parser so that other types of applications such as communication protocols including cache coherence protocols and multithreaded/distributed programs can be verified in this framework.

Third, it is necessary to investigate the abstraction techniques at a higher level to generate more constrained initial environment for better performance. Since the dissertation does not focus on how the initial component environments are obtained, the maximal environment is used as a simplification for experiment setup and also used for demonstrating the efficacy of the proposed abstraction refinement methods. On the other hand, if a more constrained initial approximate environment can be obtained for each component, the complexity of the component state space can be greatly reduced from the beginning, thus making abstraction refinement and state space reductions much more efficient. Some previous work [98, 93] has been done in this direction.

Fourth, an approach to reconstructing the concrete counter-examples needs to be developed. When a counter-example is found during the local verification on individual components or during the verification on the reduced global state space, it is usually very abstract with a lot of details abstracted away. This abstract counter-example does not return much useful feedback to the users for debugging. Another motivation is that the counter-example found during the local verification needs to be checked to determine its truth. Finding the concrete correspondent for the local counter-example can make determining its truth much easier.

Finally, it would be highly interesting to investigate how the methods developed in this dissertation apply to real-time and hybrid systems. Verification of real-time and hybrid systems is much more difficult than verifying discrete concurrent systems as large amount of additional information including timing and continuous dynamics needs to be represented in each state. For discrete concurrent systems, compositional verification considers system decomposition along a system's structure. For real-time and hybrid system, it is necessary to investigate if decomposition should be applied along the boundaries of different semantic domains in addition to system structures. As embedded systems integrating discrete and continuous components together are becoming more widely deployed in safety-critical applications, ensuring their correctness demands a lot of research efforts.

REFERENCES

[1] Jpf wiki. `http://babelfish.arc.nasa.gov/trac/jpf/`.

[2] Nusmv user manual. `http://nusmv.fbk.eu/NuSMV/userman/index-v2.html`.

[3] Spin user manual. `http://spinroot.com/spin/Man/Manual.html`.

[4] J. Ahrens. A compositional approach to asynchronous design verification with automated state space reduction. Master's thesis, University of South Florida, 2007.

[5] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *POPL 2005*, pages 98–109, 2005.

[6] R. Alur, L. de Alfaro, T. Henzinger, and F. Mang. Automating modular verification. In *Proceedings of the 10th International Conference on Concurrency Theory*, volume 1664 of *LNCS*, pages 82–97. Springer-Verlag, 1999.

[7] R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. Mocha: modularity in model checking. In *Proc. Int. Conf. on Computer Aided Verification*, volume 1427 of *LNCS*, pages 521–525. Springer-Verlag, 1998.

[8] R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *Proc. Int. Conf. on Computer Aided Verification*, volume 3576 of *LNCS*, pages 548–562. Springer-Verlag, 2005.

[9] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, Nov. 1987.

[10] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c program. In *Proceedings the PLDI'01*, volume 36 of *SIGPLAN Notices*, pages 203–213. ACM Press, June 2001.

[11] M. Ben-Ari. A primer on model checking. *ACM Inroads*, 1:40–47, 2010.

[12] S. Berezin, S. Campos, and E. M. Clarke. Compositional reasoning in model checking. In *COMPOS 98*, volume 1536 of *LNCS*, pages 81–102. Springer-Verlag, Sept. 1998.

[13] M. Bobaru, C. S. Pasareanu, and D. Giannakopoulou. Automated assume-guarantee reasoning by abstraction refinement. In *Proc. Int. Conf. on Computer Aided Verification*, volume 5123 of *LNCS*, pages 135–148. Springer-Verlag, 2008.

[14] M. G. Bobaru, D. Giannakopoulou, and C. S. Pasareanu. Refining interface alphabets for compositional verification. In *Proc. Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 4424 of *LNCS*, pages 292–307. Springer-Verlag, 2007.

[15] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[16] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond. In *Proceedings of 5th Annual IEEE Symposium on Logic in Computer Science*, pages 1–33. IEEE Computer Society Press, 1990.

[17] S. Cheung and J. Kramer. Enhancing compositional reachability analysis with context constraints. In *Proceedings of the 1st ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 115–125, 1993.

[18] S. Cheung and J. Kramer. Compositional reachability analysis of finite-state distributed systems with user-specified constraints. In *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 140–150, 1995.

[19] S. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Transations on Software Engineering and Methodology*, 5(4):334–377, 1996.

[20] S. Cheung and J. Kramer. Checking safety properties using compositional reachability analysis. *ACM Trans. Softw. Eng. Methodol.*, 8(1):49–78, 1999.

[21] J.-P. K. Christel Baier. *Principles of Model Checking.* The MIT Press, 2008.

[22] E. M. Clarke. The birth of model checking. *25 Years of Model Checking*, pages 1–26, 2008.

[23] E. M. Clarke, E. A. Emerson, and J. Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM (CACM)*, 52(11):74–84, 2009.

[24] E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at ltl model checking. *Formal Methods in System Design*, 10(1):47–71, 1997.

[25] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Intl. Conf. on Computer Aided Verification*, pages 154–169, 2000.

[26] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *JACM*, 50(5):752–794, 2003.

[27] E. M. Clarke, O. Grumberg, and D. E. Long. Verification tools for finite-state concurrent systems. In *REX School/Symposium 1993*, pages 124–175, 1993.

[28] E. M. Clarke, O. Grumberg, and D. Peled. *Model Cheking.* MIT Press, 1999.

[29] E. M. Clarke, A. Gupta, J. Kukula, and O. Shrichman. Sat based abstraction-refinement using ilp and machine learning techniques. In *Proc. International Workshop on Computer Aided Verification*, pages 265–279, 2002.

[30] E. M. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the 4th Annual Symposium on Logic in computer science*, pages 353–362, Piscataway, NJ, USA, 1989. IEEE Press.

[31] J. Cobleigh, G. Avrunin, and L. Clarke. Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning. In *Proc. the 2006 international symposium on Software testing and analysis*, pages 97–108, New York, NY, USA, 2006. ACM Press.

[32] J. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *Proc. Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 2619 of *LNCS*, pages 331–346. Springer-Verlag, 2003.

[33] A. Cohen, K. S. Namjoshi, and Y. Sa'ar. Split: A compositional ltl verifier. In *Proc. Int. Conf. on Computer Aided Verification*, volume 6174 of *LNCS*, pages 558–561. Springer-Verlag, 2010.

[34] A. Davis and S. M. Nowick. An introduction to asynchronous circuit design. Technical Report UUCS-97-013, Department of Computer Science, University of Utah, Sept. 1997.

[35] L. de Alfaro and T. Henzinger. Interface automata. *Foundations of Software Engineering*, pages 109–120, 2001.

[36] L. de Alfaro and T. henzinger. Interface theories for component-based design. In *Proceedings of the 1st International Workshop on Embedded Software*, pages 148–165, Oct 2001.

[37] L. de Alfaro and T. Henzinger. Interface-based design. *Engineering Theories of Software-intensive Systems*, 195:83–104, 2005.

[38] D. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed Independent Circuits*. PhD thesis, Carnegie Mellon University, 1988.

[39] E.Mercer. *Correctness and Reduction in Timed Circuit Analysis*. PhD thesis, University of Utah, 2002.

[40] E. A. Emerson. The beginning of model checking: A personal perspective. *25 Years of Model Checking*, pages 27–45, 2008.

[41] L. Fix and O. Grumberg. Verification of temporal properties. *Logic and Computation*, 6(3):343–361, 1996.

[42] C. Flanagan and S. Freund. Thread-modular verification for shared-memory programs. In *Proceedings of ESOP*, pages 262–277, 2002.

[43] D. Giannakopoulou and C. S. Pasareanu. Learning-based assume-guarantee verification (tool paper). In *Spin 2005*, pages 282–287, 2005.

[44] D. Giannakopoulou and C. S. Pasareanu. Interface generation and compositional verification in javapathfinder. *FASE 2009*, 5503:94–108, 2009.

[45] D. Giannakopoulou and C. S. Pasareanu. Context synthesis. *SFM*, pages 191–216, 2011.

[46] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *ICSE 2004*, pages 211–220, 2004.

[47] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Component verification with automatically generated assumptions. *Automated Software Engineering*, pages 297–320, 2005.

[48] D. Giannakopoulou, C. S. Pasareanu, and C. Blundell. Assume-guarantee testing for software components. *IEF Software*, 2(6):547–562, 2008.

[49] D. Giannakopoulou, C. S. Pasareanu, and J. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *ICSE 2004*, pages 211–220, 2004.

[50] P. Godefroid. Partial-order methods for the verification of concurrent systems: An approach to the state-explosion problem. volume 1032 of *LNCS*. Springer-Verlag, 1996.

[51] S. Graf and B. Steffen. Compositional minimization of finite state systems. In *Proc. Int. Conf. on Computer Aided Verification*, volume 531, pages 186–196, 1991.

[52] S. Graf, B. Steffen, and G. Luttgen. Compositional minimization of finite state systems using interface specifications. *Formal Aspects of Computation*, 8(5):607–616, 1996.

[53] O. Grumberg and D. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[54] C. S. P. H. Barringer, D. Giannakopoulou. Proof rules for automated compositional verification through learning. In *Proceedings of the Second Workshop on Specification and Verification of Component-based Systems*, pages 14–21, 2003.

[55] T. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Proc. Int. Conf. on Computer Aided Verification*, volume 2725 of *LNCS*, pages 262–274. Springer-Verlag, 2003.

[56] T. Henzinger, S. Qadeer, and S. Rajamani. You assume, we guarantee: methodology and case studies. In *Proc. Int. Conf. on Computer Aided Verification*, pages 440–451. Springer, 1998.

[57] T. Henzinger, S. Qadeer, and S. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In *International Conference on Computer-Aided Design, ICCAD*, pages 245–252. IEEE Computer Society, 2000.

[58] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 41(4), 2009.

[59] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of the IFIP 9th World Congress*, pages 321–332, 1983.

[60] M. Kishinevsky, J. Cortadella, and A. Kondratyev. Asynchronous interface specification, analysis and synthesis, 1998.

[61] T. Kitai, Y. Ogguro, T. Yoneda, E.Mercer, and C.Myers. Partial order reduction for timed circuit verification based on level oriented model. *IEICE Transactions on Information and Systems*, E86-D(12):2601–2611, 2003.

[62] A. Kondratyev, J. Cortadella, M. Kishinevsky, E. Pastor, O. Roig, and A. Yakovlev. Checking Signal Transition Graph implementability by symbolic BDD traversal. In *Proc. European Design and Test Conference*, pages 325–332, Paris, France, Mar. 1995.

[63] A. Kondratyev, M. Kishinevsky, A. Taubin, J. Cortadella, and L. Lavagno. The use of petri nets for the design and verification of asynchronous circuits and systems. *Circuits Systems and Computers*, 8(1):67–118, 1998.

[64] F. Lang. Refined interface for compositional verification. In *FORTE'06: Formal Techniques for Networked and Distributed Systems*, volume 4229 of *LNCS*, pages 159–174. Springer Verlag, 2006.

[65] K. Laster and O. Grumberg. Modular model checking of software. In *TACAS 98*, pages 20–35, 1998.

[66] M. Leucker. Learning meets verification. In *In FMCO06*, pages 127–151, 2006.

[67] J. Magee and J. Kramer. *Concurrency: state models and Java programs.* John Wiley and Sons, 1999.

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

[69] K. L. McMillan. *Symbolic model checking: an approach to the state explosion problem.* Kluwer Academic publisher, 1993.

[70] K. L. McMillan. A compositional rule for hardware design refinement. In *Proc. Int. Conf. on Computer Aided Verification*, volume 1454 of *LNCS*, pages 24–35. Springer-Verlag, 1997.

[71] K. L. McMillan. Circular compositional reasoning about liveness. In *CHARME'1999*, volume 1703 of *LNCS*, pages 342–346. Springer-Verlag, 1999.

[72] K. L. McMillan, S. Qadeer, and J. Saxe. Induction in compositional model checking. In *Proc. Int. Conf. on Computer Aided Verification*, volume 1855 of *LNCS*, pages 312–327. Springer-Verlag, 2000.

[73] B. Metzler. Decomposing integrated specifications for verification. *IFM 2007*, pages 459–479, 2007.

[74] B. Metzler, H. Wehrheim, and D. Wonisch. Decomposition for compositional verification. *ICFEM 2008*, pages 105–125, 2008.

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

[76] W. Nam and R. Alur. Learning-based symbolic assume-guarantee reasoning with automatic decomposition. In *Proc. Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 4218 of *LNCS*, pages 170–185, 2006.

[77] W. Nam, P. Madhusudan, and R. Alur. Automatic symbolic compositional verification by learning assumptions. *Formal Methods in System Design*, 32(3):207–234, 2008.

[78] K. Namjosi and R. Trefler. On the completeness of compositional reasoning. In *Proc. Int. Conf. on Computer Aided Verification*, volume 1855 of *LNCS*, pages 139–153. Springer-Verlag, 2000.

[79] P. Parizek, J. Adamek, and T. Kalibera. Automated construction of reasonable environment for java components. *Electr. Notes Theor. Comput. Sci.*, 253(1):145–160, 2009.

[80] P. Parizek and F. Plasil. Assume-guarantee verification of software components in sofa 2 framework. *IET Software*, 4:210–229, 2010.

[81] C. S. Pasareanu and D. Giannakopoulou. Towards a compositional spin. In *Spin*, volume 3925 of *LNCS*, pages 234–251. Springer-Verlag, 2006.

[82] C. S. Pasareanu, D. Giannakopoulou, and M. Bobaru. Learning to divide and conquer: applying l algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 32(3):175–205, 2008.

[83] E. Pastor, O. Roig, J. Cortadella, and R. M. Badia. Petri net analysis using boolean manipulation. In *Proceedings of the 15th International Conference on Application and Theory of Petri Nets*, pages 416–435. Springer-Verlag, 1994.

[84] D. Peled. Ten years of partial order reduction. In *Proc. Int. Conf. on Computer Aided Verification*, volume 1427 of *LNCS*, pages 17–28. Springer-Verlag, 1998.

[85] A. Pnueli. In transition from global to modular temporal reasoning about programs. *Logics and Models of Concurrent Systems*, 13:123–144, Oct 1984.

[86] A. Pnueli, Y. Sa'ar, and L. D. Zuck. Jtlv: A framework for developing verification algorithms. In *Proc. Int. Conf. on Computer Aided Verification*, volume 6174 of *LNCS*, pages 171–174. Springer-Verlag, 2010.

[87] W. Roever, H. Langmaack, A. Pnueli, and Eds. Compositionality: The significant difference. In *COMPOS 97*, volume 1536 of *LNCS*. Springer-Verlag, Sept. 1998.

[88] R. Singh, C. S. Pasareanu, and D. Giannakopoulou. Learning component interfaces with may and must abstractions. In *Proc. Int. Conf. on Computer Aided Verification*, volume 6174 of *LNCS*, pages 527–542. Springer-Verlag, 2010.

[89] M. Sirjani, A. Movaghar, A. Shali, and F. Boer. Model checking, automated abstraction, and compositional verification of rebeca models. *Journal of Universal Computer Science*, 11:1054–1082, 2005.

[90] M. Sirjani, A. Movaghar, A. Shali, and F. Boer. Modular verification of a component-based actor language. *Journal of Universal Computer Science*, 10:1695–1717, 2005.

[91] S.Ling and H. W. Schmidt. Using a notion of safety in petri nets to analyse real-time systems. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.146.5641`.

[92] K. Stevens, R. Ginosar, and S. Rotem. Relative timing. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 208–218, 1999.

[93] R. Thacker, K. Johns, C. Myers, and H. Zheng. Automatic abstraction for verification of cyber-physical systems. In *International Conference on Cyber-Physical Systems*, July 2010.

[94] T.Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, volume 77, pages 541–580, 1989.

[95] A. Valmari. The state explosion problem. In *Petri Nets*, volume 1491 of *LNCS*, pages 429–528. Springer-Verlag, 1996.

[96] H. Yao and H. Zheng. Automated interface refinement for compositional verification. *IEEE Transaction on Computer-aided Design of Integrated Circuits and Systems*, 28(3):433–446, 2009.

[97] T. Yoneda and T. Yoshikawa. Using partial orders for trace theoretic verification of asynchronous circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, Mar. 1996.

[98] H. Zheng. *Modular Synthesis and Verification of Timed Circuits Using Automatic Abstraction*. PhD thesis, University of Utah, 2001.

[99] H. Zheng, J. Ahrens, and T. Xia. A compositional method with failure-preserving abstractions for asynchronous design verification. *IEEE Transactions on COMPUTER-AIDED DESIGN of Integrated Circuits and Systems*, 27:1343–1347, 2008.

[100] 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*, 25(3):403–412, 2006.

[101] H. Zheng, H. Yao, and T. Yoneda. Synchronization-based abstraction refinement for modular verification of asynchronous designs. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, May 2009.

[102] H. Zheng, H. Yao, and T. Yoneda. Modular model checking of large asynchronous designs with efficient abstraction refinement. *IEEE Transactions on Computers*, 59(4):561–573, 2010.