Bounded Model Checking for Asynchronous Concurrent Systems

Manitra Johanesa Rakotoarisoa

Department of Computer Architecture
Technical University of Catalonia

Advisor: Enric Pastor Llorens

Thesis submitted to obtain the qualification of Doctor from the Technical University of Catalonia

To my mother

Contents

List of Figures

List of Tables

List of Algorithms

Nomenclature

BA	Büchi automata
BDD	Binary decision diagrams
BMC	Bounded model checking
CAD	Computer-aided design
CNF	Conjunctive normal form
CTL	Computation tree logic
DPLL	Davis-Putnam-Longeman-Loveland, name of an algorithm for solving SAT
FSM	Finite state machine
LTL	Linear temporal logic
NuSmv	A model checking tool that has BMC capabilities
PN	Petri net
SAT	Boolean satisfiability problem
TGBA	Transition-based generalized Büchi automata
TS	Transition system
ZChaff	A SAT solver

Abstract

In this thesis we study the verification of asynchronous concurrent systems using a symbolic model checking technique called *bounded model checking* (BMC). BMC is a method targeted mainly at finding bugs in a system. It answers the question whether there exists an execution path, shorter than a given number, that violates a given property. Such an execution path is known as *counterexample*. During a BMC operation each execution path is encoded into a Boolean formula, and the problem is reduced to satisfiability checking of the formula. Therefore, the operation consists mainly in constructing a Boolean formula that is satisfiable if and only if such a counterexample exists.

We model our systems with *transition systems* (TSs). In particular, we are mainly interested in synchronized product of TSs. Since concurrent systems are formed by a combination of several components communicating between each other, synchronized product of TSs is well-suited to capture the behavior of such systems. The executions of concurrent systems are commonly modeled using the so-called *interleaving execution*, which allows only one single event to fire at each step. However, due to the complexity of such systems, performing BMC with interleaving will not only require many steps but also generate long formulas. In this work, we adopt different approaches based on *breadth-first search* (BFS). Our methods reduce the necessary steps, and produce smaller formulas. In a BMC operation, the translation of the model into a Boolean formula is polynomial in the size of the model, but the solving time of the Boolean formula can be exponential in the size of the formula. Therefore, our research hypothesis is that we can improve the efficiency of BMC by generating succinct formula, and by minimizing the number of necessary steps during an execution.

We introduce several BMC techniques aimed at improving the efficiency of BMC for asynchronous concurrent systems. The techniques are grouped in two main parts (i) techniques for checking reachability properties and (ii) techniques for checking other properties written in *linear temporal logic* (LTL). In addition, we also propose some methods for minimizing the number of execution steps or bounds.

We implemented all these methods in a BMC toolset. At the end of the dissertation, we will discuss the experimental results we obtained.

Keywords: bounded model checking, asynchronous concurrent systems, satisfiability problem, reachability properties, linear temporal logic.

Acknowledgments

This thesis would not have been possible without the support of many people. It is a pleasure for me to thank all of them.

First, I would like to express my sincere gratitude to my supervisor Enric Pastor who offered invaluable assistance, support, and guidance throughout my work. With his great efforts to explain things clearly and simply, he helped make formal verification fun for me. He also provided encouragement and good company. I would have been lost without him.

I am indebted to all my colleagues at the Computer Architecture Department for providing a stimulating and pleasant working environment. Especially, I am grateful to Marc Solé, Josep Carmona, and Juan Lopez who spent a great deal of their time discussing with me, and gave sound advice that contributed in many ways to the accomplishment of this research project.

Special thanks to the external reviewers, Marco Peña and Robert Viladrosa who carefully read the preliminary version of this report and provided valuable suggestions and comments. Thanks also to all the members of the jury for evaluating and judging my thesis.

I would like to show my gratitude to the following institutions for funding this research: the Government of Catalonia under the scholarship FI, and the Ministry of Science and Education of Spain under the contract CICYT TIN 2007-63927.

I wish to thank all my friends, especially Nirina Rakotoarivelo, Rafael Ordonez, Manitra Raharijaona, and Fidel Pedregal Pimentel. They encouraged me and supported me from the beginning until the end, despite the long period of time I spent doing my Ph.D. thesis.

I would also like to thank my entire family for their unwavering support. Most importantly, I am very thankful to my mother, Marguerite Ravaoarivelo. She bore me, raised me, taught me, and has loved me throughout the years. To her I dedicate this thesis.

Last but not least, I would like to express my gratitude to all the people who have not been cited above but who helped me directly or indirectly with the realization of this thesis. Thank you so much everyone.

Chapter 1

Introduction

Complex hardware systems become more and more ubiquitous in mission critical applications such as military, satellite, and medical to name but a few. In such applications, reliability remains a primary concern because a failure that occurs during their normal operations might produce important catastrophes like loss of life or loss of money. Examples of these catastrophes include the Pentium bug [Gep95, Ade97], space launch disasters like the Mariner I [HL05] or the Ariane 5 [Har03], the Warsaw A320 crash [Lad95, CMMM95], not to mention the AT&T telephone switching failure [Ste93, Har00].

All these failures are often caused by minuscule *bug*[1] that exists inside the software which controls the systems, or within the hardware itself. In addition, most of these systems cannot be interrupted while working, even for a few seconds a year, making it difficult to repair bugs found during their normal operations.

Therefore, manufacturers of such systems have to validate their designs, before shipping them to ward off disasters caused by undetected errors. Appropriate validation techniques are then necessary to discover bugs that would affect the use of such systems.

In general, design validation can be performed using two groups of methods: (i) simulation and testing (ii) formal verification.

In *simulation and testing*, the main purpose is to carry out some experiments before getting the product on the market. This involves introducing series of inputs into the design and checking whether the produced outputs correspond to the expected ones. Simulation and testing explore only some system behaviors and they take time to find the slightest bugs in a system. Hence, they can overlook considerable errors if the system has

[1] *Bug* is a computer jargon referring to an undesirable property that makes the computer misbehave or crash. The first recorded bug was a real insect. In 1947, American mathematician and computer scientist Grace Murray Hopper (1906-1992), known also for inventing the programming language COBOL, was working on the Harvard University's Mark II computer. One day, the machine crashed. Hopper investigated and found that a moth has squeezed into one of the 17,000 relays in the machine. She removed the moth and the machine worked perfectly [Kid98, Nor05]. This is not, however the origin of the term "bug" since this term has already been used long before by other scientists, but Hopper's was the "first actual case of a bug being found" according to a note she put on her log book.

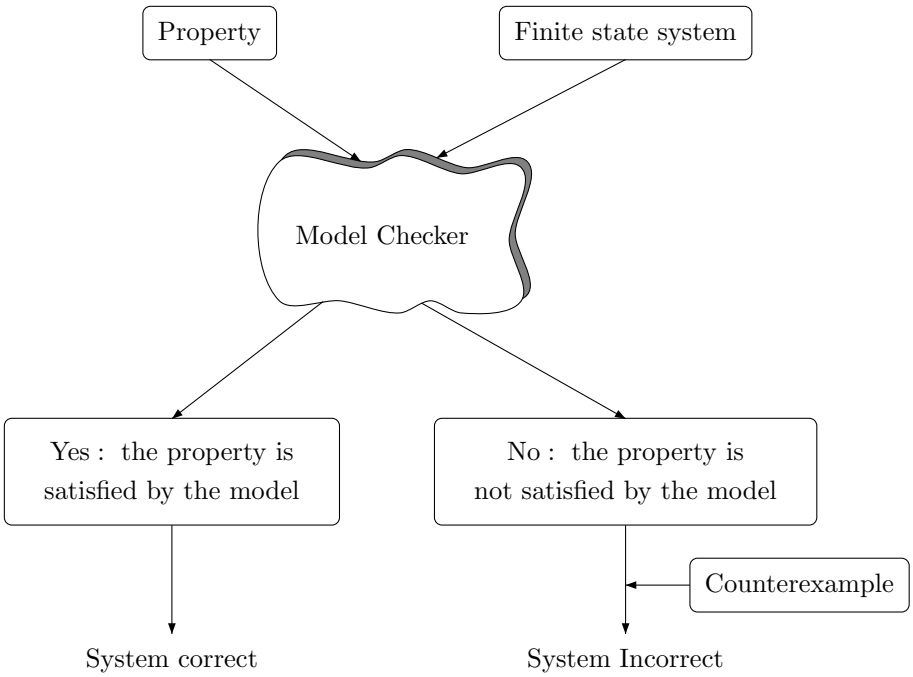


Figure 1.1: Model Checking.

huge state number. Nonetheless, many industries nowadays still use these techniques as standard ways to validate their designs.

Formal verification [Das06, Kro99], on the other hand, does not rely on direct experiments. Instead, it uses mathematical formalisms to check whether the *model* of a system satisfies a given *property*. The system is correct if this property is satisfied, otherwise it is incorrect. Formal verification offers exhaustive coverage of all system behaviors, which makes it more suitable for detecting bugs in extremely concurrent systems. There exist two main approaches to formal verification. Namely, theorem proving and model checking.

Theorem proving is a more theoretical approach. It uses mathematical axioms and proof rules to prove the correctness of a system. Hence, it can only be performed by people who have solid experience in logical reasoning. In this method, both the system and the desired property are expressed as formulas in some mathematical logic. The system is correct if a proof for the property can be constructed from the axioms of the system. The whole process is rather time-consuming, yet it is proven to be more effective for detecting software bugs in case the source code contains thousands of lines with complex logical operations [Ros05]. In addition, it has the ability to verify infinite state systems.

Model checking [CGP99, CK96, QS82] is used to verify *finite state systems*. It is fully automatic, terminates with *true* if the system is correct, or alternatively provides an execution path known as *counterexample* that shows how the system violates the given

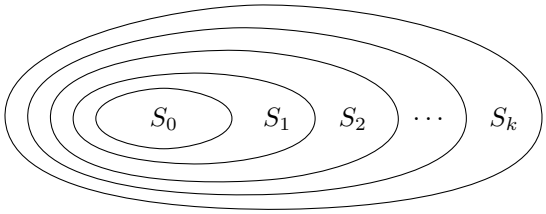


Figure 1.2: Image computation.

property (see Figure 1.1). Counterexamples often correspond to subtle design errors and can therefore be used for debugging the system. In *symbolic model checking* [BCM+90], the system's state space is represented symbolically with Boolean encoding rather than with explicit representation.

1.1 Symbolic Model Checking

1.1.1 BDD-based approach

Binary decision diagram (BDD) [Bry86], which offers a compact and canonical representation of functions, is traditionally used in symbolic model checking to represent the system's state space. Numerous studies, coupled with successful practical experiments, have been done so far to exploit the performance of this technique. In particular, the model checking tool called *system model verifier* (SMV) [McM93], developed by McMillan in 1993, was one of the efficient tools which showed the potentiality of BDD-based approaches, and became a touchstone for other implementations. This tool has been successfully used to verify industrial designs, see e.g. [CYLR01, BLP95].

In general, BDD-based model checking relies on the approach known as *reachability analysis* which involves computing all the reachable states in the system starting from a given set and analyzing the results. Basically, there exist two fundamental variants of this approach: *forward* and *backward* methods.

Forward Reachability Analysis

In forward reachability analysis, the goal is to determine whether a state violating an invariant property can be reached from the initial states. To achieve this goal, a reachability procedure performs repeatedly *image computation*. That is, it computes the image of the newly reached states from one iteration using a transition relation, and repeat this process until no more new states can be generated. In this case, a *least fixpoint* (lfp) is reached and the procedure stops (see Figure 1.2). The transitions and the states are represented using BDDs. The reached states are analyzed later on to determine whether any of them violates the property of interest. Formally, these images are computed as follow. Let S_0 be a set of initial states and $T(v, v')$ be a transition relation, where v and v'

denotes the current and next state variables respectively. Let us use $S(v)$ and $T(v, v')$ to denote the propositional formulas related to the set of states and transitions respectively. Then, the image of $S(v)$ through $T(v, v')$ is given as:

$$Img(S(v')) = \exists v.T(v, v') \wedge S(v) \tag{1.1}$$

Now, starting from the initial states, the least fixed point can be computed by repeatedly applying the following equation in which the final results of the set R contains all forward reached states:

$$R = lfp\ R.(S_0 \vee Img(R)) \tag{1.2}$$

For instance, assume we want to check whether a system satisfies a safety property expressing that something should be true at every state of the system. In this case, the model checker applies repeatedly image computation to determine if there is an execution path from an initial state to a state where this property fails. If such a path does not exist, then the system is correct, otherwise the system is incorrect and the path found by the model checker corresponds to a counterexample.

Algorithm 1.1 illustrates the forward reachability procedure. The set R contains all reachable states, while S_i contains the newly discovered states in each iteration. The function `img()` perform the image computation from S_i. The algorithm stops when S_i is empty.

Algorithm 1.1: forwardReachability

input : S_0 a set of initial states
output: R a set of reachable states

1 $R \longleftarrow \varnothing$;
2 $i \longleftarrow 0$;
3 **while** *($S_i \neq \varnothing$)* **do**
4 $\quad R \longleftarrow R \cup S_i$;
5 $\quad S_{i+1} \longleftarrow$ `img(`S_i`)` $\setminus R$;
6 $\quad i \longleftarrow i + 1$;
7 **end**
8 **return** R

Note that, for large systems, the final set R may become too big. To avoid this situation, some people perform *on-the-fly* check of the invariant property. That is, at each iteration the newly reached states is checked whether any of them violates the property. If the answer is positive, then the algorithm stops. This method prevents exploration through the entire state space.

Backward Reachability Analysis

Backward reachability is mainly used to perform diagnostic of a failure in a system. As its name suggests, it is the opposite of forward reachability. The aim in here is to know whether a specified set of target (i.e violating) states is reachable from any of the initial states. To do so, the procedure starts from the set of violating states and perform *preimage computation* repeatedly until a fixpoint is reached. Then, it checks if the reached set contains one of the initial states. As before, all states and transitions are encoded with BDDs. Formally a preimage computation is performed as follow. Let $T(v.v')$ be a transition relation and S_t a set of target states. The preimage of $S(v')$ through $T(v, v')$ is given as:

$$PreImg(S(v)) = \exists v'.T(v, v') \wedge S(v') \tag{1.3}$$

and the least fixed point is obtained by repeatedly applying the following equation from the target set:

$$R = lfp\ R.(S_t \vee PreImg(R)) \tag{1.4}$$

The algorithm for backward reachability is similar to Algorithm 1.1. We need only to start from S_t instead of S_0. On-the-fly methods can also be applied to backward reachability in order to handle large systems.

State Explosion Problem

Although successful, BDD-based approaches are subject to the so-called *state explosion problem*, or the situation where the size of the state space grows exponentially beyond the available computer resources. This situation arises if the system being checked has many components that can make transitions in parallel. For that reason, certain types of designs cannot be handled efficiently with BDD-based approaches. Below, we will review briefly some interesting techniques for dealing with the state explosion problem. Namely, *partial order reduction, compositional reasoning, abstraction, induction*, and *symmetry*.

Partial Order Reduction This technique is designed for concurrent systems with communicating processes [GW94, Pel93, WW96, NG02]. With this technique, when concurrent events are independent, it suffices to consider one of them since, in this case, they all lead to the same state. Therefore, during state space exploration, only a subset of the enabled transitions is considered in each state rather than all of them. This decreases the size of the state space while preserving the properties of interest. The chosen subsets are called *ample sets* [God96, HP95], or *stubborn sets* [Val90], or *sleep sets* [God90], or *persistent sets* [God90], or also *stamper sets* [PVK01].

Compositional Reasoning This technique applies to large systems composed of parallel processes. For such systems, it is possible to divide up a given property into local

properties, each of them relates to a specific component of the system. Compositional reasoning methods [CLM89, GS90, SG90] abate the state explosion problem in the following way. Check first whether each system's component satisfies its local property, then make sure the conjunction of all local properties yields the property of interest. If all the above are verified, then the whole system satisfies the given property as well.

When the interconnections between the system's components are too complicated, a variant approach called *assume-guarantee reasoning* [GL94] is more appropriate. In this case, the behavior of one component hinges upon the other components' behaviors. Thus, when verifying one component, the user must make assumptions about the properties of other components. If these *assumed* properties are satisfied, the correctness of the entire system could be verified, in a suitable way, without generating the whole state graph.

Abstraction In this technique, the goal is to simplify the verification process by reducing the model of the system into a much simpler representation [CGL92, DGG97]. In general, the reduction consists in excluding some features of the model, by forbidding some behaviors, for example, or by merging some identical states, or by removing some data. This technique is useful especially when the original model is too large to handle by the verification tool at hand. But doing abstraction may result in a reduced model that no longer has the same behavior as the original one. Hence, the user must ensure beforehand that the chosen abstraction technique will preserve the property of interest.

Induction This method deals with *parametrized systems*, that is systems whose configuration depends on a certain parameter [KM95, CGJ95, WL89]. For instance, a network of n computers, or a mutual exclusion involving n processes. Here, the goal is to generalize the result, i.e. to check whether a property is satisfied by all systems in a given class. This is a hard problem. Still, in some cases, it could be resolved by generating first an *invariant* that represents the behavior of an arbitrary member of the class. Then, use this invariant to check the property of all class members.

Symmetry This technique is relevant for systems containing replicated components. It consists in replacing the system's states with different structure known as *orbit representatives* [CJEF96]. Unfortunately, finding orbit representatives also constitutes a hard problem, hence this method remains impractical in most cases.

1.1.2 SAT-based Approach

Combining model checking with the *Boolean satisfiability problem* (SAT) is another efficient way to avoid the state explosion. Known as *bounded model checking* (BMC) [BCCZ99, CBRZ01, BCC+03], this approach is mainly targeted at finding bugs in a system. More precisely, it consists in finding a counterexample of a fixed bound, say k, for a given property and constructing a Boolean formula that is satisfiable if and only if such a counterexample exists. To achieve this goal, a typical BMC algorithm works in the

following way. It first unrolls the system k times. Then, it combines the unrolled system with the negation of the given property, and encodes the whole as a Boolean formula. Finally, it calls a SAT-solver to check whether this formula is satisfiable. If the SAT-solver returns a satisfying assignment, then the system is incorrect and this assignment corresponds to a counterexample of length k. Otherwise, the SAT-solver can provide a proof of unsatisfiability. The unrolling process is polynomial in the size of the system, whereas the satisfiability check can be exponential in the size of the formula. This technique has also been successfully applied to real-world designs (see e.g. [BCRZ00, BLM01, CFF+01]).

BMC overcomes the state explosion problem because it does not use canonical representation of the state space. Furthermore, it consumes much less space than BDD-based techniques because typical SAT-solvers, used for determining the satisfiability of the constructed formula, require no more than polynomial amount of memory. Hence, it can handle huge systems with thousands of state variables, in contrast to BDD-based approaches which can only be applied to systems with, at best, hundreds of variables.

Apart from overcoming the sate explosion problem, BMC also has the following advantages: (i) The Boolean formula translation from a single unrolling can be replicated without additional analysis, thus it can be done in linear time. (ii) Due to this replication possibility, *incremental learning* is possible and is helpful in many cases. (iii) Due to the advances in SAT-solving techniques, many fast SAT-solvers are now available.

Algorithm 1.2: standardBmcAlgorithm

input : M a system model
$maxL$ maximum BMC length
ϕ a property

output: *true* if a counterexample has been found
$false$ otherwise

```
1 bound ⟵ 1 ;
2 isSat ⟵ false ;
3 while (bound ⩽ maxL) and (isSat = false) do
4     F ⟵ unroll(M, φ, bound) ;
5     isSat ⟵ solve(F) ;
6     bound ⟵ bound + 1 ;
7 end
8 return isSat
```

In practice, a standard BMC algorithm works by increasing the bound progressively to search for a deep counterexample up to a certain maximum number. At each step, the algorithm calls a SAT-solver to determine whether a counterexample exists or not for the current bound. Algorithm 1.2 illustrates this operation. It starts at bound 1 and stops when the maximum BMC length is reached or the formula becomes satisfiable. F corresponds to the formula generated at each bound by the unrolling operation. The

function `solve()` calls the SAT-solver to determine whether F is satisfiable.

The speed of the SAT-solver is crucial during the execution because it affects the performance of the whole BMC operation. When the bound is increased, the number of variables in the formula increases linearly, whereas the solving complexity increases exponentially. For example, when moving from a bound k to k', with $k < k'$, the complexity increases $2^{n.k}$ to $2^{n.k'}$, where n is the number of gates in the circuit. This situation implies that the deeper the exploration goes, the slower the execution.

During a BMC session, if the maximum number is reached without finding any counterexample, no conclusion can be drawn concerning the correctness of the system since the counterexample may exists at a bound beyond the maximum number. This inability to draw conclusion makes BMC somewhat incomplete compared to BDD-based approaches. Nevertheless, many techniques have been proposed so far to work around this completeness issue. We will review some of them in Chapter 3.

The standard BMC method mentioned above is well suited for synchronous systems, a kind of system in which the execution is coordinated by a clock and all variables change together at the same time. But if we apply it to *asynchronous concurrent systems*, a kind of system composed of many components that can communicate and work concurrently, it will fare badly. Due to the inherent interleaved nature of these systems, the number of times we have to invoke a SAT-solver will be much larger—maybe an order of magnitude. In addition, some of them tend to produce large formulas sooner. Therefore, the above-mentioned execution slowdown may appear at an early stage because not only the SAT-solver will be invoked many times, but it may also handle large formulas. Hence, these systems need more elaborate methods for handling them.

1.2 Synchronous Versus Asynchronous Systems

In this section we will discuss the main differences between synchronous and asynchronous systems. We will start with the synchronous ones.

1.2.1 Synchronous systems

The synchronous methodology has been used successfully for the design and implementation of safety-critical embedded systems such as flight control systems in flight-by-wire avionics and antiskidding or anticollision equipment on automobiles. They are also present in other applications such as trains, nuclear plants, and cell phones.

Basically, a synchronous circuit contains the following components:

- circuit inputs,
- gates,
- latches.

The operation in such a circuit is controlled by a global clock. All values in its storage components change simultaneously following the clock signals. During a clock cycle, each circuit input receives a random value *true* or $false$. The output of a gate is formed by a Boolean combination of its inputs. A latch corresponds to a memory unit with one input and one output. The inputs of gates and latches are connected to the inputs and outputs of other gates and latches. Consequently, the output of a latch in a given clock-cycle is a Boolean combination of inputs and outputs of latches in the previous clock-cycle. The behavior of the whole circuit is entirely predictable because each input reaches its final value before the next clock occurs. In practice, some delay may be needed for each logical operation in order to limit the speed at which the system can run. To determine the maximum safe speed, static timing analysis is often used.

Advantages of Synchronous Systems

Here are some of the most important advantages of synchronous systems.

Easier to model Synchronous systems are often modeled with *finite state machine* (FSM). To model concurrency correctly, it is sometimes necessary to use a combination of several FSMs for one system. Since synchronous models are deterministic, it is much more easier to formally reason about the models and check certain properties, especially in case of safety-critical systems.

Easier to design A synchronous circuit can be designed easily in an improvised fashion. For instance, a designer can simply define the combinational logic to compute a given function, and surround it with latches. In addition, nearly all of the most important design-related operations for synchronous systems exist in many of today's computer-aided design (CAD) tools.

Drawbacks of Synchronous systems

In the synchronous paradigm, time is defined as a sequence of instants between which nothing interesting happens. In each instant, some events (inputs) occur in the environment, and a reaction (output) is computed instantly by the modeled design. Therefore, despite of all the advantages we discussed above, reasoning and verification based on synchronous models are meaningful only if a completely synchronous implementation of the whole system is possible, and if we are sure that for the implemented system the reaction time (including internal communications) is negligible compared to the rate of external events. Furthermore, it is practically impossible to model a large system using synchronous models.

1.2.2 Asynchronous systems

Asynchronous systems can be found in many applications, ranging from CD players to avionics systems. Their features meet the necessary requirements for assembling large-scale heterogeneous and scalable systems. Typically, they are constructed by combining several hardware components together to form a correct working system. Communication between these components is assured through *channels*, which are used both to synchronize operations and to pass data. Like their synchronous counterparts, asynchronous systems also use binary signals. However, they do not use a global clock. Instead, their components work concurrently using *handshaking* between them in order to perform the necessary synchronization, communication, and sequencing of operations. This mechanism results in a behavior similar to local clocks that are not in phase.

Advantages of Asynchronous Systems

Asynchronous systems have numerous advantages. We will describe some of the them.

Low power dissipation Basically, the clock in a synchronous circuit has to control all parts of the circuit including those that are not used in the current computation. For instance, even though a floating point unit on a processor may not be used in an ongoing instruction stream, the clock still has to control the unit. By contrast, for asynchronous circuits, the components are activated only when needed, so they make signal transitions only when performing some work or communicating. Because of this feature, they work fast, yet have low power dissipation.

No clock skew Clock skew refers to the difference in arrival times of the clock signal at different parts of the circuit. The absence of a global clock in synchronous circuits prevent them from having clock skew problems.

Less electromagnetic interference Electromagnetic interference (EMI) is a kind of interference caused by the high speed clock switching all over a chip at approximately the same time. Asynchronous systems generate less emission of EMI because they use local handshaking whose signals are not correlated in time.

Easing of global timing issue In systems such as synchronous microprocessors, most portions of the circuit must be carefully optimized in order to obtain the highest clock rate and to achieve better performance. On the other hand, for asynchronous systems, rarely used portions of the circuit can be left unoptimized without affecting system performance.

Easier technology migration Basically, integrated circuits are built using different technologies during their lifetime. For instance, early systems may be built with gate arrays, while later production may migrate to custom ICs. For asynchronous systems, there is no need to migrate all system components in order to achieve a greater performance.

Migrating only the most critical parts is sufficient. In addition, components with different delays can be substituted without altering other elements.

Automatic adaptation to physical properties The delay in a circuit can be affected by many factors including variations in fabrication, temperature, and power-supply voltage. Asynchronous systems can adapt themselves to these different factors, and will run as fast as the current physical properties allow.

Drawbacks of Asynchronous Systems

Asynchronous systems have their drawbacks, nonetheless. First of all, due to their complexity, they are more difficult to design. Next, the ordering of operations is too difficult to handle, especially for complex systems. Finally, there are not too many CAD tools devoted to asynchronous circuits. Many of the most essential operations existing in today's CAD tools such as placement, routing, partitioning, logic synthesis are either need special modifications for asynchronous systems, or are not applicable at all.

1.3 Scope of This Work

The main purpose of this work is to build efficient verification techniques for asynchronous concurrent systems. Because of the pivotal roles these systems assume in a given application, designers of such systems must keep development and maintenance costs under control and meet nonfunctional constraints on the design of the system, such as cost, power, weight, or the system architecture by itself. But most importantly, they must assure their customers as well as the certification authorities that both the design and its implementation are correct. Otherwise, they may end up shipping unsafe systems to the market, and the consequences of this action would be catastrophic. To achieve this goal, designers need efficient methods and tools to assist them in verifying the correctness of the design. These methods should be built on solid mathematical foundations in order to be able to reason formally about the operation of the system.

We focus our study on BMC. We aim at developing BMC methods supported by tools for analyzing the behavior of asynchronous concurrent systems and verify their correctness, considering all the benefits and challenges in this area. BMC is traditionally used for verifying synchronous systems because the characteristics of these systems match well a BMC operation. However, in this work, we apply it to asynchronous systems. The following points explain our choice.

- There exist already numerous BDD-based methods and tools for asynchronous concurrent systems. Most of them have been successfully applied in the industry. Still, the state explosion problem limits the size of the problem they can solve. On the other hand, SAT-based methods do not have this kind of problem, thus can handle much bigger systems.

- With the success story of BMC with synchronous systems, we believe that with a little effort, it is also possible to repeat this success for asynchronous systems. For this reason, we are interesting in building verification methods and tools, based on BMC, for asynchronous systems.

- SAT-solving methods and tools continue to improve. Not only they become faster, but the size of the formulas they can handle continues to grow. Therefore, we believe that even though asynchronous systems are complex, using these state-of-the-art solvers will help to tackle the problems.

To handle asynchronous concurrent systems properly, and to tackle all the BMC problems explained in the previous section, two main related tasks should be considered. First, developing encoding techniques that provide compact Boolean formulas to be solved by a SAT-solver. Second, building efficient unrolling techniques to minimize the bound needed for finding a counterexample.

We propose in this thesis several encoding techniques for systems modeled as *transition systems*: methods for checking reachability properties as well as for checking properties expressed in *linear temporal logic* (LTL). We also propose methods for minimizing the bound during the BMC operation. We have implemented all these methods in a BMC toolset for asynchronous systems.

1.4 Structure of the Thesis

The rest of this thesis is structured as follows. In Chapter 2 we give all the definitions we need throughout the report. In Chapter 3 we review most of the important BMC techniques that exist in the literature. We introduce our approaches in Chapters 4 and 6. In Chapter 7, we describe our BMC toolset. Case studies with experimentation results are shown in Chapter 8. We conclude the thesis in Chapter 9 and give some possible future directions.

Chapter 2

Background

In this chapter, we present the definitions and notations to be used throughout the report. We will first describe the transition system model, along with other popular formalisms for modeling concurrent systems. Next, we will explain what a property is and how to express it formally. After that, we will discuss the satisfiability problem, and review some of the most effective methods for solving it. Finally, we will show, with an example, how to use BMC to verify safety properties.

2.1 Transition Systems

In order to verify the correctness of a system, it is necessary to use an appropriate formalism that can describe clearly the system's behavior. Furthermore, the model should be able to capture the properties of interest. For finite concurrent systems, the behavior is mainly determined by three elements: *states*, *transitions*, and *computations*. A state corresponds to a group of variables needed for describing the system at a particular instant. When an action or event occurs, the system changes from one state to another, and the values of the state before and after such an event determine a transition. A computation or also *execution path* is formed by an infinite sequence of states, in which two consecutive states are connected via a transition. In this section, we discuss a well-known model called *transition system* (TS) [Arn94], which is also the one we use in our study.

2.1.1 Definitions

Definition 2.1.1 (Transition system)
A **transition system** *is tuple* $\mathcal{A} = (S, \Sigma, T, s_0)$ *where*

- S *is a non-empty set of* **states***;*
- Σ *is a non-empty set of* **events***;*
- $T \subseteq S \times \Sigma \times S$ *is a transition relation whose elements are called* **transitions***;*

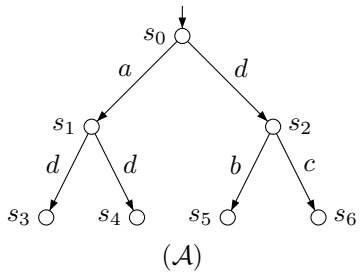


Figure 2.1: A Transition system.

- s_0 *is the* **initial state**.

A transition from a state s to a state s' is written $s \xrightarrow{e} s'$ or (s, e, s') where e denotes the event associated to the transition. ■

A transition system can be represented as a directed graph whose nodes correspond to the states, and the edges to the transitions. The transitions are labeled with their associated events.

Example 2.1.1 Figure 2.1 depicts an example of TS. We have $\mathcal{A} = (S, \Sigma, T, so)$ where $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6\}$; $\Sigma = \{a, b, c, d\}$; $T = \{s_0 \xrightarrow{a} s_1, s_0 \xrightarrow{d} s_2, s_1 \xrightarrow{d} s_3, s_1 \xrightarrow{d} s_4, s_2 \xrightarrow{b} s_5, s_2 \xrightarrow{c} s_6\}$.

Definition 2.1.2 (Enabling)

Let $\mathcal{A} = (S, \Sigma, T, s_0)$ be a transition system, and e an event in Σ. e is **enabled** *at state s if $\exists\ s \xrightarrow{e} s' \in T$. If e is enabled, it can* **fire**. *The set of enabled events at state s is denoted by $\mathcal{E}(s)$.* ■

Definition 2.1.3 (Run)

Let $\mathcal{A} = (S, \Sigma, T, s_0)$ be a transition system. A **run** *or* **execution** *from a state s is a sequence of transitions*

$$\sigma = s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} \dots \xrightarrow{e_{k-1}} s_k \tag{2.1}$$

such that $s_0 = s$ and $\forall i \geqslant 0$, $s_i \xrightarrow{e_i} s_{i+1} \in T$. A state s_k is **reachable** *if there exists a run from the initial state s_0 to s_k.* ■

It is, therefore, possible to construct a run from a set of states S_0 to a set of states S_k denoted as $S_0 \xrightarrow{E_0} S_1 \xrightarrow{E_1} \dots \xrightarrow{E_{k-1}} S_k$, where E_i designs the set of events that fire from S_i.

Definition 2.1.4 (To Set)

Given a transition system $\mathcal{A} = (S, \Sigma, T, s_0)$. Let $s \in S$ be a state. We define the set $To(s)$ as follows:

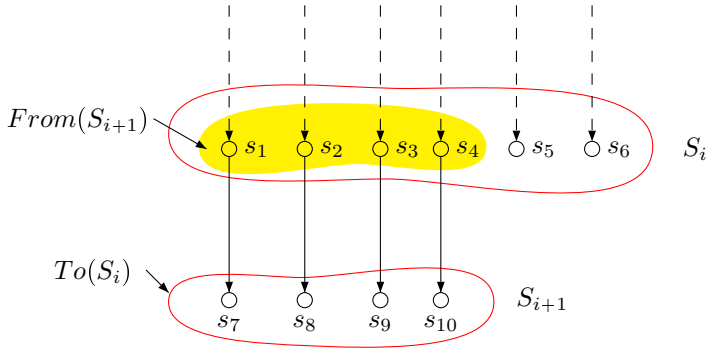


Figure 2.2: To and $From$ sets.

$$To(s) = \{s' \in S \mid \exists e \in \Sigma, s \xrightarrow{e} s' \in T\} \tag{2.2}$$

Likewise, given a set of states $P \subset S$ the set $To(P)$ is a set $P' \subset S$ such that $\forall s' \in P', \exists s \in P$ such that $s' \in To(s)$. ■

Definition 2.1.5 (From Set)

Given a transition system $\mathcal{A} = (S, \Sigma, T, s_0)$. Let $s' \in S$ be a state. We define the set $From(s')$ as follows:

$$From(s') = \{s \in S \mid \exists e \in \Sigma, s \xrightarrow{e} s' \in T\} \tag{2.3}$$

Likewise, given a set of states $P' \subset S$ the set $From(P')$ is a set $P \subset S$ such that $\forall s' \in P', \exists s \in P, s \in From(s')$. ■

Note that in a given run $S_0 \xrightarrow{E_0} S_1 \xrightarrow{E_1} \ldots \xrightarrow{E_{k-1}} S_k$, we always have $To(S_i) = S_{i+1}$, but sometimes we may have $From(S_{i+1}) \neq S_i$. This situation happens when some of the states in S_i has no outgoing transitions (see Figure 2.2).

2.1.2 Symbolic Representation

To represent transition systems symbolically, we use *Boolean algebras* [Kop89]. The representation is needed when we construct the Boolean formulas during the BMC unrolling operation. We will start this section by describing Boolean algebras, then show how to use them for representing a transition system.

Definition 2.1.6 (Boolean Algebra)

A Boolean algebra *is a tuple $(B, +, ., 0, 1)$, where B is a set; $+$ an $.$ are binary operators on B satisfying the commutative and distributive law; 0 and 1 are elements of B representing the neutral element for $+$ and $.$ respectively. i.e., $\forall\, b \in B$, $b + 0 = b$ and $b \,.\, 1 = b$.*

Each element $b \in B$ has a complement *$\overline{b} \in B$ such that $b + \overline{b} = 1$ and $b \,.\, \overline{b} = 0$.* ■

Definition 2.1.7 (Boolean Function)
A Boolean function *is an n-variable logic function* $f : \mathbb{B}^n \rightarrow \mathbb{B}$, *where* $\mathbb{B} = \{0, 1\}$, *and* f *transforms each element* $(v_0, \ldots, v_{n-1}) \in \mathbb{B}^n$ *into an element of* $\mathbb{B}$.

A Boolean function f *is a* zero *function if* $f(v_0, \ldots, v_{n-1}) = 0,\ \forall (v_0, \ldots, v_{n-1}) \in \mathbb{B}^n$. *Likewise, a Boolean function* f *is a* one *function if* $f(v_0, \ldots, v_{n-1}) = 1,\ \forall (v_0, \ldots, v_{n-1}) \in \mathbb{B}^n$.

Each element of $\mathbb{B}^n$ *is called* vertex. *A* literal *is either a variable* v_i *or its complement* $\overline{v}_i$. *A* cube c *is a set of literals, such that if* $v_i \in c$ *then* $\overline{v}_i \notin c$ *and vice versa.* ■

Example 2.1.2 Below are some interesting examples of Boolean algebras.

- $(\mathbb{B}, \vee, \wedge, 0, 1)$, where $\mathbb{B} = \{0, 1\}$; $\vee$ and $\wedge$ correspond to the logical operators OR and AND respectively.

- $(F_n(\mathbb{B}), +, ., 0, 1)$, where $F_n(\mathbb{B})$ is the set of n-variable logic functions on $\mathbb{B}$; $+$ and $.$ represent the addition and multiplication of n-variable logic functions respectively; 0 and 1 correspond to the *zero* and *one* functions respectively.

- $(2^S, \cup, \cap, \varnothing, S)$, where S is a set; $\cup$ and $\cap$ correspond to the set operators union and intersection respectively.

We will now give a symbolic representation of a transition system using Boolean algebra. Given a TS $\mathcal{A} = (S, \Sigma, T, s_0)$. We can then build a Boolean algebra $(2^S, \cup, \cap, \varnothing, S)$ from the set of states S. However, since in BMC, we are dealing with Boolean variables rather than sets, we need to find a way to map each state in S into Boolean variables. In this way, we will have an encoding that reasons in terms of logic operators such as $\vee$ and $\wedge$, rather than set operators such as $\cup$ and $\cap$.

The mapping can be done as follows. We represent each state $s \in S$ by en encoding function $\mathcal{Q} : S \rightarrow \mathbb{B}^n$ such that $n \geqslant \lceil log_2(|S|) \rceil$ and $\mathbb{B} = \{0, 1\}$. That is, given a set of Boolean variables $V = \{v_0, \ldots, v_{n-1}\}$ each state $s \in S$ is encoded into a vertex $(v_0, \ldots, v_{n-1}) \in \mathbb{B}^n$. We choose $n = |S|$, and for each state, all variables in its associated vertex equal zero except for the one which has the same index as the state. i.e. given a state s_i, we have:

$$s_i \leftrightarrow (v_0 = 0, \ldots, v_i = 1, \ldots, v_{n-1} = 0) \tag{2.4}$$

Example 2.1.3 Assume $|S| = 4$. Then, we have $n = 4$, i.e $V = \{v_0, v_1, v_2, v_3\}$, and each state is mapped as follows $s_0 \leftrightarrow (1, 0, 0, 0)$, $s_1 \leftrightarrow (0, 1, 0, 0)$, $s_2 \leftrightarrow (0, 0, 1, 0)$, $s_3 \leftrightarrow (0, 0, 0, 1)$.

Any set of states $P \subset S$ can be represented by a *characteristic (Boolean) function* $\mathcal{X}_P : \mathbb{B}^n \rightarrow \mathbb{B}$ that evaluates to 1 for those vertices in $\mathbb{B}^n$ that correspond to states in P, encoded using $\mathcal{Q}$.

Using the state encoding above, the characteristic function of a set of states P can be expressed by the product of those variables that evaluate to 1 for each state in P.

Formally, we have:

$$\mathcal{X}_P = \bigwedge_{s_i \in P} v_i \tag{2.5}$$

This kind of characteristic function is much common in Petri net-based methods such as the one in [OTK04], but in this work, we apply it to transition systems.

Example 2.1.4 As a an example, consider the TS $\mathcal{A}$ depicted in Figure 2.1. We have:

$$\mathcal{X}_{\{s_0,s_1,s_2\}} = v_0 \wedge v_1 \wedge v_2; \quad \mathcal{X}_{\{s_3\}} = v_3;$$

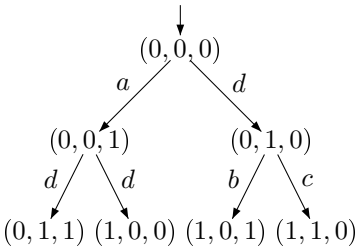
It is also possible to represent each state using the binary number equivalent to the state's index. This representation requires at least $log_2(|S|)$ Boolean variables. For instance, assume $|S| = 8$, then we need $n = log_2(8) = 3$ variables to represent each state, i.e. $V = \{v_0, v_1, v_2\}$. In binary number with three variables, we have $0 = 000$, $1 = 001$, $2 = 010$, $3 = 011$, etc. Thus, each state is mapped as follows $s_0 \leftrightarrow (0,0,0)$, $s_1 \leftrightarrow (0,0,1)$, $s_2 \leftrightarrow (0,1,0)$, $s_3 \leftrightarrow (0,1,1)$, etc. For each state, the number of variables that have the value 1 varies from 0 to n. Therefore, the characteristic function $\mathcal{X}_P$ given in equation (2.5) does not apply for this representation. Rather, it is defined as follows. Assume we have a set of states $P = \{s_0, s_1, s_2\}$, and $n = 3$. Then the function $\mathcal{X}_P$ is given as $\mathcal{X}_P = (\overline{v}_0 \wedge \overline{v}_1 \wedge \overline{v}_2) \vee (\overline{v}_0 \wedge v_1 \wedge \overline{v}_2) \vee (\overline{v}_0 \wedge v_1 \wedge v_2)$. As we can see, the derived Boolean formula is larger than the one in Example 2.1.4. Hence, this representation is not quite appropriate for SAT-based approaches because it will give extra work to the solving process. It is much more common in BDD-based methods (see e.g. [Pen03]). Figure 2.3 depicts the difference between these two representations.

2.2 Other Models for Concurrent Systems

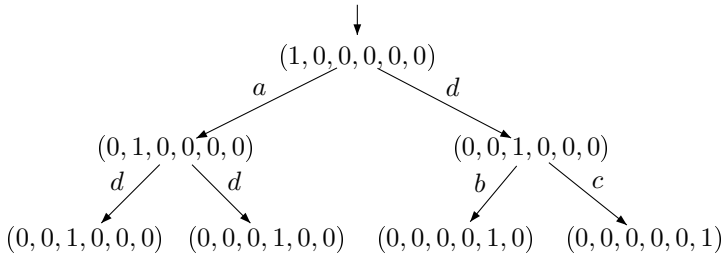
In this section, we discuss other important formalisms for modeling concurrent systems. Namely, *Kripke structure, Petri nets*, and *automata*.

2.2.1 Kripke Structure

A *Kripke structure* [CGP99] is a formalism used for describing the behavior of *reactive systems*, i.e. systems that interact frequently with their environment. It is a graph-based model, whose nodes represent the system's states and whose edges represent the transitions between states. This formalism also uses a labeling function that maps each node with a set of properties that are true in the corresponding state (See Figure 2.4). Below is a formal definition of a Kripke structure.



Representation appropriate for BDD.



Representation appropriate for SAT.

Figure 2.3: Different ways for representing the states.

Definition 2.2.1 (Kripke Structure)

A **Kripke Structure** *is a tuple* $\mathcal{M} = (S, I, T, \ell)$ *where:*

- S *is a finite set of* **states**;
- $I \subseteq S$ *is the set of* **initial states**
- $T \subseteq S \times S$ *is a* **transition relation** *between states;*
- $\ell : S \to 2^{AP}$ *is a* **labeling** *function that labels each state with the set of atomic propositions (AP) true in that state.* ■

Definition 2.2.2 (Paths, Runs)

A **path** *from a state* s *is an infinite sequence of states* $\pi = s_0 s_1 s_2 \ldots$ *such that* $s_0 = s$ *and* $(s_i, s_{i+1}) \in T$ *for all* $i \geqslant 0$. *The set of all paths starting from a state* s *is denoted* $\Pi(s)$.

A **run** *is a path that starts from an initial state. The set of runs of* $\mathcal{M}$ *is defined as* $\Pi(\mathcal{M}) = \bigcup_{s \in I} \Pi(s)$.

A run constitutes a possible behavior of $\mathcal{M}$ *and* $\Pi(\mathcal{M})$ *is called the* **linear-time behavior** *of* $\mathcal{M}$. ■

Definition 2.2.3 (Computation Tree)

A **tree** *rooted at a state* s *is the infinite tree* $T(s)$ *obtained by unfolding* $\mathcal{M}$ *from* s.

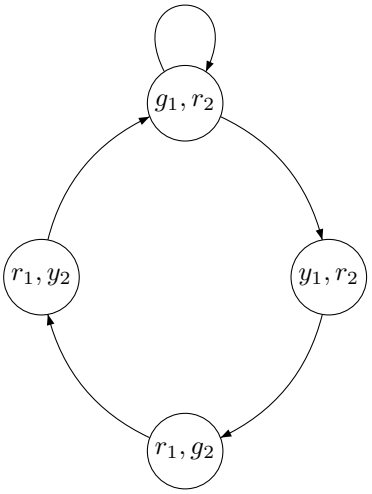


Figure 2.4: A Kripke structure model of a simple traffic light controller.

A **computation tree** *is a tree starting from an initial state. The set of computation trees of $\mathcal{M}$ is defined as $T(\mathcal{M}) = \{T(s) \mid s \in I\}$, it shows all possible executions in $\mathcal{M}$ starting from an initial state. $T(\mathcal{M})$ is also called the* **branching-time behavior** *of $\mathcal{M}$.* ■

Example 2.2.1 To illustrate the Kripke structure model, let us take a simple and typical example: a controller that operates the traffic lights at an intersection between a busy highway and a farm road. The controller is designed as follows. A sensor detects the presence of a car on the farm road in either direction. By default, the highway light remains green and the farm light remains red until a car is detected by the sensor on the farm road. In this case, the light on the highway turns yellow, for a short period of time, before switching into red. Once the highway light becomes red, the light on the farm road changes into green to allow its traffic to move, and the sensor is disabled. After a time interval long enough for a few cars to pass on the farm road, its light turns yellow momentarily. Then both lights switch back to the default position, and the sensor is again enabled.

Let us associate the highway with a variable called $road_1$ and the farm road with $road_2$. Assume that for each $road_i$ we have a set of atomic propositions $L_i = \{g_i, y_i, r_i\}$ such that:

- g_i : the light is green on $road_i$,
- y_i : the light is yellow on $road_i$,
- r_i : the light is red on $road_i$.

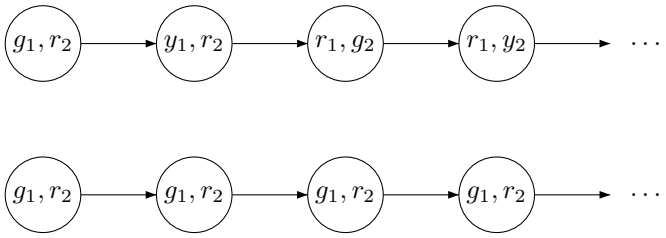


Figure 2.5: Runs from the traffic light controller.

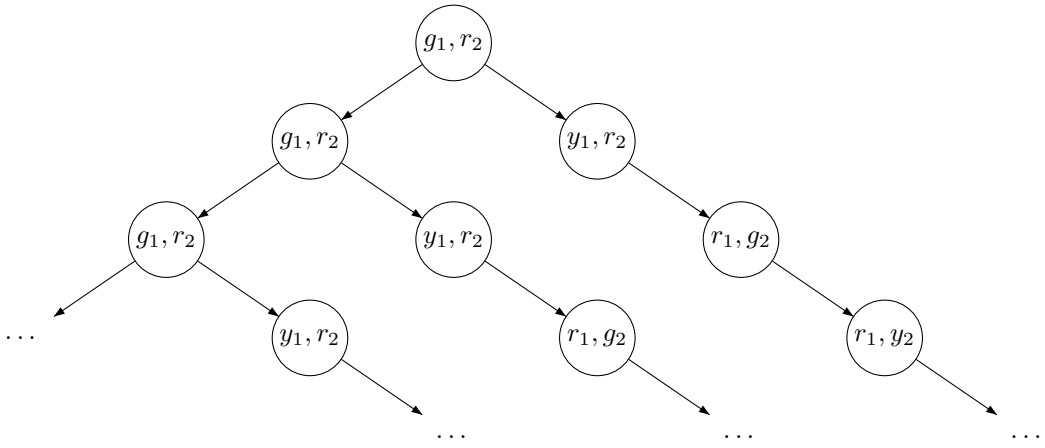


Figure 2.6: Computation tree of the traffic light controller.

Since we are only concerned with the changes of lights on both roads, we will neglect the notion of time. Now, according to the design of our controller, we can build the Kripke structure $\mathcal{M} = (S, I, T, \ell)$ where:

- $S = L_1 \times L_2$
- $I = \{(g_1, r_2)\}$
- $T = \{((g_1, r_2), (y_1, r_2)),\ ((y_1, r_2), (r_1, g_2)),\ ((r_1, g_2), (r_1, y_2)),\ ((r_1, y_2), (g_1, r_2)),$ $((g_1, r_2), (g_1, r_2))\}$
- $\ell(g_1, r_2) = \{g_1, r_2\},\ \ell(y_1, r_2) = \{y_1, r_2\},\ \ell(r_1, g_2) = \{r_1, g_2\}$, and $\ell(r_1, y_2) = \{r_1, y_2\}$
- From the initial state (g_1, r_2) the model has a path π defined as:

  $\pi = (g_1, r_2)\,(y_1, r_2)\,(r_1, g_2)\,(r_1, y_2)\,(g_1, r_2)\,(g_1, r_2)\,\ldots$

The graphical representation of the model is depicted in Figure 2.4. Examples of runs and a computation tree are shown in Figure 2.5 and 2.6 respectively.

2.2.2 Petri Nets

A *Petri net* [Pet81, Rei85, Mur89] is a graphical as well as mathematical modeling tool invented in 1962 by A. Petri [Pet62, Pet66]. It has a simple, yet powerful representation comparable to flow charts, block diagrams, and networks. Moreover, through its mathematical nature, it is possible to generate state equations that represent the system's behavior. Petri nets can be used for describing numerous classes of systems, including software [BE77], computer systems [ACA78, KL82], communication networks [BT82, Dia87], process control systems [BM86, DS83], and manufacturing systems [Des89, ZDR92, BK86], just to name a few.

To model all these systems efficiently, Petri nets exist in different types including *colored Petri nets* [Jen97], *high level Petri nets* [JR91], *timed Petri nets* [Zub80, Wan98], and *Stochastic Petri nets* [MBC+98, BK02, Haa02]. In this report, we are concerned only with "classical" Petri nets. It is defined formally as follows.

Definition 2.2.4 (Petri Net)
A **Petri net** *is a tuple* $\mathcal{N} = (P, T, F, M_0)$ *where:*

- P *is a set of* **places**;
- T *a set of* **transitions**;
- F *a function* $(P \times T) \cup (T \times P) \to \{0, 1\}$;
- M_0 *is the* **initial marking**.

The sets P *and* T *are disjoint, and their elements are called* **nodes**. *An* **arc** *exists from node* x *to node* y *if* $F(x, y) = 1$. *The set* ${}^\bullet x = \{y \in P \cup T \mid F(y, x) = 1\}$ *denotes the* **preset** *of a node* x. *The set* $x^\bullet = \{y \in P \cup T \mid F(x, y) = 1\}$ *denotes the* **postset** *of a node* x.

For a Petri net $\mathcal{N}$ *we have:*
$Min(\mathcal{N}) = \{x \in P \mid {}^\bullet x = \varnothing\}$ *and* $Max(\mathcal{N}) = \{x \in P \mid x^\bullet = \varnothing\}$. ■

Definition 2.2.5 (Marking)
A **marking** M *of a Petri net* $\mathcal{N} = (P, T, F, M_0)$ *is a mapping of* P *to the natural numbers* $\mathbb{N}$, *i.e.* $M : P \to \mathbb{N}$. M *can be identified with the multiset containing* $M(p)$ *copies of* p *for each* $p \in P$. ■

Graphically, transitions and places are represented with rectangles and circles, respectively. Tokens inside places indicates that they hold a marking.

Example 2.2.2 Figure 2.7 depicts a Petri net version of the traffic light controller described in Example 2.2.1. Formally, this Petri net can be represented as $\mathcal{N} = (P, T, F, M_0)$, where:

- $P = \{g_1, y_1, r_1, g_2, y_2, r_2\}$,

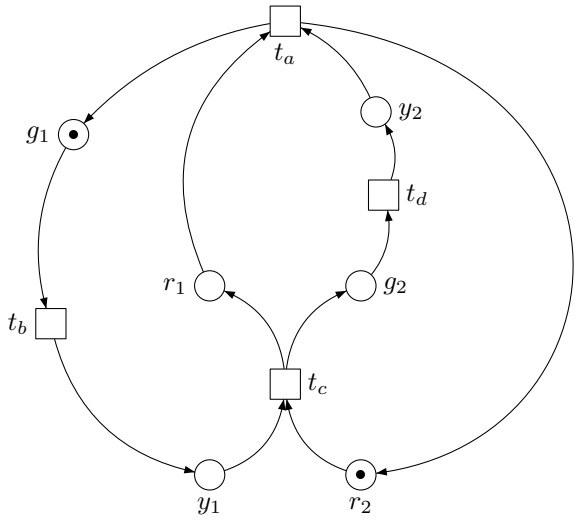


Figure 2.7: A Petri net version of the traffic light controller.

- $T = \{t_a, t_b, t_c, t_d\}$,
- $M_0 = \{g_1, r_2\}$.

2.2.3 Automata

Automata [Hro04, Sip97, HMU01, Koz97, Kel95] is the simplest computing model in computer science. It is used for describing *finite state machines*, i.e. a system that, given a set of input symbols, evolves through a series of states following a transition function. The automaton reads the input symbols one by one until there is nothing left. The next state is determined by the current symbol as well as the current state. When all the symbols are read, the automaton stops. If it stops in one of the states called *accepting states*, it is said to *accept* the input. Otherwise, it *rejects* the input.

As in the Kripke structure model, states are represented graphically with circles, while transitions are depicted as arrows. Each transition is labeled with the current symbol. An incoming arrow without origin identifies the initial state. Accepting states are marked with a double circle (see Figure 2.8).

Automata can be classified into different types depending on the kind of input they accept. We will only focus on *finite automata*. For the other classes, see e.g. [AD94] and the references cited above. Applications of automata to model checking can be found in [BBF+99, CGP99].

Definition 2.2.6 (Finite Automata)
A **finite automata** *is a tuple* $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ *such that:*

- Q *is a finite set of* **states**;

- Σ *an alphabet called, the* **input alphabet**;

- $\delta : Q \times \Sigma \longrightarrow Q$ *is the* **transition function**;

- $q_0 \in Q$ *is the* **initial state**;

- $F \subseteq Q$ *is the set of* **accepting states**. ■

Definition 2.2.7 (Language Accepted by an Automaton)
Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be an automaton and $w = e_1 e_2 ... e_n$ a word (i.e. a sequence of symbols) over Σ^*.

The automaton $\mathcal{A}$ **accepts** *w if there exists a sequence of states $r_0, r_1, ..., r_n$ in Q satisfying the following three conditions:*

- $r_0 = q_0$,

- $\delta(r_i, e_{i+1}) = r_{i+1}$, *for* $0 \leqslant i \leqslant n-1$, *and*

- $r_n \in F$.

The **language** $\mathcal{L}(A)$ **accepted by** $\mathcal{A}$ *is defined as*

$$\mathcal{L}(A) = \{w \in \Sigma^* \mid \mathcal{A} \textit{ accepts } w\} \tag{2.6}$$

which corresponds to the set of all words accepted by $\mathcal{A}$. ■

Example 2.2.3 To illustrate the use of automata let us take a typical example: a digicode that controls the opening of building doors. This example is a simplified version of the one in [BBF+99]. We assume that only two keys are available, a and b. The door opens upon keying in the sequence aab, regardless of any previous try. We can model the digicode using an automaton with 4 states and 8 transitions as in Figure 2.8.

The automaton can be described formally by writing $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where

- $Q = \{1, 2, 3, 4\}$

- $q_0 = 1$

- $\delta(1, b) = 1;\ \delta(1, a) = 2;\ \delta(2, a) = 3;\ \delta(2, b) = 1;\ \delta(3, a) = 3;\ \delta(3, b) = 4;$
  $\delta(4, a) = 4;\ \delta(4, b) = 4;$

- $F = \{4\}$.

$\mathcal{A}$ accepts all inputs containing the string aab.

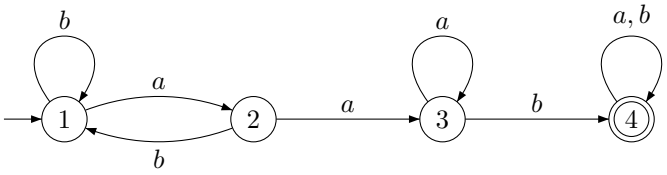


Figure 2.8: An automaton model of a digicode.

2.3 Linear Temporal Logic

Temporal logic [Eme90, CW96] is a form of logic used for describing the behavior of concurrent systems. It is aimed at expressing succession of events without mentioning time explicitly. It uses simple and clear notations, called *temporal operators*, that correspond roughly to words in a natural language such as *until*, *next*, and *eventually*. To express properties of states, the logic also uses atomic propositions combined with Boolean connectives such as $\neg$, $\wedge$, and $\vee$.

In this section, we will focus on a temporal logic called *linear temporal logic* (LTL) [GPSS80, Lam80, BBF+99] which uses the following temporal operators:

- X ("next") indicates that the next state satisfies a property;
- F ("in the future or eventually") specifies that a future state on the path will satisfy a property;
- G ("globally or always") implies that a property will hold at all the future states in the path;
- U ("until") relates two properties. It states that a property holds along the path until a second property is verified.
- R ("release") is the logical dual of U. It requires that the second property holds along the path up to the first state where the first property holds (or forever if such a state does not exist).

Let P be a set of atomic propositions. The syntax and semantics of an LTL formula is given by the following definitions.

Definition 2.3.1 (Syntax)

The set of **LTL formulas** *on the set P is defined by the grammar $\psi = p \,|\neg\psi\,|\psi_1 \wedge \psi_2\,|\psi_1 \vee \psi_2\,|\,\mathsf{X}\psi\,|\,\psi_1\,\mathsf{U}\,\psi_2$, where p ranges over P.* ■

Definition 2.3.2 (semantics)

Let $\xi = u_0u_1u_2\ldots$ be an infinite word over the alphabet 2^P. Let ψ be an LTL formula. In the semantics below, $\xi \models \psi$ means ξ **models** *ψ. We also define $\xi(i) = u_i$ and $\xi^i = u_iu_{i+1}\ldots$ for $i \in \mathbb{N}$. The semantics of LTL is then given as follows:*

- $\xi \models p$ *iff* $p \in \xi(0)$, *for* $p \in P$

- $\xi \models \psi_1 \wedge \psi_2$ *iff* $\xi \models \psi_1$ *and* $\xi \models \psi_2$,

- $\xi \models \psi_1 \vee \psi_2$ *iff* $\xi \models \psi_1$ *or* $\xi \models \psi_2$,

- $\xi \models \mathsf{X}\,\psi$ *iff* $\xi^1 \models \psi$,

- $\xi \models \psi_1\ \mathsf{U}\ \psi_2$ *if* $\exists i\ [\xi^i \models \psi_2$ *and* $\forall j,\ j < i$ *we have* $\xi^j \models \psi_1]$, ■

We will also use the abbreviations $true = p \vee \neg p$ and $false = \neg true$. The temporal operators F, G, and R are derived from the above definition as follows:

$$\mathsf{F}\psi = true\,\mathsf{U}\,\psi \tag{2.7}$$

$$\mathsf{G}\psi = false\,\mathsf{R}\,\psi \tag{2.8}$$

$$\psi_1\ \mathsf{R}\ \psi_2 = \neg(\neg\psi_1\ \mathsf{U}\ \neg\psi_2) \tag{2.9}$$

Example 2.3.1 Let us give some examples of LTL formula:

- $\mathsf{G}(barOpen\,\mathsf{U}\,midnight)$ reads "it is always true that the bar opens until midnight,"

- $\mathsf{X}(ourTrain)$ reads "our train will come just next,"

- $\mathsf{G}\neg(critSec1 \wedge critSec2)$ reads "both processes will never be in their critical section at the same time,"

- $\mathsf{G}(Req \rightarrow \mathsf{F}\,Sat)$ reads "any request will eventually be satisfied."

For concurrent systems, the next-time operator X does not have significant meaning because, in these types of systems, it is not really important to make a distinction between one execution step and two or more subsequent steps. These steps are considered to be observationally equivalent, especially for high level abstract specifications. For that reason, some researchers do not include this operator when they study LTL with concurrent systems (e.g. [Sor02]). The resulted logic is often known as LTL-X. In [Lam83], Lamport gives a detailed explanation as to why the next-time operator can be omitted in temporal logic.

Some people also focus their study on the logic known as PLTL which is a type of LTL combined with past operators such as Y (yesterday), O (once), and H (historically) [BC03, BHJ+06]. In theoretical point of view, there is no big difference between PLTL and future-only LTL since they both have the same degree of expressiveness. In practice, however, past operators can be more useful as they keep specifications short and simple. Hence, they can help model checker users to formulate the desired properties in a more succinct and natural way.

Negative Normal Form

Many LTL-based model checking methods require the formula to be written into *negative normal form* (NNF)—i.e. a formula in which every negation appears only before a literal—using the operators $\vee, \wedge$ X, U, and R. NNF formulas can be obtained by applying the following equivalences:

$$true \equiv p \vee \neg p \tag{2.10}$$
$$false \equiv \neg true \tag{2.11}$$
$$\psi_1 \wedge \psi_2 \equiv \neg(\neg\psi_1 \vee \neg\psi_2) \tag{2.12}$$
$$\psi_1 \,\mathsf{R}\, \psi_2 \equiv \neg(\neg\psi_1 \,\mathsf{U}\, \neg\psi_2) \tag{2.13}$$
$$\mathsf{F}\,\psi \equiv true \,\mathsf{U}\, \psi \tag{2.14}$$
$$\mathsf{G}\psi \equiv \neg\, \mathsf{F}\, \neg\psi \tag{2.15}$$

Logic for Nondeterministic Choices

The logic LTL can only describe events along a single computation path. Therefore, it cannot express the possibility of nondeterministic choices that may exist along an execution (run) at some instants. Because of this feature, LTL formulas are also called *path formulas*. To express nondeterministic choices, one can use the logic known as *computation tree logic* (CTL) [EH86, CES86] whose formulas may contain the following path quantifiers:

- A, known as *universal path quantifier*, suggests that a property is satisfied by *all the computation paths* starting from the current state;
- E, known as *existential path quantifier*, illustrates that *there exists a computation path* from the current state that satisfies a property.

Example 2.3.2 To see the difference between CTL and LTL representations, let us take again one of the formulas from Example 2.3.1. In LTL, the statement "both processes will never be in their critical section at the same time," is expressed as $\mathsf{AG}\neg(critSec1 \wedge critSec2)$

More examples of properties expressed in temporal logic can be found in [BBF+99, Pnu77]. The complexity of temporal logic with model checking is discussed in [Shn02, Sa85].

2.4 Satisfiability Problem

The *Boolean satisfiability problem (SAT)* is a decision problem of finding if there exists some assignment of the values 0 and 1 to the variables in a Boolean formula that makes it evaluate to 1 (*true*). If this assignment exists, the formula is said to be *satisfiable*.

The formula to be solved is often written in *conjunctive normal form* (CNF), that is a conjunction of clauses with disjunctive literals as in the following example.

Example 2.4.1 The CNF formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ has the satisfying assignment $x_1 = 1,\ x_2 = 1,\ x_3 = 0$.

SAT has been applied to many fields in computer science including problems in electronic design automation (EDA) [FDK98, CG96, DKMW93, SSA91], microprocessor verification [VB01], field-programmable gate array (FPGA) layout [NASR04], automatic test pattern generation (ATPG) [SBSV96, TGH97], to name but a few. (See [GPFW97] for a complete list of applications that can be formulated as SAT.)

Although, SAT belongs to the class of NP-complete problems [Coo71], numerous studies have been conducted to find better algorithms and methods for it. Some of them are software-related approaches [GPD97], while others consist in customizing the hardware itself to match the requirements of SAT [AS97, SYSN01, SdBF04]. As a result, many practical SAT instances can now be solved in minutes, or even seconds on a regular computer. But since no one has ever proven that all SAT problems can be solved in polynomial time, there must be out there some SAT instance which takes huge amount of time to solve. Note that, interesting SAT benchmarks are available online at [Vel, DIM, SAT]. Benchmarks designed especially for BMC can also be found in [Zar05, Zar04].

In practice, there exist two classes of algorithms for solving SAT: *incomplete* and *complete algorithms*. Based on heuristic search algorithms, incomplete algorithms can provide results very fast. Still, they cannot prove unsatisfiability and, in some cases, terminate without giving any result at all. For that reason, most state-of-the-art SAT solvers use complete algorithms which can always lead to a result and are able to prove unsatisfiability. We will describe two well-known algorithms for solving SAT. First, a complete algorithm known as DPLL . Second, an incomplete one invented by Stålmarck.

2.4.1 DPLL Algorithm

Invented by Martin Davis, Hilary Putnam, Goerge Longeman, and Donald Loveland in 1962 [DLL62, DP60], DPLL accepts as input Boolean formulas written in CNF. It is a backtracking search that relies on the following rules: *unit clause rule*, *implication*, and *Boolean constraint propagation*.

Unit Clause Rule A Clause in which there is only one unassigned literal is called *unit clause*. If the rest of the literals in that clause have the value 0, then the unassigned literal must take the value 1 to make the clause satisfiable.

Implication A variable is forced to be 1 (true) or 0 (false) depending on the previous assignment. The term *conflict* refers to the situation where both 1 and 0 are implied to the same variable.

Algorithm 2.1: dpll

input : formula in CNF
output: *sat* or *unsat*

```
1  while 1 do
2      decide() ;
3      while 1 do
4          status ⟵ bcp() ;
5          if status = conflicts then
6              blevel ⟵ analyzeConflict() ;
7              if blevel = 0 then
8                  return unsat ;
9              else
10                 backtrack(blevel) ;
11             end
12         else if status = satisfiable then
13             return sat ;
14         else
15             break ;
16         end
17     end
18 end
```

Boolean constraint Propagation (BCP) It refers to the task of applying the unit clause rule iteratively until there is no unit clause available.

Algorithm 2.1 depicts a simplified version of DPLL. The function `decide()` chooses an unassigned variable to branch and gives it a value. `bcp()` carries out BCP operation until a satisfying assignment is found, or until a conflict occurs. When conflict occurs, `analyzeConflict()` determines the level at which to backtrack. This level corresponds to a variable that has not been assigned both ways (1 and 0). If no such a variable exists, the formula is unsatisfiable. A new clause, known as *conflict clause*, which describes the conflicting branch can also be added during the conflict analysis in order to avoid searching the same path again in the future. `backtrack (`*blevel*`)` flips the branching variable at *blevel*, and undoes all implications up to that level.

DPLL is implemented in many solvers such as ZChaff [MMZ+01], Berkmin [GN02], Sato [Zha97], and Grasp [SS99] to name but a few.

2.4.2 Stålmarck's Algorithm

Stålmarck's k-saturation algorithm [SS00] is based on the so-called *dilemma proof system*, which is used for proving whether a propositional logic formula is a tautology. Here, k

represents a natural number called *saturation level*. The algorithm exhaustively searches for a proof of depth k for a given formula. It runs fast with a time complexity $O(n^{2k+1})$, where n corresponds to the size of the formula. This method is known to be useful for verifying properties of industrial designs, in which, $k \leqslant 2$ appears to be enough in most cases. Unlike DPLL, Stalmårck's algorithm can take non-CNF formulas as input. In addition, implication ($\Rightarrow$), and equivalence ($\Leftrightarrow$) are also allowed. Before describing the algorithm, we will first review some terminologies used in the dilemma proof system.

Triplets A *triplet* is a compound formula of the form $x = y\ o\ z$, where x represents the compound formula itself, y and z are literals, and o a binary operator. It has been proved that all Boolean logic formulas can be reduced into triplets.

Example 2.4.2 As an example, the formula $F = (x_1 \vee (x_2 \wedge x_3)) \Leftrightarrow (x_1 \vee x_2) \wedge (x_1 \vee x_3)$ can be reduced to the following triplets:

$a = x_1$
$b = x_2$
$c = x_3$
$d = x_2 \wedge x_3$
$e = a \vee d$
$f = x_1 \vee x_2$
$g = x_1 \vee x_3$
$h = f \wedge g$
$i = e \Leftrightarrow h$

Formula Relation, Equivalence Class Let F be a Boolean formula, and $Sub(F)$ a set containing all subformulas in F along with their complements. A *formula relation* on F is an equivalence relation $\equiv$ over the domain $Sub(F)$, with the constraint that if $P \equiv Q$, then $P' \equiv Q'$, where P' and Q' correspond to the complement of P and Q respectively.

If $P \equiv Q$ holds, this means P and Q belong to the same *equivalence class*, and have the same truth value.

Association Let R be a formula relation, and P, Q subformulas of R. The relation $\mathsf{R}[P \equiv Q]$ denotes the least formula relation containing R and relates P and Q. The notation $P \equiv Q$ is called an *association* on R. If ψ is the association $P \equiv Q$, then ψ' denotes the *complementary association* $P \equiv Q'$.

Propagation Rules A *propagation rule* or a *simple rule* for a binary operator is a rule for assigning a truth value, i.e. $\top$ (*true*) or $\bot$(*false*), to a formula, given that the truth value of some of its subformulas are known. In some cases, the formula may be assigned to another formula or complement of another formula.

For example, if the conjunction of two subformulas $P \wedge Q$ is true, then P must be true and Q must be true. This statement is expressed with the pair of rules below:

$$\frac{P \wedge Q \equiv \top}{P \equiv \top} \qquad \frac{P \wedge Q \equiv \top}{Q \equiv \top} \tag{2.16}$$

Other interesting rules for $\wedge$ include:

$$\frac{P \equiv \top}{P \wedge Q \equiv Q} \qquad \frac{Q \equiv \top}{P \wedge Q \equiv P} \tag{2.17}$$

$$\frac{P \equiv \bot}{P \wedge Q \equiv \bot} \qquad \frac{Q \equiv \bot}{P \wedge Q \equiv \bot} \tag{2.18}$$

$$\frac{P \equiv Q}{P \wedge Q \equiv P} \qquad \frac{P \equiv Q}{P \wedge Q \equiv Q} \tag{2.19}$$

Dilemma Rule The *dilemma rule* is a branching rule for deriving new equivalence relation from a formula relation. It has the form below:

$$\frac{\begin{array}{c}\mathsf{R}\end{array}}{\begin{array}{cc}\mathsf{R}[P \equiv Q] & \mathsf{R}[P \equiv \neg Q] \\ \pi_0 & \pi_1 \\ R_0 & R_1\end{array}} \quad \begin{array}{c} \\ \\ \\ R_0 \cap R_1 \end{array} \tag{2.20}$$

The dilemma rule can be interpreted as follows: first, two subformulas P and Q are taken from a formula relation R. P and Q must be from different, but non-complementary, equivalence classes in R. Then, two derivations π_1 and π_2 are built from the relations $\mathsf{R}[P \equiv Q]$ and $\mathsf{R}[P \equiv \neg Q]$ respectively. This leads to new equivalence classes R_0 and R_1. Finally, all common associations in R_0 and R_1 are combined in order to form the intersection $R_0 \cap R_1$.

The Dilemma Proof system

Recall that the dilemma proof system is used for proving whether a given formula is a tautology. Briefly put, it works in the following way. At the beginning, the formula is assumed to be false. It is first decomposed into set of triplets. Then, the propagation rules are applied to its subformulas to determine if a contradiction exists. If yes, then the formula is a tautology. Otherwise, the formula is false. The dilemma rule is used only if none of the propagation rules can be applied. If still a contradiction is found from the left and right branch of the dilemma rule, then the formula is a tautology. More detailed explanation about the dilemma proof system can be found in [SS00].

To implement the dilemma proof system, Stålmarck introduced the k-saturation algorithm which has two parts: 0-saturation, used for implementing the propagation rules, and k-saturation (with $k \geqslant 1$), used for the dilemma rule. The formula to be proved is represented as a set of triplets. Starting from a given relation, the algorithm tries to derive a new relation by applying some rules related to the triplets. The given relation corresponds to the one obtained after assuming that the formula is false. This is often

done by assigning the value false (or zero) to *the top of the formula tree* and then put this assignment as initialization of the relation. A contradiction exists if no new relation can be derived, and this also means that the formula is a tautology. For instance, in Example 2.4.2 the top of the formula tree corresponds to the variable i in the set of triplets. Thus, if we want to assume that F is false, we could simply assign the value false (or zero) to the variable i.

Once the result of a tautology checking is known, the satisfiability of the formula can also be concluded. For instance, it has been proved that if a formula is not a tautology, its negation will be unsatisfiable (see e.g. [WBCG00]). In [CG05], Cook *et al.* describe a class of formula for which 1-saturation is sufficient to prove unsatisfiability.

We will now describe the saturation algorithm.

Algorithm 2.2: zeroSaturate

input : R a formula relation
T a set of triplets
output: a formula relation

```
1  while T ≠ ∅ do
2      t ⟵ retrieve(T) ;
3      R′ ⟵ propagate(R,t) ;
4      if R′ = R then
5          return R ;
6      else
7          T ⟵ T ∪ affected(R′, R) ;
8          R′ ⟵ R ;
9      end
10 end
11 return R ;
```

0-saturation algorithm

0-saturation, depicted in Algorithm 2.2, takes as input a formula relation R, and a set of triplets T. Here the goal is to apply propagation rules to the relation until no more new associations can be obtained. The function `retrieve(`T`)` chooses one triplet from T and removes it from the set. `propagate(R,`t`)` checks whether any rules corresponding to the chosen triplet t can be applied to R, and applies it. If no rules are applicable, or inconsistency was found, `propagate(R,`t`)` simply returns R and the algorithm stops. Otherwise, `affected(`R′, R`)` identifies all triplets affected by the rules and put them back into T for further exploration. These steps continue until T is empty.

Algorithm 2.3: saturate

input : R a formula relation
T a set of triplets
k a number $\geqslant 1$
output: a formula relation

```
1 if k = 0 then
2     return zeroSaturate(T, R)
3 end
4 foreach x ∈ var(T) do
5     R_0 ⟵ saturate(k − 1, T, R ∪ {x = 0});
6     R_1 ⟵ saturate(k − 1, T, R ∪ {x = 1});
7     R ⟵ R_0 ∩ R_1
8 end
9 return R ;
```

k-saturation algorithm ($k \geqslant 1$)

k-saturation, with $k \geqslant 1$, implements the dilemma rule. The algorithm computes iteratively $(k-1)$-saturation, $(k-2)$-saturation, etc., until $k = 0$. Logically, it calls 0-saturation when $k = 0$. The version of k-saturation, depicted in Algorithm 2.3, takes each triplet variable x, and branches over the two possible values of x (i.e *true* or *false*). The two branches produce new relations R_1 and R_0 respectively. Then, $\mathsf{R}_0 \cap \mathsf{R}_1$ is computed to form a new relation R, as in the dilemma rule.

2.4.3 Other Methods for Solving SAT

Methods for non-CNF Formulas

To solve non-CNF formulas (also known as *non-clausal* formulas), the typical way involves translating them into CNF, and then applying DPLL-based solvers. Although many CNF translation techniques have been developed, they are largely based on the Tseiting encoding [Tse83], which consists in introducing new variables for every subformula, along with new clauses to define the relation between the new variables and the subformula. The addition of these variables and clauses must not, in any case, change the truth value of the original formula. This method produces a CNF representation linear in the size of the original formula. Table 2.1 depicts a typical CNF translation of the gates AND, OR, and NOT.

But this translation technique causes some problems nonetheless. Namely, it may introduce too many variables, hence increasing greatly the size of the formula to be solved by the DPLL procedure. Furthermore, a large amount of structural information can be lost during the translation process. For example, in a translated CNF formula, there is no difference between primary input variables, flip-flops, etc. Nevertheless, there exist in

Gate	CNF clauses
$o \leftarrow \mathsf{AND}(i_1, i_2, ..., i_n)$	$(i_1 \vee \neg o) \wedge$ $(i_2 \vee \neg o) \wedge$ ... $(\neg i_1 \vee \neg i_2 \vee ... \vee \neg i_n \vee o)$
$o \leftarrow \mathsf{OR}(i_1, i_2, ..., i_n)$	$(\neg i_1 \vee o) \wedge$ $(\neg i_2 \vee o) \wedge$ ... $(i_1 \vee i_2 \vee ... \vee i_n \vee \neg o)$
$o \leftarrow \mathsf{NOT}(i)$	$(\neg i \vee \neg o) \wedge$ $(i \vee o)$

Table 2.1: Typical CNF Translation for the gates AND, OR, and NOT

the literature several techniques for restoring structural information from a CNF formula [RMB04, FM07, OGMS02]. Such information might be useful for a SAT solver in order to make more appropriate decisions.

Methods for coping with the above problems include optimizing the translation in order to have more compact CNF formulas as in [NRW98, Vel04b, She04, dlT92], adapting the DPLL procedure to work with non-CNF formula [GS99, TBW04], creating SAT solvers devoted to non-CNF formulas such as NoClause [TBW04] and BcSat [JN00], or hybrid solvers such as CirCUs [JS04] and Pueblo [SS06] which work by combining CNF clauses with other formalisms, or also adapting the CNF translation in order to preserve much of the structural information as in [PG86].

Note that it is also possible to generate the proof of unsatisfiability from a SAT-solver [MA03]. The set of clauses from the original SAT-instance needed to generate such a proof is called *unsatisfiable core* [LMS04, ZM03].

SMT-based Methods

Recently, as an effort to improve the DPLL procedure, some researchers have developed a method called DPLL(T) [GHN+04] which is mainly used to determine the satisfiability of quantifier-free formulas composed of linear equalities or inequalities. The DPLL(T) method combines solvers based on DPLL with arithmetic solvers capable of deciding the satisfiability of conjunctions of linear constraints. Existing tools that use this approach often integrate in their implementations some well known algorithms such as Fourier-Motzkin elimination [DE73] (used by CVClite [BB04], CVC [SBD02], SVC [BDL96]), Simplex methods [CLRS09] (used by MathSAT [BBC+05], ICS [FORS01], Simplics, Yices [DdM06a, DdM06b], ARIO [SS05]), and graph algorithms (e.g. Barcelogic [NO05], Slice [WIGG05]). These tools are collectively known as *satisfiability modulo theory* (SMT)-based solvers. A comparison of some of them can be found at [BdMS07]. Obviously, there

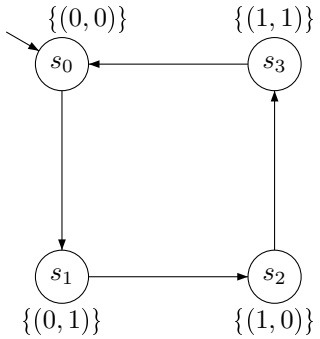


Figure 2.9: A two bit counter.

also exist BMC approaches based on SMT solvers (see e.g. [GG06, GG08a, GG08b]).

2.5 Bounded Model Checking

2.5.1 BMC Idea

The idea behind BMC can be described as follows. Given a system $\mathcal{M}$, a temporal logic formula ϕ, and a bound k, build a propositional formula that is satisfiable iff there is a counterexample of length k. More precisely, the BMC problem $\mathcal{M} \models_k \phi$ is equivalent to the satisfiability problem on the following formula:

$$[\![\mathcal{M}, \phi]\!]_k \equiv [\![\mathcal{M}]\!]_k \wedge [\![\neg\phi]\!]_k \tag{2.21}$$

where

$$[\![\mathcal{M}]\!]_k \coloneqq I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}). \tag{2.22}$$

and

$$[\![\neg\phi]\!]_k = (\neg\phi(s_0) \vee \ldots \vee \neg\phi(s_k)) \tag{2.23}$$

In the above formulas, $I(s_0)$ corresponds to the *characteristic function* of the initial state, $T(s_i, s_{i+1})$ denotes the transition relation, and $[\![\mathcal{M}]\!]_k$ expresses the *unrolling* of the transition relation up to the bound k. $[\![\neg\phi]\!]_k$ corresponds to the negation of the property ϕ. It states that $\neg\phi$ holds along the execution path from bound 0 to k. Therefore, if the formula (2.21) is satisfiable, a counterexample has been found, which means that the model violates the property ϕ. In practice, the bound is increased progressively to look for longer possible counterexample up to a given maximum length.

2.5.2 Safety Check Example

The following example explains how to use BMC to check safety property in a two-bit counter. We use Kripke structure as a model of the system (see Figure 2.9).

Let (a, b) be the Boolean variables that represent the value of bits at each state. The initial state and the transition relation are given as follows:

- initial state $I(S_0) : (\neg a_0 \wedge \neg b_0)$,

- transition relation $T(s_i, s_{i+1}) : (a_{i+1} = (a_i \neq b_i) \wedge b_{i+1} = \neg b_i)$.

The safety property to be checked is $\phi = AG(\neg a \vee \neg b)$, which means "it is always true that either of the two bits is false (or 0)." To determine whether a counterexample exists, we are more concerned with the negation of ϕ, that is $EF(a \wedge b)$, which means "it is possible to reach a state where both bits are true (or 1)."

By applying the equations (2.22), (2.21), and (2.23) for $k = 2$, we obtain the following:

$$[\![\mathcal{M}, \phi]\!]_2 \coloneqq I(S_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge [\![\neg\phi]\!]_2$$

that is,

$$\begin{aligned} [\![\mathcal{M}, \phi]\!]_2 \quad \coloneqq \quad & (\neg a_0 \wedge \neg b_0) \wedge \\ & (a_1 = (a_0 \neq b_0) \wedge b_1 = \neg b_0) \wedge \\ & (a_2 = (a_1 \neq b_1) \wedge b_2 = \neg b_1) \wedge \\ & ((a_0 \wedge b_0) \vee (a_1 \wedge b_1) \vee (a_2 \wedge b_2)). \end{aligned}$$

The formula $[\![M, \phi]\!]_2$ is unsatisfiable. Thus, unrolling the transition relation up to $k = 2$ provides no counterexample. On the other hand, if we continue with $k = 3$, we could see that $[\![M, \phi]\!]_3$ provides a satisfiable formula that violates the safety property.

2.5.3 BMC and Liveness Properties

While it is straightforward to check safety properties with BMC (see e.g.[PICW04]), liveness properties are quite complicated. For example, to check whether the liveness property AG EFP holds, we need to verify that all paths from the initial state lead to P. This is typically done in the following way: first, check if a state satisfying $\neg P$ can be reached within bound 1. If yes, then P does not hold in all existing paths within bound 1, thus we need to increase the bound to 2. We continue the process for bound 2 while asserting that $\neg P$ holds for bound 1. If we still find a state satisfying $\neg P$ in bound 2, we can keep going by increasing the bound. However, if we obtain an unsatisfaiable assignment at any given bound, we can conclude that the liveness property holds. More details on how to use BMC for checking liveness properties can be found in [BCCZ99, GGA05].

Note that, when Biere *et al.* first presented the BMC idea in [BCCZ99], they used Kripke structures to model their system. Subsequent works on BMC adopt alternative modeling techniques. Still, the idea which underlies these works remains the same as the original one. For instance, timed automata is used in [WZP03, Sor02], Petri nets in [Hel01, WLP04], and transition systems in [JHN03, Jus04]. As mentioned before, we will also use transition systems in this work.

Chapter 3

Exixting Techniques

At first, research on BMC was largely revolved around synchronous systems. The characteristic of these systems, in which all variables change their values simultaneously, matches perfectly a BMC operation. More recently, however, a great deal of work on BMC for asynchronous systems has been published. In this chapter we will review some of the most interesting BMC techniques that exist in the literature. Analyses and comparisons of most of these techniques can be found in [PBG05, AKMM03, ADA+05]. We will also discuss briefly how to deal with the completeness issue, and how to combine SAT with unbounded model checking.

3.1 Standard Methods

In general, all BMC techniques fall into two categories: (i) techniques related to the SAT solving and (ii) techniques related to the BMC process itself. The goal, though, is always the same in every technique: to find a counterexample as fast as possible, if it exists. These techniques include reduction of the BMC problem into another problem, tuning SAT solvers for BMC, learning and preprocessing, incremental SAT solving, and strengthening and encodings of properties.

Reduction to Another Problem Some techniques consist in reducing the BMC problem into a different kind of problem. For instance, the technique in [HN01a, HN01b] use the logic programming known as *answer set programming* [MT99] to reduce the BMC problem for 1-safe Petri nets into computation of stable models of logic programs. Another approach for 1-safe Petri nets, introduced in [Hel01], consists in reducing the problem into Boolean circuit satisfiability checking, using three semantics known as process, step, and interleaving semantics. In [ACKS02], BMC has been applied to timed system. In this technique the BMC problem is reduced into satisfiability problem of a math-formulas, that is a Boolean combination of propositional variables and linear mathematical relations over real variables.

Tuning SAT Solvers for BMC Other researches focus on modifying some features in the SAT solvers to match the requirements of a BMC operation [Sht00, JAS04, SZ03]. These solvers are mainly based on the DPLL algorithm, which solves a formula by traversing a search tree as described in Section 2.4.1. The structure of a BMC formula makes it possible to use alternative branching methodologies during the traversal [WJHS04, Zar04], or to extend the default algorithm so that more conflict clauses can be added [Sht00, Str04]. In addition, the formulas can be simplified using different methods such as the ones in [Kue04, Vel04a] before feeding them to the solvers. These techniques enable the solvers to speed their execution. Therefore, the performance of BMC using them will also improve.

Learning and Preprocessing Using learning mechanisms to extract information from a circuit can improve both SAT solving and BMC. For example, [CNQ03, GGW+03] add some information learned from a BDD-based analysis to the SAT engine to speed its execution. The method in [CNQ03] performs first a BDD-based approximate reachability analysis, then converts the BDD at each step into clauses to constrain the SAT solver, whereas the one in [GGW+03] extracts clauses dynamically or statically, and limits the learning process only to local regions to prevent excessive computational and memory costs.

For every unsatisfiable CNF formula, it is possible to generate a proof of unsatisfiability from the solver. The set of clauses from the original formula needed for generating such a proof is called *unsatisfiable core*. Extracting such information is proven to be useful especially in an abstraction-refinement process because it gives an explanation as to why the formula is unsatisfiable and can guide the refinement. [MA03] suggests a method for generating such a proof. [McM03, ZPHS05, GS05, CCK+02] show how to use it in BMC.

The authors in [AH04] introduce a preprocessing engine called *SIMP2C*, which converts indirect implications learned via logic simulation into constraint clauses. These clauses are replicated throughout the whole BMC run and provided to the SAT solver to speed up its decision process. This reduction technique is also adopted in the *Hypre* preprocessor [BW03], which, in addition, can perform equality reduction.

The preprocessing engine called *NiVER* introduced in [SP05] uses a variable elimination technique that does not increase the size of the CNF formula. Though not directly related to clause learning, this technique also improves SAT solving. Hypre and NiVER are designed for general purposed SAT-solver, yet they can also be used for a BMC operation.

Incremental SAT Solving As it turns out, a CNF formula generated at a given BMC step is, in fact, an extension of the one generated immediately before it. Consequently, all propositional formulas obtained during a BMC operation maintain the same structure. Many SAT solvers exploit this feature to perform *incremental* solving [Een03, JS05, ZPH04, WKS01, Str01]. That is, the possibility of reusing some conflict clauses learned during one SAT check when solving the next instance. These learned

clauses prevent the solver from examining some regions in the search space that contain no solution. As a result, the solving speed improves, and so does the entire BMC operation. BMC encodings, built especially for incremental SAT such as the one in [HJL05], will improve even more the whole process. The authors in [JS05] use a general criterion for filtering the clauses to be reused, and a *distillation* technique for preventing the exploration of unneeded regions in the search space.

Property Strengthening While incremental learning can improve the speed of SAT solving, it does not reduce the depth needed to prove a property. On the other hand, *property strengthening* allows a property to be falsified or verified within small depth. It does so, by enlarging the property so that a counterexample can be detected at an earlier depth. For example, the framework for proving a property via structural analysis, introduced in [BKA02], combines a BDD-based backward unfolding with a *target enlargement* mechanism in order to make unreachability proof much more easier. It computes first an overapproximation of the so-called system diameter. Then, performs a mix SAT-based forward unfolding and the BDD-based backward one from the target states. In [dMRS03], Moura *et al.* strengthen the property by adding to it a counterexample generated from a failed induction step. In this BMC-induction combination, the strengthening power depends largely on the generated counterexample.

Encodings of Properties It is also possible to improve the performance of BMC by finding a better way to encode the properties. For LTL properties, some methods use encodings that allow direct translation of the formula into propositional formula [CPRS02, LBHJ04], while others encode the LTL property using Büchi automata [dMRS02, CKOS04]. Some works also deal with CTL properties such as the ones presented in [PWZ02a, PWZ02b].

3.2 Completeness

Numerous techniques have been developed so far in order to make BMC complete, that is, allowing it not only to look for a counterexample, but also to provide a "yes" or "no" answer, just as in standard model checking (see Figure 1.1). These techniques include computing the completeness threshold, temporal induction, interpolation, ATPG-based method, and abstraction.

Computing the Completeness Threshold The idea consists in computing the value of a bound known as *completeness threshold* [CKOS04] or also *termination length* [AS06], within which a counterexample is expected to be found, if it exists. However, computing this bound is not trivial. Furthermore, its value depends on the property, the model, and the termination criterion. For reachability properties, for example, completeness threshold coincides with the *system diameter* [BCCZ99, BK04] which is the longest shortest path

between two states. Thus, to check whether these properties hold, it is enough to increase the bound up to the system diameter, and if no counterexample appears within this limit, then the system is correct. But determining the diameter is no easy task, and its value can be too large to handle. For this reason, Biere *et al.* propose in [BCCZ99] an overapproximation called *recurrence diameter* which corresponds to the longest loop-free path between two states. This number, while big, is proven to be much easier to compute than the diameter [KS03].

Temporal Induction Here, the goal is to prove the following two conditions which constitute the base case and the induction step respectively. Given a bound k, (i) the property holds in all paths of length k from the initial states and (ii) for any path of length k where the property holds in all states, it is not possible from the last state of the path to reach a state where the property does not hold. In addition, to ensure completeness, all states of the paths in the base condition must be distinct [SSS00, Een03]. Induction algorithms increase the value of k until the two conditions above are proven or a counterexample is found.

Interpolation An interpolant for the pair (A, B) is a formula P such that (i) $A \Rightarrow P$ (ii) $P \wedge B$ is unsatisfiable and (iii) P refers only to the common variables of A and B.

If a BMC formula is unsatisfiable, it can be partitioned into conjunction of two formulas A and B, such that A represents the initial states with the first transition, whereas B encodes the rest of the transition up to an initial bound k along with the property. In [McM03, McM05] McMillan introduces an algorithm that computes the interpolant of these two formulas. This interpolant corresponds to an overapproximation of the set of states reachable from an initial state in one step. Using this interpolant, McMillan's algorithm can determine whether a fixpoint is reached at bound k, or a real counterexample occurs. If it reaches a fixpoint, then the property holds and the algorithm stops. If no real counterexample appears, the algorithm increases the value of k until either it finds a real counterexample or reaches the system diameter. If it reaches the diameter without finding a real counterexample, then the system is correct.

ATPG-based Method In [IPC03], Iyer *et al.* suggest a quite different approach based on the technique known as automated test pattern generation (ATPG). Their algorithm performs an ATPG-based backtracking traversal from states where the property is false. If it reaches an initial state during the traversal, then it generates a counterexample. Otherwise, the property is proven to be true and the algorithm stops. To guarantee completeness, it uses additional Boolean constraints to mark already visited states, thus avoiding searching them again in future iterations.

Abstraction Completeness can also be achieved through abstraction-refinement techniques, most of which use the unsatisfiable core to refine the model. The algorithm in

[CCK+02], for example, starts with an initial abstraction of the model. Using a BDD-based procedure, it checks first whether the property holds in the abstract model. If yes, then the system is correct. Otherwise, it runs a BMC session on the concrete model, taking the length of the abstract counterexample as a BMC length. If the BMC operation still returns a counterexample, then the system is incorrect. Otherwise, it uses the unsatisfiable core to refine the abstract model and restart again with the BDD-based procedure.

The *proof-based* method introduced in [MA03], on the other hand, starts with a BMC session with an initial length k. If the formula is unsatisfiable, the algorithm uses the unsatisfiable core to create a new abstraction of the model. Then, it runs a standard BDD-based model checker on the abstract model to determine whether it is correct or not. If it is correct, the algorithm terminates. Otherwise, the BDD-based model checker produces a counterexample whose length k', which is always greater than k, is used later on as the next BMC bound.

The algorithm in [Kro05] creates an abstraction by using a method known as *cut-point insertion*. That is, it removes some parts of the design and replaces them with nondeterministically chosen inputs. The result is an abstract model that overapproximates the concrete one, and preserves safety properties. This algorithm also uses the unsatisfiable core to refine the model, and to ensure completeness, it can identify reasonably small completeness threshold.

3.3 SAT with Unbounded Model Checking

Adapting SAT to work with unbounded model checking (UMC) makes the image computation process more tractable. SAT-based UMC performs the image computation using *all-solution* SAT-solvers. That is, solvers that compute *all* satisfying solutions (see e.g. [LHS04]), as opposed to general-purposed SAT-solvers which provide only a single solution if it exists. In contrast to BDD-based approaches, this technique stores the set of reached states as set of clauses, and adds some *Blocking clauses* in order to avoid reaching the same solution space in the future. The clauses generated at each iteration can be used as the starting states for the next iteration.

For example, [GYA+01] exploits the strengths of both SAT and BDDs in order to improve the image computation during a model checking operation. This is achieved by decomposing first the image computation problem into many sub-problems using a SAT-based algorithm. Each of these sub-problems is handled later on by a conventional BDD approach. This allows the state space to be represented as BDDs, and the transition relation as a CNF formula. The same authors also introduce in [GYAG00] a CNF partitioning technique which, combined with the previous method, improves the efficiency of the associated SAT algorithm. In [CCK03], Chauchan *et. al* show how to perform an image computation with a pure SAT procedure. Unlike the method in [GYA+01], they use clausal form to represent both the state space and the transition relation.

The method in [ABE00, BC00], on the other hand, adapts existing BDD-based algorithms to work with SAT-solvers. This adaptation increases the class of systems that can be verified using traditional symbolic methods. In [GGA04], the main purpose is to use some special features from SAT-based algorithm for tackling the quantifier elimination problem, which is one of the most important operations in symbolic model checking.

SAT procedure is also used in [Cla02] to determine whether a counterexample obtained from an abstract model is real or spurious. In case the counterexample is spurious, the algorithm refines the model using integer linear programming and machine learning techniques.

3.4 Existing BMC Tools

Many BMC-related tools have been developed so far. While some of them are built especially to perform BMC operation, others can also handle BDD-based model checking. The most widely used is the one called NuSmv [CCGR99] which has incorporated BMC implementation since its version 2.0. This tool uses *reduced Boolean circuits* [ABE00] as an intermediate representation for the propositional formulas.

The BMC feature has also been included in Vis [BHSV+96] since its version 2.0. Unlike NuSmv, it does not use intermediate representation for generating formulas.

Chapter 4

Encoding Methods

In this chapter, we introduce our BMC encoding methods for systems modeled as synchronized product of transition systems. Recall that BMC is performed by increasing the bound progressively to look for a longer possible counterexample up to a given length (see Algorithm 1.2). The objective is mainly to generate a Boolean formula out of the unrolling operation, and doing so in a way to obtain a formula easily manageable by a SAT-solver. Normally, when we increase the bound, the size of the generated Boolean formula grows as well. This situation, combined with the possible huge number of steps needed for finding a counterexample in concurrent systems, constitutes a major hurdle in BMC. To cope with this issue, it is then necessary to use an encoding that not only generates succinct formula, but also reduces the bound required for finding a counterexample. In this way, the SAT-solver's task will be alleviated, and the execution will go faster. With this end in view, we based our method on the following fundamental points.

Fire multiple events at once Traditionally, concurrent systems are unrolled using the so-called *interleaving execution*. However, interleaving allows us only to fire one event at a time, which is not quite helpful for concurrent systems, especially when dealing with synchronized product ones. In our method, we provide a way for firing several events together. The idea is to unroll the system using *breadth-first search* approaches and provide an encoding that can capture all reachable states in a given step.

Reduce the bound using chaining traversal We can reduce the bound by combining the above approach with the method known as chaining traversal. This method allows consecutive enabled events to be fired together in one step, thus creates a domino effect that substantially reduces the bound. In order to work properly, however, this method requires a well-established event order. For that, we also provide some event ordering techniques to be used with the chaining operation.

The rest of the chapter is organized as follows. In Section 4.1 we review some work in the literature related to this topic. In Section 4.2 we describe how to represent a

transition system symbolically. In Section 4.3 we introduce a BMC encoding devoted for single independent systems. In Section 4.4 we detail the encoding for synchronized transition systems. In Section 4.5 we show how to express reachability properties. We end the Chapter with the chaining method in Section 4.6.

4.1 Related Work

We will now review some BMC encodings in the literature related to asynchronous concurrent systems. In [Jus04, JHN03] Jussila *et al.* introduce a BMC technique for systems modeled as labeled transition systems (LTSs). Their idea consists in translating LTSs into Boolean formula using some predefined translation predicates. This technique allows several independent actions to happen simultaneously. As a result, it can shorten the bound needed to detect a property violation. The encoding in [Hel01] also adopts a similar idea for Petri net models. It uses some translation rules to obtain a *Boolean circuit* (BC) from the unrolling operation, and uses a BC-based solver to check the satisfiability of the generated circuit. By using translation rules, as in the above approaches, the Boolean formula is not constructed directly from the model itself. Instead, it is derived from the newly obtained representation after applying the rules.

By contrast, the methods in [OTK04] and [Sor02], which apply to Petri nets and timed automata respectively, derive the Boolean formula directly from the definition of the system itself rather than relying on transformation rules. In addition, the authors of [Sor02] show how to make BMC for timed automata complete by setting up a lower and an upper bounds for the length k of counterexample. On the other hand, while the method in [OTK04] has produced interesting results, its performance depends on a preprocessing algorithm that computes a total order of the transitions.

In [GG08b], Ganai *et al.* apply BMC to multi-thread concurrent systems with shared memory, such as multi-processor computers. They use their method to check safety properties such as *data races* for these types of systems, which they model using *extended finite state machine* (EFSM). In their approach, they first create independent models for each individual thread in the system, then, to ensure synchronization, explicitly add additional synchronization constraints into the model. To reduce the size of these constraints they do not allow local wait cycles (i.e. there are no self-loops in local read/write blocks with shared accesses.) They also transform the model using some property preserving techniques, which further reduce the size of concurrency constraints. They check the satisfiability of the generated formula with SMT-based solver instead of a standard one.

The methods most closely related to ours are the ones introduced in [Jus04, JHN03], which also deal with synchronized product of transition systems. However, the principles are quite different since, in our work, we do not use transformation rules. The method in [OTK04], while using direct transformation, differs from ours in the type of system to which it applies—Petri net vs. transition system. The method in [Sor02] also use direct transformation but it is for timed systems.

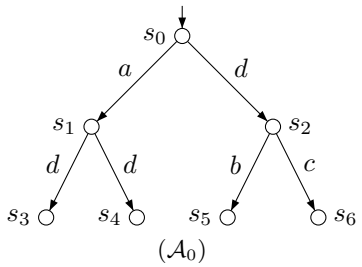


Figure 4.1: A Transition system.

4.2 Symbolic Representation

Representing a Single Transition

A transition system $\mathcal{A}$ is usualy represented as $\mathcal{A} = (S, \Sigma, T, s_0)$ (see Section 2.1). We can extend this representation as $\mathcal{A} = (V, V', S, \Sigma, T, s_0)$, where V and V' denote respectively the set of *current* and *next-state* Boolean variables for $\mathcal{A}$, such that for each current-state variable $v \in V$, there exists a corresponding next-state variable $v' \in V'$ and vice versa.

Now, let s_p and s_q be two states such that $s_q \in To(s_p)$. We define $T(s_p, s_q)$, the transition from s_p to s_q, as follows:

$$T(s_p, s_q) = \begin{cases} \mathcal{X}_{\{s_p\}} \wedge \neg v'_p \wedge v'_q \wedge \bigwedge_{v_k \in V \setminus \{v_p, v_q\}} (v_k \leftrightarrow v'_k) & \text{if } s_p \neq s_q \text{ and } \mathcal{E}(s_q) \neq \varnothing \\ \mathcal{X}_{\{s_p\}} \wedge \neg v'_p \wedge \neg v'_q \wedge \bigwedge_{v_k \in V \setminus \{v_p, v_q\}} (v_k \leftrightarrow v'_k) & \text{if } s_p \neq s_q \text{ and } \mathcal{E}(s_q) = \varnothing \\ \mathcal{X}_{\{s_p\}} \wedge v'_p \wedge \bigwedge_{v_k \in V \setminus \{v_p\}} (v_k \leftrightarrow v'_k) & \text{if } s_p = s_q \end{cases} \tag{4.1}$$

In practice, to express a transition between two steps i and $i+1$, we encode the current-state variable using the index i and the next-state variable using $i+1$. Thus, assuming that s_p is reached at step i and s_q at step $i+1$, equation (4.1) above becomes:

$$T(s_{p,i}, s_{q,i+1}) = \begin{cases} \mathcal{X}_{\{s_{p,i}\}} \wedge \neg v_{p,i+1} \wedge v_{q,i+1} \\ \quad \wedge \bigwedge_{v_k \in V \setminus \{v_p, v_q\}} (v_{k,i} \leftrightarrow v_{k,i+1}) & \text{if } s_p \neq s_q \text{ and } \mathcal{E}(s_q) \neq \varnothing \\ \mathcal{X}_{\{s_{p,i}\}} \wedge \neg v_{p,i+1} \wedge \neg v_{q,i+1} \\ \quad \wedge \bigwedge_{v_k \in V \setminus \{v_p, v_q\}} (v_{k,i} \leftrightarrow v_{k,i+1}) & \text{if } s_p \neq s_q \text{ and } \mathcal{E}(s_q) = \varnothing \\ \mathcal{X}_{\{s_{p,i}\}} \wedge v_{q,i+1} \\ \quad \wedge \bigwedge_{v_k \in V \setminus \{v_p, v_q\}} (v_{k,i} \leftrightarrow v_{k,i+1}) & \text{if } s_p = s_q \end{cases} \tag{4.2}$$

Example 4.2.1 For the TS $\mathcal{A}_0$ in Figure 4.1, the equation for $T(s_{0,0}, s_{1,1})$ is given as:

$$\begin{aligned} T(s_{0,0}, s_{1,1}) \quad = \quad & v_{0,0} \wedge \neg v_{0,1} \wedge v_{1,1} \wedge (v_{2,0} \leftrightarrow v_{2,1}) \wedge (v_{3,0} \leftrightarrow v_{3,1}) \wedge (v_{4,0} \leftrightarrow v_{4,1}) \\ & \wedge (v_{5,0} \leftrightarrow v_{5,1}) \wedge (v_{6,0} \leftrightarrow v_{6,1}) \end{aligned}$$

Representing Multiple Transitions

Given two sets of states S_p and S_q such that $To(S_p) = S_q$. Assume S_p is reached at step i and S_q at step $i+1$. To capture the execution of multiple transitions together from S_p, we define $T(S_{p,i}, S_{q,i+1})$ as follows:

$$T(S_{p,i}, S_{q,i+1}) = \bigwedge_{s_p \in S_p; s_q \in S_q} T(s_{p,i}, s_{q,i+1}) \tag{4.3}$$

Equation 4.3 is useful for capturing the firing of multiple events together from the states where they are enabled.

Example 4.2.2 For the TS $\mathcal{A}_0$, assume that at step 0, we start with $S_0 = \{s_0\}$, and at step 1 we reach $S_1 = \{s_1, s_2\}$. Then $T(S_{0,0}, S_{1,1})$ is given as:

$$\begin{aligned} T(S_{0,0}, S_{1,1}) \quad = \quad & v_{0,0} \wedge \neg v_{0,1} \wedge v_{1,1} \wedge v_{2,1} \wedge (v_{3,0} \leftrightarrow v_{3,1}) \wedge (v_{4,0} \leftrightarrow v_{4,1}) \\ & \wedge (v_{5,0} \leftrightarrow v_{5,1}) \wedge (v_{6,0} \leftrightarrow v_{6,1}) \end{aligned}$$

Note that, the example above may appear incomplete as it concerns only a transition between two consecutive steps as described in equation (4.3). We will see in the next section how to encode a complete BMC execution starting from the initial state up to a given bound.

4.3 Encoding for Independent Systems

In this section, we will discuss how to encode a BMC operation for a single independent TS. We will start with the standard interleaving execution, and continue with a method based on the so-called breadth-first search.

4.3.1 Interleaving Execution

Introduction

To capture the behavior of concurrent systems it is not uncommon to use the method known as *interleaving execution* [CGP99], which allows only one event, within one transition, to fire at each step. As a result, with this technique, we can reach at most one single state at each step, and can only obtain an execution path formed by succession of single transitions from bound 0 to k. The events in this path follow a linear order known as *interleaving sequence*.

Application to BMC

Given a TS $\mathcal{A} = (S, \Sigma, T, s_0)$, a property ϕ, and a bound k. The BMC formula for $\mathcal{A}$, defined in terms of interleaving execution, is given as

$$int[\![\mathcal{A}, \phi]\!]_k = int[\![\mathcal{A}]\!]_k \wedge [\![\neg\phi]\!]_k \tag{4.4}$$

where

- $int[\![\mathcal{A}]\!]_k$ defines the unrolling of $\mathcal{A}$ up to bound k using interleaving execution, i.e.

$$int[\![\mathcal{A}]\!]_k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \tag{4.5}$$

- $I(s_0)$ denotes the characteristic function of the initial state, and is defined as

$$I(s_0) = \mathcal{X}_{\{s_{0,0}\}} \wedge \bigwedge_{s_p \neq s_0} \neg v_{p,0} \tag{4.6}$$

- $T(s_i, s_{i+1}) = T(s_{p,i}, s_{q,i+1})$ such that s_p is a state reached at step i, and s_q is one reached at step $i+1$.

- $[\![\neg\phi]\!]_k$ corresponds to the negation of the property at bound k.

For the moment we are only interested in the unrolling part of the BMC formula. Hence, we will not detail $[\![\neg\phi]\!]_k$ until Section 4.5.

Example 4.3.1 For the system $\mathcal{A}_0$ in Figure 4.1 we can obtain the following interleaving execution for bound $k = 2$

$$\begin{aligned} int[\![\mathcal{A}]\!]_2 \quad &= \quad I(s_0) \\ &\quad \wedge T(s_{0,0}, s_{1,1}) \\ &\quad \wedge T(s_{1,1}, s_{4,2}) \\ \\ &= \quad \big(v_{0,0} \wedge \neg v_{1,0} \wedge \neg v_{2,0} \wedge \neg v_{3,0} \wedge \neg v_{4,0} \wedge \neg v_{5,0} \wedge \neg v_{6,0}\big) \\ &\quad \wedge \Big(v_{0,0} \wedge \neg v_{0,1} \wedge v_{1,1} \wedge (v_{2,0} \leftrightarrow v_{2,1}) \wedge (v_{3,0} \leftrightarrow v_{3,1}) \\ &\qquad \wedge (v_{4,0} \leftrightarrow v_{4,1}) \wedge (v_{5,0} \leftrightarrow v_{5,1}) \wedge (v_{6,0} \leftrightarrow v_{6,1})\Big) \\ &\quad \wedge \Big(v_{1,1} \wedge \neg v_{1,2} \wedge v_{4,2} \wedge (v_{2,1} \leftrightarrow v_{2,2}) \wedge (v_{3,1} \leftrightarrow v_{3,2}) \\ &\qquad \wedge (v_{5,1} \leftrightarrow v_{5,2}) \wedge (v_{6,1} \leftrightarrow v_{6,2}) \wedge (v_{0,1} \leftrightarrow v_{0,2})\Big) \end{aligned}$$

Theorem 4.3.1
Given a transition system $\mathcal{A}$, a property ϕ, and a bound k. If $int[\![\mathcal{A}, \phi]\!]_k$ is satisfiable, then $\mathcal{A} \not\models \phi$.

Proof Assume that $int[\![\mathcal{A}, \phi]\!]_k$ is satisfiable. From the equation (4.5) we have $T(s_i, s_{i+1}) = 1$ iff s_{i+1} is reachable from s_i in one step. Obviously, this implies that $T(s_0, s_1) \wedge \ldots \wedge T(s_{k-1}, s_k) = 1$ iff $s_0, \ldots, s_k$ represents a run from the initial state. Since $int[\![\mathcal{A}, \phi]\!]_k = I(s_0) \wedge T(s_0, s_1) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge [\![\neg\phi]\!]_k$ is satisfiable, we can conclude that a state violating the property ϕ is reachable in k steps or less from the initial state. ■

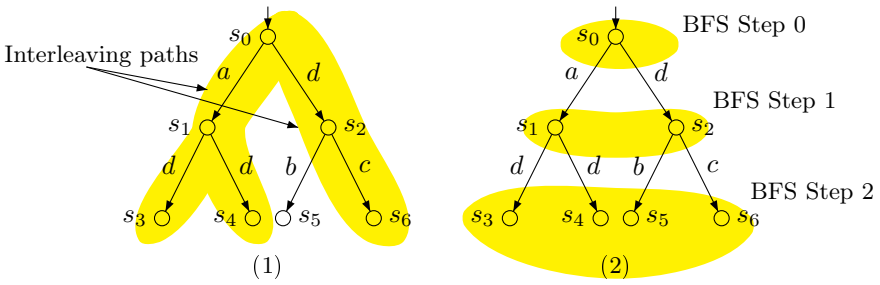


Figure 4.2: Interleaving (1) vs. BFS. (2)

Theorem 4.3.2

Given a transition system $\mathcal{A}$, a property ϕ, and a bound k. $\mathcal{A} \not\models \phi$ implies $\exists k \in \mathbb{N}$ such that $int[\![\mathcal{A}, \phi]\!]_k$ is satisfiable.

PROOF Assume that $\mathcal{A} \not\models \phi$, i.e. there exists a path from the initial state to a state where ϕ is violated. Assume that k is the length of that path. Since ϕ is violated at execution step k, its negation is thus satisfied at that step. Therefore, we have $int[\![\mathcal{A}]\!]_k \wedge [\![\neg\phi]\!]_k$ is satisfiable, which means $int[\![\mathcal{A}, \phi]\!]_k$ is satisfiable. ■

With the interleaving method, we need to analyze all possible paths in the system between bound 0 and k in order to ensure that no counterexample has been missed. Doing so is onerous, however, and time-consuming for concurrent systems. One way to deal with this issue is to use breadth-first search method as we will explain in the next section.

4.3.2 Breadth-First Search Execution

Introduction

Breadth-first search (BFS) is a kind of algorithm for searching a graph. It systematically explores the edges of the graph to "discover" *all* reachable vertices from a given *source*. It also computes the distance—smallest number of edges—from the source to each reachable vertex.

A vertex is *discovered* the first time it is encountered during the search. The BFS algorithm expands the *search frontier* between *discovered* and *undiscovered* vertices uniformly across the breadth of the frontier. That is, it discovers all vertices at distance k from the source before discovering any vertices at distance $k + 1$. To do so, BFS works as follows. Starting at the source, it visits all the adjacent vertices. Then for each of those new vertices, it visits their undiscovered adjacent ones, and so on, until every vertex has been visited. Detailing the BFS algorithm is out of the scope of this work, though. Interested reader is invited to check [CLRS09].

Using BFS allows us to fire multiple events together at once. This implies that, at each step, we reach a set of states rather than a single state as in the interleaving execution. Hence, from bound 0 to bound k we obtain an execution path formed by sets of states. In addition, since BFS can discover all reachable states from the initial one, using it ensures us that no counterexamples will be missed during the BMC operation.

Example 4.3.2 Figure 4.2 shows the difference between interleaving and BFS execution. With the interleaving execution (Figure 4.2 (1)) we have to analyze all the following paths:

(P1) $s_0 \xrightarrow{a} s_1 \xrightarrow{c} s_3$

(P2) $s_0 \xrightarrow{a} s_1 \xrightarrow{c} s_4$

(P3) $s_0 \xrightarrow{c} s_2 \xrightarrow{b} s_5$

(P4) $s_0 \xrightarrow{c} s_2 \xrightarrow{d} s_6$

If we omit the path (P2), for instance, and a counterexample appears at s_4, then we will miss it. On the other hand, with BFS (Figure 4.2 (2)) we only need to analyze the following path.

$$\{s_0\} \xrightarrow{\{a,c\}} \{s_1, s_2\} \xrightarrow{\{c,b,d\}} \{s_3, s_4, s_5, s_6\}$$

The real hurdle with the BFS method, however, is to determine the exact state in which the property violation occurs. For instance, in Example 4.3.2 above, if a property is violated at bound 2, the encoding we use should be able to tell us in which of the states in the set $\{s_3, s_4, s_5, s_6\}$ this property violation really occurs. We will see how to cope with this problem later in this chapter.

Application to BMC

To unroll the system using BFS, we proceed as follows. We apply equation (4.3) which describes the transition between $From$ and To sets. At each step, we put into the $From$ set all states reached at the current step, and we construct the To set by firing all the events that are currently enabled. We initialize the $From$ set with the initial state.

Now, given a TS $\mathcal{A} = (S, \Sigma, T, s_0)$, a property ϕ, and a bound k. The BMC problem $\mathcal{A} \models_k \phi$, defined in terms of BFS execution, is equivalent to the satisfiability problem of the following formula:

$$bfs[\![\mathcal{A}, \phi]\!]_k = bfs[\![\mathcal{A}]\!]_k \wedge [\![\neg\phi]\!]_k \tag{4.7}$$

where

- $bfs[\![\mathcal{A}]\!]_k$ defines the unrolling of $\mathcal{A}$ up to bound k using BFS execution, i.e.

$$bfs[\![\mathcal{A}]\!]_k = I(S_0) \wedge \bigwedge_{i=0}^{k-1} T(S_i, S_{i+1}) \tag{4.8}$$

- $S_0 = \{s_0\}$
- $I(S_0) = I(s_0)$,
- $T(S_i, S_{i+1}) = T(S_{p,i}, S_{q,i+1})$ such that S_p is the set of states reached at step i, and S_q is the one reached at step $i+1$.

Note that, at each step we have $S_i \xrightarrow{E_i} S_{i+1}$, where E_i denotes the set of events that fire at step i. For a single independent system we have $E_i = \mathcal{E}_i$, where $\mathcal{E}_i$ corresponds to the set of events that are enabled at S_i (see Definition 2.1.2).

Example 4.3.3 Let us take again the TS $\mathcal{A}_0$ in Figure 4.1. The BMC formula $bfs[\![\mathcal{A}]\!]_2$ for bound $k = 2$ is given as,

$$\begin{aligned} bfs[\![\mathcal{A}]\!]_2 \;=\; & I(S_0) \\ & \wedge T(S_{0,0}, S_{1,1}) \\ & \wedge T(S_{1,1}, S_{2,2}) \\ \\ =\; & \big(v_{0,0} \wedge \neg v_{1,0} \wedge \neg v_{2,0} \wedge \neg v_{3,0} \wedge \neg v_{4,0} \wedge \neg v_{5,0} \wedge \neg v_{6,0}\big) \\ & \wedge \Big(v_{0,0} \wedge \neg v_{0,1} \wedge v_{1,1} \wedge v_{2,1} \wedge (v_{3,0} \leftrightarrow v_{3,1}) \\ & \quad \wedge (v_{4,0} \leftrightarrow v_{4,1}) \wedge (v_{5,0} \leftrightarrow v_{5,1}) \wedge (v_{6,0} \leftrightarrow v_{6,1})\Big) \\ & \wedge \Big(v_{1,1} \wedge v_{2,1} \wedge \neg v_{1,2} \wedge \neg v_{2,2} \wedge v_{3,2} \wedge v_{4,2} \wedge v_{5,2} \wedge v_{6,2} \\ & \quad \wedge (v_{0,1} \leftrightarrow v_{0,2})\Big) \end{aligned}$$

where,

$$\begin{aligned} & S_0 = \{s_0\};\ S_1 = \{s_1, s_2\}; S_2 = \{s_3, s_4, s_5, s_6\}; \\ & E_0 = \{a, d\}; E_1 = \{c, b, d\} \end{aligned}$$

Theorem 4.3.3
Given a transition system $\mathcal{A}$, a property ϕ, and a bound k. If $bfs[\![\mathcal{A}, \phi]\!]_k$ is satisfiable, then $\mathcal{A} \not\models \phi$.

PROOF $T(S_i, S_{i+1})$ can be seen as a combination of several individual transitions of the form $T(s_i, s_{i+1})$ between two steps i and $i+1$. This assumption applies for all $T(S_i, S_{i+1}), i \in [0, k]$. We can, therefore, obtain many runs $s_0, \ldots, s_k$ formed by these individual transitions from step 0 to k. It follows that, we also have a run $S_0, \ldots, S_k$ formed by the combination of these individual runs. Applying Theorem 4.3.1, we have $T(s_0, s_1) \wedge \ldots \wedge T(s_{k-1}, s_k) = 1$ for each of the individual runs. Hence, we also have $T(S_0, S_1) \wedge \ldots \wedge T(S_{k-1}, S_k) = 1$ since the run $S_0, \ldots, S_k$ is obtained by combining all $s_0, \ldots, s_k$ runs. Therefore, since $bfs[\![\mathcal{A}, \phi]\!]_k = I(S_0) \wedge T(S_0, S_1) \wedge \ldots \wedge T(S_{k-1}, S_k) \wedge [\![\neg\phi]\!]_k$ is satisfiable, a state violating the property ϕ is reachable in k steps or less from the initial state. ∎

Theorem 4.3.4

Given a transition system $\mathcal{A}$, a property ϕ, and a bound k. $\mathcal{A} \not\models \phi$ implies $\exists k \in \mathbb{N}$ such that $bfs[\![\mathcal{A}, \phi]\!]_k$ is satisfiable.

Proof The proof of this theorem is similar to the one for Theorem 4.3.2 except that here, we are dealing with set of states rather than a single state. Assume that $\mathcal{A} \not\models \phi$. This means there exists a reachable set of states containing a state that violates the property ϕ. This set is reached from the initial state through a run formed by all the reachable states at each execution step. Assume k is the length of the run. Since ϕ is violated at step k, its negation is thus satisfied at that step. Therefore, we have $I(S_0) \wedge T(S_0, S_1) \wedge \ldots \wedge T(S_{k-1}, S_k) \wedge [\![\neg\phi]\!]_k$ is satisfiable, which means $bfs[\![\mathcal{A}, \phi]\!]_k$ is satisfiable. ■

4.4 Encoding for Synchronized Systems

Asynchronous concurrent systems are composed of many components that communicate between each other. We model them using *synchronized product of transition systems*. The approach in Section 4.3 allows us only to encode the BMC execution of a single component—considered independent—without taking into account the synchronization. Starting from now on, we will always work on synchronized systems.

In this section, we introduce some encoding methods for these systems. As before, we will start with the interleaving execution and continue with the BFS one.

4.4.1 Interleaving Execution

Introduction

Let us first start by defining the synchronized product of TSs in terms of interleaving execution.

Definition 4.4.1 (Synchronized Product of TSs)

Let $\mathcal{A}_0, \ldots, \mathcal{A}_m$ be TSs. Their synchronized product *defined in terms of interleaving execution, denoted $(\mathcal{A}_0, \ldots, \mathcal{A}_m)$ is the TS $\mathcal{A} = (S, \Sigma, T, s_0)$ such that,*

- $\mathcal{A}_j = (S^j, \Sigma^j, T^j, s_0^j),\ j \in [0, m]$
- $S = S^0 \times \ldots \times S^m$
- $s_0 = (s_0^0, \ldots, s_0^m)$
- $\Sigma = \bigcup_{j=0}^{m} \Sigma^j$,
- $T \subseteq S \times \Sigma \times S$

The firing of an event at each step i should respect the following synchronization condition: given a transition $t = ((s_i^0, \ldots, s_i^m), e_i, (s_{i+1}^0, \ldots, s_{i+1}^m))$, for all $0 \leqslant j \leqslant m$ if $e_i \in \Sigma^j$ then $(s_i^j, e_i, s_{i+1}^j) \in T^j$ otherwise $s_{i+1}^j = s_i^j$. That is, we can fire an event only if it is a local event—an event that exists only in one component—or it is synchronized, and is enabled in all components that have it in their alphabets. s_i^j denotes the state reached at step i for component j. ■

To respect the rules in interleaving execution, we have to fire only one event at each step. In addition, each component of the system must execute at most one transition related to the enabled event at a time. However, since we are dealing with synchronized product of TSs, we need to take into account the fact that the event may be enabled in several components. Thus, we have to fire it in each of these components to guarantee the synchronization. All this means that, at each step, we fire only one event, but possibly, many transitions from different components.

Example 4.4.1 In Figure 4.3, we combine the system $\mathcal{A}_0$ with another one $\mathcal{A}_1$. We then obtain a synchronized system $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$. We can obtain the following interleaving execution for bound $k = 2$:

$$(s_0, s_7) \xrightarrow{a} (s_1, s_8) \xrightarrow{d} (s_4, s_8)$$

Application to BMC

Now, consider a synchronized TS $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_m)$, a property ϕ, and a bound k. We define the BMC formula for $\mathcal{A}$, in terms of interleaving execution, as follows.

$$Int[\![\mathcal{A}, \phi]\!]_k = Int[\![\mathcal{A}]\!]_k \wedge [\![\neg\phi]\!]_k \tag{4.9}$$

where

- $Int[\![\mathcal{A}]\!]_k$ defines the unrolling of $\mathcal{A}$ up to bound k using interleaving execution, i.e.

$$Int[\![\mathcal{A}]\!]_k = I_{Int}(s_0) \wedge \bigwedge_{i=0}^{k-1} T_{Int}(s_i, s_{i+1}) \tag{4.10}$$

- $I_{Int}(s_0)$ denotes the characteristic function of the initial state, such that

$$I_{Int}(s_0) = \mathcal{X}_{\{s_0^0, \ldots, s_0^m\}} \wedge \bigwedge_{s_p \in \bigcup_{j=0}^m S^j \setminus \{s_0^0, \ldots, s_0^m\}} \neg v_{p,0} \tag{4.11}$$

- $T_{Int}(s_i, s_{i+1})$ defines the transition between two steps i and $i+1$ such that

$$T_{Int}(s_i, s_{i+1}) = T(\{s_i^0, \ldots, s_i^m\}, \{s_{i+1}^0, \ldots, s_{i+1}^m\}) \tag{4.12}$$

Recall that the firing of an event at each step should respect the synchronization criterion given in Definition 4.4.1.

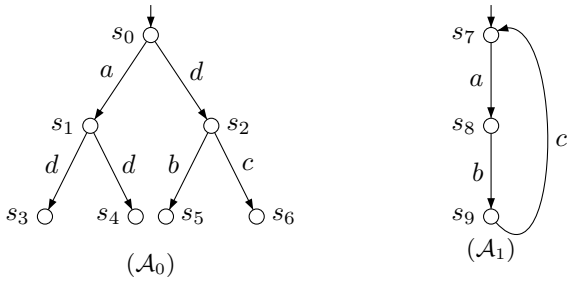


Figure 4.3: Synchronized product of TSs.

Example 4.4.2 For the system $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ depicted in Figure 4.3, we can generate the following interleaving execution formula from bound 0 to 2.

$$
\begin{aligned}
Int[\![\mathcal{A}]\!]_2 \quad = \quad & I_{Int}((s_0^0, s_0^1)) \\
& \wedge T_{Int}((s_0^0, s_0^1), (s_1^0, s_1^1)) \\
& \wedge T_{Int}((s_1^0, s_1^1), (s_2^0, s_2^1)) \\
\\
= \quad & \big(v_{0,0} \wedge v_{7,0} \wedge \neg v_{1,0} \wedge \neg v_{2,0} \wedge \neg v_{3,0} \wedge \neg v_{4,0} \wedge \neg v_{5,0} \wedge \neg v_{6,0} \wedge \neg v_{8,0} \\
& \quad \wedge \neg v_{9,0}\big) \\
& \wedge \Big(v_{0,0} \wedge v_{7,0} \wedge \neg v_{0,1} \wedge \neg v_{7,1} \wedge v_{1,1} \wedge v_{8,1} \wedge (v_{2,0} \leftrightarrow v_{2,1}) \wedge (v_{3,0} \leftrightarrow v_{3,1}) \\
& \quad \wedge (v_{4,0} \leftrightarrow v_{4,1}) \wedge (v_{5,0} \leftrightarrow v_{5,1}) \wedge (v_{6,0} \leftrightarrow v_{6,1}) \wedge (v_{9,0} \leftrightarrow v_{9,1})\Big) \\
& \wedge \Big(v_{1,1} \wedge \neg v_{1,2} \wedge v_{4,2} \wedge (v_{0,1} \leftrightarrow v_{0,2}) \wedge (v_{2,1} \leftrightarrow v_{2,2}) \wedge (v_{3,1} \leftrightarrow v_{3,2}) \\
& \quad \wedge (v_{5,1} \leftrightarrow v_{5,2}) \wedge (v_{6,1} \leftrightarrow v_{6,2}) \wedge (v_{7,1} \leftrightarrow v_{7,2}) \wedge (v_{8,1} \leftrightarrow v_{8,2}) \\
& \quad \wedge (v_{9,1} \leftrightarrow v_{9,2})\Big)
\end{aligned}
$$

where

$$
\begin{aligned}
& s_0^0 = s_0;\ s_0^1 = s_7;\ s_1^0 = s_1;\ s_1^1 = s_8; s_2^0 = s_4;\ s_2^1 = s_8; \\
& e_0 = a;\ e_1 = d
\end{aligned}
$$

Theorem 4.4.1

Given a synchronized system $\mathcal{A} = (A_0, \ldots, A_m)$, a property ϕ, and a bound k. If $Int[\![\mathcal{A}, \phi]\!]_k$ is satisfiable, then $\mathcal{A} \not\models \phi$.

Proof Assume $Int[\![\mathcal{A}, \phi]\!]_k$ is satisfiable, i.e. $I_{Int}(s_0) \wedge T_{Int}(s_0, s_1) \wedge \ldots \wedge T_{Int}(s_{k-1}, s_k) \wedge [\![\phi]\!]_k$ is satisfiable. We have $T_{Int}(s_i, s_{i+1}) = T(\{s_i^0, \ldots, s_i^m\}, \{s_{i+1}^0, \ldots, s_{i+1}^m\})$. Each $T_{Int}(s_i, s_{i+1})$ is then of the form $T(S_i, S_{i+1})$, where $S_i = \{s_i^0, \ldots, s_i^m\}$. Since $Int[\![\mathcal{A}, \phi]\!]_k$ is satisfiable, we can therefore say that $I(S_0) \wedge T(S_0, S_1) \wedge \ldots \wedge T(S_{k-1}, S_k) \wedge [\![\phi]\!]_k$ is also satisfiable. Using the results from the proof of Theorem 4.3.3 we conclude that a state violating the property ϕ is reached in k steps of less from the initial state. ■

Theorem 4.4.2

Given a synchronized transition system $\mathcal{A}$, a property ϕ, and a bound k. $\mathcal{A} \not\models \phi$ implies $\exists k \in \mathbb{N}$ such that $Int[\![\mathcal{A}, \phi]\!]_k$ is satisfiable.

PROOF Assume $\mathcal{A} \not\models \phi$ is satisfiable. This means there is a path from the initial state to a state violating the property ϕ. Assume k is the length of that path. In a synchronized system using the interleaving execution, a path of length k is a succession of transitions $T_{Int}(s_i, s_{i+1})$, $0 \leqslant i \leqslant k-1$. Since ϕ is violated at step k, then its negation is satisfied a that step. Therefore we have $I_{Int}(s_0) \wedge T_{Int}(s_0, s_1) \wedge \ldots \wedge T_{Int}(s_{k-1}, s_k) \wedge \neg\phi$ is satisfiable, i.e. $Int[\![\mathcal{A}, \phi]\!]_k$ is satisfiable. ■

4.4.2 Breadth-First Search Execution

Introduction

As before, we first start with the definition of synchronized product of TSs in terms of BFS execution.

Definition 4.4.2

Let $\mathcal{A}_0, \ldots, \mathcal{A}_m$ be TSs. Their synchronized product*, defined in terms of BFS, denoted $(\mathcal{A}_0, \ldots, \mathcal{A}_m)$ is the TS $\mathcal{A} = (S, \Sigma, T, s_0)$ such that,*

- $\mathcal{A}_j = (S^j, \Sigma^j, T^j, s_0^j),\ j \in [0, m]$,
- $S = 2^{S^0} \times \ldots \times 2^{S^m}$,
- $s_0 = (S_0^0, \ldots, S_0^m)$, *where* $S_0^j = \{s_0^j\},\ j \in [0, m]$,
- $\Sigma = \bigcup_{j=0}^{m} \Sigma^j$,
- $T \subseteq S \times 2^{\Sigma} \times S$.

The firing of an event at each step i should respect the following synchronization condition: given a transition $t = ((S_i^0, \ldots, S_i^m), E_i, (S_{i+1}^0, \ldots, S_{i+1}^m))$, for all $0 \leqslant j \leqslant m$ if $\exists e_i \in E_i$ such that $e_i \in \Sigma^j$ then $\exists (s_i^j, e_i, s_{i+1}^j) \in T^j$ such that $s_i^j \in S_i^j$ and $s_{i+1}^j \in S_{i+1}^j$ otherwise $S_{i+1}^j = S_i^j$. That is, we can fire an event only if it is a local event—an event that exists only in one component—or it is synchronized, and is enabled in all components that have it in their alphabets. S_i^j denotes the set of states reached at step i for component j■

The above synchronization condition is similar to the one in Definition 4.4.1, except that here we are firing a set of events rather than a single event. Using BFS allows us to fire not only synchronized events but also independent ones, all together in one step. However, because of the above condition, and unlike the BFS method for independent systems, not all enabled events could be fired. Once we know the set of fireable events, we fire them using a standard BFS procedure as explained earlier.

Example 4.4.3 From the synchronized system $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ shown in Figure 4.4 we can obtain the following BFS execution from bound 0 to 2 (each dashed line in the figure corresponds to one step):

$$(S_0^0, S_0^1) \xrightarrow{E_0} (S_1^0, S_1^1) \xrightarrow{E_1} (S_2^0, S_2^1)$$

where

$S_0^0 = \{s_0\};\ S_0^1 = \{s_7\};\ S_1^0 = \{s_1, s_2\};\ S_1^1 = \{s_8\}; S_2^0 = \{s_3, s_4, s_5\};\ S_2^1 = \{s_9\};$
$E_0 = \{a, d\};\ E_1 = \{d, b\}$

Application to BMC

Now, given a synchronized system $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_m)$, a property ϕ, and a bound k. We define the BMC formula for $\mathcal{A}$, in terms of BFS execution, as follows.

$$Bfs[\![\mathcal{A}, \phi]\!]_k = Bfs[\![\mathcal{A}]\!]_k \wedge [\![\neg\phi]\!]_k \tag{4.13}$$

where

- $Bfs[\![\mathcal{A}]\!]_k$ defines the unrolling of $\mathcal{A}$ up to bound k using BFS execution, i.e.

$$Bfs[\![\mathcal{A}]\!]_k = I_{Bfs}(s_0) \wedge \bigwedge_{i=0}^{k-1} T_{Bfs}(s_i, s_{i+1}) \tag{4.14}$$

- $I_{Bfs}(s_0)$ corresponds to the characteristic of the initial state, such that

$$I_{Bfs}(s_0) = \mathcal{X}_{\bigcup_{j=0}^{m} S_0^j} \wedge \bigwedge_{s_p \in \bigcup_{j=0}^{m} S^j \setminus \bigcup_{j=0}^{m} S_0^j} \neg v_{p,0} \tag{4.15}$$

- $T_{Bfs}(s_i, s_{i+1})$ defines the transitions between step i and $i+1$ such that

$$T_{Bfs}(s_i, s_{i+1}) = T(\bigcup_{j=0}^{m} S_i^j, \bigcup_{j=0}^{m} S_{i+1}^j) \tag{4.16}$$

Example 4.4.4 To illustrate this representation, let us continue the Example 4.4.3 and apply BMC for bound $k = 2$. The formula $Bfs[\![\mathcal{A}]\!]_2$ is given as:

$$\begin{aligned}
Bfs[\![\mathcal{A}]\!]_2 \quad = \quad & I_{Bfs}((S_0^0, S_0^1)) \\
& \wedge T_{Bfs}((S_0^0, S_0^1), (S_1^0, S_1^1)) \\
& \wedge T_{Bfs}((S_1^0, S_1^1), (S_2^0, S_2^1)) \\
\\
= \quad & \big(v_{0,0} \wedge v_{7,0} \wedge \neg v_{1,0} \wedge \neg v_{2,0} \wedge \neg v_{3,0} \wedge \neg v_{4,0} \wedge \neg v_{5,0} \wedge \neg v_{6,0} \wedge \neg v_{8,0} \wedge \\
& \neg v_{9,0}\big) \\
& \wedge \Big(v_{0,0} \wedge v_{7,0} \wedge \neg v_{0,1} \wedge \neg v_{7,1} \wedge v_{1,1} \wedge v_{2,1} \wedge v_{8,1} \wedge (v_{3,0} \leftrightarrow v_{3,1}) \\
& \wedge (v_{4,0} \leftrightarrow v_{4,1}) \wedge (v_{5,0} \leftrightarrow v_{5,1}) \wedge (v_{6,0} \leftrightarrow v_{6,1}) \wedge (v_{9,0} \leftrightarrow v_{9,1})\Big) \\
& \wedge \Big(v_{1,1} \wedge v_{2,1} \wedge v_{8,1} \wedge \neg v_{1,2} \wedge \neg v_{2,2} \wedge \neg v_{8,2} \wedge v_{3,2} \wedge v_{4,2} \wedge v_{5,2} \\
& \wedge v_{9,2} \wedge (v_{0,1} \leftrightarrow v_{0,2}) \wedge (v_{6,1} \leftrightarrow v_{6,2}) \wedge (v_{7,1} \leftrightarrow v_{7,2})\Big)
\end{aligned}$$

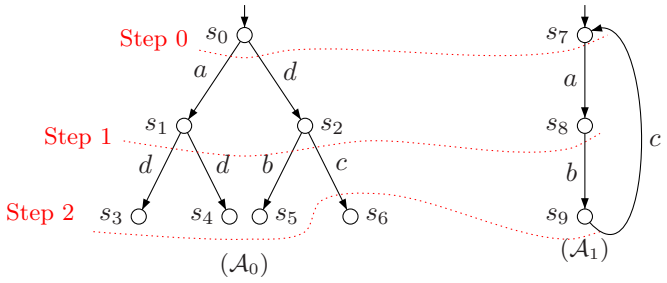


Figure 4.4: Typical BFS execution for synchronized systems.

Theorem 4.4.3

Given a synchronized system $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_m)$, a property ϕ, and a bound k. If $Bfs[\![\mathcal{A}, \phi]\!]_k$ is satisfiable then $\mathcal{A} \not\models \phi$.

Proof Assume $Bfs[\![\mathcal{A}, \phi]\!]_k$ is satisfiable i.e. $I_{Bfs}(s_0) \wedge T_{Bfs}(s_0, s_1) \wedge \ldots \wedge T_{Bfs}(s_{k-1}, s_k) \wedge [\![\phi]\!]_k$ is satisfiable. From equation (4.16) we can say that each $T_{Bfs}(s_i, s_{i+1})$ is of the form $T(S_i, S_{i+1})$ where $S_i = \bigcup_{j=0}^{m} S_i^j$. Using the result of the proof of Theorem 4.3.3 and 4.4.1, we can conclude that a state violating the property ϕ is reachable in k step or less from the initial state. ■

Theorem 4.4.4

Given a synchronized transition system $\mathcal{A}$, a property ϕ, and a bound k. $\mathcal{A} \not\models \phi$ implies $\exists k \in \mathbb{N}$ such that $Bfs[\![\mathcal{A}, \phi]\!]_k$ is satisfiable.

Proof To prove this theorem, we only need to follow the reasoning in the proof of Theorem 4.3.4 by replacing S_i into $\bigcup_{j=0}^{m} S_i^j$. ■

Remark 4.4.1 In our encoding, each state is encoded into two variables—current and next-state variables, thus we need at most two Boolean variables to represent each state. Still, the resulted formula is compact with a size $O(\Sigma^j(|S^j|).k)$ where S^j and Σ^j represent the set of states and the alphabet for component j respectively, and k is the bound.

Remark 4.4.2 The trace is built on the fly. That is the algorithm keeps track on the states it has visited. Then, once the formula becomes satisfiable the counterexample trace can be output right away. Therefore it does not add more complexity to the whole operation.

4.5 Expressing Reachability Properties

So far, we have been interested only in the unrolling part of the BMC formula. In this section, we will show how to define the property ϕ. The proposed encoding can be used

for checking *reachability properties*, which express that some particular situation can be reached. Checking such properties is equivalent to determining whether there exists an execution path leading to a state satisfying the property. These properties can be *simple,* for instance "we can enter a critical section," or *conditional,* for instance "we can enter a critical section without passing through $n = 0$," or also in negative form such as "we cannot have $n < 0$."

In particular, we use our method to check if there is a *deadlock* in a system, i.e. if there exists a reachable state with no firing events. Formally, a deadlock run in a transition system is defined as follows.

Definition 4.5.1

Let $\sigma = s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} \ldots \xrightarrow{e_{k-1}} s_k$ be a run of a TS $\mathcal{A}$. σ is a deadlock *run iff the state s_k does not have any outgoing transitions.* ■

For synchronized systems, two cases should be considered in order to express deadlock conditions:

- the deadlock state is reached after firing a local event,
- the deadlock state is reached after firing a synchronized event and each synchronized counterpart of that event also reaches a deadlock state.

Let $\mathcal{R}$ be the set of reachable states for all components at a given bound k. $\mathcal{R}$ can be divided into two subset $\mathcal{R}_l$ and $\mathcal{R}_s$ which correspond to the states reached by local events and the synchronized ones respectively. $\mathcal{R}_s$ can still be divided into several subsets $L_1, \ldots, L_n$ such that each L_i contains states reached by related events—i.e. events with their synchronized counterparts.

The deadlock condition for a system can be expressed by the negation of the following "live" condition, which expresses that at each step we can always fire some events from each reachable state:

$$[\![Live]\!]_k = \Big(\bigwedge_{s_i \in \mathcal{R}_l} v_{i,k} \wedge \bigwedge_{L \subset \mathcal{R}_s} \bigvee_{s_j \in L} v_{j,k} \Big) \tag{4.17}$$

The negation of the property above expresses that we can obtain a reachable state with no firing events, which means a deadlock appears in that state. The deadlock condition at a bound k is thus given as follow:

$$[\![\neg Live]\!]_k = \neg\Big(\bigwedge_{s_i \in \mathcal{R}_l} v_{i,k} \wedge \bigwedge_{L \subset \mathcal{R}_s} \bigvee_{s_j \in L} v_{j,k} \Big) \tag{4.18}$$

Example 4.5.1 Let us take again our running example depicted in Figure 4.3. The deadlock condition for this system at bound 2, using the BFS execution (see Figure 4.4), is given as:

$$[\![\neg Live]\!]_2 = \neg\big(v_{3,2} \wedge v_{4,2} \wedge (v_{5,2} \vee v_{9,2})\big)$$

such that

$$\mathcal{R}_l = \{s_3, s_4\};\ L_1 = \{s_5, s_9\};\ \mathcal{R}_s = L_1.$$

To verify whether a deadlock appears in the system $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ using the BFS approach, we only need to combine the equation above with the one obtained from Example 4.4.4. We can use the same principle with the interleaving execution.

Theorem 4.5.1
Given a synchronized system $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_m)$ and a bound k, such that the formulas $Int[\![\mathcal{A}]\!]_k$, and $Bfs[\![\mathcal{A}]\!]_k$ are satisfiable. Let $[\![\neg Live]\!]_k$ be the deadlock condition for $\mathcal{A}$ at bound k. If $[\![\neg Live]\!]_k$ is SAT, then $\mathcal{A} \not\models live$.

PROOF Assume that $Int[\![\mathcal{A}]\!]_k$, and $Bfs[\![\mathcal{A}]\!]_k$ are satisfiable. Then all we need to do is replace ϕ in the proof of Theorem 4.4.1, and 4.4.3 into *live*. ■

4.6 Reducing the Bound Using Chaining

4.6.1 Introduction

We have seen that BFS methods are more advantageous compared with the interleaving ones. We can still improve the performance of BFS methods by combining them with the mechanism known as *chaining traversal* whose main objective is to reduce the number of execution steps. Chaining is much more common in BDD-based approaches (see e.g. [SP02]), but in this work, we apply it to BMC.

Briefly put, chaining is performed as follows. We first define an event order. Then, at each step, we fire the events respecting the chosen order. After firing one event, we add to the $From$ set the newly reached state. Then we fire the next event in the order provided it is enabled at the states in the new $From$ set. The procedure thus produces a sort of chain in the execution, hence its name. Only when all the fireable events at one step have been fired will we move to the next step.

Example 4.6.1 An example of a chaining execution is shown in Figure 4.5. Assume we chose the following event order $[a, c, b, d]$. At step 0, we have $From = \{s_0\}$ because s_0 is the initial state. Events a and d are enabled at s_0, thus we can fire them both at this step. But respecting the event order, we have to fire a first. We reach the state s_1 after firing a, and update the $From$ set accordingly. We then obtain a new set $From = \{s_0, s_1\}$. Now, we can fire d which is enabled at both s_0 and s_1. Firing d from these states leads us to s_2, s_3, and s_4. There is no more event to be fired, so we arrive at step 1 with the following final sets $From = \{s_0, s_1\}$ and $To = \{s_2, s_3, s_4\}$.

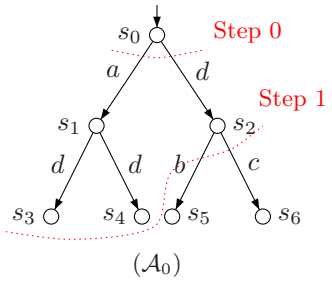


Figure 4.5: Chaining execution with event order $[a, c, b, d]$.

4.6.2 Application to BMC

The BMC formula related to chaining is slightly different from the one given in equation (4.13) because we need to take into account the fact that the $From$ set is updated at each iteration. To perform this update we define two sets To' and $S_i'^j$ as follows.

$$To'(S_i^j) = \{s_i' \in To(S_i^j) \mid \forall s_i \in S_i^j,\ \mathcal{E}(s_i) \cap \mathcal{E}(s_i') \neq \varnothing\} \tag{4.19}$$

$$S_i'^j = S_i^j \cup To'(S_i^j) \tag{4.20}$$

$S_i'^j$ is the new $From$ set obtained after adding into S_i^j the states recently generated. We have $S_i^j \subseteq S_i'^j$ and $To'(S_i^j) \subseteq To(S_i^j)$ for all $0 \leqslant j \leqslant m$.

Now, given a synchronized system $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_m)$, a property ϕ, and a bound k. The BMC formula for $\mathcal{A}$ using chaining is defined as:

$$Chain[\![\mathcal{A}, \phi]\!]_k = Chain[\![\mathcal{A}]\!]_k \wedge [\![\neg\phi]\!]_k \tag{4.21}$$

where

- $Chain[\![\mathcal{A}]\!]_k$ defines the unrolling of $\mathcal{A}$ up to bound k using chaining execution, i.e.

$$Chain[\![\mathcal{A}]\!]_k = I_{Chain}(s_0') \wedge \bigwedge_{i=0}^{k-1} T_{Chain}(s_i', s_{i+1}) \tag{4.22}$$

- $s_i' = (S_i'^0, \ldots, S_i'^m)$,

- $I_{Chain}(s_0')$ defines the characteristic function of the initial state, such that

$$I_{Chain}(s_0') = \mathcal{X}_{\bigcup_{j=0}^m S_0'^j} \wedge \bigwedge_{s_p \in \bigcup_{j=0}^m S^j \setminus \bigcup_{j=0}^m S_0'^j} \neg v_{p,0} \tag{4.23}$$

- $T_{Chain}(s_i', s_{i+1})$ defines the transition between step i and $i+1$ such that

$$T_{Chain}(s_i', s_{i+1}) = T(\bigcup_{j=0}^m S_i'^j, \bigcup_{j=0}^m S_{i+1}^j) \tag{4.24}$$

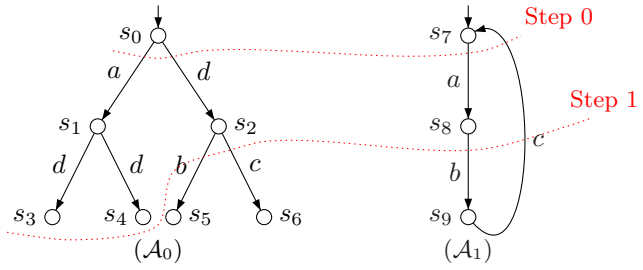


Figure 4.6: Chaining execution for synchronized systems.

Obviously, when applying chaining for synchronized products of TSs, we also have to respect the synchronization criteria described in Definition 4.4.2. This means, chaining can be performed if the chosen event is either a local event, or it is enabled in all its components at the current step.

Example 4.6.2 If we apply the chaining operation described in Figure 4.5 to the synchronized systems $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ in Figure 4.4, we obtain the following BMC formula $Chain[\![\mathcal{A}]\!]_1$ for bound $k = 1$ (the execution is depicted in Figure 4.6):

$$
\begin{aligned}
Chain[\![\mathcal{A}]\!]_1 \quad &= \quad I_{Chain}((S_0'^0, S_0'^1)) \\
&\quad \wedge T_{Chain}((S_0'^0, S_1'^1), (S_1^0, S_1^1)) \\
&= \quad \big(v_{0,0} \wedge v_{1,0} \wedge v_{7,0} \wedge \neg v_{2,0} \wedge \neg v_{3,0} \wedge \neg v_{4,0} \wedge \neg v_{5,0} \wedge \neg v_{6,0} \wedge \neg v_{8,0} \wedge \neg v_{9,0}\big) \\
&\quad \wedge \Big(v_{0,0} \wedge v_{1,0} \wedge v_{7,0} \wedge \neg v_{0,1} \wedge \neg v_{1,1} \wedge \neg v_{7,1} \wedge v_{2,1} \wedge v_{3,1} \wedge v_{4,1} \wedge v_{8,1} \\
&\qquad \wedge (v_{5,0} \leftrightarrow v_{5,1}) \wedge (v_{6,0} \leftrightarrow v_{6,1}) \wedge (v_{9,0} \leftrightarrow v_{9,1})\Big)
\end{aligned}
$$

where,

$$
\begin{aligned}
&S_0^0 = \{s_0\};\ S_0'^0 = \{s_0, s_1\};\ S_0^1 = \{s_7\};\ S_0'^1 = \{s_7\};\ S_1^0 = \{s_2, s_3, s_4\};\ S_1^1 = \{s_8\}; \\
&E_0 = \{a, d\}
\end{aligned}
$$

We then have the following chaining execution:

$$(S_0'^0, S_0'^1) \xrightarrow{E_0} (S_1^0, S_1^1)$$

Comparing the formula above with the one in Example 4.4.4 for $k = 2$, we can see that the equations for the transitions which lead to the states s_3 and s_4 can already be expressed at bound 1. Therefore, a counterexample that exists in those states can be found at bound 1 instead of bound 2, if we choose the right order of events. In addition, the formula we obtain in here is clearly smaller than the one in Example 4.4.4.

Theorem 4.6.1

Given a synchronized system $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_m)$, a property ϕ, and a bound k. If $Chain[\![\mathcal{A}, \phi]\!]$ is satisfiable then $\mathcal{A} \not\models \phi$.

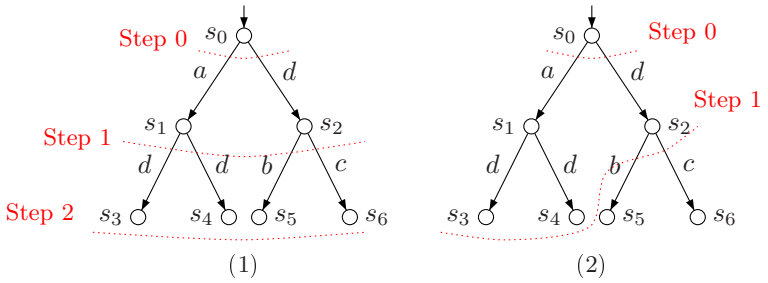


Figure 4.7: BFS with chaining using two different event orders.

PROOF The proof is straightforward from the one for Theorem 4.4.3. We need only to change $Bfs[\![\mathcal{A}, \phi]\!]_k$ with $Chain[\![\mathcal{A}, \phi]\!]_k$. ■

Theorem 4.6.2

Given a synchronized transition system $\mathcal{A}$, a property ϕ, and a bound k. $\mathcal{A} \not\models \phi$ implies $\exists k \in \mathbb{N}$ such that $Chain[\![\mathcal{A}, \phi]\!]_k$ is satisfiable.

PROOF The proof follows similar arguments to the proof of Theorem 4.4.4. We need only to change $Bfs[\![\mathcal{A}, \phi]\!]_k$ with $Chain[\![\mathcal{A}, \phi]\!]_k$. ■

Some event orders may fare badly compared to others, even though they are applied to the same system. To illustrate this situation, we show in Figure 4.7 two different chaining executions applied to the same system. First, we fire the events following the order $[d, c, b, a]$. The execution progresses much the same as in a typical BFS, and it takes two iterations to reach the states s_3 and s_4, for instance, from the initial state s_0 (see Figure 4.7 (1)). On the other hand, when we fire the events with the order $[a, d, b, c]$, only one iteration is enough to reach s_3 and s_4 (see Figure 4.7 (2)). Choosing the right order of event is then crucial in order to make BFS with chaining effective.

In BMC, fewer iterations means smaller formula in most cases. In addition, the speed of a BMC operation depends to a large extent on the speed of the SAT-solver, which in turn depends on the length of the formula. Therefore, since smaller formula takes shorter time to solve, choosing the right order of events can speed up the whole BMC operation. However, this situation may not be true if the chaining operation adds many states in one iteration. In our case, we limit the number of states to be included in one iteration by applying chaining only to those events that are enabled at the current step.

4.6.3 Ordering the Events

One way of ordering the events is by taking into account the *causality relations* between them. In this section, we will give a heuristic that computes the causality between two events and show how to use this information for ordering them. Taking into account the

causality gives us a higher probability of obtaining the right order. Still, as we will see later in the experimental results in Section 8.1, there might be some cases in which this situation is not always true. This heuristic is a simplified version of the one in [SP02].

Let T_{e_i} and T_{e_j} be the transitions associated to the events e_i and e_j respectively. We define $To(e_i)$ as

$$To(e_i) = \{s' \in S \ / \ \exists s \in S, s \xrightarrow{e_i} s' \in T\} \tag{4.25}$$

it gives the set of states reached after firing the event e_i.

The heuristic $causality(e_i \rightarrow e_j)$ is defined as

$$causality(e_i \rightarrow e_j) = |To(e_i) \cap FR(e_j)| \tag{4.26}$$

it indicates the number of states where e_j becomes fireable after firing e_i.

Intuitively, big values of $causality(e_i \rightarrow e_j)$ show that the activation of transition T_{e_i} will tend to produce states in which the application of transition T_{e_j} is possible. Therefore, in a chaining operation, if e_i and e_j are all enabled at the current step, firing e_i before e_j will generate more new states than firing e_j before e_i.

To order the firing of the events using the above heuristic, we proceed as follows. First, before starting the BMC operation, we compute the causality between each pair of events in the system and store the result in a *causality matrix*. Next, at each step, we check the matrix to determine the causality between all executable events for this step. We then build a list of events in which they are arranged in the following way: given two pairs of events e_i and e_j, if $causality(e_i \rightarrow e_j) > causality(e_j \rightarrow e_i)$, then e_i will go before e_j in the list. Obviously, if both causalities are equal, there is no restrictions between the two events. This procedure is carried out for the rest of the executable events, it ensures that the event which stays at the top of the list has higher probability to generate states where the rest of the executable events are fireable. When the list is built, we fire the event at the top and remove it from the list, and we continue that way until the list is empty.

Example 4.6.3 To illustrate this event ordering method, consider the system $\mathcal{A}_0$. The causality matrix for this TS is given as follows.

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

At step 0, the events a and d are enabled. Looking at the above matrix, we can see that $causality(a \rightarrow d) = 1$ whereas $causality(d \rightarrow a) = 0$. Therefore, we obtain the following list $[a, d]$ that is, we will fire a before d. In fact, we used this event order for the execution depicted in Figure 4.7 (2).

4.6.4 Chaining Algorithm

Algorithm 4.1 shows the BMC procedure related to chaining. `computeFireableEvents()` creates a list of fireable events an orders them using the heuristic above. The function `selectEvent()` chooses an event from the list. `chainFire()` fires the chosen event, taking into account all conditions described in this section, and building the BMC formula F in the process. The function `solve()` calls the SAT-solver. It returns *true* if F is satisfiable, *false* otherwise. The algorithm stops whether the iteration reaches the maximum BMC length $maxL$ or F is satisfiable.

Algorithm 4.1: Bmc

input : $\mathcal{A} = (\mathcal{A}_0 \dots \mathcal{A}_n)$ synchronized TSs
$maxL$ maximum BMC length
output: *true* if a counterexample has been found
$false$ otherwise

```
1  bound ⟵ 1 ;
2  isSat ⟵ false ;
3  while (bound ⩽ maxL) and (isSat = false) do
4      for A_j ∈ A do
5          list ⟵ computeFireableEvents(A_j) ;
6          while list ≠ ∅ do
7              e ⟵ selectEvent(list) ;
8              F ⟵ chainFire(e) ;
9          end
10     end
11     isSat ⟵ solve(F) ;
12     bound + + ;
13 end
14 return isSat
```

Remark 4.6.1 Note that, to check a deadlock property with chaining we will also use the formulas in equations (4.17) and (4.18).

Chapter 5

Leap-Based Approach

In this Chapter, we will present a method for increasing the speed of a BMC operation. Normally, in a typical BMC algorithm (see Algorithm 1.2), we call a SAT solver at each execution step. However, due to the possible huge number of steps needed for exploring concurrent systems, we may need to perform many calls to a SAT solver during the whole BMC operation. In addition, since solving SAT instances occupies a great deal of time during the BMC operation, calling a solver too often will probably slow down the execution, especially when dealing with sizeable formulas at a time.

One obvious way to cope with this problem is to use a fast SAT solver. But this solution is helpful only for Boolean formulas with small sizes. For most of today's solvers, the problem still remains when they are called repeatedly to solve large formulas. The solution we propose consists in reducing the number of solver calls during a BMC operation, especially when the formulas become large. In this way, the speed will increase because the solver has to solve only few SAT instances.

Basically, our technique is performed as follows. When we move from one bound to another, we do not carry out a simple one by one incrementation as in a standard BMC algorithm. Instead, we use *leaps*. That is, we combine several unrolling steps together in a single iteration before calling a SAT solver. One single bound in our method is, therefore, equivalent to several consecutive steps in a standard BMC algorithm.

The leap-based technique also applies to reachability properties. In order for it to work properly, it is important to ensure that we can catch all failure states occurring between BMC bounds even though we do not run a solver at the exact points where these failures occur. Otherwise, we would miss some counterexamples along the execution. The unrolling methods we propose in this Chapter can handle this situation. They generate at each bound a BMC formula that contains all necessary information needed for identifying failure states from earlier steps.

Figure 5.1 depicts the different steps for a leap execution. We have two types of iterators, namely i and j, related to the BMC bounds and the steps needed between bounds respectively. All states s_j correspond to the ones reached during the unrolling

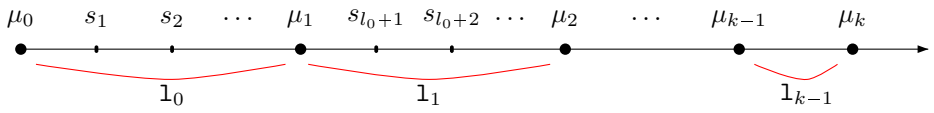


Figure 5.1: Unrolling using leaps.

iterations between two bounds i and $i+1$. In a standard BMC algorithm, we would call a solver at every step j. By contrast, for the leap method, we call a solver only when we reach a bound i. Each leap value l_i determines the number of unrolling iterations required between i and $i+1$. For instance, assume $l_0 = 3$, then to move from 0 to 1, we allow the unrolling to iterate three times before calling the SAT solver. We do so for all $i \in [0, k-1]$. Each state μ corresponds to the one reached at each bound i. Obviously, we have $\mu_0 = s_0$, and $\mu_i = s_n \; \forall \; 1 \leqslant i \leqslant k-1$ with $n = \sum_{j=1}^{i-1} l_j$ for $1 \leqslant i \leqslant k-1$. We will use this notation when we formalize the unrolling executions related to the leap approach.

The rest of the Chapter is organized as follows. In Section 5.1 we will review some of the existing work related to this topic. In Section 5.2 and Section 5.3 we will introduce the leap-based unrolling methods related to deadlock and other reachability properties respectively. In Section 5.4 we describe two jumping strategies that apply with the leap approach. In Section 5.5 we show how to find the shortest counterexample. An finally, in Section 5.6 we discuss the time performance for the leap approach.

5.1 Related Work

There exist many BMC techniques in the literature designed especially for reducing the number of solver calls. In a mechanism known as *lazy satisfiability check*, Ábráham *et al.* exploit the conflict clauses generated by the solver [ÁBKS05]. They optimize the SAT formula in order to solve it efficiently with solvers designed for linear hybrid systems. At a given BMC bound, if the formula is unsatisfiable, the algorithm analyzes and memorizes the conflicts returned by the solver so that, in a future bound, the solver will not be called again if the same conflicts ever appear. This technique is further explored in [ÁHBS06] where, combined with parametric data structures, it not only minimizes the number of solver calls, but also reduces the amount of memory needed by the solver.

The *proof-based* technique introduced in [MA03] also relies on the information generated by a SAT solver. Designed for synchronous systems, it combines BMC with standard BDD-based model checking, which allows it not only to look for a counterexample, but also to determine that the system is actually correct if no counterexample has been found. In this technique, the algorithm starts with a length k. If the formula is unsatisfiable, the proof of unsatisfiability provided by the solver will be used for creating a new abstraction of the system. Then, a standard BDD-based model checker is ran on the abstracted model to determine whether it is correct or not. If it is correct, the algorithm will terminate. Otherwise, the BDD-based model checker will produce a counterexample whose length

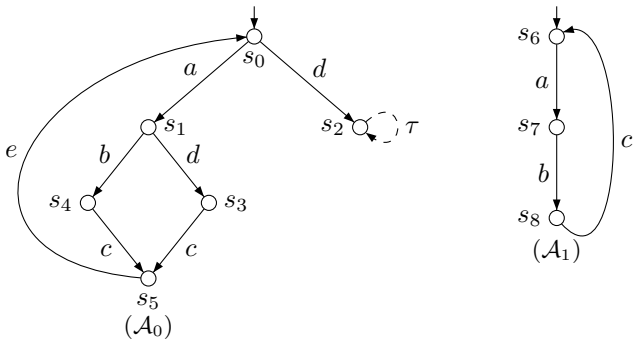


Figure 5.2: A synchronized system with idle steps.

k', which is always greater than k, is used later on as the next BMC step. Since k' is not necessarily equals $k+1$, this technique also reduces the number of calls to a SAT solver.

In a method called *path compression* the idea is to compress a chain of transitions into a single jump transition [KLY02, KM99, Yor94, dSU96]. These transitions are irrelevant to the property to be checked so they can be safely skipped during the traversal. This method constitutes an improvement to partial order reduction techniques, and suitable to properties written with nexttime free temporal logics (i.e. temporal logics without the nexttime operator such as LTL$-_X$ and CTL$-_X$). The method known as *local transition merging* presented in [Jus05], adapts similar ideas to BMC.

In a sense, the underlying idea in our approach is much the same as in the path compression method, in which a long jump transition is comparable to a leap in our method. However, both approaches differ in the way of how to safely skip transitions so that no counterexample would be missed. With the path compression, transitions can be skipped based on their relevance to the property to be checked. We use a quite different approach that relies on the model's structure itself. In addition, unlike the ones in [ÁBKS05, MA03], we do not analyze the conflict clause provided by a SAT solver.

5.2 Unrolling Method for Deadlock Property

5.2.1 Introduction

Our principle is based on the following observation. Normally, concurrent systems change from one state to another when at least one event occurs. However, any concurrent system may remain at a given state for an unbounded period of time. These *idle steps* are often represented with τ-labeled transitions in a TS graph. This behavior is usually excluded from models but it will work to our advantage. By including such behavior in a BMC model, once a state is reached at a step j it will repeat itself at steps $j+1$, $j+2$, etc.

Because a deadlock state has no outgoing transitions, it will behave as an idle state if it ever occurs in-between BMC bounds, and will remain so until the next call for a SAT solver. Therefore, once a deadlock state is reached, our unrolling algorithm will consider

it as an idle state, and will repeat it over the execution. As a results, the state will keep its characteristics, and even though we do not call the SAT solver at the exact step where the deadlock occurs, we would not miss it at the next solver call.

For instance, assume a deadlock appears at the state s_2 in Figure 5.2. If idle steps were not allowed, then s_2 would only be reachable by traces of length 1: $s_0 \xrightarrow{b} s_2$, or length 5: $s_0 \xrightarrow{a} s_1 \xrightarrow{b} s_4 \xrightarrow{d} s_5 \xrightarrow{d} s_0 \xrightarrow{b} s_2$, or length 9 etc. But if idle steps are allowed, we can reach s_2 with traces of any length above 1, e.g. $s_0 \xrightarrow{b} s_2 \xrightarrow{\tau} s_2 \xrightarrow{\tau} s_2 \ldots$. We express formally this key observation in the next theorem.

Theorem 5.2.1

Let $[\![\mathcal{A}]\!]_k$ be the SAT formula related to the unrolling of the model $\mathcal{A}$ up to the bound k. If the unrolling allows idle steps, then there is a satisfying assignment of $[\![\mathcal{A}]\!]_k$ iff there is a reachable state in $\mathcal{A}$, reachable through sequence of steps of length $i \leqslant k$ (once idle steps have been removed).

Proof The proof of the theorem is straightforward as it is clear that given any valid unrolling, if a step is allowed to be idle, then all possible executions of length k include all possible executions of some smaller length. For instance, all valid executions of length $k-1$ can be obtained from the executions of length k by changing the last step by the idle step. ∎

5.2.2 BMC Equations

Let us now give the BMC equations related to this deadlock checking method. As usual, we consider interleaving, BFS, and Chaining executions. The big difference between these equations and the ones in Section 4.4 can be seen in the transitions used. Here, we need to take into account the fact that we repeat each deadlock state over and over, until the next solver call. Note that, there is no change for the deadlock conditions. We will still use the equations (4.17) and (4.18).

Given a synchronized model $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_m)$, a property ϕ, a bound k, and natural numbers $l_i > 1$ with $i \in [0, k-1]$. The BMC formulas for $\mathcal{A}$ related to deadlock checking, defined in terms of the leaps l_i, is given as follows.

Interleaving Execution

$$lInt[\![\mathcal{A}, \phi]\!]_k = lInt[\![\mathcal{A}]\!]_k \wedge [\![\neg\phi]\!]_k \tag{5.1}$$

where

$$lInt[\![\mathcal{A}]\!]_k = I_{Int}(s_0) \wedge \bigwedge_{i=0}^{k-1} TI_{l_i}(\mu_i, \mu_{i+1}) \tag{5.2}$$

$$TI_{l_0}(\mu_0, \mu_1) = \bigwedge_{j=0}^{l_0-1} T_{lInt}(s_j, s_{j+1}) \tag{5.3}$$

$$TI_{l_i}(\mu_i, \mu_{i+1}) = \bigwedge_{j=n}^{n+l_i-1} T_{lInt}(s_j, s_{j+1}) \tag{5.4}$$

$$T_{lInt}(s_i, s_{i+1}) = T(\{s_i^0, \ldots, s_i^m\}, \{s_{i+1}^0, \ldots, s_{i+1}^m\}) \wedge \bigwedge_{s_d \in \{s_{i+1}^0, \ldots, s_{i+1}^m\};\, \mathcal{E}(s_d) = \varnothing} \neg v_{d,i} \tag{5.5}$$

Recall that $n = \sum_{j=1}^{i-1} l_j$ for $1 \leqslant i \leqslant k-1$.

BFS Execution

$$lBfs[\![\mathcal{A}, \phi]\!]_k = lBfs[\![\mathcal{A}]\!]_k \wedge [\![\neg\phi]\!]_k \tag{5.6}$$

where

$$lBfs[\![\mathcal{A}]\!]_k = I_{Bfs}(s_0) \wedge \bigwedge_{i=0}^{k-1} TB_{l_i}(\mu_i, \mu_{i+1}) \tag{5.7}$$

$$TB_{l_0}(\mu_0, \mu_1) = \bigwedge_{j=0}^{l_0-1} T_{lBfs}(s_j, s_{j+1}) \tag{5.8}$$

$$TB_{l_i}(\mu_i, \mu_{i+1}) = \bigwedge_{j=n}^{n+l_i-1} T_{lBfs}(s_j, s_{j+1}) \tag{5.9}$$

$$T_{lBfs}(s_i, s_{i+1}) = T(\bigcup_{j=0}^{m} S_i^j, \bigcup_{j=0}^{m} S_{i+1}^j) \wedge \bigwedge_{s_d \in \bigcup_{j=0}^{m} S_i^j;\, \mathcal{E}(s_d) = \varnothing} \neg v_{d,i} \tag{5.10}$$

Chaining Execution

$$lChain[\![\mathcal{A}, \phi]\!]_k = lChain[\![\mathcal{A}]\!]_k \wedge [\![\neg\phi]\!]_k \tag{5.11}$$

where

$$lChain[\![\mathcal{A}]\!]_k = I_{Chain}(s'_0) \wedge \bigwedge_{i=0}^{k-1} TC_{l_i}(\mu_i, \mu_{i+1}) \tag{5.12}$$

$$TC_{l_0}(\mu_0, \mu_1) = \bigwedge_{j=0}^{l_0-1} T_{lChain}(s'_j, s_{j+1}) \tag{5.13}$$

$$TC_{l_i}(\mu_i, \mu_{i+1}) = \bigwedge_{j=n}^{n+l_i-1} T_{lChain}(s'_j, s_{j+1}) \tag{5.14}$$

$$T_{lChain}(s'_i, s_{i+1}) = T(\bigcup_{j=0}^{m} S_i'^j, \bigcup_{j=0}^{m} S_{i+1}^j) \wedge \bigwedge_{s_d \in \bigcup_{j=0}^{m} S_i'^j;\, \mathcal{E}(s_d) = \varnothing} \neg v_{d,i} \tag{5.15}$$

Figure 5.3 shows a sketch of an unrolling execution between two bounds i and $i+1$ using the leap method described above. ϕ corresponds to the deadlock property, while

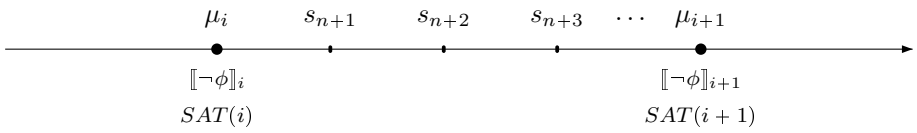


Figure 5.3: Leap methods for deadlock property.

$SAT(i)$ indicates that we call the solver at bound i. We put the constraints related to the negation of ϕ only at each bound i. Yet, by allowing idle steps, we could catch any counterexample occurring at s_{n+1}, s_{n+2}, s_{n+3}, etc.

Example 5.2.1 Given the system depicted in Figure 5.2. As we can see, the state s_2 has no outgoing transitions. Therefore, the algorithm assumes there is an idle step in this state, which is then repeated over and over until the next solver call. Using the BFS unrolling formulas from equations (5.6) and (5.9), we have the following formula for a leap execution between $i = 0$ and $i = 1$, with $l_0 = 3$:

$$
\begin{aligned}
lBfs[\![\mathcal{A}]\!]_1 \quad = \quad & \big(v_{0,0} \wedge v_{6,0} \wedge \neg v_{1,0} \wedge \neg v_{2,0} \wedge \neg v_{3,0} \wedge \neg v_{4,0} \wedge \neg v_{5,0} \neg v_{7,0} \neg v_{8,0}\big) \\
& \wedge \Big(v_{0,0} \wedge v_{6,0} \wedge \neg v_{6,1} \wedge \neg v_{0,1} \wedge \neg v_{2,1} \wedge v_{1,1} \wedge v_{7,1} \wedge (v_{3,0} \leftrightarrow v_{3,1}) \\
& \quad \wedge (v_{4,0} \leftrightarrow v_{4,1}) \wedge (v_{8,0} \leftrightarrow v_{8,1}) \wedge (v_{5,0} \leftrightarrow v_{5,1})\Big) \\
& \wedge \Big(v_{1,1} \wedge v_{7,1} \wedge \neg v_{2,1} \wedge \neg v_{1,2} \wedge \neg v_{2,2} \wedge \neg v_{7,2} \wedge v_{3,2} \wedge v_{4,2} \wedge v_{8,2} \\
& \quad \wedge (v_{0,1} \leftrightarrow v_{0,2}) \wedge (v_{6,1} \leftrightarrow v_{6,2}) \wedge (v_{5,1} \leftrightarrow v_{5,2})\Big) \\
& \wedge \Big(v_{4,2} \wedge v_{3,2} \wedge v_{8,2} \wedge \neg v_{2,2} \wedge \neg v_{4,3} \wedge \neg v_{3,3} \wedge \neg v_{2,3} \wedge v_{5,3} \wedge v_{6,3} \\
& \quad \wedge (v_{1,2} \leftrightarrow v_{1,3}) \wedge (v_{7,2} \leftrightarrow v_{7,3}) \wedge (v_{0,2} \leftrightarrow v_{0,3})\Big) \\
& \wedge \neg\big(v_{1,3} \wedge v_{2,3}\big)
\end{aligned}
$$

5.3 Unrolling Method for Other Reachability Properties

5.3.1 Introduction

In a BMC operation, the normal way is to add the constraints related to the negation of a property only when we call a SAT solver. This means that, in a leap-based unrolling, we should put these constraints only at each bound i. As we have seen before, this approach works perfectly for deadlock detection combined with idle step. But doing so with other reachability properties would result in missed counterexamples because the information related to these properties could not be maintained even though we repeat idle states over and over, so we would never know if any of the states in between BMC bounds violates the property to be checked. Therefore, to deal with other types of reachability properties, we need a different approach.

Our idea consists in putting the constraints related to the negation of the property not only at each bound i but also at each step j. In this way, even though we call the solver at a later step we would not miss the counterexample. For sure, this method will increase the size of the resulted formula, which would normally slows down the execution. However, we still can avoid this situation experimentally, by using the jumping methods introduced in Section 5.4. Figure 5.4 depicts an unrolling operation performed using this approach.

5.3.2 BMC Equations

Now, given a synchronized model $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_m)$, a property ϕ, a bound k, and natural numbers $l_i > 1$ with $i \in [0, k-1]$. The BMC formulas for $\mathcal{A}$ reachability checking, defined in terms of the leaps l_i, is given as follows.

Interleaving Execution

$$RlInt[\![\mathcal{A}]\!]_k = I_{Int}(s_0) \wedge \bigwedge_{i=0}^{k-1} RI_{l_i}(\mu_i, \mu_{i+1}) \tag{5.16}$$

where

$$RI_{l_0}(\mu_0, \mu_1) = \bigwedge_{j=0}^{l_0-1} T_{Int}(s_j, s_{j+1}) \wedge [\![\neg\phi]\!]_{j+1} \tag{5.17}$$

$$RI_{l_i}(\mu_i, \mu_{i+1}) = \bigwedge_{j=n}^{n+l_i-1} T_{Int}(s_j, s_{j+1}) \wedge [\![\neg\phi]\!]_{j+1} \tag{5.18}$$

$T_{Int}(s_i, s_{i+1})$ has been given in equation (4.12), and as usual, we have $n = \sum_{j=1}^{i-1} l_j$ for $1 \leqslant i \leqslant k-1$.

BFS Execution

$$RlBfs[\![\mathcal{A}]\!]_k = I_{Bfs}(s_0) \wedge \bigwedge_{i=0}^{k-1} RB_{l_i}(\mu_i, \mu_{i+1}) \tag{5.19}$$

where

$$RB_{l_0}(\mu_0, \mu_1) = \bigwedge_{j=0}^{l_0-1} T_{Bfs}(s_j, s_{j+1}) \wedge [\![\neg\phi]\!]_{j+1} \tag{5.20}$$

$$RB_{l_i}(\mu_i, \mu_{i+1}) = \bigwedge_{j=n}^{n+l_i-1} T_{Bfs}(s_j, s_{j+1}) \wedge [\![\neg\phi]\!]_{j+1} \tag{5.21}$$

The relation $T_{Bfs}(s_j, s_{j+1})$ has been given in equation (4.16).

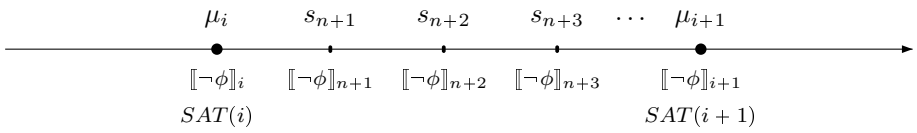


Figure 5.4: Leap methods for other reachability properties.

Chaining Execution

$$RlChain[\![\mathcal{A}]\!]_k = I_{Chain}(s'_0) \wedge \bigwedge_{i=0}^{k-1} RC_{l_i}(\mu_i, \mu_{i+1}) \tag{5.22}$$

where

$$RC_{l_0}(\mu_0, \mu_1) = \bigwedge_{j=0}^{l_0-1} T_{Chain}(s'_j, s_{j+1}) \wedge [\![\neg\phi]\!]_{j+1} \tag{5.23}$$

$$RC_{l_i}(\mu_i, \mu_{i+1}) = \bigwedge_{j=n}^{n+l_i-1} T_{Chain}(s'_j, s_{j+1}) \wedge [\![\neg\phi]\!]_{j+1} \tag{5.24}$$

The relation $T_{Chain}(s'_j, s_{j+1})$ has been given in equation (4.24).

Example 5.3.1 Consider again the system depicted in Figure 5.2. As we are no longer verifying deadlock properties, idle steps do not apply anymore. Therefore, we put the constraints related to the property at each step j. Using the BFS unrolling from equations (5.19), (5.20), and (5.21), we have the following formula for a leap execution between $i = 0$ and $i = 1$, with $l_0 = 3$:

$$\begin{aligned} leap[\![\mathcal{A}, \phi]\!]_1 \quad = \quad & \big(v_{0,0} \wedge \neg v_{1,0} \wedge \neg v_{2,0} \wedge \neg v_{3,0} \wedge \neg v_{4,0} \wedge \neg v_{5,0}\big) \\ & \wedge \Big(v_{0,0} \wedge \neg v_{0,1} \wedge v_{1,1} \wedge v_{2,1} \wedge (v_{3,0} \leftrightarrow v_{3,1}) \wedge (v_{4,0} \leftrightarrow v_{4,1}) \\ & \quad \wedge (v_{5,0} \leftrightarrow v_{5,1})\Big) \wedge [\![\neg\phi]\!]_1 \\ & \wedge \Big(v_{1,1} \wedge v_{2,1} \wedge \neg v_{1,2} \wedge v_{3,2} \wedge v_{4,2} \wedge v_{2,2} \wedge (v_{0,1} \leftrightarrow v_{0,2}) \\ & \quad \wedge (v_{5,1} \leftrightarrow v_{5,2})\Big) \wedge [\![\neg\phi]\!]_2 \\ & \wedge \Big(v_{4,2} \wedge v_{3,2} \wedge v_{2,2} \wedge \neg v_{4,3} \wedge \neg v_{3,3} \wedge v_{2,3} \wedge v_{5,3} \wedge (v_{1,2} \leftrightarrow v_{1,3}) \\ & \quad \wedge (v_{0,2} \leftrightarrow v_{0,3})\Big) \wedge [\![\neg\phi]\!]_3 \end{aligned}$$

Obviously, each $[\![\neg\phi]\!]_j$ should be replaced with the formula related to the property to be checked.

Remark 5.3.1 Although, we express all the equations in terms of synchronized transition systems, we will see in Chapter 8 that the leap approach can also be applied to other types of models such as Petri net.

5.4 Jumping Methods

The efficiency of the leap approach depends to a large extent on the value of a leap l_i. Obviously, a big value of l_i indicates that the unrolling operation performs a long jump at bound i. Similarly, a small value means a short jump. It is therefore, crucial to choose the right value of each l_i. We will present in this section two heuristic algorithms used for choosing resoanable leap values.

5.4.1 Using Logarithmic Functions

As explained in Section 1.1.2, the size of the BMC formula is relatively small at an earlier stage of the execution and, consequently, the solver will run fast during this period. It is, therefore, beneficial to perform bigger jumps at the beginning in order to avoid calling a SAT solver frequently just for solving a small formula at a time.

Based on the above observation, we want the leaps to be bigger when the BMC execution starts, and we want them to become smaller at upper bounds. When we increase the bound, the formula grows in size as well, hence the solver will need more time and resources for solving it. Therefore, by performing short jumps at upper bounds, we can ensure that the BMC formula will not be too big to handle by the solver.

The growth of base 2 logarithmic function $log_2()$, depicted in Figure 5.6, matches exactly our above-mentioned scenario. If we construct a graph showing the execution time of a BMC session, by associating the x axis to the CPU time spent at each bound, and the y axis to the bound value, we will obtain a graph similar to Figure 5.6. For that reason, we choose the $log_2()$ function as a way of determining the leaps.

We determine the leap in the following way. We set up various CPU time thresholds, each of them corresponds to the one consumed by a SAT solver at each execution step. When the value is really small, i.e. below a given minimum threshold, or when it is too big, i.e. beyond a given maximum threshold, we use a small constant leap. We also set up another middle value between the minimum and maximum time interval. In order to go deep faster, we multiply the logarithmic function with the small constant leap before we reach the middle value.

When the maximum value is reached, we use only the small constant in order to have smaller leaps up to a certain huge CPU time limit. This limit is reached when the formula becomes too big to be solved by a SAT solver. In this case, we backtrack to the previous step, recompute the leap by dividing the latest one by two, and rejump accordingly.

Using the above method allows us always to obtain a BMC formula manageable by the solver even though we put several steps together in a single SAT check.

These different leap values are depicted in Figure 5.5, where `k` is the constant leap, `log2` is the base 2 logarithmic function, and `b` is the bound.

Note that in this jumping strategy we assume the BMC execution to follow the general case, that is, the deeper you go, the slower the execution. However, in certain situations, the solver may run faster at a very deep depth value.

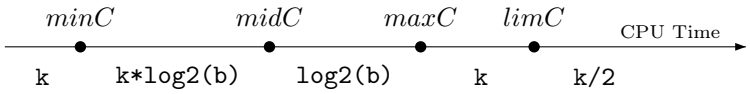


Figure 5.5: Different leap values.

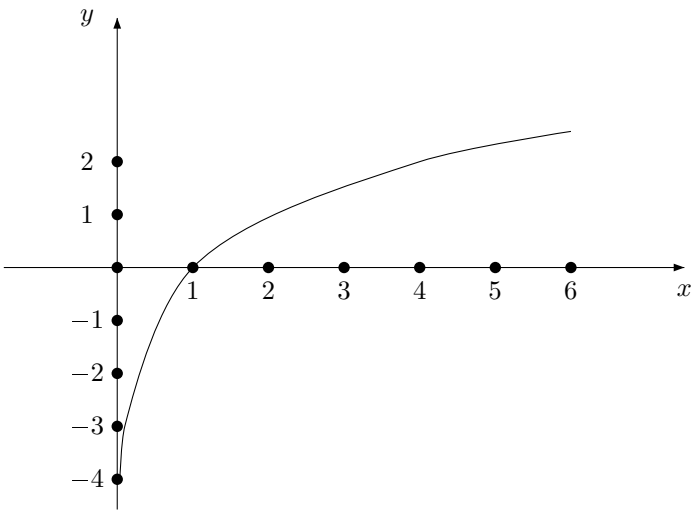


Figure 5.6: Plot for the base 2 logarithmic function.

Algorithm 5.1 illustrates the above method. The jumping strategy starts from line 7 to line 17. $fixLeap$ holds the small constant leap. The variable $minC$, $midC$, $maxC$, and $limC$ determine the CPU time thresholds. The function `backtrack()` performs the backtrack operation when $limC$ is reached. $maxL$ corresponds to the maximum BMC length. The function `unroll()` is used for calling any unrolling technique for reachability that implements one of the techniques in Section 5.2 and Section 5.3. F contains the formula obtained after the transformation process. The function `solve()` calls the SAT solver. It returns *true* if F is satisfiable, $false$ otherwise. `updateCpuTime()` computes the CPU time consumed at each step, which is stored into $solvingTime$. The variables *bound* and *leap* are initialized by the constant leap, and the algorithm stops when $maxL$ is reached or F is satisfiable.

Remark 5.4.1 For clarity purposes, the jumping strategy in Algorithm 5.1 illustrates only our basic idea. However, there are some implementation details we have taken into account, that are not mentioned in Algorithm 5.1.

1. In the first situation, i.e. $solvingTime < minC$, we also count the number of iterations in the main loop. If this number exceeds a given value, we move to the next condition, that is $minC \leqslant solvingTime < midC$.

Algorithm 5.1: JumpByLogarithm

input : P a 1-safe Petri net
$maxL$ maximum BMC length
$minC, midC, maxC, limC$ time thresholds
$fixLeap$ fixed constant leap
$idleMethod$ unrolling method with idle step

output: *true* if a counterexample has been found
$false$ otherwise

```
1  bound ⟵ fixLeap ;
2  isSat ⟵ false ;
3  while (bound ⩽ maxL) and (isSat = false) do
4      F ⟵ unroll(P, bound, idleMethod) ;
5      isSat ⟵ solve(F) ;
6      solvingTime ⟵ updateCpuTime () ;
7      if solvingTime < minC then
8          leap ⟵ fixLeap ;
9      else if minC ⩽ solvingTime < midC then
10         leap ⟵ fixLeap * log2(bound) ;
11     else if midC ⩽ solvingTime < maxC then
12         leap ⟵ log2(bound) ;
13     else if maxC ⩽ solvingTime < limC then
14         leap ⟵ fixLeap ;
15     else if solvingTime ⩽ limC then
16         leap ⟵ leap / 2 ;
17         backtrack(bound) ;
18     bound ⟵ bound + leap ;
19 end
20 return isSat
```

2. We restrict the $fixLeap$ value to be a very small integer, but greater than 1. In this way, we could avoid the problem discussed at the beginning of this Chapter related to the speed of the whole leap operation. So far the values we used for our benchmarks are around 4, i.e. 3, 4, or 5.

3. When $limC$ is reached, we also count the number of backtrack operations, and the algorithm stops when it comes back to the latest bound the solver was able to check. In addition, dividing the leap by 2 may yields a value equals 1. In this case, the algorithm stops.

4 For the the CPU time thresholds $minC$, $midC$, $maxC$, and $limC$, there are no standard values that work for all benchmarks. These constants are chosen while

performing the experimentations. In addition, some of those constants may not be used at all. For instance, if the formula is satisfiable between $midC$ and $maxC$, then we will not reach the constant $maxC$ and $limC$. The important thing here is to understand the idea behind the method, but one should performs a couple of tests in order to know which values work best for each benchmark.

5.4.2 Using Interpolation

In this heuristic we use interpolation-extrapolation methods instead of the base 2 logarithmic function as explained below.

In many cases, SAT solving times steadily increase throughout the BMC execution until we reach a point where the circuit becomes satisfiable. Therefore, we want to estimate, for each benchmark, the bound at which the CPU time limit could be reached. We also prefer not to spend too much time in smaller bounds and try to use the available resources to go as deep as possible. All this can be achieved by jumping directly to the estimated bound just after a few execution steps.

To perform the estimation, a number of data points containing the bounds and the CPU time spent for each bound have to be collected during the initial phase of a BMC execution. A fix constant leap is used while collecting the data points, and the collection process stops when a given CPU time is reached. A curve is then constructed out of these points, and extrapolated to reach the CPU time limit. However, for the extrapolation function to work properly, these data points need to be smoothed. We use the smoothing method introduced in [Rei67] and the well-known Newton polynomial for the extrapolation. When all these operations are finished, the algorithm jumps directly to the estimated bound obtained by the extrapolation function. However, the generated bound value is merely an estimation. That is, it doesn't necessarily mean that the CPU time limit is actually reached at this bound. But, in case this limit is really reached we do the same procedure as in the previous jumping method. Figure 5.7 explains the interpolation method.

Algorithm 5.2: jumpByInterpolation

```
1 ...
2 data ⟵ addPoint (bound, solvingTime) ;
3 if solvingTime ⩽ midC then
4     leap ⟵ fixLeap ;
5 else if midC < solvingTime < limC then
6     leap ⟵ interpolate (data) ;
7 ...
```

Now, if we use this jumping scheme, the lines 7 to 14 in Algorithm 5.1 will be replaced by the lines 2 to 6 in Algorithm 5.2. Here, the points to be interpolated are stored in the variable *data*. These points consist of the bounds with their corresponding solving

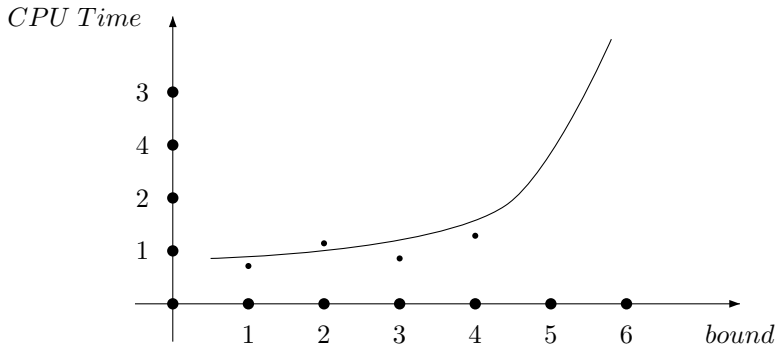


Figure 5.7: The interpolation method with the collected points.

times. The function `addPoint()` adds a point into *data* , while `interpolate()` performs the interpolation/extrapolation. The rest of the variables are the same as in Algorithm 5.1.

5.5 Finding the Shortest Counterexample

Once a SAT formula derived from a Boolean circuit is proved satisfiable, a counterexample is found. However, since our heuristics skip some bounds, this counterexample may not be the shortest unless the bound was increased by one since the last nonsatisfiable bound. Therefore, to find the shortest counterexample, we proceed as follows. If k was the last nonsatisfiable bound and n is the bound for which a counterexample has been found, the shortest counterexample must be of length i with $k < i \leqslant n$. This can be further restricted by removing any idle steps found in the counterexample. For instance, suppose that two idle steps appear in the counterexample, then the bound for the shortest counterexample can be restricted to $k < i \leqslant n - 2$. From this point on, we proceed as in dichotomic search. That is, take the bound in the middle of the possible range of i and check again. If at this new bound, say n', the formula is not satisfiable, then the range for i will be updated as $n' < i \leqslant n - 2$. Otherwise, idle steps will be removed from n' (assume there are j idle steps) and the next range will be updated as $n < i \leqslant n' - j$. This procedure continues until the range reduces to a single number and the shortest counterexample is found. The whole operation requires at most $\log_2(n - k)$ steps.

5.6 Time Performance

For the leap approach, we normally obtain at each bound a formula bigger than the one generated from a single step in a standard BMC. Which also suggests that the leap approach may take longer time to solve than the standard one. To evaluate the time performance of the leap approach, the following two cases need to be considered.

Case 1 The formula is satisfiable in a bound between leaps and there is a big distance between that bound and the next leap. In this case, standard methods might go faster, especially if it happens at an earlier stage of a BMC execution. More precisely let $SAT(j)$ be the time spent by a SAT solver for solving the formula at a step j. Assume that we have a leap $l = 10$. If the formula is satisfaible at step $j+2$, then we have $SAT(j)+SAT(j+l) \geqslant SAT(j) + SAT(j+1) + SAT(j+2)$, which simply means standard methods win over the leap one. The jumping methods we introduced in Section 5.4 allow us to always have reasonable leap values, and help us avoid this situation.

Case 2 The formula is unsatisfiable in all the bounds between leaps. In this case the leap method will always go faster. We have the following theorem to prove this assertion.

Theorem 5.6.1

Given a natural number $l \geqslant 1$. Let $SAT(j)$ be the time spent by a SAT solver for solving the BMC formula generated at a step j. If the formula is unsatisfiable in each step j with $0 \leqslant j \leqslant l$, then we have $SAT(j) + SAT(j+l) \leqslant SAT(j) + SAT(j+1) + \ldots + SAT(j+l)$.

Proof The size of a BMC formula is usually polynomial in the number of its variables. Assume we have a Boolean formula ϕ with n variables. If the size of ϕ is polynomial in n, then the time complexity for solving ϕ is $\Omega(2^n)$, which means 2^n is the lower bound time for solving ϕ (see [CLRS09]).

Now, assume that n_j is the number of variables at a state s_j for $0 \leqslant j \leqslant l$, we then have

$$2^{n_j} \leqslant SAT(j)$$

Therefore,

$$2^{n_j} + 2^{n_{j+1}} + \ldots + 2^{n_{j+l}} \leqslant SAT(j) + SAT(j+1) + \ldots + SAT(j+l).$$

Obviously, we can also state that

$$2^{n_j} + 2^{n_{j+l}} < 2^{n_j} + 2^{n_{j+1}} + \ldots + 2^{n_{j+l}}$$

and

$$2^{n_j} + 2^{n_{j+l}} \leqslant SAT(j) + SAT(j+l).$$

The previous two inequalities imply that

$$SAT(j) + SAT(j+l) \leqslant 2^{n_j} + 2^{n_{j+1}} + \ldots + 2^{n_{j+l}}.$$

Combining everything, we have

$$\begin{aligned}
&2^{n_j} + 2^{n_{j+l}} \\
&\leqslant SAT(j) + SAT(j+l) \\
&\leqslant 2^{n_j} + 2^{n_{j+1}} + \ldots + 2^{n_{j+l}} \\
&\leqslant SAT(j) + SAT(j+1) + \ldots + SAT(j+l).
\end{aligned}$$

We then conclude that

$$SAT(j) + SAT(j+l) \leqslant SAT(j) + SAT(j+1) + \ldots + SAT(j+l)$$ ■

We should also mention that many of today's BMC tools also include a switch to allow the user to specify a (fixed) increment size. Our heuristics may sound a little complicated, but the advantage of using them is that, they allow the incrementation to be done automatically and not in fixed size, especially for the interpolation method. In addition, these methods can be very helpful when the user is first attacking a new problem since it is not always evident to start directly with a fixed increment size unless one knows what to expect from the system. Hence, using a method that allows automatic incrementing can be helpful in this situation. Moreover, the learning and incremental mechanisms that exist in today's SAT solver help to maintain the information from the previous check. Thus, it helps as well to speed up the execution.

Chapter 6

An Automata-Theoretic Approach

Even though they are efficient, all the encodings discussed in Chapter 4 and Chapter 5 apply only to reachability. In this chapter, we improve the methods in order to check other types of properties. Here, we adopt the logic LTL as a specification formalism since it can express a broad range of properties including safety, liveness, and fairness. Furthermore, it has a path-like structure well suited for a BMC operation.

To perform the unrolling, we do not construct a Boolean formula directly out of an LTL specification. Rather, we apply the technique known as *automata-theoretic* approach, in which the LTL formula is first translated into an automaton on infinite words, known as *Büchi automaton*, before encoding it into a Boolean formula [VW86, WVS83]. The objective is to associate each LTL formula with a Büchi automaton that accepts exactly all infinite words modeling the formula. This approach reduces the LTL model checking problem into known automata-theoretic problems. In this way, some existing automata-related algorithms can be used to solve the model checking problem. Translating the LTL formula into an automaton may appear to add extra cost to the model checking process, but it has been shown experimentally that for most of the LTL formulas regularly used in formal verification, the resulting automata is often small [Wol02]. Consequently, the size of the Boolean formula constructed from the automaton can be small as well, which is really helpful especially for a BMC operation. Moreover, creating a Boolean formula from an automaton is often simpler than creating it directly from the LTL formula itself.

Automata-theoretic approaches are performed in the following way: build first the automaton representing the state graph of the system, next translate the negation of the LTL formula into a Büchi automaton, then build a new automaton formed by the synchronized product of the two, and finally conduct an *emptiness check* of the product, i.e. check if there is an infinite word accepted by the new automaton. If such a word exists, it corresponds to a counterexample.

The above-mentioned steps are commonly used to deal with unbounded model check-

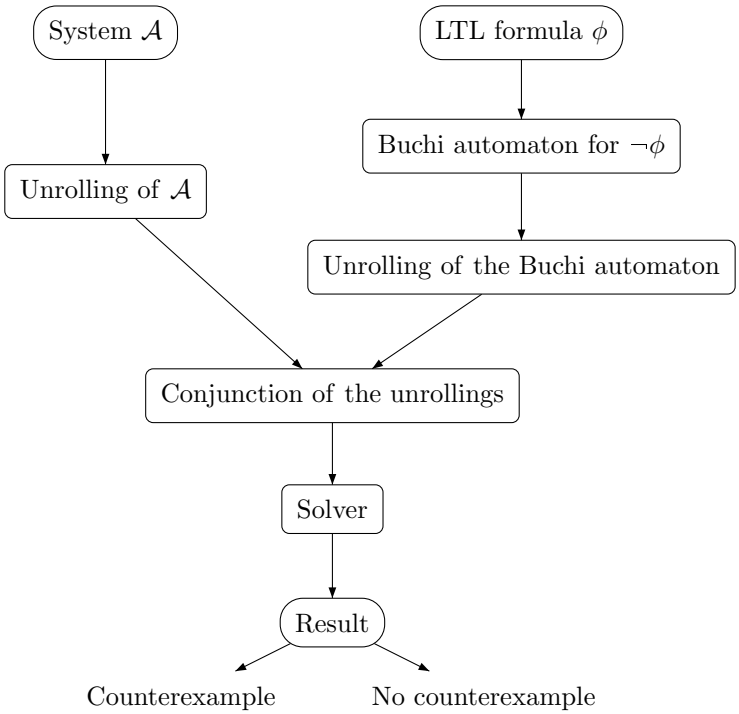


Figure 6.1: Automata-theoretic approach to BMC.

ing. Since our focus is on BMC, our technique is quite different, especially in the last step. We do not build a new product automaton. Rather, we translate separately the two automata into Boolean formulas, and then construct the conjunction of the two formulas. The whole operation is divided in four principal tasks (see Figure 6.1):

1. build the unrolling of the system up to a bound k;
2. translate the negation of the LTL formula into a Büchi automaton;
3. build the unrolling of the automaton up to k;
4. compute the conjunction of the two unrollings.

We apply our technique to synchronized product of transition systems. The LTL specifications we are checking concern the events of the system. Therefore, to perform the unrolling of the TSs, we extend the encodings introduced in Chapter 4 by including into the formulas some information concerning the enabled events.

We are mainly interested in the type of Büchi automata known as *transition-based generalized Büchi automata* (TGBA). There are two main reasons for this choice. First, in theoretical viewpoint, the TS model and TGBA are graph-based formalisms that have roughly the same structure. As a result, it is much simpler to build the conjunction between the TS unrolling and the one constructed from the automata. Second, practically,

it has been proven that translating LTL formulas into TGBA yields smaller automata (see e.g. [DLP04]).

The rest of the chapter is organized as follows. In Section 6.1, we review some work in the literature related to this topic. In Section 6.2, we will discuss the Büchi automata encodings. In Section 6.3, we will introduce the BMC encodings.

6.1 Related Work

Automata-theoretic approaches are much more common in unbounded model checking [DLP04, AEF+05, She05, Var95, VW86, Var06]. Some people also apply it to on-the-fly model checking, in which the principal steps consist in building on demand the automaton related to an LTL formula, and then performing an on-the-fly emptiness check on the product automaton [CVWY92, GH93, Hol88, CDLP05]. This approach can be combined with Tarjan's algorithm as discussed in [JV04, GH93]. In [Cou99], Courveur proposed another variant of this algorithm, which can perform fairness checking without needless overhead.

The most closely related to ours is the BMC approach introduced in [Sor02], which also generates the Boolean formula by constructing the conjunction of the two formulas obtained from the LTL transformation and the system model respectively. However, the big difference is that the method in [Sor02] applies to timed automata rather than transition systems.

The BMC method in [dMRS02, CKOS04], by contrast, is performed by searching for *fair loop* counterexamples in the product system. Although the search can sometimes go deeper than in other methods, it does not produce linear sized Boolean formulas. Combining this method with other encodings such as the ones in [CPRS02, LBHJ04] could possibly improve the results.

A method for translating an LTL formula into a *symbolic Büchi automaton* is also introduced in [CGH94]. The translation produces a result ready to be used for symbolic model checking, and it runs in time $O(|\phi|)$, where ϕ is the LTL formula. Nevertheless, the number of states in the automaton may sometimes blow up exponentially. Shneider suggested a way to go around this problem in [Sch99], while Schuppan *et al.* discussed a possible application of this method to BMC in [SB05].

Note Most early work on BMC with LTL is based on the encoding introduced by Biere *et al.* [BCCZ99], in which they translate LTL specifications directly into Boolean formulas. They also encode loop and nonloop executions differently, whereas subsequent techniques use the same encoding for both (e.g. [CPRS02, FSW02]).

6.2 Representation of Büchi Automata

A *Büchi automaton* (BA) [AD94] is a finite state automaton that accepts infinite inputs. Invented by the Swiss mathematician Julius Richard Büchi in 1962, it is useful for specifying the behavior of nonterminating systems, such as hardware or operating systems.

We start this section by describing different types of Büchi automata, especially those frequently used in formal verification. The definitions given here is slightly different from the one in [AD94] because we need to adapt them to match the requirements of an LTL formula.

6.2.1 Different Types of Büchi Automata

Let P be a finite set of atomic propositions, and $Bool(P)$ the set of Boolean formulas over P. Rather than defining the label of an automaton over the alphabet 2^P, we are interested in using the set $Bool(P)$.

Definition 6.2.1 (Büchi Automata)
A **Büchi Automaton** *is a tuple* $\mathcal{B} = (Q, \delta, Q_0, F)$*, where*

- Q *is a finite nonempty set of states,*
- $\delta \subseteq Q \times Bool(P) \times Q$ *is the transition relation,*
- $Q_0 \subseteq Q$ *is the set of initial states,*
- $F \subseteq Q$ *is the set of accepting states.*

A transition from a state q_i *to a state* q_j *is denoted* $q_i \xrightarrow{\sigma} q_j$ *or* $\langle q_i, \sigma, q_j \rangle$*, where* $\sigma \in Bool(P)$ *determines the* **label** *of the transition.*

A **run** *of* $\mathcal{B}$ *is an infinite sequence* $\rho = q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} q_2 \xrightarrow{\sigma_2} \ldots$ *such that* $q_0 \in Q_0$*, and* $\forall i \geqslant 0, \langle q_i, \sigma_i, q_{i+1} \rangle \in \delta$. ■

Let σ be an element of $Bool(P)$. For each $p \in P$, we say that $p \in \sigma$ if p appears in the terms of σ. Likewise, for each $u \in 2^P$, we say that $u \in \sigma$ if $\forall p \in u, p \in \sigma$.

Definition 6.2.2 (Accepting Run for BA)
Given a Büchi automaton $\mathcal{B} = (Q, \delta, Q_0, F)$*. A run* $\rho = q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} q_2 \xrightarrow{\sigma_2} \ldots$ *of* $\mathcal{B}$ *is* accepting *if and only if there exists* $q_i \in F$ *that appears infinitely often in* ρ*. The automaton accepts an infinite word* $\xi = u_0 u_1 u_2 \ldots$ *over* 2^P *if there exists an accepting run* ρ *such that for all transitions* $\langle q_i, \sigma_i, q_{i+1} \rangle$ *in* ρ*, with* $i \geqslant 0$*, we have* $u_i \in \sigma_i$*. The set of words accepted by* $\mathcal{B}$ *is denoted* $\mathcal{L}(\mathcal{B})$. ■

A Büchi automaton can be represented as a graph, in which the nodes correspond to the states, and the edges to the transitions. Nodes with double circles indicate the accepting states, while a single incoming arrow—not coming from another state—identify each initial state.

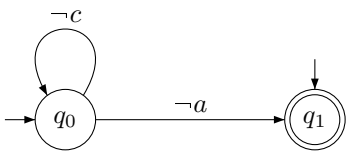


Figure 6.2: A Büchi automaton.

Example 6.2.1 Figure 6.2 depicts an example of a Büchi automaton. Formally, we have $\mathcal{B} = (Q, \delta, Q_0, F)$ where $Q = \{q_0, q_1\}$; $Q_0 = \{q_0, q_1\}$; and $F = \{q_1\}$.

There exist other variants of Büchi automata. The most interesting ones, in terms of verification, are *generalized Büchi automata* (GBA) with multiple acceptance conditions defined in terms of states, and *transition-based generalized Büchi automata* (TGBA) with acceptance conditions on the transitions. We give below the definitions of these two types.

Definition 6.2.3 (Generalized Büchi Automata)

A **generalized Büchi Automaton** *is a tuple* $\mathcal{B} = (Q, L, \delta, Q_0, \mathcal{F})$*, where*

- Q *is a finite nonempty set of states,*
- $L : Q \to Bool(P)$ *is a state labeling function,*
- $\delta \subseteq Q \times Q$ *is the transition relation,*
- $Q_0 \subseteq Q$ *is the set of initial states,*
- $\mathcal{F} \subseteq 2^Q$ *is the set of sets of accepting states.* ■

Definition 6.2.4 (Accepting Run for GBA)

Given a generalized Büchi automaton $\mathcal{B} = (Q, L, \delta, Q_0, \mathcal{F})$*. A run* $\rho = q_0 q_1 q_2 \ldots$ *of* $\mathcal{B}$ *is* **accepting** *if and only if* $\forall\ F_j \in \mathcal{F}$ *there exists* $q_i \in F_j$ *that appears infinitely often in* ρ*. The automaton accepts an infinite word* $\xi = u_0 u_1 u_2 \ldots$ *over* 2^P *if there exists an accepting run* $\rho = q_0 q_1 q_2 \ldots$ *such that* $\forall i \geqslant 0,\ u_i \in L(q_i)$*.* ■

Definition 6.2.5 (Transition-based Generalized Büchi Automata)

A **transition-based generalized Büchi Automaton** *is a tuple* $\mathcal{B} = (Q, \delta, Q_0, \mathcal{T})$*, where*

- Q *is a finite nonempty set of states,*
- $\delta \subseteq Q \times Bool(P) \times Q$ *is the transition relation,*
- $Q_0 \subseteq Q$ *is the set of initial states,*
- $\mathcal{T} \subseteq 2^\delta$ *is the set of sets of accepting transitions.* ■

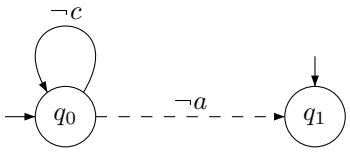


Figure 6.3: A Transition-based Büchi automaton.

Definition 6.2.6 (Accepting Run for TGBA)

Given a transition-based generalized Büchi automaton $\mathcal{B} = (Q, \delta, Q_0, \mathcal{T})$. *A run* $\rho = q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} q_2 \xrightarrow{\sigma_2} \ldots$ *of* $\mathcal{B}$ *is* accepting *if and only if* $\forall\, T_j \in \mathcal{T}$ *there exists* $t_i \in T_j$ *that appears infinitely often in* ρ. *The automaton accepts an infinite word* $\xi = u_0 u_1 u_2 \ldots$ *over* 2^P *if there exists an accepting run* ρ *such that for all transitions* $\langle q_i, \sigma_i, q_{i+1} \rangle$ *in* ρ, *with* $i \geqslant 0$, *we have* $u_i \in \sigma_i$. ■

Lemma 6.2.1

Given a generalized Büchi automaton, one can build an equivalent transition-based generalized Büchi automaton, and vice versa.

Proof (⇒) All we need to do is move the labels and the acceptance conditions from states to transitions. An accepting transition in a TGBA is simply one that leads to an accepting state in the corresponding GBA. Therefore, assume that we have a GBA with an accepting set $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$, we can build a TGBA with an accepting set $\mathcal{T}$, such that $|\mathcal{T}| = |\mathcal{F}|$, and the elements of a set $T_i \in \mathcal{T}$ correspond to all incoming transitions of the states in F_i. i.e. for all $t = q_a \xrightarrow{\sigma} q_b \in T_i$ in the TGBA, we have $q_b \in F_i$ in the corresponding GBA.

(⇐) Follows by similar argumentation. ■

Example 6.2.2 Figure 6.3 depicts a TGBA version of the automaton in Example 6.2.1. Accepting transitions are represented as dashed lines. Formally, we have $\mathcal{B} = (Q, \delta, Q_0, \mathcal{T})$ where, $Q = \{q_0, q_1\}$; $Q_0 = \{q_0, q_1\}$; and $\mathcal{T} = \{\{q_0 \xrightarrow{\neg a} q_1\}\}$.

6.2.2 Translating an LTL Formula Into a TGBA

Translating an LTL formula into a Büchi automata is a whole independent topic that has already been investigated thoroughly by many researchers (see e.g. [DGV99, SB00, DLP04, Tau03, EH00, GO01]). In our work, we use the technique introduced in [GPVW95]. With this translation, the labels and acceptance conditions are defined in terms of states rather than transition. Therefore, to obtain a TGBA, we simply push the labels into the transitions after the translation, and adjust the acceptance conditions accordingly.

The translation is performed using the tableau rules shown in Table 6.1. The rules are applied to a formula ϕ until it becomes an expression composed only of *elementary*

Table 6.1: Tableau Rules

ϕ	$\alpha_1(\phi)$	$\alpha_2(\phi)$
$\phi_1 \wedge \phi_2$	$\{\phi_1, \phi_2\}$	$\varnothing$
$\phi_1 \vee \phi_2$	$\{\phi_1\}$	$\{\phi_2\}$
$\phi_1 \mathsf{U} \phi_2$	$\{\phi_2\}$	$\{\phi_1, \mathsf{X}(\phi_1 \mathsf{U} \phi_2)\}$
$\phi_1 \mathsf{R} \phi_2$	$\{\phi_2, \phi_1\}$	$\{\phi_2, \mathsf{X}(\phi_1 \mathsf{R} \phi_2)\}$

subformulas—i.e. a constant, or an atomic proposition, or a formula starting with the X operator. These subformulas put in disjunctive normal form (DNF) constitute an *elementary cover* of ϕ. Each term of the cover corresponds to a state of the automaton. The atomic propositions in the term, as well as their negations are used to construct the label of the state. The other subformulas in the term define the transitions out of the state, its *next part*, along with the acceptance conditions.

The translation process is applied to the next part of each state until no more new covers can be generated, that is, until we obtain a closed set of elementary covers. The automaton is obtained by connecting each state to its next part. The states in the elementary cover of ϕ correspond to the initial states. Acceptance conditions are added to each elementary subformula of the form $\mathsf{X}(\phi_1 \mathsf{U} \phi_2)$. The acceptance condition concerns every state s whose label does not imply $\phi_1 \mathsf{U} \phi_2$, or whose label implies ϕ_2. Then we move the labels and acceptance conditions into the transitions, in order to obtain a TGBA.

Before translating an LTL formula, we usually transform it into NNF, as explained in Section 2.3. During the translation, if there are more than one atomic proposition in a term of an elementary cover, we combine them with the logic operator $\wedge$ to form the label. Thus, $\wedge$ will be the only operator that may appear in each label.

Example 6.2.3 As an example, consider an LTL formula $\phi = \mathsf{F}a \wedge \mathsf{F}\neg a$. After applying the NNF transformation rules and the ones in Table 6.1, we obtain the following three-term elementary cover:

$$\Big(\mathit{true} \wedge \mathsf{X}(\mathit{true} \mathsf{U} a) \wedge \mathsf{X}(\mathit{true} \mathsf{U} \neg a)\Big) \vee \Big(a \wedge \mathsf{X}(\mathit{true} \mathsf{U} \neg a)\Big) \vee \Big(\neg a \wedge \mathsf{X}(\mathit{true} \mathsf{U} a\Big)$$

We thus obtain three initial states, which we call q_0, q_1, and q_2 respectively. We could see, for instance, that the state q_1 has a label a, and next part $(\mathit{true} \mathsf{U} \neg a)$. If we apply the rules to the next part of q_1 we obtain the following cover:

$$\Big(\mathit{true} \wedge \mathsf{X}(\mathit{true} \mathsf{U} \neg a)\Big) \vee \Big(\neg a\Big)$$

Hence, q_1 generates two new states, say q_3 and q_4, labeled as true and $\neg a$ respectively. The state q_4 has no next part, while q_3 has again $(\mathit{true} \mathsf{U} \neg a)$. Therefore, from q_3, no more *new* covers could be generated, which means we do not obtain new states anymore. Instead, we have two outgoing transitions which go to q_3 (a self-loop) and q_4 respectively.

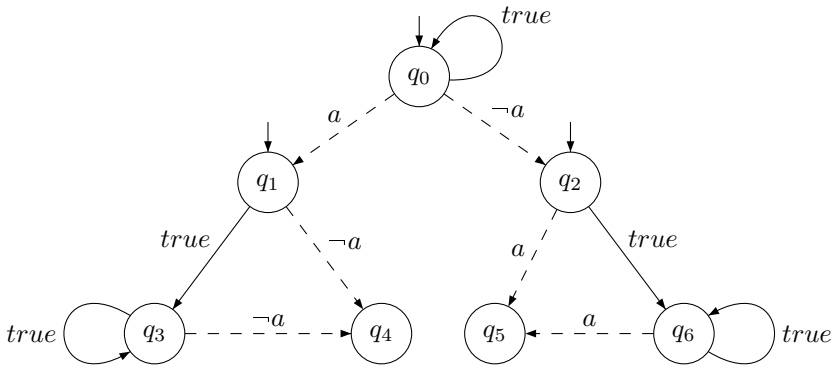


Figure 6.4: TGBA for the LTL formula ($\mathsf{F}a \wedge \mathsf{F}\neg a$).

After pushing the labels on the transitions and adapting the acceptance conditions accordingly, we obtain the TGBA depicted in Figure 6.4. The states q_5 and q_6 are obtained from q_2 using the same mechanism as before. Formally, we have $\mathcal{B} = (Q, \delta, Q_0, \mathcal{T})$ where, $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$; $Q_0 = \{q_0, q_1, q_2\}$; and $\mathcal{T} = \{T_1, T_2\}$, such that $T_1 = \{\{q_0 \xrightarrow{a} q_1\}; \{q_2 \xrightarrow{a} q_5\}; \{q_6 \xrightarrow{a} q_5\}\}$ and $T_2 = \{\{q_0 \xrightarrow{\neg a} q_2\}; \{q_1 \xrightarrow{\neg a} q_4\}; \{q_3 \xrightarrow{\neg a} q_4\}\}$.

Definition 6.2.7 (TGBA Associated to an LTL Formula)

Given an LTL formula ϕ. Let us denote $Sub(\phi)$ the set of subformulas of ϕ, and Acc the set of until subformulas of ϕ, i.e. $Acc = \{\phi_1 \, \mathsf{U} \, \phi_2 \in Sub(\phi)\}$. We define $\mathcal{B}_\phi$, the TGBA associated to ϕ, as follows:

- $Q = 2^{Sub(\phi)}$,
- $\delta = Q \times Bool(P) \times Q$,
- $Q_0 = \{\phi\}$,
- $\mathcal{T} = \{T_1, T_2, ..., T_n\}$, *such that* $\forall t = q_a \xrightarrow{\sigma} q_b \in T_i$ *we have* $q_a \in Q$ *and* $q_b \in 2^{Acc}$. ■

After the translation, the resulted automaton should accept all infinite words modeling a given LTL formula. The following theorem formalizes this assertion.

Theorem 6.2.2

Let ϕ be an LTL formula, and $\mathcal{B}_\phi = (Q, \delta, Q_0, \mathcal{T})$ the TGBA associated to ϕ. Then $\mathcal{L}(\mathcal{B}_\phi)$ corresponds to all infinite words over 2^P that models ϕ. Formally, we have $\mathcal{L}(\mathcal{B}_\phi) = \{\xi \in (2^P)^\omega \mid \xi \vDash \phi\}$.

Proof ($\Rightarrow$) The idea is to show that $\forall \varphi \in Sub(\phi)$, the language accepted by $\mathcal{B}_\phi$ with $Q_0 = \varphi$ is equal to $\{\xi \in (2^P)^\omega \mid \xi \vDash \varphi\}$.

Let φ be a element of $Sub(\phi)$, and $\xi = u_0 u_1 u_2 \ldots$ an infinite word accepted by $\mathcal{B}_\phi$ with $Q_0 = \varphi$. One of the cases below will arise:

(1) $\varphi = p$ with $p \in P$. Since $Q_0 = \{p\}$, and p is the only proposition that appears in Q_o, then we have $p \in \sigma_0$, thus $p \in \xi(0)$. As a result, $\xi \vDash p$ (see Definition 2.3.2).

(2) $\varphi = g \wedge h$ with $g \in P$ and $h \in P$. We then have $Q_0 = \{g \wedge h\}$, i.e. $g \in \sigma_0$ and $h \in \sigma_0$, which implies $g \in \xi(0) \wedge h \in \xi(0)$. Thus, $\xi \vDash g$ and $\xi \vDash h$. As a result, we conclude $\xi \vDash g \wedge h$.

(3) $\varphi = g \vee h$ with $g \in P$ and $h \in P$. We then have $Q_0 = \{g \vee h\}$, i.e. $g \in \sigma_0$ or $h \in \sigma_0$, which implies $g \in \xi(0) \vee h \in \xi(0)$. Thus, $\xi \vDash g$ or $\xi \vDash h$. As a result, we conclude $\xi \vDash g \vee h$.

(4) $\varphi = \mathsf{X}g$ with $g \in P$. We have $Q_0 = \{\mathsf{X}g\}$ and $Q_1 = \{g\}$. This implies that $g \in \xi(1)$. We then can say that $\xi^1 \vDash g$. Hence, $\xi \vDash \mathsf{X}g$ (see Definition 2.3.2).

(5) $\varphi = g \,\mathsf{U}\, h$ with $g \in P$ and $h \in P$. Applying the tableau rules, we have $g \,\mathsf{U}\, h \equiv h \vee (g \wedge \mathsf{X}(g \,\mathsf{U}\, h))$. Therefore, proving $\xi \vDash g \,\mathsf{U}\, h$ is then equivalent to proving $\xi \vDash h \vee (g \wedge \mathsf{X}(g \,\mathsf{U}\, h))$.
We have
$$Q_0 = \{h \vee (g \wedge \mathsf{X}(g \,\mathsf{U}\, h))\},$$
thus
$$\Big(h \in \xi(0)\Big) \vee \Big(g \in \xi(0) \wedge g \in \xi(1) \wedge h \in \xi(1)\Big).$$
i.e.
$$\Big(\xi \vDash h\Big) \vee \Big(\xi \vDash g \wedge \xi^1 \vDash g \ \wedge \xi^1 \vDash h)\Big).$$
We then found $i = 1$ such that $\xi^i \vDash h$ and $\forall j < i$, $\xi^i \vDash g$. Hence, following Definition 2.3.2, we conclude that $\xi \vDash g \,\mathsf{U}\, h$.

Note that if g and h are elements of $2^{sub(\phi)}$ the proof can still be constructed recursively using the above results. The proof of all derived operators can also be obtained from these results.

($\Leftarrow$) Let $\xi = u_0 u_1 u_2 \ldots$ be an infinite word modeling the formula ϕ. We will prove that ξ is accepted by $\mathcal{B}_\phi$. To do that, we need to show that $\forall T_j \in \mathcal{T}$, there exists $t_i \in T_j$ that appears infinitely often in a run of the form $\rho = q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} q_2 \xrightarrow{\sigma_2} \ldots$, such that $\forall i \geqslant 0$, $u_i \in \sigma_i$.

Let T_j be a set of transition in $\mathcal{T}$. The following two facts can be established.

(i) We have seen before that if $\xi \vDash \phi$, then for a subformula $\varphi = g \,\mathsf{U}\, h$ we will also have $\xi \vDash \varphi$ with $Q_0 = \varphi$.

(ii) By definition of $\mathcal{B}_\phi$, each element $t \in T_j$ is of the form $q_a \xrightarrow{\sigma} q_b$ where $q_b \in Acc$ i.e. $q_b \in \{g \,\mathsf{U}\, h \in Sub(\phi)\}$.

Combining the facts (i) and (ii), we can say that any element of T_j will appear infinitely often in the run ρ, which means ρ is accepted by $\mathcal{B}_\phi$. Thus, ξ is accepted by $\mathcal{B}_\phi$. ■

6.2.3 TGBA Encoding

Consider a TGBA $\mathcal{B} = (Q, \delta, Q_0, \mathcal{T})$. The encoding of $\mathcal{B}$ at bound k is given as:

$$[\![\mathcal{B}]\!]_k = I(Q_0) \wedge T(Q_0, Q_1) \wedge \ldots \wedge T_{Acc}(Q_{k-1}, Q_k) \tag{6.1}$$

where

- Q_i is the set of states reached at step i,
- $I(Q_0)$ is the characteristic function for the set of initial states,
- $T(Q_i, Q_{i+1})$ encodes the transitions between steps i and $i+1$, $\forall i \in [0, k-2]$,
- $T_{Acc}(Q_{k-1}, Q_k)$ encodes the transitions between steps $k-1$ and k, as well as the acceptance conditions for bound k.

We denote the Boolean variable related to a state q_n at step i as $q_{n,i}$. The characteristic function for a set of states Q_j, denoted as $\mathcal{X}_{Q_j}$, is formed by the product of the Boolean variables related to the states in Q_j. The characteristic function for the set of initial states is given as:

$$I(Q_0) = \mathcal{X}_{Q_0} \wedge \bigwedge_{q_n \in Q \setminus Q_0} \neg q_{n,0} \tag{6.2}$$

For example, for the TGBA depicted in Figure 6.4 we have:

$$I(Q_o) = q_{0,0} \wedge q_{1,0} \wedge q_{2,0} \wedge \neg q_{3,0} \wedge \neg q_{4,0} \wedge \neg q_{5,0} \wedge \neg q_{6,0}$$

Let q_n be a state. The equation of a transition t from a state q_n to another state q_m between two steps i and $i+1$ is given as:

$$t(q_{n,i}, q_{m,i+1}) = q_{n,i} \wedge \sigma \wedge q_{m,i+1} \tag{6.3}$$

where σ corresponds to the label of the transition t. In practice, we give an index i to all the variables in σ. For instance, given a transition $t = q_{0,0} \xrightarrow{\neg a} q_{2,1}$, we have $t(q_{0,0}, q_{2,1}) = q_{0,0} \wedge \neg a_0 \wedge q_{2,1}$.

Recall that $\wedge$ is the only operator that may exists in a label σ. Therefore, we treat the constant *true* as a neutral element if it ever appears among the operands in σ. For example, assume $\sigma = true \wedge a$, this means $\sigma = a$, and we have $t(q_{n,i}, q_{m,i+1}) = q_{n,i} \wedge a_i \wedge q_{m,i+1}$.

We then define $T(Q_i, Q_{i+1})$, $i \in [0, k-2]$ as:

$$T(Q_i, Q_{i+1}) = \bigwedge_{q_n \in Q_i} \bigvee_{t \in T_{q_n}} t(q_{n,i}, q_{m,i+1}) \tag{6.4}$$

where T_{q_n} is the set of all outgoing transitions from q_n, and $q_m \in Q_{i+1}$.

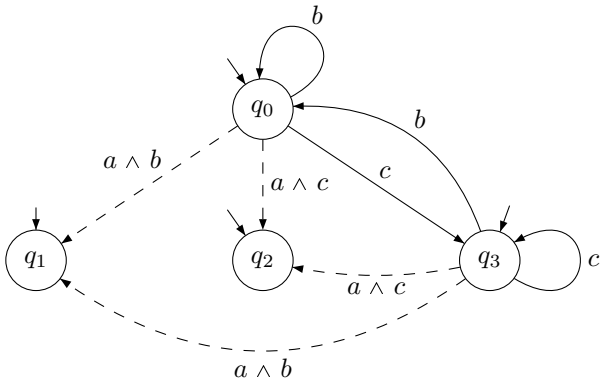


Figure 6.5: TGBA for the LTL formula $\phi = a \,\mathsf{R}\, (c \vee b)$.

Let q_n be a state. The equation of an accepting transition t_A from a state q_n to another state q_m between two steps i and $i+1$ is given as:

$$t_A(q_{n,i}, q_{m,i+1}) = q_{n,i} \wedge \big(\sigma_A \vee \bigvee_{t \in T_{q_n} \setminus A_{q_n}} \neg\sigma\big) \wedge q_{m,i+1} \tag{6.5}$$

where A_{q_n} is the set of accepting transitions from q_n, σ_A is the label associated to t_A, and σ the one associated to t.

We define $T_{Acc}(Q_{k-1}, Q_k)$ as:

$$T_{Acc}(Q_{k-1}, Q_k) = \bigwedge_{q_n \in Q_{k-1}} \Big(\bigvee_{t \in T_{q_n} \setminus A_{q_n}} t(q_{n,k-1}, q_{m,k}) \vee \bigvee_{t_A \in A_{q_n}} t_A(q_{n,k-1}, q_{m,k1}) \Big) \tag{6.6}$$

where $q_m \in Q_k$.

Example 6.2.4 Consider the LTL formula $\phi = a \,\mathsf{R}\, (c \vee b)$. The TGBA obtained from this formula is depicted in Figure 6.5. The Boolean equation for this automaton at bound $k = 2$ is given as:

$$[\![\mathcal{B}]\!]_2 \quad = \quad I(Q_0) \wedge T(Q_0, Q_1) \wedge T_{Acc}(Q_1, Q_2)$$

where

$$
\begin{aligned}
I(Q_0) &= q_{0,0} \wedge q_{1,0} \wedge q_{2,0} \wedge q_{3,0} \\
T(Q_0, Q_1) &= \big((q_{0,0} \wedge b_0 \wedge q_{0,1}) \vee (q_{0,0} \wedge (a_0 \wedge b_0) \wedge q_{1,1}) \vee (q_{0,0} \wedge (a_0 \wedge c_0) \wedge q_{2,1}) \\
&\quad \vee (q_{0,0} \wedge c_0 \wedge q_{3,1})\big) \\
&\quad \wedge \big((q_{3,0} \wedge (a_0 \wedge b_0) \wedge q_{1,1}) \vee (q_{3,0} \wedge (a_0 \wedge c_0) \wedge q_{2,1}) \vee (q_{3,0} \wedge b_0 \wedge q_{0,1}) \\
&\quad \vee (q_{3,0} \wedge c_0 \wedge q_{3,1})\big) \\
T_{Acc}(Q_1, Q_2) &= \big((q_{0,1} \wedge ((a_1 \wedge b_1) \vee \neg c_1 \vee \neg b_1) \wedge q_{1,2}) \\
&\quad \vee (q_{0,1} \wedge ((a_1 \wedge c_1) \vee \neg c_1 \vee \neg b_1) \wedge q_{2,1}) \\
&\quad \vee (q_{0,1} \wedge b_1 \wedge q_{0,2}) \vee (q_{0,1} \wedge c_1 \wedge q_{3,2})\big) \\
&\quad \wedge \big((q_{3,1} \wedge ((a_1 \wedge b_1) \vee \neg b_1 \vee \neg c_1) \wedge q_{1,2}) \\
&\quad \vee (q_{3,1} \wedge ((a_1 \wedge c_1) \vee \neg b_1 \vee \neg c_1) \wedge q_{2,2}) \\
&\quad \vee (q_{3,1} \wedge b_1 \wedge q_{0,2}) \vee (q_{3,1} \wedge c_1 \wedge q_{3,2})\big)
\end{aligned}
$$

In this example, $Q_0 = Q_1 = Q_2 = \{q_0, q_1, q_2, q_3\}$. This situation happens mainly because of the self-loop on q_0 and q_3.

6.3 Building the BMC Formulas

The BMC formulas are obtained by building the conjunction of the TS unrolling and the TGBA one. To unroll the TS, we extend the techniques introduced in Chapter 4 by also including into the formulas the enabled events at each step. As before, we will take into consideration the three unrolling executions. Namely, interleaving, BFS, and Chaining.

6.3.1 Interleaving Execution

Consider a synchronized TS $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_m)$, a bound k, and an LTL property ϕ such that $\mathcal{B}$ is its corresponding Büchi automaton. We define the BMC formula for $\mathcal{A}$, in terms of interleaving execution and LTL as follows.

$$LInt[\![\mathcal{A}, \phi]\!]_k = LInt[\![\mathcal{A}]\!]_k \wedge [\![\mathcal{B}]\!]_k \tag{6.7}$$

where

$$LInt[\![\mathcal{A}]\!]_k = I_{Int}(s_0) \wedge \bigwedge_{i=0}^{k-1} T_{LInt}(s_i, s_{i+1}) \tag{6.8}$$

and

$$T_{LInt}(s_i, s_{i+1}) = T_{Int}(s_i, s_{i+1}) \wedge e_i \wedge \bigwedge_{e_l \neq e_i} \neg e_l \tag{6.9}$$

Example 6.3.1 To illustrate the interleaving execution, let us take the synchronized system $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ depicted in Figure 6.6, and an LTL formula $\phi = c \,\mathsf{R}\, a$ which means the event a is always enabled along the execution until the first position where c is enabled, or forever if such a position does not exist.

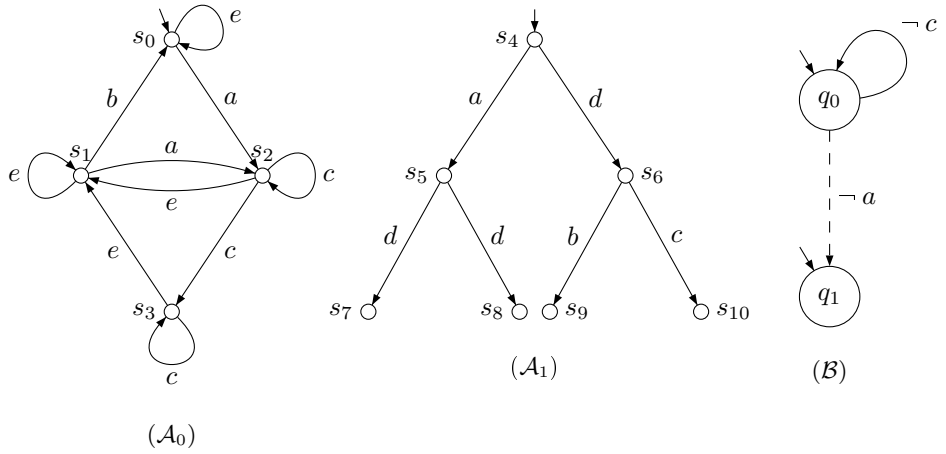


Figure 6.6: A complete example: two TSs and a TGBA.

The encoding of $\mathcal{A}$ at bound $k = 2$ is given as:

$$
\begin{aligned}
LInt[\![\mathcal{A}]\!]_2 \;=\;& I_{Int}((s_0^0, s_0^1)) \\
& \wedge T_{LInt}((s_0^0, s_0^1), (s_1^0, s_1^1)) \\
& \wedge T_{LInt}((s_1^0, s_1^1), (s_2^0, s_2^1)) \\
\\
=\;& \Big(v_{0,0} \wedge v_{4,0} \wedge \neg v_{1,0} \wedge \neg v_{2,0} \wedge \neg v_{3,0} \wedge \neg v_{5,0} \wedge \neg v_{6,0}\Big) \\
& \wedge \Big(v_{0,0} \wedge v_{4,0} \wedge \neg v_{0,1} \wedge \neg v_{4,1} \wedge v_{5,1} \wedge v_{2,1} \wedge (v_{1,0} \leftrightarrow v_{1,1}) \wedge (v_{3,0} \leftrightarrow v_{3,1}) \\
& \quad \wedge (v_{6,0} \leftrightarrow v_{6,1}) \wedge (v_{7,0} \leftrightarrow v_{7,1}) \wedge (v_{8,0} \leftrightarrow v_{8,1}) \wedge (v_{9,0} \leftrightarrow v_{9,1}) \\
& \quad \wedge (v_{10,0} \leftrightarrow v_{10,1}) \wedge a_0 \wedge \neg b_0 \wedge \neg c_0 \wedge \neg d_0 \wedge \neg e_0\Big) \\
& \wedge \Big(v_{2,1} \wedge \neg v_{2,2} \wedge v_{1,2} \wedge (v_{0,1} \leftrightarrow v_{0,2}) \wedge (v_{3,1} \leftrightarrow v_{3,2}) \wedge (v_{4,1} \leftrightarrow v_{4,2}) \\
& \quad \wedge (v_{5,1} \leftrightarrow v_{5,2}) \wedge (v_{6,1} \leftrightarrow v_{6,2}) \wedge (v_{7,1} \leftrightarrow v_{7,2}) \wedge (v_{8,1} \leftrightarrow v_{8,2}) \\
& \quad \wedge (v_{9,1} \leftrightarrow v_{9,2}) \wedge (v_{10,1} \leftrightarrow v_{10,2}) \wedge \neg a_1 \wedge \neg b_0 \wedge \neg c_0 \wedge \neg d_0 \wedge e_0\Big)
\end{aligned}
$$

where

$$
\begin{aligned}
& s_0^0 = s_0;\ s_0^1 = s_4;\ s_1^0 = s_2;\ s_1^1 = s_5; s_2^0 = s_1;\ s_2^1 = s_5; \\
& e_0 = a;\ e_1 = e
\end{aligned}
$$

The negation of the formula ϕ is $\neg c\ \mathsf{U}\ \neg a$. The translation of this formula into a TGBA yields the one shown in Figure 6.6 ($\mathcal{B}$). Formally, we have $\mathcal{B} = (Q, \delta, Q_0, F)$ where, $Q = \{q_0, q_1\}$, $Q_0 = \{q_0, q_1\}$, and $\mathcal{T} = \{\{q_0 \xrightarrow{\neg a} q_1\}\}$.

The translation of the automaton into Boolean formula yields

$$
\begin{aligned}
[\![\mathcal{B}]\!]_2 &= I(Q_0) \\
&\quad \wedge T(Q_0, Q_1) \\
&\quad \wedge T_{Acc}(Q_1, Q_2) \\
&= \Big(q_{0,0} \wedge q_{1,0}\Big) \\
&\quad \wedge \Big((q_{0,0} \wedge \neg c_0 \wedge q_{0,1}) \vee (q_{0,0} \wedge \neg a_0 \wedge q_{1,1})\Big) \\
&\quad \wedge \Big((q_{0,1} \wedge \neg c_1 \wedge q_{0,2}) \vee (q_{0,1} \wedge (\neg a_1 \vee c_1) \wedge q_{1,2})\Big)
\end{aligned}
$$

Here again we have $Q_0 = Q_1 = Q_2 = \{q_0, q_1\}$ because of the self-loop on q_0.

The BMC equation is obtained by the conjunction of the two formulas. We have, $LInt[\![\mathcal{A}, \phi]\!]_2 = LInt[\![\mathcal{A}]\!]_2 \wedge [\![\mathcal{B}]\!]_2$.

Using a SAT solver one can find a satisfying assignment for the formula at bound $k = 2$. We then have the following counterexample:

$$(s_0, s_4) \xrightarrow{a} (s_2, s_5) \xrightarrow{e} (s_1, s_5)$$

Theorem 6.3.1

Given a synchronized system $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_m)$, an LTL formula ϕ, and a bound k. If $LInt[\![\mathcal{A}, \phi]\!]_k$ is satisfiable, then $\mathcal{A} \not\models \phi$.

Proof Assume that $LInt[\![\mathcal{A}, \phi]\!]_k$ is satisfiable, i.e. $LInt[\![\mathcal{A}]\!]_k \wedge [\![\mathcal{B}]\!]_k$ is satisfiable. This also means that $I_{Int}(s_0) \wedge T_{LInt}(s_0, s_1) \wedge \ldots \wedge T_{LInt}(s_{k-1}, s_k)$, which represents the encoding of the execution from step 0 to step k, is satisfiable. Since $[\![\mathcal{B}]\!]_k$, which represents the encoding of the negation of the property ϕ at bound k, is also satisfiable, we can therefore say that we found a path of length k from the initial state to a state where the LTL property is violated, which means we have $\mathcal{A} \not\models \phi$. ∎

Theorem 6.3.2

Given a synchronized system $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_m)$, an LTL formula ϕ, and a bound k. $\mathcal{A} \not\models \phi$ implies that $\exists k \in \mathbb{N}$ such that $LInt[\![\mathcal{A}, \phi]\!]_k$ is satisfiable.

Proof To prove this theorem, all we need to do is use the results from the proof of Theorem 4.4.2 and replace $\neg\phi$ into $\mathcal{B}$. ∎

6.3.2 Breadth First Search Execution

Given a synchronized system $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_m)$, and a bound k, and an LTL formula ϕ such that $\mathcal{B}$ is its corresponding Büchi automaton. We define the BMC formula for $\mathcal{A}$, in terms of BFS execution and LTL, as follows.

$$LBfs[\![\mathcal{A}, \phi]\!]_k = LBfs[\![\mathcal{A}]\!]_k \wedge [\![\mathcal{B}]\!]_k \tag{6.10}$$

where

$$LBfs[\![\mathcal{A}]\!]_k = I_{Bfs}(s_0) \wedge \bigwedge_{i=0}^{k-1} T_{LBfs}(s_i, s_{i+1}) \tag{6.11}$$

and

$$T_{LBfs}(s_i, s_{i+1}) = T_{Bfs}(s_i, s_{i+1}) \wedge \bigwedge_{e_i \in E_i} e_i \wedge \bigwedge_{e_l \in \Sigma \setminus E_i} \neg e_l \tag{6.12}$$

Recall that $\Sigma = \bigcup_{j=0}^{m} \Sigma^j$, while E_i corresponds to the set of events that fire at step i (see Definition 4.4.2).

Example 6.3.2 As an Example, consider the synchronized system $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ depicted in Figure 6.6, and the LTL formula $\phi = c \,\mathsf{R}\, a$. The unrolling of $\mathcal{A}$ using the BFS method for k = 2 is given as:

$$\begin{aligned}
LBfs[\![\mathcal{A}]\!]_2 \;=\;& I_{Bfs}((S_0^0, S_0^1)) \\
& \wedge T_{LBfs}((S_0^0, S_0^1), (S_1^0, S_1^1)) \\
& \wedge T_{LBfs}((S_1^0, S_1^1), (S_2^0, S_2^1)) \\
\\
=\;& \Big(v_{0,0} \wedge v_{4,0} \wedge \neg v_{1,0} \wedge \neg v_{2,0} \wedge \neg v_{3,0} \wedge \neg v_{5,0} \wedge \neg v_{6,0}\Big) \\
& \wedge \Big(v_{0,0} \wedge v_{4,0} \wedge v_{0,1} \wedge \neg v_{4,1} \wedge v_{2,1} \wedge v_{5,1} \wedge v_{6,1} \wedge (v_{1,0} \leftrightarrow v_{1,1}) \\
& \quad \wedge (v_{3,0} \leftrightarrow v_{3,1}) \wedge (v_{7,0} \leftrightarrow v_{7,1}) \wedge (v_{8,0} \leftrightarrow v_{8,1}) \wedge (v_{9,0} \leftrightarrow v_{9,1}) \\
& \quad \wedge (v_{10,0} \leftrightarrow v_{10,1}) \wedge a_0 \wedge \neg b_0 \wedge \neg c_0 \wedge d_0 \wedge e_0\Big) \\
& \wedge \Big(v_{2,1} \wedge v_{5,1} \wedge v_{6,1} \wedge v_{0,1} \wedge v_{0,2} \wedge v_{1,2} \wedge v_{2,2} \wedge v_{3,2} \wedge v_{7,2} \wedge v_{8,2} \\
& \quad \wedge v_{10,2} \wedge (v_{4,1} \leftrightarrow v_{4,2}) \wedge (v_{9,1} \leftrightarrow v_{9,2}) \\
& \quad \wedge \neg a_0 \wedge \neg b_0 \wedge c_0 \wedge d_0 \wedge e_0\Big)
\end{aligned}$$

where

$S_0^0 = \{s_0\};\ S_0^1 = \{s_4\};\ S_1^0 = \{s_0, s_2\};\ S_1^1 = \{s_5, s_6\};\ S_2^0 = \{s_0, s_1, s_2, s_3\};\ S_2^1 = \{s_7, s_8, s_{10}\};$
$E_0 = \{a, d, e\};\ E_1 = \{c, d, e\}$

To verify whether the system satisfy the property ϕ we simply build the BMC equation $LBfs[\![\mathcal{A}, \phi]\!]_2 = LBfs[\![\mathcal{A}]\!]_2 \wedge [\![\mathcal{B}]\!]_2$, where $[\![\mathcal{B}]\!]_2$ is the same as in Example 6.3.1.

We then obtain the following counterexample at bound $k = 2$:

$$(\{s_0\}, \{s_4\}) \xrightarrow{\{a,d,e\}} (\{s_0, s_2\}, \{s_5, s_6\}) \xrightarrow{\{c,d,e\}} (\{s_0, s_1, s_2, s_3\}, \{s_7, s_8, s_{10}\})$$

Theorem 6.3.3
Given a synchronized system $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_m)$, an LTL formula ϕ, and a bound k. If $LBfs[\![\mathcal{A}, \phi]\!]_k$ is satisfiable, then $\mathcal{A} \not\models \phi$.

Proof Assume that $LBfs[\![\mathcal{A}, \phi]\!]_k$ is satisfiable. This means $LBfs[\![\mathcal{A}]\!]_k \wedge [\![\mathcal{B}]\!]_k$ is satisfiable. This implies that $I_{Bfs}(s_0) \wedge \bigwedge_{i=0}^{k-1} T_{LBfs}(s_i, s_{i+1})$ is satisfiable (see equation (6.14)).

Hence, there exists a path of length k from the initial state. Since $[\![\mathcal{B}]\!]_k$ represents the negation of the property, and since the product $LBfs[\![\mathcal{A}]\!]_k \wedge [\![\mathcal{B}]\!]_k$ is satisfiable, we conclude that we found a path leading to a state where the property ϕ is violated. Therefore, we have $\mathcal{A} \not\models \phi$. ■

Theorem 6.3.4
Given a synchronized system $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_m)$, an LTL formula ϕ, and a bound k. $\mathcal{A} \not\models \phi$ implies that $\exists k \in \mathbb{N}$ such that $LBfs[\![\mathcal{A}, \phi]\!]_k$ is satisfiable.

Proof Assume $\mathcal{A} \not\models \phi$. This means that, using the BFS execution, we can construct a path from the initial state to a state where the property ϕ is violated. Assume that k is the length of that path. We then have a path $s_0, \ldots, s_k$. Since s_{i+1} is reachable from s_i in one step, we can say that $T_{Bfs}(s_i, s_{i+1}) = 1$. Therefore, we also have $I_{Bfs}(s_0) \wedge T_{Bfs}(s_0, s_1) \wedge \ldots \wedge T_{Bfs}(s_{k-1}, s_k) = 1$. Hence $LBfs[\![\mathcal{A}]\!]_k = I_{Bfs}(s_0) \wedge T_{LBfs}(s_0, s_1) \wedge \ldots \wedge T_{LBfs}(s_{k-1}, s_k) = 1$ because each $T_{LBfs}(s_i, S_{i+1})$ is an extension of $T_{Bfs}(s_i, s_{i+1})$ by simply adding the transition label into the equation, and adding the label will not change the satisfiability of the formula. The property ϕ is violated at step k. Therefore, we conclude that $LBfs[\![\mathcal{A}]\!]_k \wedge, [\![\mathcal{B}]\!]_k$ is satisfiable as $\mathcal{B}$ represents the negation of the property ϕ. ■

6.3.3 Chaining

Given a synchronized system $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_m)$, and a bound k, and an LTL formula ϕ such that $\mathcal{B}$ is its corresponding Büchi automaton. We define the BMC formula for $\mathcal{A}$, in terms of chaining execution and LTL, as follows.

$$LChain[\![\mathcal{A}, \phi]\!]_k = LChain[\![\mathcal{A}]\!]_k \wedge [\![\mathcal{B}]\!]_k \tag{6.13}$$

where

$$LChain[\![\mathcal{A}]\!]_k = I_{Chain}(s'_0) \wedge \bigwedge_{i=0}^{k-1} T_{LChain}(s'_i, s_{i+1}) \tag{6.14}$$

and

$$T_{LChain}(s'_i, s_{i+1}) = T_{Chain}(s'_i, s_{i+1}) \wedge \bigwedge_{e_i \in E_i} e_i \wedge \bigwedge_{e_l \in \Sigma \setminus E_i} \neg e_l \tag{6.15}$$

Example 6.3.3 Let us take again the synchronized system $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ depicted in Figure 6.6 and the LTL formula: $\phi = c \,\mathsf{R}\, a$. The unrolling of $\mathcal{A}$ at bound $k = 2$ using the chaining execution is given by the following equation:

$$
\begin{aligned}
LChain[\![\mathcal{A}]\!]_1 &= I_{LChain}((S_0'^0, S_0'^1)) \\
&\quad \wedge T_{LChain}((S_0'^0, S_1'^1), (S_1^0, S_1^1)) \\
&= \Big(v_{0,0} \wedge v_{2,0} \wedge v_{4,0} \wedge v_{5,0} \wedge \neg v_{1,0} \wedge \neg v_{3,0} \wedge \neg v_{6,0} \wedge \neg v_{7,0} \neg v_{8,0} \\
&\quad \wedge \neg v_{9,0} \wedge \neg v_{10,0}\Big) \\
&\quad \wedge \Big(v_{0,0} \wedge v_{2,0} \wedge v_{4,0} \wedge v_{5,0} \wedge \neg v_{4,1} \wedge \neg v_{2,1} \wedge \neg v_{5,1} \wedge v_{0,1} \wedge v_{1,1} \wedge v_{6,1} \wedge v_{7,1} \wedge v_{8,1} \\
&\quad \wedge (v_{3,0} \leftrightarrow v_{3,1}) \wedge (v_{9,0} \leftrightarrow v_{9,1}) \wedge (v_{10,0} \leftrightarrow v_{10,1}) \\
&\quad \wedge a_0 \wedge \neg b_0 \wedge c_0 \wedge d_0 \wedge e_0\Big)
\end{aligned}
$$

where,

$S_0^0 = \{s_0\};\ S_0'^0 = \{s_0, s_2\};\ S_0^1 = \{s_4\};\ S_0'^1 = \{s_4, s_5\};\ S_1^0 = \{s_0, s_1\};\ S_1^1 = \{s_6, s_7, s_8\};$
$E_0 = \{a, d, e\}$

In this example we use the chaining order $[a, d, e]$.

As before, the BMC equation is obtained with the conjunction $LChain[\![\mathcal{A}, \phi]\!]_2 = LChain[\![\mathcal{A}]\!]_2 \wedge [\![\mathcal{B}]\!]_2$, where $[\![\mathcal{B}]\!]_2$ is the same as in Example 6.3.1.

We obtain the following counterexample:

$$(\{s_0, s_2\}, \{s_4, s_5\}) \xrightarrow{E_0} (\{s_0, s_1\}, \{s_6, s_7, s_8\})$$

Theorem 6.3.5

Given a synchronized system $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_m)$, an LTL formula ϕ, and a bound k. If $LChain[\![\mathcal{A}, \phi]\!]_k$ is satisfiable, then $\mathcal{A} \not\models \phi$.

Proof To prove this theorem, all we need to do is use the reasoning from the proof of Theorem 6.3.3 and change $LBfs$ into $LChain$. ■

Theorem 6.3.6

Given a synchronized system $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_m)$, an LTL formula ϕ, and a bound k. $\mathcal{A} \not\models \phi$ implies that $\exists k \in \mathbb{N}$ such that $LChain[\![\mathcal{A}, \phi]\!]_k$ is satisfiable.

Proof The proof of this theorem is also similar to the one for Theorem 6.3.4. All we need to do is change $LBfs$ into $LChain$. ■

6.4 Discussion

An automata-theoretic approach offers numerous advantages when it is combined with model checking in general. However, it also has its drawbacks. We will discuss some of them in this section.

First, since the transformation algorithm itself may presents some errors, it can be difficult sometimes to know whether the obtained automaton correctly represents the

given LTL formula. Therefore, it may be necessary to verify the transformation algorithm beforehand or, alternatively, test several transformation techniques on the same formula and compare the results. The work in [TH00] demonstrates how to conduct such tests.

Second, an automaton transformation algorithm has an exponential time complexity in the worst and best case [Wol02]. Still, it is practically feasible because, as mentioned earlier, most of the LTL formulas used in verification produce small automata. One solution of this problem is to use *alternating automata* [GO01], which can be generated in linear time. It has been pointed out that finding the optimal size for a Büchi automaton is a PSPACE-hard problem [EH00].

Despite the problems mentioned above, automata-theoretic approaches have been successfully implemented into numerous model checking tools including Java PathFinder [VHBP00] developed at the NASA Ames Research Center, and SPIN [Hol97] developed by Bell Labs.

Chapter 7

Implementation

We have implemented our methods in a toolset written in C++. The tool accepts as input transition systems and Petri nets. Although all the BMC methods we described in Chapter 4, Chapter 5, and Chapter 6 are devoted to TS models, we have integrated in the tool some options allowing the user to perform deadlock detection for Petri nets. We have also implemented the heuristic algorithms discussed in Chapter 5 for Petri nets. Concerning the TS models, the tool can perform deadlock detection using the method described in Chapter 4, and LTL verification using the method in Chapter 6. Basically, all the major functions in the tool are initiated through command lines entered by the user. We call the tool BMC++. We will describe it in this chapter, and show with some examples the different tasks it can accomplish.

7.1 Different Modules

The tool has four major modules: LTL2BA, BMC, CNF translator, and solver (see Figure 7.1).

7.1.1 LTL2BA

Description

If the user enters an LTL specification, it will be first translated into a Büchi automaton, as explained in Chapter 6. The LTL2BA module assumes this task. The module negates the formula, transforms it into NNF, and stores it as a tree structure. Then it uses the generated tree to build the BA.

Implementation Details

The core of this module is formed by four principal functions. Namely, a parser, an NNF translator, a function that builds the tree structure, and the one which generates the BA.

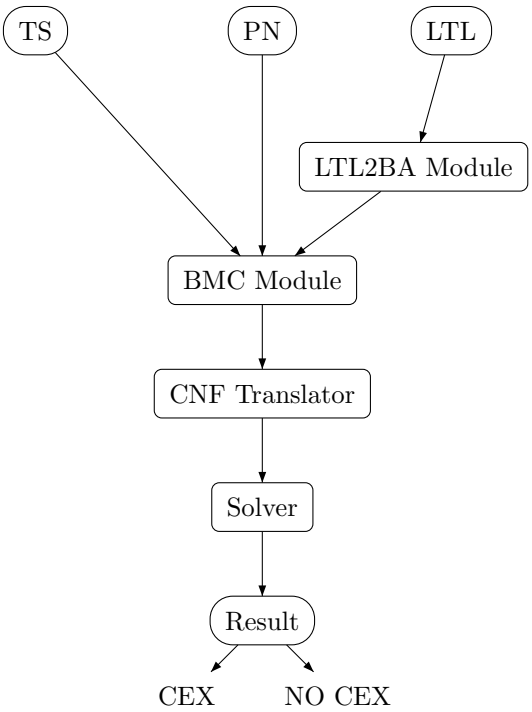


Figure 7.1: BMC++ major modules.

Once the formula has been entered, the parser will parse it. If the formula is correct, then the parser will negate it. Otherwise, the process stops and the tool prompts the user to reenter the formula. To simplify the task of the parser, the formula should be entered without space in between. Parentheses are allowed if the user prefers to put more emphasis on the operators. Table 7.1 shows the list of the most used LTL operators and how the user should enter them.

Table 7.1: LTL operators and their keyboard equivalents

Operator	**Keyboard Equivalent**
$\neg$	!
$\wedge$	&
$\vee$	|
$\Rightarrow$	>
=	=

After parsing the formula, the parser will give its negation to the NNF translator, which simply applies the rules in equations (2.11) to (2.15) to create an NNF formula. Then, another function will receive the generated NNF formula and translate it into tree structure. This step is necessary because it is much more easier to handle the formula if

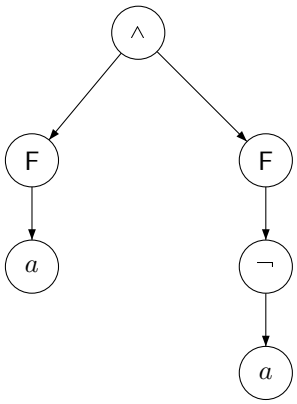


Figure 7.2: The tree representation of the LTL formula $\mathsf{F}a \wedge \mathsf{F}\neg a$.

it is represented as a tree. As an example, Figure 7.2 shows a tree representation of the LTL formula $Fa \wedge F\neg a$.

The final step in this module is assumed by the function which actually performs the BA translation. It takes the tree structure and repeatedly apply the algorithm described in Section 6.2.2 until the final BA is formed. All the above steps are summarized in Figure 7.3 (1).

7.1.2 BMC Module

Description

This is the core of the tool. It performs all the unrolling techniques described in Chapters 4, 5, and 6. For a BMC session involving a TS, if the user entered an LTL specification, this module takes the BA generated by the LTL2BA module, and build a Boolean formula out of it. Obviously, the module also generate a Boolean formula from the unrolling operation. It ends its parts after generating these formulas.

Implementation Details

All operations involving this module are launched through command lines. It begins its tasks after receiving an input file from the user. If it receives a TS file, then all subsequent commands and operations will be devoted only for TSs. The same applies for PNs. Obviously, the first thing it does is fetch the file from the disk and parse it. If there is no errors in the file, then the tool waits for the next command from the user. Otherwise, it prompts an error message. To deal with synchronized TSs, the user must enter the file one by one, and the tool will make the necessary synchronization.

When all the input files have been entered, the user can choose among several options concerning the unrolling operation. Or alternatively, he or she can accept the default options. In case of a TS, the available options are: interleaving execution, BFS execution,

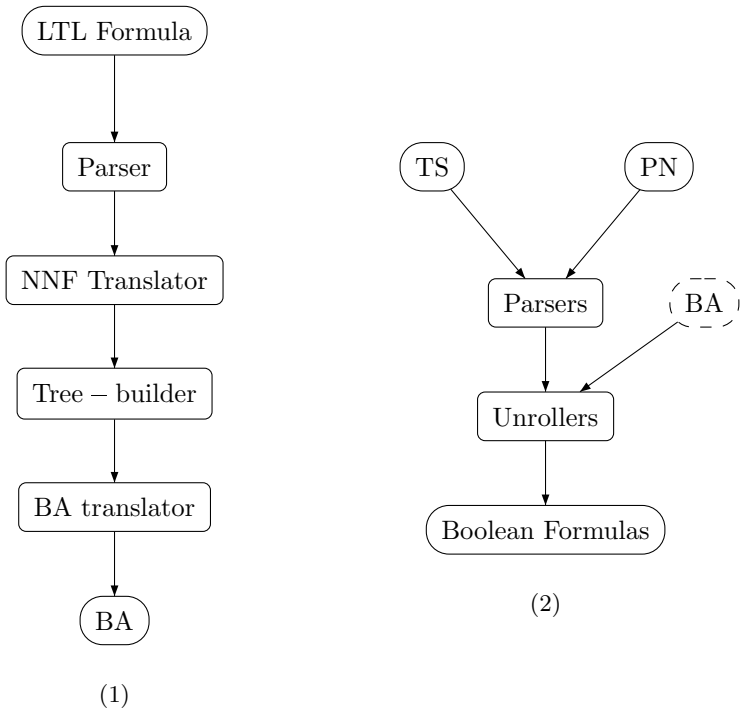


Figure 7.3: Details of the LTL2BA Module (1), and the BMC Module (2).

and chaining execution. Of course, they can also choose whether to check an LTL formula or perform a deadlock detection. If the user chooses LTL formula, then the module calls a function that takes the BA generated by LTL2BA and builds a Boolean formula out of it. For the PN, the options are: interleaving execution, step execution, and process execution. The latest two options are functions we integrated inside the tool based on the method described in [Hel01]. Briefly put, step execution is similar to a standard BFS but adapted to work for Petri nets, while process execution is a version of the step one that takes only into account the shortest path among all possible step executions leading to the same marking. Most importantly, the user can also choose whether to adopt the leap execution or not. Only deadlock detection can be performed with PNs. Figure 7.3 (2) summarizes all the above steps.

Inside the tool, each TS, PN, and BA are mainly stored using a graph-like data structure. The only difference is that the edges and nodes carry more information than a standard graph.

Note that, once the unrolling operation has been launched, no more input files can be entered.

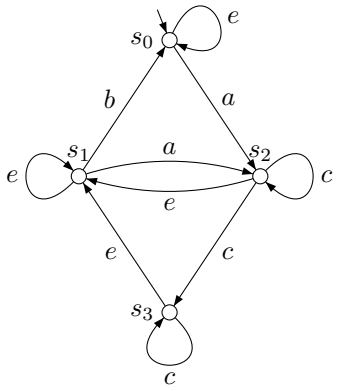


Figure 7.4: A transition system.

7.1.3 CNF translator

This module serves as an intermediate step between BMC module and the solver. It translates the Boolean formulas generated by the BMC module into conjunctive normal form (CNF). This step is necessary because we use a CNF-based solver.

Implementation Details

The translation is performed using Tseiting encoding [Tse83] as explained in Section 2.4.3. To generate a CNF formula, the algorithm applies recursively the rules in Table 2.1. We also combined the translation with the reduction techniques described in [Vel04b]. In case of an LTL check, we transform separately the Boolean formulas from the unrolling and the one from the BA, then put the conjunction of the two to form one single CNF.

7.1.4 Solver

The solver module performs the last step of the whole operation. In fact, we did not build our own solver. Instead, we integrated into our tool a state-of-the-art CNF-based SAT solver called ZChaff [MMZ+01]. This module takes the formula generated by the CNF translator and gives it to the solver, which determines whether the formula is satisfiable or not.

The reason why we chose ZChaff can be explained as follows. After conducting numerous experiments with different SAT-solvers, we observed that ZChaff produces the more stable results in terms of speed. We tried five SAT-solvers: ZChaff, SATO, Grasp, Berkmin, MiniSat, and BCSat. We observed that the running time of those solvers, including ZChaff, varies a lot even though they were run on the same benchmark. Nevertheless, the CPU time consumed by ZChaff is more stable compared to those of the others.

7.2 Input Formats

7.2.1 TS File format

The TS file format has three parts named header, action_names, and transitions. Inside the header, we define the following variables: State_cnt, Transition_cnt, Initial_states which correspond to the number of states, number of transitions, and initial state id respectively. Each event id and its associated name is described in the action_names part. Each line in the transitions part identifies the source, destination, and event related to a transition. Below we formalize the TS format.

Begin TS

Begin Header
State_cnt = $< number\ of\ states >$
Transition_cnt = $< number\ of\ transitions >$
Initial_states = $< id\ of\ the\ initial\ state >$;
End Header

Begin Action_names
$< event\ id1 >$ = "$< event\ name1 >$"
$< event\ id2 >$ = "$< event\ name2 >$"
...
End Action_names

Begin Transitions
$< source\ id >$: $< destination\ id >$,$< event\ id >$ ... $< destination\ id >$,$< event\ id >$;
...
End Transitions

End TS

Example 7.2.1 As an example, bellow we give the file for the TS depicted in Figure 7.4, which is, in fact, the same as the system ($\mathcal{A}_0$) depicted in Figure 6.6.

Begin TS

Begin Header
State_cnt = 4
Transition_cnt = 10
Initial_states = 0;
End Header

Begin Action_names
1 = "a"
2 = "b"
3 = "c"
4 = "e"
End Action_names

Begin Transitions
0: 0,4 2,1;
1: 0,2 1,4 2,1;
2: 2,3 3,3 1,4;
3: 1,4 3,3;
End Transitions

End TS

7.2.2 PEP File format

For a Petri net model, we adopt the format used by the PEP tool [GB96]. Briefly put, the PEP format consist of seven principal parts. Namely, header, some default values, list of places, list of transitions, list of transition to place arcs, list of place to transition arcs, and list of some additional information.

The PEP tool can handle many types of files, so the header is used for specifying the type of the current file (a PEP file is identified by the keyword "PEP" in the header). Since the PEP tool is also a graphical one, some default values specifying the positions of certain nodes of labels in the graph are needed (e.g. position of the initial marking or the place name). Each line in the list of places are constructed with the place identification followed by some description, which may include the name of the place and its position in the graph. The same applies for the list of transitions. Each transition to place arc is defined by both transition and place identifications separated by the character "<". The place to transition arcs are constructed in the same way using the character ">". The last part, which is an optional one, is needed mainly for putting some graphical information such as the caption of the graph. Below we formalize the PEP format.

PEP
PetriBox
FORMAT_N

PL
< *place id* >"*place description*"
< *place id* >"*place description*"

...
TR
$<$ *transition id* $>$"*transition description*"
$<$ *transition id* $>$"*transition description*"
...
TP
$<$ *transition id* $> <$ *place id* $>$
$<$ *transition id* $> <$ *place id* $>$
...
PT
$<$ *place id* $> >$ *transition id* $>$
$<$ *place id* $> >$ *transition id* $>$
...
TX

Example 7.2.2 The following is a PEP file related to the Petri net in Figure 7.5.

PEP
PetriBox
FORMAT_N

PL
1"$g1$"M1m1
2"$g2$"
3"$r1$"
4"$r2$"M1m1
5"$y1$"
6"$y2$"
TR
1"ta"
2"tb"
3"tc"
4"td"
TP
$1 < 1$
$1 < 4$
$2 < 5$
$3 < 3$
$3 < 2$
$4 < 6$
PT

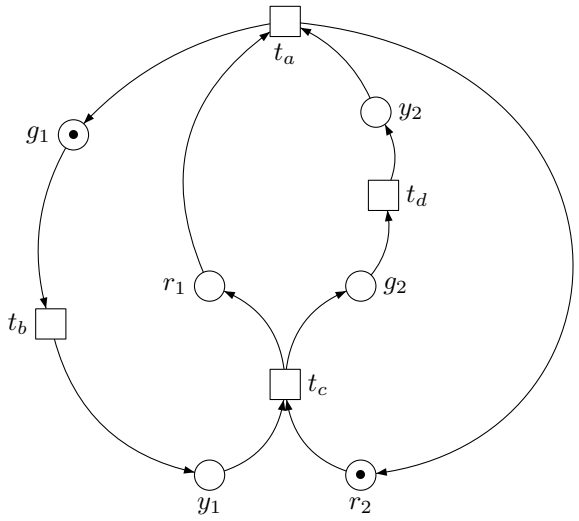


Figure 7.5: A Petri net.

1 > 2
2 < 4
3 < 1
4 < 3
5 < 3
6 < 1
TX
"Example net"

The notation "M1m1" next to the description of places $g1$ and $r2$ means that these places have the initial markings and each of them has one token. The keyword TX identifies the last part in the file.

7.3 Important commands

Almost all the major operations in the tool are launched through command lines. These operations includes reading the input files, the BMC itself, reading the LTL formula. In this section, we will briefly describe the most important commands available in the tool.

read_pn

Usage:	read_pn < *PN filename* >
Description:	Opens the PN model given as argument.

read_ts

Usage:	read_ts $< TS\ filename >$
Description:	Opens the TS model given as argument.

enable_leap

Usage:	enable_leap $< leapMethod >$
Description:	Enables the leap options during a BMC operation for Petri nets.
	This command should have, as an argument, one of the leap methods listed below.
Arguments:	$< interpolation >$ use the interpolation method
	$< log >$ use the logarithmic method.

enable_chaining

Usage:	enable_chaining $[option]$
Description:	Enables chaining during a BMC operation.
Options:	$-c$ use causality for ordering the events.
	If no option is given the event will be chosen randomly.

go_bmc

Usage:	go_bmc $[bmcLength]$
Description	Perform a BMC computation. If no length is given, the tool will stop at length 10. By default, chaining is not used for TS models.

help

Usage:	help $[commandName]$
Description	Gives a description of the command given as argument. If no specific command is given, the tool will simply list all available commands

set_default

Usage:	set_default
Description:	Resets the tool to the default configuration.

source

Usage:	source $< filename >$
Description:	Runs all commands written inside the file given as argument.

reset

Usage:	reset
Description:	Resets the BMC session and reinitialize all variables.
	A BMC session launched after this command will start from bound 1.

Note that, there exist other commands than the ones listed above. To know them the user needs only to issue the command help.

Chapter 8

Case Studies

8.1 Experiments Related to Reachability

We have conducted several tests with a set of deadlock checking benchmarks. The main purpose of our experiments is to show the difference between using chaining or not during a BMC operation. The experiments were conducted on a machine with 2.4 Ghz Intel Pentium processor, 1 GB memory, and Linux Red hat 9 operating system. In all experiments in this thesis, we use ZChaff as SAT solver. The following points explain the columns shown in Table 8.1.

- **Model**: the name of the model.
- $|\mathbf{\Sigma}|$: number of events in the model.
- **No Chain**: results obtained using standard BFS.
- **Chain**: results obtained using chaining.
- **CEX**: the smallest bound at which a counterexample has been found.
- **TotalTime**: running time, in seconds, for the whole BMC computation. We call the C++ function `getrusage()` to obtain these times.

Some of the benchmarks are taken from [JHN03]. They are TSs versions of the deadlock checking benchmarks collected by Corbett [Cor96]. The rest of the benchmarks are from [MR90]. We used the deadlock condition described in Section 4.5 during the experiments.

For each benchmark, we incremented the bound until a counterexample was found. The running times shown in Table 8.1 are the total time for the entire BMC session starting from bound 1 until the CEX bound and including the solver time, as opposed to the ones reported in [JHN03] which show only the times obtained for the CEX bound.

As we can see from Table 8.1, using chaining decreases the number of bound required for finding a counterexample. This decrease is manifested clearly for the *key*s and *mmgt*s

Table 8.1: Deadlock detection results

Model	$\|\Sigma\|$	NoChain		Chain	
		CEX	TotalTime	CEX	TotalTime
DARTES	205	32	6.8	31	6.1
KEY(2)	34	52	5.8	25	1.16
KEY(3)	47	54	8.28	24	1.28
KEY(4)	60	31	2.75	23	1.55
KEY(5)	73	31	3.69	25	2.41
MMGT(3)	19	9	0.6	4	0.1
MMGT(4)	24	11	1.31	4	0.1
HART(25)	50	51	3.82	44	3.34
HART(50)	100	101	29.29	92	26.95
HART(75)	150	151	99.17	142	92.71
HART(100)	200	201	239.7	192	228
SENTEST(25)	34	33	2.3	31	2.23
SENTEST(50)	59	58	12.38	56	11.91
SENTEST(75)	84	83	36.66	81	35.54
SENTEST(100)	109	108	81.96	106	80.02
SPEED(1)	10	3	0.04	1	0.01

benchmarks. For the chaining operation, we allowed the computer to order the events randomly at each step.

In some cases, there is no big difference between the running time obtained with the two methods. As explained in Section 4.6.2 this situation occurs when many states must be added in one iteration for the chaining operation. Still, the experiments show that the chaining method performs faster than standard BFS.

8.1.1 Explication of the Benchmarks

Dartes Program (DARTES) The communication skeleton of a complex Ada program with 32 tasks.

Keyboard Program (KEY) The communication skeleton of an Ada program that manages keyboard/screen interaction in a window manager. The program is scaled by making the number of customer tasks a parameter (m). The size (m) means there are $m + 5$ tasks.

Distributed Memory Manager (MMGT) The communication skeleton of a complex Ada program implementing a memory management scheme with m users. The size m version has $m + 4$ tasks.

Hartstone Program (HART) The communication skeleton of an Ada program in which one task starts and then stops m worker tasks.

Sensor Test Program (SENTEST) The communication skeleton of an Ada program that starts up m tasks to test sensors. The size (m) means there are $m + 4$ tasks.

Table 8.2: Comparison with NuSMV

Model	|Σ|	Chain		NuSMV(BMC)	
		CEX	TotalTime	CEX	TotalTime
DARTES	205	31	6.1	32	1111
KEY(2)	34	25	1.16	60	4076
KEY(3)	47	24	1.28	64	4498
KEY(4)	60	23	1.55	64	5446
KEY(5)	73	25	2.41	64	2960
MMGT(3)	19	4	0.1	10	922
MMGT(4)	24	4	0.1	12	1107
HART(50)	100	92	26.95	>100	117
HART(100)	200	192	228	>200	986

Speed Regulation Program (SPEED) The communication skeleton of an Ada program with 10 tasks that monitor and regulate the speed of a car.

8.1.2 Comparison with the Tool NuSMV

We have also conducted some experiments comparing the results we obtained above with the state-of-the-art model checking tool NuSMV. The tool can perform BDD-based model checking and BMC. We compared our results using the BMC option in the tool. Since the best performances we obtained in the experiments in Table 8.1 are related to the chaining execution, we compared only the NuSMV results with the chaining execution as seen in Table 8.2. The time reported for the NuSMV BMC operations are the sum of each execution time from bound 0 to the CEX bound. We were not able to perform the comparison with all the benchmarks because only a few of the Corbett benchmarks are available in the NuSMV format. However, we believe that the results shown in Table 8.2 gives an idea on how the missing benchmarks would perform with the tool NuSMV. To perform the NuSMV BMC operations we use the following commands:

set input_file < *filename* >;
set sat_solver zchaff;
go_bmc;
check_ltlspec_bmc -p "G (Live)" -k 50 -l X;
quit;

The command set input_file provides the input file related to the current benchmark. set sat_solver defines what SAT solver to use during the BMC operation. In our case, we use ZChaff. The command go_bmc configures the systems for BMC. The command check_ltlspec_bmc -p "G (Live)" -k 50 -l X checks whether the given property holds along the execution up to the given bound which is 50 in this particular example. In our case, the property is defined as an LTL formula "G (Live)" where live is a predicate that evaluates to true iff some action can be executed in the reached state. The last portion of the

command -l X says we are interested only in non-loop executions during the search for counterexample.

In Table 8.2 **NuSMV(BMC)** gives the results from NuSMV BMC. For all of the HARTs benchmarks, we did not find a counterexample using the NuSMV tool. We report only the results from HART(50) and HART(100) for which we limit the search respectively at bound 100 and 200.

8.2 Experimental Results Related to LTL

We took five problems from the literature, namely *gas station, bakery algorithm, readers-writers, sleeping barber,* and *leader election protocol.* As explained in [ACL05, GG08b], choosing what benchmarks to use is no easy feat, especially when it comes to asynchronous concurrent systems. Furthermore, the tasks become much more complex if we try to compare the method with other implementations. We will describe each of these benchmarks in the following subsections.

8.2.1 Gas Station

The gas station problem [PDH99, HL85] consists of simulating a self-service gas station with an operator, a number of pumps and customers. When a customer arrives, she first prepays the operator to be able to fill from a specific pump. The operator then activates the pump, assigned an ID to the customer, and pass it through a queue associated with the pump. Only after successful ID verification can a customer start using a pump. When the customer is done, the pump passes the charge back to the operator, who charges the customer and returns the change, if any. The whole service is performed in a first-in-first-out manner.

8.2.2 Bakery Algorithm

The bakery algorithm [BGR95, Bul00, BB06, BGP99] deals with the mutual exclusion problem for n processes $P_1, \ldots, P_n$ that want to access a shared critical section of a code. Each process Pi has a shared register readable by the other processes but writable only by P_i. A natural number n_i is associated to each register. Initially, all n_i $(1 \leqslant i \leqslant n)$ are set to zero. When a process P_i wants to access the critical section, it gives its n_i a value greater than all n_j $(j \neq i)$. Then P_i keeps reading the registers of the other process P_j and wait until $n_j = 0$ or $n_j > n_i$ or $n_j = n_i \wedge j > i$. Then P_i enters the critical section. Once it leaves the critical section, P_i resets the value of n_i to zero.

8.2.3 Readers-writers Problem

The readers-writers problem [Bul00, CHP71] is a typical problem in concurrency. It concerns the situation in which many processes must access a shared memory at the same

time. Multiple readers can read the memory together, but when a writer is writing no other readers or writers are allowed to share access to the memory.

8.2.4 Sleeping Barber

The sleeping barber problem [Bul00, BB06] is a synchronization problem between multiple processes waiting to access a single resource. It is analogous to a barber shop with one barber, a barber chair, and a number of chairs in a waiting room. The barber works when there are customers, but sleeps in his chair when there are none. When a customer arrives, he either awakens the barber and gets a haircut, or if the barber is busy, he sits in one of the vacant chairs in the waiting room. If there is no vacant chair, the newly arrived customer simply leaves. After cutting a customer's hair, the barber serves the next one in the waiting room, or goes back to sleep if the room is empty.

8.2.5 Leader Election Protocol

The leader election protocol [LLEL02, ERKCK91, KKM90, KKK95, SK91, Sin92] is necessary for the election of a leader in a distributed system formed by several autonomous computing nodes working on a joint task and can communicate with each other. When a failure appears, all the nodes should be able to adapt themselves to the new condition so that they may continue on their joint task. The reorganization is managed by a special node called the *leader* which is elected by all the active nodes as a first step in any reorganization procedure. After the election, the leader starts reorganizing the system and normal operation resumes.

There are many ways for electing a leader. As an example let us describe the one introduced in [ERKCK91]. The system consists of a unidirectional ring of n processors $P_1, \ldots, P_n$. The processors are numbered in counter-clockwise order, while communication is carried out clockwise. The processors need not to be awake at the same time, but no processor is allowed to wake after receiving a message from an awakened processor. When a processor P_i wakes up, it sends a message with its own ID to its neighbor P_j. If it is a participating processor, P_j checks the ID of the received message. If $j > i$ it forward the message to the next processor, but if $j < i$ the message is killed at node j. If P_j is not a participating processor, it just passes along the message to the next node.

Any message sent should eventually meet all other processors in the ring before it comes back to its sender. A message that returns back to its sender causes the sender to become elected leader.

8.2.6 Properties and Instances

The properties we checked for these benchmarks are listed below.

- Gas station: "If customer 2 prepays while customer 1 is using the pump then the operator will activate the pump for customer 2 next"

Table 8.3: LTL checking results

Models	**Bound**	**Time**	**CEX**
$4g$	120	4.45	No
$2b$	120	3.1	No
$4b$	120	4.9	No
$2rw$	1	0.01	Yes
$4rw$	1	0.01	Yes
$8l$	36	4.15	Yes
$10s$	120	6.22	No

LTL: $\mathsf{G}((fillCustomer1 \wedge prepayCustomer2) \rightarrow (\neg activate1 \,\mathsf{U}\, (activate2 \vee \mathsf{G}\neg activate1))$

- Bakery: "Only one process can enter the critical section"

  LTL: $\mathsf{G}\neg(criticalSection1 \wedge criticalSection2 \wedge \ldots)$

- Readers-writers: "At any time, there must be at least one writer writing and at least one reader reading."

  LTL: $\mathsf{G}((w1 \vee w2 \vee \ldots) \wedge (r1 \vee r2 \vee \ldots))$

- Leader election: "If process 1 is the elected processor then it will be the only one that enter into the elected state"

  LTL: $\mathsf{G}(leaderP1 \rightarrow \neg(leaderP2 \vee leaderP3 \vee \ldots))$

- Sleeping Barber: "The barber cut one person at a time"

  LTL: $\mathsf{G}\neg(customer1 \wedge customer2 \wedge \ldots)$

For the gas station problem we used an instance with four customers, one pump and one queue ($g4$). For the bakery algorithm we used a version with two processors ($2b$) and another with four processors ($4b$). We did the same with readers-writers ($2rw$ and $4rw$). We used a ring with eight nodes for the leader election ($8l$), and finally a model with ten customers for the sleeping barber ($10s$).

As before, the experiments were conducted on a machine with 2.4 Ghz Intel Pentium processor, 1 GB memory, and Fedora 10 operating system. The following points explain the columns shown in Table 8.3.

- **Models**: the model instances as explained above,

- **Bound**: the deepest bound used for the tests,

- **Time**: running time, in seconds, for the whole BMC computation starting from bound 1 including the solver time.

Table 8.4: Comparison of leap and standard BMC

Model	|P|	|T|	Max L.	Standard BMC	Leap BMC	
					Log.	Inter.
KEY(2)	94	92	36	836.95	6.49	13.52
KEY(3)	129	133	36	2458.89	5.64	17.36
KEY(4)	164	174	36	4231.53	9.96	26.71
DEMUX(16)	352	258	>80	153.58	16.89	20.86
REG(4)	77	96	>50	490.72	29.33	11.93
REG(8)	153	184	>20	342.61	57.45	100.32
SDL-ARQ	163	96	>44	5313.13	1446.14	1423.41
FIR-4-A1M1	1394	1131	>80	8043.52	1526.8	690.7
FIR-4-A2M2	1444	1158	>80	9478.42	2067.2	972.18
IIR-A1M1	576	476	>80	1513.35	188.71	188.09
IIR-A2M2	544	437	>80	1534.71	277.99	221.1
LF2	315	267	>90	11131.23	4037.65	3618.57

- **CEX**: a "yes" or "no" column that indicates whether a counterexample has been found or not at the current bound.

For each benchmark, we incremented the bound little by little. We limited the search at bound 120 for this experiments. For the properties we checked we found counterexamples only for the readers-writers and leader election problems.

8.3 Experiments Related to Leap

As usual, the experiments were conducted on a machine with 2.4 Ghz Intel Pentium processor, 1 GB memory, and Linux Red hat 9 operating system. As mentioned in Remark 5.3.1 we conducted the leap experiments using Petri net models. The *Keys* benchmarks are taken from the Petri net versions of Corbett's case studies [Cor96]. *Demux*, *Regs*, and *sdl-arq* are converted from STG benchmarks, while $firs$, $iirs$, and lf belong to the STG benchmark family used in [YOM04]. We performed deadlock detection using the deadlock condition described in [Hel01]. For the standard BMC the method used is similar to a standard BFS execution but adapted to work for Petri nets. The following points explain the columns shown in Table 8.4.

- **Model**: the name of the model.

- **|P|**: number of places in the model.

- **|T|**: number of transitions in the model.

- **Max L.**: maximal length of a BMC session. A number preceded by ">" indicates that no counterexample has been found up to this bound.

- **Standard BMC**: total CPU time, in seconds, for the whole BMC computation for standard BMC, starting from bound 1 until the maximal length, including the SAT-solver's time. We call the C++ function `getrusage()` to obtain these times.
- **Leap BMC**: the same as in **Standard BMC** but for the leap approach. Here the bound starts from $fixLeap$.
- **Log.**: results obtained when using the logarithmic jumping method.
- **Inter.**: results obtained when using the interpolation/extrapolation method.

Our goal is to show the efficiency of the big leap approach compared with standard BMC. Table 8.4, shows the results we obtained from the comparison. We report the running time obtained using the two jumping methods. As you can see, in general, the interpolation method runs slower than the logarithmic one. Still, both methods outperform the standard BMC approach.

8.4 Example of a Leap Execution

Below is a detailed explanation of a big leap execution using Algorithm 5.2. It consists of the result obtained in Table 8.4, column 6, last line, with the $lf2$ benchmark.

We used the following parameters:

- Model: $lf2$
- BMC method: big leap
- Jumping method: logarithmic
- Fix leap = 4
- $minC$ = 1 sec.
- $midC$ = 60 sec.
- $maxC$ = 150 sec.
- $limC$ = 1 hr.
- Maximum iteration for $minC$ = 9
- Maximum BMC Length = 90

We obtained the following results. (Explanation is given after the whole results.)

- leap = 4

  Solving time: 0

  No counterexample found with bound 4

- leap = 4

  Solving time: 0.01

  No counterexample found with bound 8

- leap = 4

  Solving time: 0.04

  No counterexample found with bound 12

- leap = 4

  Solving time: 0.06

  No counterexample found with bound 16

- leap = 4

  Solving time: 0.12

  No counterexample found with bound 20

- leap = 4

  Solving time: 0.17

  No counterexample found with bound 24

- leap = 4

  Solving time: 0.41

  No counterexample found with bound 28

- leap = 4

  Solving time: 0.44

  No counterexample found with bound 32

- leap = 4

  Solving time: 0.68

  No counterexample found with bound 36

- leap = 20

  Solving time: 3.04

  No counterexample found with bound 56

- leap = 24

  Solving time: 140

  No counterexample found with bound 80

- leap = 6

  Solving time: 291.6

  No counterexample found with bound 86

- leap = 4

  Solving time: 3601.08

  No counterexample found with bound 90

- Total CPU time: 4037.65

The execution starts with $leap = 4$ and continues this way until bound 36. During this period, we have $solvingTime < minC$. At this bound, the algorithm computes a new leap value because the maximum iteration for $minC$ is reached (see explanation in Section 5.4). The leap becomes $leap = 4 \times log_2(36) \simeq 20$, which corresponds to the formula used for the second condition in the algorithm. At bound 56, the algorithm still uses this formula because the solving time is still less than $midC$, so we have $leap = 4 \times log_2(56) \simeq 24$. At bound 80, we reach the situation where $midC \leqslant solvingTime < maxC$, then the algorithm computes another new leap value, which is $leap = log_2(80) \simeq 6$. $maxC$ is surpassed at bound 86, and the leap becomes 4 again. The algorithm stops at bound 90 because the maximum BMC Length is reached.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

In this dissertation, the research goal has been to develop efficient verification techniques for asynchronous concurrent systems. We focused on BMC using synchronized products of transition systems (TSs) as a model. Our idea is to replace the standard interleaving technique with techniques based on the breadth-first search (BFS) execution. While interleaving techniques allow us only to fire one event at a time, with breadth-first search we can fire multiple events together, which is really helpful when dealing with concurrent systems because it reduces the number of paths to be explored during the operation.

We have shown that combining BFS with chaining traversal can improve the BMC performance. The main objective in chaining is to reduce the number of the execution steps by firing the enabled events following a predefined order. The efficacy of chaining depends to a large extent on the chosen event order. It is, therefore, important to use the right event order during a chaining operation.

To increase the speed of a BMC operation, we have proposed a method based on *leap*. That is, we do not call the SAT solver at every single bound but at intervals. The number of bounds to be skipped is determined by the leap values. Normally, skipping bounds suggest that some counterexamples could be missed if they appeared in some of those bounds that have been skipped. To cope with this problem, we introduced a method that allows us to catch all errors if they exist all along the execution. We also introduced two jumping heuristics that can be used for selecting the bounds to be skipped. Experience have shown that the leap approach greatly improve the speed of the operation.

All the methods above are built for checking reachability properties. We have also proposed a method for verifying other properties written with the logic LTL. The method is based on the so-called automata-theoretic approach in which the LTL specification is first translated into Büchi automata before creating a Boolean formula. The advantage of using an automata as an intermediate step is that it is simpler to generate a Boolean formula from the automata and to combine it with the one generated form the TS unrolling.

We have implemented all these methods in a BMC toolset that can perform deadlock detection, and also can check LTL specifications. We conducted some experiments using the tool and compare our results with the state-of-the-art NuSMV tool. The methods show promise in the experimental results conducted using the tool.

9.2 Future Work

The following points summarizes possible future research directions related to this work.

Path compression For the methods devoted to reachability properties, combining them with path compression techniques such as the one in [KLY02] may bring some improvements. This idea could be interesting, because it will diminish the path to be searched, thus making the BMC formula even smaller.

Event ordering To improve the chaining method, the focus should be on the event ordering since a good event order is key to the success of the operation. In this work, we have used only one event ordering technique which works better for some kinds of systems. However, it is crucial to use a method guaranteed to work with any type of system.

Reduction techniques For the leap approach, integrating some reduction techniques, such as the ones in [Ber85, PRCB94, ES01], prior to the actual BMC computation could be interesting. This is useful especially when dealing with large systems since this reduces the size of the model, thus reduces the number of bounds to be explored, and decreases even further the execution time during search for counterexamples.

Checking CTL formulas It is also interesting to improve the encoding so that it can be used not only for checking LTL formulas, but other types of temporal logic such as CTL for instance.

Bibliography

[ABE00] P.A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT solvers. In S. Graf and M. Schwartzbach, editors, *Proc. of the 6th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *LNCS*, pages 411–425, Berlin, Germany, March 2000. Springer-Verlag.

[ÁBKS05] E. Ábrahám, B. Becker, F. Klaedtke, and M. Steffen. Optimizing bounded model checking for linear hybrid systems. In R. Cousot, editor, *Proc. of the 6th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI 2005)*, volume 3385 of *Lecture Notes in Computer Science*, pages 396–412, Paris, France, January 17-19 2005. Springer-Verlag.

[ACA78] T. Agerwala and Y. Choed-Amphai. A synthesis rule for concurrent systems. In *Proc. of the 15th conf. on Design Automation*, pages 305–311, Las Vegas, Nevada, United States, June 1978. IEEE Press.

[ACKS02] G. Audemard, A. Cimatti, A. Kormilowicz, and R. Sebastiani. Bounded model checking for timed systems. In *Proc. of the 22nd Joint Int. Conf. on Formal Techniques for Networked and Distributed Systems (FORTE 2002)*, volume 22 of *LNCS*, pages 243–259, Houston, Texas, U.S., November 2002. Springer-Verlag.

[ACL05] D-A. Atiya, N. Catano, and G. Lüttgen. Towards a benchmark for model checkers of asynchronous concurrent systems. Technical report, University of Warwick, U.K., 2005.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[ADA+05] N. Amla, X. Du, A.Kuehlmann, R.P. Kurshan, and K.L. McMillan. An analysis of SAT-based model checking techniques in an industrial environment. In D. Borrione and W. J. Paul, editors, *Proc. of the 13th Int. Conf. on Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725 of *Lecture Notes in Computer Science*, pages 254–268, Saarbrücken, Germany, October 2005. Springer-Verlag.

[Ade97] A. Adelman. The mathematics of the pentium bug. *SIAM Review*, 39(March):54–67, 1997.

[AEF+05] R. Armoni, S. Egorov, R. Fraer, D. Korchemny, and M. Vardi. Efficient LTL compilation for SAT-based model checking. In *Proc. of the Int. Conf. on Computer-Aided Design (ICCAD'05)*, pages 877–884. IEEE Computer Society, November 2005.

[AH04] R Arora and M. S. Hsiao. Enhancing SAT-based bounded model checking using sequential logic implications. In *Proc. of the 17th Int. Conf. on VLSI Design*, pages 784–787, 2004.

[ÁHBS06] E. Ábrahám, M. Herbstritt, B. Becker, and Martin Steffen. Bounded model checking with parametric data structures. In *Proc. of the 4th Int. workshop on Bounded Model Checking (BMC'06)*, Seattle,Washington, U.S., August 15 2006. Elsevier Science B.V.

[AKMM03] N. Amla, R. Kurshan, K. McMillan, and R. Medel. Experimental analysis of different techniques for bounded model checking. In H. Garavel and J. Hatcliff, editors, *Proc. of the 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 34–48. Springer-Verlag, April 2003.

[Arn94] A. Arnold. *Finite Transition Systems*. Masson, Paris, 1994.

[AS97] M. Abramovici and D.G. Saab. Satisfiability on reconfigurable hardware. In *Proc. of the 7th Int. Workshop on Field-Programmable Logic and Applications*, volume 1304 of *Lecture Notes in Computer Science*, pages 448–456. Springer-Verlag, 1997.

[AS06] M. Awedh and F. Somenzi. Termination criteria for bounded model checking: extensions and comparison. *Electronic Notes in Theoretical Computer Science*, 144(1):51–66, 2006.

[BB04] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proc. of the 16th Int. Conf. on Computer Aided Verification (CAV'2004)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518, Boston, Massachusetts, U.S., July 2004. Springer-Verlag.

[BB06] C. Bartzis and T. Bultan. Efficient BDDs for bounded arithmetic constraints. *Int. Journal on Software Tools for Technology Transfer*, 8(1):26–36, 2006.

[BBC+05] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. The MathSAT 3 system. In Robert Nieuwenhuis, editor, *Proc. of the 20th Int. Conf. on Automated Deduction (CADE'05)*, volume 3632 of *Lecture Notes in Computer Science*, pages 315–321, Tallinn, Estonia, July 2005. Springer-Verlag.

[BBF+99] B. Bérard, M. Bioit, A. Finkel, F. Latoussinie, A. Petit, L. Petrucci, Ph. Schnoebelen, and P. Mckenie. *Systems and software verification.* Springer-Verlag, 1999.

[BC00] P. Bjesse and K. Claessen. SAT-based verification without state space traversal. In W.A. Hunt Jr. and S.D. Johnson, editors, *Proc. of. the 3rd Int. Conf. on Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 372–389. Springer-Verlag, November 2000.

[BC03] M. Benedetti and A. Cimatti. Bounded model checking for past LTL. In H. Garavel and J. Hatcliff, editors, *Proc. of the 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 18–33, Warsaw, Poland, April 2003. Springer-Verlag.

[BCC+03] A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in computer sciences*, 58, 2003.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of the Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, March 1999.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[BCRZ00] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC using symbolic model checking without BDDs. In *Proc. of the Int. Conf. on Formal Methods in Computer Aided Design*, volume 1633 of *Lecture Notes in Computer Science*, pages 60–71. Springer-Verlag, November 2000.

[BDL96] C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In M. Srivas and A. Camilleri, editors, *Proc. of the 1st Int. Conf. on Formal Methods in Computer Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201, Palo Alto, California, U.S., November 1996. Springer-Verlag.

[BdMS07] C. Barrett, L. de Moura, and A. Stump. Design and results of the 2nd annual satisfiability modulo theories competition (SMT-COMP 2006). *Formal Methods in System Design*, 31(3):221–239, 2007.

[BE77] J. Baer and C.S. Ellis. Model, design, and evaluation of a compiler for a parallel processing environment. *IEEE Transaction on Software Engineering*, 3(6):394–405, 1977.

[Ber85] G. Berthelot. Checking properties of nets using transformation. In *Advances in Petri Nets, covers the 6th European Workshop on Applications and Theory in Petri Nets*, volume 222 of *Lecture Notes in Computer Science*, pages 19–40, Espoo, Finland, June 1985. Springer-Verlag.

[BGP99] T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):747–789, 1999.

[BGR95] E. Börger, Y. Gurevich, and D. Rosenzweig. The bakery algorithm: yet another specification and verification. pages 231–243, 1995.

[BHJ+06] A. Biere, K. Heljanko, T. A. Junttila, T. Latvala, and V. Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2(5:5):1–64, 2006.

[BHSV+96] R.K. Brayton, G.D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G. Swamy, and T. Villa. VIS: a system for verification and synthesis. In R. Alur and T.A. Henzinger, editors, *Proc. of. the 8th Int. Conf. on Computer Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432, New Brunswick, NJ, U.S., July 1996. Springer-Verlag.

[BK86] C.L. Beck and B.H. Krogh. Models for simulation and discrete control of manufacturing systems. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*, pages 305–310, San Francisco, U.S., 1986. IEEE Computer Society Press.

[BK02] F. Bause and P. Kritzinger. *Stochastic Petri nets – An introduction to the theory (2nd edition)*. Vieweg Verlag, 2002.

[BK04] J. Baumgartner and A. Kuehlmann. Enhanced diameter bounding via structural transformation. In *Proc. of. the Conf. Design Automaton and Test in Europe (DATE'2004)*, pages 36–41, 2004.

[BKA02] J. Baumgartner, A. Kuehlmann, and J. Abraham. Property checking via structural analysis. In E. Brinksma and K. G. Larsen, editors, *Proc. of the 14th Int. Conf. on Computer Aided Verification, (CAV 2002)*, volume 2404 of *Lecture Notes in Computer Science*, pages 151–165, Copenhagen, Denmark., July 2002. Springer-Verlag.

[BLM01] P. Bjesse, T. Leonard, and A. Mokkedem. Finding bugs in Alpha microprocessor using satisfiability solvers. In *Proc. of the 13th Int. Conf. on Computer*

Aided Verification (CAV'2001), volume 2102 of *Lecture Notes in Computer Science*, pages 454–464. Springer-Verlag, 2001.

[BLP95] J. Bormann, J. Lohse, and M. Payer. Model checking in industrial hardware design. In *Proc. of the 32th Int. Conf. on Design Automation (DAC'95)*, pages 298–303, San Fransisco, California, U.S., June 1995.

[BM86] G. Bruno and G. Marchetto. Process-translatable Petri nets for the rapid prototyping of process control systems. *IEEE Transaction on Software Engineering*, 12(2):346–357, 1986.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulations. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[BT82] G. Berthelot and R. Terrat. Petri nets theory for the correctness of protocols. In C.A. Sunshine, editor, *Proc. of the IFIP WG6.1 2nd Int. Workshop on Protocol Specification, Testing and Verification (PSTV)*, pages 325–342, Idyllwild, CA, U.S., 1982. North-Holland.

[Bul00] T. Bultan. Bdd vs. constraint-based model checking: an experimental evaluation for asynchronous concurrent systems. In S. Graf and M. Schwartzbach, editors, *Proc. of the 6th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *LNCS*, pages 441–455, Berlin, Germany, March 2000. Springer-Verlag.

[BW03] F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In E. Giunchiglia and A. Tacchella, editors, *Proc. of the 6th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *Lecture Notes in Computer Science*, pages 341–355, Santa Margherita Ligure, Italy,, May 2003. Springer-Verlag.

[CBRZ01] E.M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal methods in system design*, 19(1):7–34, July 2001.

[CCGR99] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model verifier. In N. Hallbwachs and D. Peled, editors, *Proc. of. the 11th Int. Conf. on Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 495–499, Trento, Italy, July 1999. Springer-Verlag.

[CCK+02] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In M. Aagaard and J. W. O'Leary, editors, *Proc. of the 4th Int. Conf. on Formal Methods in Computer Aided Design (FMCAD'02)*, pages 33–51, Portland, Oregon, U.S., November 2002. Springer-Verlag.

[CCK03] P. Chauhan, E. Clarke, and D. Kroening. Using SAT based image computation for reachability analysis. Technical Report CMU-CS-03-151, Carnegie Mellon University, School of Computer Science, 2003.

[CDLP05] J-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In P. Godefroid, editor, *Proc. of the 12th Int. SPIN Workshop on Model Checking of Software*, volume 3639 of *LNCS*, pages 169–184, San Francisco, CA, U.S., August 22-24 2005. Springer-Verlag.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[CFF+01] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M.Y. Vardi. Benefits of bounded model checking at an industrial setting. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. of the 13th Int. Conf. on Computer Aided Verification, (CAV 2001)*, volume 2102 of *Lecture Notes in Computer Science*, pages 436–453, Paris, France, July 18–22 2001. Springer-Verlag.

[CG96] C.-A. Chen and S.K. Gupta. A satisfiability-based test generator for path delay faults in combinational circuits. In *Proc. of the 33rd annual conference on Design automation*, pages 209–214, Las Vegas, Nevada, U.S., 1996. ACM Press.

[CG05] B. Cook and G. Gonthier. Using stålmarck's algorithm to prove inequalities. In K. Lau and R. Banach, editors, *Proc. of 7th Int. Conf. on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *Lecture Notes in Computer Science*, pages 330–344, Manchester, UK, November 2005. Springer-Verlag.

[CGH94] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In David L. Dill, editor, *Proc. of the 6th Int. Conf. Computer Aided Verification (CAV'1994)*, volume 818 of *LNCS*, pages 415–427, Stanford, California, U.S., June 21-23 1994. Springer.

[CGJ95] E.M. Clarke, O. Grumberg, and S. Jha. Veryfying parameterized networks using abstraction and regular languages. In I. Lee and S.A. Smolka, editors, *Proc of the 6th Int. Conf. on Concurrency Theory (CONCUR '95)*, volume 962 of *Lecture Notes in Computer Science*, pages 395–407, Philadelphia, PA, U.S., August 21–24 1995. Springer-Verlag.

[CGL92] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *Proc. of the 19th ACM SIGPLAN-SIGACT symposium on Principles of*

programming languages (POPL), pages 342–354, Albuquerque, New Mexico, U.S., 1992. ACM Press.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking*. MIT Press, 1999.

[CHP71] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with "readersÂťÂť and "writersÂťÂť. *Communications of the ACM*, 14(10):667–668, 1971.

[CJEF96] E.M. Clarke, S. Jha, R. Enders, and T. Filkorn. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2):77–104, 1996.

[CK96] E.M. Clarke and R.P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[CKOS04] E. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. Completeness and complexity of bounded model mhecking. In *Proc. of the 5th Int. Conf. on Verification, Model Checking, and Abstract Interpretation, (VMCAI 2004)*, volume 2937 of *Lecture Notes in Computer Science*, pages 85–96, Venice, January 11–13 2004. Springer-Verlag.

[Cla02] E.M. Clarke. SAT-based counterexample guided abstraction refinement. In *Proc. of the 9th Int. SPIN Workshop on Model Checking of Software*, page 1, Grenoble, France, April 2002. Springer-Verlag.

[CLM89] E.M. Clarke, D.E. Long, and K.L. McMillan. A language for compositional specification and verification of finite state hardware controllers. In J. A. Darringer and F. J. Ramming, editors, *Proc. of the 9th Int. Symposium on Computer Hardware Description Languages and Their Applications*, pages 281–295, North Holland, 1989.

[CLRS09] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithm*. The MIT Press, third edition, 2009.

[CMMM95] A. Coombes, J. McDermid, J. Moffett, and P. Morris. Requirements analysis and safety: a case study (using GRASP). In G. Rabe, editor, *Proc. of the 14th Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP'95)*, Belgirate, Italy, October 1995. Springer-Verlag.

[CNQ03] G. Cabodi, S. Nocco, and S. Quer. Improving SAT-based bounded model checking by means of BDD-based approximate traversal. In *Proc. of. the Design Automaton and Test in Europe (DATE)*, pages 898–903, March 2003.

[Coo71] S.A. Cook. The complexity of theorem-proving procedures. In *Proc. of the 3rd annual ACM symposium on Theory of computing*, pages 151–158, Shaker Heights, Ohio, U.S., 1971.

[Cor96] J.C. Corbett. Evaluating deadlock detection methods for concurrent systems. *IEEE Transactions on Software Engineering*, 22(3):161–180, 1996.

[Cou99] J. Couvreur. On-the-fly verification of linear temporal logic. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Proc. of the World Congress on Formal Methods in the Development of Computing Systems FM'99*, volume 1708 of *LNCS*, pages 253–271, Toulouse, France, September 1999. Springer.

[CPRS02] A. Cimatti, M. Pistore, M. Roveri, and R. Sebastiani. Improving the encoding of LTL model checking into SAT. In *Proc. of the 3rd Int. Conf. on Verification, Model Checking and Abstract Interpretation VMCAI'2002*, volume 2294 of *Lecture Notes in Computer Science*, pages 196–207. Springer-Verlag, 2002.

[CVWY92] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[CW96] E.M. Clarke and J. M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[CYLR01] H. Choi, B. Yun, Y. Lee, and H. Roh. Model checking of S3C2400X industrial embedded SOC product. In *Proc. of the 38th Int. Conf. on Design Automation (DAC'01)*, pages 611–616, Las Vegas, Nevada, U.S., June 2001.

[Das06] P. Dasgupta. *A roadmap for formal property verification.* Springer, The Netherlands, 2006.

[DdM06a] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In T. Ball and R. Jones, editors, *Proc. of the 18th Int. Conf. Computer Aided Verification (CAV'2006)*, volume 4144 of *LNCS*, pages 81–94, Seattle, WA, U.S., August 2006. Springer.

[DdM06b] B. Dutertre and L. de Moura. Integrating simplex with DPLL(T). Technical Report SRI-CSL-06-01, SRI International, 2006.

[DE73] G. Dantzig and B. Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory*, 14(3):288–297, 1973.

[Des89] A. A. Desrochers. *Modeling and control of automated manufacturing systems.* IEEE Computer Society Press, 1989.

[DGG97] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2):253–291, 1997.

[DGV99] M. Daniel, F. Giunchiglia, and M Vardi. Improved automata generation for linear temporal logic. In N. Halbwachs and D. Peled, editors, *Proc. of the 11th Int. Conf. on Computer Aided Verification (CAV'1999)*, volume 1633 of *LNCS*, pages 249–260, Trento, Italy, 1999. Springer.

[Dia87] M. Diaz. Petri net based models in the specification and verification of protocols. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course*, volume 255 of *Lecture Notes in Computer Science*, pages 135–170. Springer-Verlag, 1987.

[DIM] DIMACS. Challenge SAT benchmarks. Available at ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/.

[DKMW93] S. Devadas, K. Keutzer, S. Malik, and A. Wang. Computation of floating mode delay in combinational circuits: practice and implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(12):1924–1936, December 1993.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *JACM*, 5(7):394–97, 1962.

[DLP04] A. Duret-Lutz and D. Poitrenaud. Spot: An extensible model checking library using transition-based generalized Büchi automata. In Doug DeGroot, Peter G. Harrison, Harry A. G. Wijshoff, and Zary Segall, editors, *Proc. of the 12th Int. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2004)*, pages 76–83, Vollendam, The Netherlands, October 2004. IEEE Computer Society.

[dlT92] T. Boy de la Tour. An optimality result for clause form translation. *Journal of Symbolic Computation*, 14(4):283 – 301, 1992.

[dMRS02] L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In A. Voronkov, editor, *Proc. of the 18th Int. Conf. on Automated Deduction (CADE'02)*, volume 2392 of *Lecture Notes in Computer Science*, pages 438–455, Copenhagen, Denmark, July 27–30 2002. Springer-Verlag.

[dMRS03] L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. In W.A. Hunt Jr. and F. Somenzi, editors, *Proc. of the 15th Int. Conf. on Computer Aided Verification, (CAV 2003)*,

volume 2725 of *Lecture Notes in Computer Science*, pages 14–26, Boulder, Colorado, U.S., July 2003. Springer-Verlag.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *JACM*, 7(3):201–215, 1960.

[DS83] D. Dubois and K.E. Stecke. Using Petri nets to represent production processes. In *Proc. of the 22nd IEEE Conference on Decision and Control*, volume 3, pages 1062–1067. IEEE, 1983.

[dSU96] H.V. der Schoot and H. Ural. An improvement on partial order model checking with ample sets. Technical Report TR-96-11, University of Ottawa, Canada, 1996.

[Een03] N. Een. Temporal induction by incremental SAT solving. In *Proc. of the Int. Workshop on Bounded Model Checking (BMC'2003)*, July 2003.

[EH86] E.A. Emerson and J.Y. Halpern. "Sometimes" and "Not Never" revisited: on branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[EH00] K. Etessami and G.J. Holzmann. Optimizing Büchi automata. In C. Palamidessi, editor, *Proc. of the 11th Int. Conf. on Concurrency Theory (CONCUR'2000)*, volume 1877 of *LNCS*, pages 153–167, University Park, PA, U.S., August 2000. Springer.

[Eme90] E.A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. 1990.

[ERKCK91] M. El-Ruby, J. Kenevan, R. Carlson, and K. Khalil. A linear algorithm for election in ring configuration networks. In *Proc. of the 24th Annual Hawaii International Conference on System Sciences (HICSS 1991)*, volume i, pages 117–123, Kauai, HI, U.S., January 1991.

[ES01] J. Esparza and C. Schröter. Net reductions for LTL model-checking. In T. Margaria and T. F. Melham, editors, *Proc. of the 11th IFIP WG 10.5 Advanced Research Working Conf. on Correct Hardware Design and Verification Methods (CHARME' 2001)*, volume 2144 of *Lecture Notes in Computer Science*, pages 310–324, Livingston, Scotland, UK, September 4–7 2001.

[FDK98] F. Fallah, S. Devadas, and K. Keutzer. Functional vector generation for hdl models using linear programming and 3-satisfiability. In *Proc.of the 35th annual conference on Design automation*, pages 528–533, San Francisco, California, U.S., 1998. ACM Press.

[FM07] Z. Fu and S. Malik. Extracting logic circuit structure from conjunctive normal form descriptions. In *In Proc. of the Int. Conf. on VLSI design (VLSI Design 2007)*, pages 37–42, Bangalore, India, 2007. IEEE Computer Society.

[FORS01] J-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: integrated canonizer and solver. In *Proc. of the 13th Int. Conf. on Computer Aided Verification (CAV'2001)*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249. Springer-Verlag, 2001.

[FSW02] A. Frisch, D. Sheridan, and T. Walsh. A fixpoint based encoding for bounded model checking. In M. Aagaard and J. W. O'Leary, editors, *Proc. of the 4th Int. Conf. on Formal Methods in Computer Aided Design (FMCAD'02)*, pages 238–255, Portland, Oregon, U.S., November 2002. Springer-Verlag.

[GB96] B. Grahlmann and E. Best. PEP - more than a petri net tool. In T. Margaria and B. Steffen, editors, *Proc. of the 2nd Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems(TACAS'96)*, volume 1055 of *LNCS*, pages 397–401, Passau, Germany, March 1996. Springer-Verlag.

[Gep95] L. Geppert. Biology 101 on the internet: dissecting the pentim bug. *IEEE Spectrum*, (February):16–17, 1995.

[GG06] M.K. Ganai and A. Gupta. Accelerating high-level bounded model checking. In *Proc. of the Int. Conf. on Computer-Aided Design (ICCAD 2006)*, pages 794–801, San Jose, CA, U.S., November 2006. ACM.

[GG08a] M.K. Ganai and A. Gupta. Completeness in SMT-based BMC for software programs. In *Proc. of the Design, Automation and Test in Europe Conference and Exhibition (DATE'08)*, pages 831–836, Munich Germany, March 2008. IEEE.

[GG08b] M.K. Ganai and A. Gupta. Efficient modeling of concurrent systems in BMC. In K. Havelund, R. Majumdar, and J. Palsberg, editors, *Proc. of the 15th Int. SPIN Workshop on Model Checking of Software*, volume 5156 of *LNCS*, pages 114–133, Los Angeles, CA, U.S., August 10-12 2008. Springer-Verlag.

[GGA04] M. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based unbounded symbolic model checking using circuit cofactoring. In *Proc. of the Int. Conf. on Computer-Aided Design (ICCAD 2004)*, pages 510–517, San Jose, California, U.S., November 2004. IEEE Computer Society / ACM.

[GGA05] M. Ganai, A. Gupta, and P. Ashar. Beyond safety: customized SAT-based model checking. In *Proc. of the 42nd Design Automation Conference (DAC'2005)*, pages 738–743, San Diego, CA, U.S., June 2005. ACM.

[GGW+03] A. Gupta, M. Ganai, C. Wang, Z. Yang, and P. Ashar. Learning from BDDs in SAT-based bounded model checking. In *Proc. of. the 40th Design Automation Conference (DAC'03)*, pages 824–829, Anaheim, California, U.S., June 2003.

[GH93] P. Godefroid and G. J. Holzmann. On the verification of temporal properties. In A.. Danthine, G. Leduc, and P. Wolper, editors, *Proc. of the IFIP WG6.1 13th Int. Symposium on Protocol Specification, Testing and Verification (PSTV)*, pages 109–124, Liège, Belgium, 1993. North-Holland.

[GHN+04] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): fast decision procedures. In *Proc. of the 16th Int. Conf. on Computer Aided Verification (CAV'2004)*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188, Boston, Massachusetts, U.S., July 2004. Springer-Verlag.

[GL94] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[GN02] E. Goldberg and Y. Novikov. Berkmin: a fast and robust SAT-solver. In *Proc. of the Design, Automation and Test in Europe Conference and Exhibition (DATE'02)*, pages 142–149, Paris, France, March 2002. IEEE Computer Society.

[GO01] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. of the 13th Int. Conf. on Computer Aided Verification (CAV'2001)*, volume 2102 of *LNCS*, pages 53–65. Springer, 2001.

[God90] P. Godefroid. Using partial orders to improve automatic verification methods. In E.M. Clarke and R.P. Kurshan, editors, *Proc. of the 2th Int. Conf. on Computer Aided Verification (CAV'90)*, volume 531 of *Lecture Notes in Computer Science*, pages 176–85, New Brunswick, NJ, U.S., June 1990. Springer-Verlag.

[God96] P. Godefroid. *Partial-Order methods for the verification of concurrent systems - An approach to the state-explosion problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[GPD97] J. Gu, P. M. Pardalos, and D. Du, editors. *Satisfiability problem: theory and applications (DIMACS) series in discrete mathematics and theoretical computer science)*, volume 35. American Mathematical Society, October 1997.

[GPFW97] J. Gu, P. W. Purdom, J. Franco, and B. W. Wah. Algorithms for the satisfiability (SAT) problem: A survey. In D. Du, J. Gu, and P. M. Pardalos, editors, *Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–150. American Mathematical Society, 1997.

[GPSS80] D.M. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proc. of 7th ACM symp. Principles of Programming Languages (POPL'80)*, pages 163–173, Las Vegas, Nevada, U.S., 1980.

[GPVW95] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. of the 15th IFIP WG6.1 Int. Symp. on Protocol Specification, Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[GS90] S. Graf and B. Steffen. Compositional minimization of finite state systems. In E.M. Clarke and R.P. Kurshan, editors, *Proc. of the 2th Int. Conf. on Computer Aided Verification (CAV'90)*, volume 531 of *Lecture Notes in Computer Science*, pages 186–196, New Brunswick, NJ, U.S., June 1990. Springer-Verlag.

[GS99] E. Giunchiglia and R. Sebastiani. Applying the Davis-Putnam procedure to non-clausal formulas. In *AI*IA 99: Advances in Artificial Intelligence: 6th Congress of the Italian Association for Artificial Intelligence*, volume 1792 of *Lecture Notes in Computer Science*, pages 84–93, Bologna, Italy,, September 1999. Springer-Verlag.

[GS05] A. Gupta and O. Strichman. Abstraction refinement for bounded model checking. In *Proc. of the 17th Int. Conf. on Computer Aided Verification (CAV'2005)*, volume 3576 of *Lecture Notes in Computer Science*, pages 112–124. Springer-Verlag, 2005.

[GW94] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305–326, 1994.

[GYA+01] A. Gupta, Z. Yang, P. Ashar, L. Zhang, and S. Malik. Partition-based decision heuristics for image computation using SAT and BDDs. In *Proc. of. the Int. Conf. on Computer Aided Design (ICCAD'01)*, pages 289–292, San Jose, California, U.S., 2001. IEEE Press.

[GYAG00] A. Gupta, Z. Yang, P. Ashar, and A. Gupta. SAT-based image computation with application in reachability analysis. In W.A. Hunt Jr. and S.D. Johnson, editors, *Proc. of the 3rd Int. Conf. on Formal Methods in Computer Aided Design (FMCAD'00)*, pages 354–371, Austin, Texas, U.S., November 2000. Springer-Verlag.

[Haa02] P. Haas. *Stochastic Petri nets: modelling, stability, simulation.* Springer-Verlag, 2002.

[Har00] D. Harel. *Computers ltd. What they really can't do.* Oxford University Press, 2000.

[Har03] B. Harvey. *Europe's space programme: to ariane and beyond.* Springer-Verlag, Praxis Publishing, 2003.

[Hel01] K. Heljanko. Bounded reachability checking with process semantics. In K.G. Larsen, editor, *Proc. of the 12th Int. Conf. on Concurrency Theory (CONCUR'2001)*, volume 2154 of *LNCS*, pages 218–232, Aalborg, Denmark, 2001. Springer-Verlag.

[HJL05] K. Heljanko, T. Junttila, and T. Latvala. Incremental and complete bounded model checking. In *Proc. of the 17th Int. Conf. on Computer Aided Verification (CAV'2005)*, volume 3576 of *Lecture Notes in Computer Science*, pages 98–111. Springer-Verlag, 2005.

[HL85] D. Heimbold and D. Luckham. Debugging ada tasking programs. *IEEE Software*, 2(2):47–57, 1985.

[HL05] D.M. Harland and R.D. Lorenz. *Space systems failures: disasters and rescues of satellites, rockets and space probes.* Praxis, 2005.

[HMU01] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition).* Addison Wesley, 2001.

[HN01a] K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. In *Proc. of the 6th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR'2001)*, pages 200–212, Vienna, Austria, September 2001.

[HN01b] K. Heljanko and I. Niemlä. Answer set programming and bounded model checking. In *Proc. of the AAAI Spring 2001 Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 90–96, Stanford, U.S., March 2001. AAAI Press.

[Hol88] G.J. Holzmann. An improved protocol reachability analysis technique. *Software, Practice & Experience*, 18(2):137–161, 1988.

[Hol97] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[HP95] G. J. Holzmann and D. Peled. An improvement in formal verification. In *Proc. of the 7th IFIP WG6.1 Int. Conf. on Formal Description Techniques (FORTE'94)*, pages 197–211, Berne, Switzerland, 1995. Chapman & Hall.

[Hro04] J. Hromkovič. *Theoretical computer science. Introduction to automata, computability, complexity, algorithmics, randomization, communication, and cryptography.* Springer-Verlag, 2004.

[IPC03] M. Iyer, G. Parthasarathy, and K. Cheng. SATORI - a fast sequential SAT engine for circuits. In *Proc. of the Int. Conf. on Computer-Aided Design (ICCAD 2003)*, pages 320–325, San Jose, California, U.S., November 2003. IEEE Computer Society / ACM.

[JAS04] H. Jin, M. Awedh, and F. Somenzi. CirCUs: A satisfiability solver geared towards bounded model checking. In *Proc. of the 16th Int. Conf. on Computer Aided Verification (CAV'2004)*, volume 3114 of *Lecture Notes in Computer Science*, pages 519–522, Boston, Massachusetts, U.S., July 2004. Springer-Verlag.

[Jen97] K. Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use. Volume 3.* Springer-Verlag, 1997.

[JHN03] T. Jussila, K. Heljanko, and I. Niemelä. BMC via on-the-fly determinization. In *Proc. of the 1st Int. Workshop on Bounded Model Checking (BMC'2003)*, pages 197–206, Boulder, Colorado, U.S., July 2003.

[JN00] T.A. Juntilla and I. Niemelä. Towards an efficient tableau method for Boolean circuit satisfiability checking. In *Proc. of the 1st Int. Conf. on Computational Logic (CL 2000)*, volume 1861 of *LNCS*, pages 553–567, London, UK, 2000. Springer-Verlag.

[JR91] K. Jensen and G. Rozenberg, editors. *High-level Petri nets. Theory and application.* Springer-Verlag, 1991.

[JS04] H. Jin and F. Somenzi. CirCUs: A hybrid satisfiability solver. In *Proc. of the 7th Int. Conf. on Theory and Applications of Satisfiability Testing, SAT 2004*, Vancouver, Canada, May 10–13 2004.

[JS05] H. Jin and F. Somenzi. An incremental algorithm to check satisfiability for bounded model checking. *Electronic Notes in Computer Science*, 119(2):51–65, 2005.

[Jus04] T. Jussila. BMC via dynamic atomicity analysis. In *Proc. of the 4th Int. Conf. on Application of Concurrency to System Design (ACSD'04)*, pages 197–206, Hamilton, Ontario, Canada, June 2004.

[Jus05] T. Jussila. *On bounded model checking of asynchronous systems.* PhD thesis, Helsinki University of Technology, Laboratory for Theoretical Computer Science, 2005.

[JV04] J. and A. Valmari. Tarjan's algorithm makes on-the-fly LTL verification more efficient. In K. Jensen and A. Podelski, editors, *Proc. of the 10th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 205–219, Barcelona, Spain, March 29 - April 2 2004. Springer-Verlag.

[Kel95] D. Kelley. *Automata and Formal Languages. An Introduction.* Prentice Hall, 1995.

[Kid98] P.A. Kidwell. Stalking the elusive computer bug. *IEEE Annals of the history of computing*, 20(4):5–9, October-December 1998.

[KKK95] T.w. Kim, E.H. Kim, and J.K. Kim. A leader election algorithm in a distributed computing system. In *Proc. of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS 1995)*, page 481. IEEE Computer Society, 1995.

[KKM90] E. Korach, S. Kutten, and S. Moran. A modular technique for the design of efficient distributed leader finding algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):84–101, January 1990.

[KL82] W.E. Kluge and K. Lautenbach. The orderly resolution of memory access conflicts among competing channel processes. *IEEE Transactions on Computers*, 31(3):194–207, 1982.

[KLY02] R.P. Kurshan, V. Levin, and H. Yenigün. Compressing transitions for model checking. In *Proc. of the 14th Int. Conf. on Computer Aided Verification (CAV'2002)*, volume 2404 of *Lecture Notes in Computer Science*, pages 569–581. Springer-Verlag, 2002.

[KM95] R.P. Kurshan and K.L. McMillan. A structural induction theorem for processes. *Information and Computation*, 117(1):1–11, 1995.

[KM99] S. Katz and H. Miller. Saving space by fully exploiting invisible transitions. *Formal methods in system design*, 14(3):311–332, May 1999.

[Kop89] S. Koppelberg. *General theory of Boolean aAlgebras*, volume 1 of *Handbook of Boolean Algebras.* 1989.

[Koz97] D.C. Kozen. *Automata and Computability.* Springer-Verlag, 1997.

[Kro99] T. Kropf. *Introduction to formal hardware verification.* Springer, 1999.

[Kro05] D Kroening. Computing over-approximations with bounded model checking. In *Proc. of the 3th Int. workshop on Bounded Model Checking (BMC'05)*, volume 144 of *Electronic Notes in Theoretical Computer Science*, pages 79–92, Boulder, Colorado, U.S., July 2005. Elsevier Science B.V.

[KS03] D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In L.D. Zuck, P.C. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Proc. of the 4th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI 2003)*, volume 2575 of *Lecture Notes in Computer Science*, pages 298–309, New York, NY, U.S., 2003. Springer-Verlag.

[Kue04] A. Kuehlmann. Dynamic transition relation simplification for bounded property checking. In *Proc. of the Int. Conf. on Computer-Aided Design (ICCAD 2004)*, pages 50–57, San Jose, California, U.S., November 2004. IEEE Computer Society / ACM.

[Lad95] P. Ladkin. Analysis of a technical description of the airbus A320 braking system. *High Integrity Systems*, 1(4):331–349, 1995.

[Lam80] L. Lamport. "Sometimes" is sometimes "Not Never" – On the temporal logic of programs. In *Proc. of the 7th ACM Symposium on Principles of Programming Languages (POPL'20)*, pages 174–185, 1980.

[Lam83] L. Lamport. What good is temporal logic? In *Proc of the 2th IFIP Congress*, pages 657–668, 1983.

[LBHJ04] T. Latvala, A. Biere, K. Heljanko, and T. Junttila. Simple bounded ltl model checking. In A. J. Hu and A. K. Martin, editors, *Proc. of 5th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD 2004)*, volume 3312 of *Lecture Notes in Computer Science*, pages 186–200, Austin, Texas, U.S., November 2004. Springer-Verlag.

[LHS04] B. Li, M.S. Hsiao, and S. Sheng. A novel SAT all-solutions solver for efficient preimage computation. In *Proc. of. the Conf. Design Automaton and Test in Europe (DATE'2004)*, pages 272–277, 2004.

[LLEL02] A. Lluch-Lafuente, S. Edelkamp, and S. Leue. Partial order reduction in directed model checking. In *Proc. of the 9th Int. SPIN Workshop on Model Checking of Software*, volume 2318, page 1, Grenoble, France, April 2002. Springer-Verlag.

[LMS04] I. Lynce and J.P. Marques-Silva. On computing minimum unsatisfiable cores. In *Proc. of the 7th Int. Conf. on Theory and Applications of Satisfiability Testing, SAT 2004*, Vancouver, Canada, May 10–13 2004.

[MA03] K. McMillan and N. Amla. Automatic abstraction without counterexamples. In H. Garavel and J. Hatcliff, editors, *Proc. of the 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 2–17, Warsaw, Poland, April 2003. Springer-Verlag.

[MBC+98] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. Modelling with generalized stochastic Petri nets. *SIGMETRICS Perform. Eval. Rev.*, 26(2):2, 1998.

[McM93] K.L. McMillan. *Symbolic model checking.* Kluwer Academic Publisher, 1993.

[McM03] K. McMillan. Interpolation and SAT-based model checking. In W. A. Hunt Jr. and F. Somenzi, editors, *Proc. of the 15th Int. Conf. on Computer Aided Verification, (CAV 2003)*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13, Boulder, Colorado, U.S., July 2003. Springer-Verlag.

[McM05] K.L. McMillan. Applications of craig interpolants in model checking. In N. Halbwachs and L.D. Zuck, editors, *Proc. of the 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 1–12, Edinburgh, UK, April 2005. Springer-Verlag.

[MMZ+01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of the 38th Int. Conf. on Design Automation (DAC'01)*, pages 530–535, Las Vegas, Nevada, United States, June 2001.

[MR90] S.P. Masticola and B.G. Ryder. Static infinite wait anomaly detection in polynomial time. In *Proc. of the Int. Conf. on Parallel Processing*, volume 2, pages 78–87, Urbana-Champaign, IL, U.S., August 1990. Pennsylvania State University Press.

[MT99] V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K. Apt, W. Marek, M. Truszczyński, and D. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999.

[Mur89] T. Murata. Petri nets: properties and applications. In *Proceedings of the IEEE*, volume 77, pages 541–580, April 1989.

[NASR04] G. Nam, F. Aloul, K.A. Sakallah, and R.A. Rutenbar. A comparative study of two boolean formulations of FPGA detailed routing constraints. *IEEE Transactions on Computers*, 53(6):688–696, June 2004.

[NG02] R. Nalumasu and G. Gopalakrishnan. An efficient partial order reduction algorithm with an alternative proviso implementation. *Formal Methods in System Design*, 20(3):231–247, 2002.

[NO05] R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *Proc. of the 17th Int. Conf. on Computer Aided Verification (CAV'2005)*, volume 3576 of *Lecture Notes in Computer Science*, pages 321–334. Springer-Verlag, 2005.

[Nor05] A.L. Norberg. *Computers and commerce: a Study of technology and management at Eckert-Mauchly Computer Company, Engineering Research Associates, and Remington Rand, 1946-1957.* The MIT Press, 2005.

[NRW98] A. Nonnengart, G. Rock, and C. Weidenbach. On generating small clause normal forms. In *Proc. of the 15th Int. Conf. on Automated Deduction: Automated Deduction*, volume 1421 of *Lecture Notes in Computer Science*, pages 397–411. Springer-Verlag, 1998.

[OGMS02] R. Ostrowski, E. Grégoire, B. Mazure, and L. Sais. Recovering and exploiting structural knowledge from CNF formulas. In *In Proc. of the 8th Int. Conf. on Principles and Practice of Constraint Programming (CP 2002)*, Lecture Notes in Computer Science, pages 185–199, Ithaca, NY, U.S., 2002. Springer-Verlag.

[OTK04] S. Ogata, T. Tsuchiya, and T. Kikuno. SAT-based verification of safe Petri nets. In *Proc. of the 2nd Int. Conf. on Automated Technology for Verification and Analysis (ATVA 2004)*, volume 3299 of *Lecture Notes in Computer Science*, pages 79–92, Taipei, Taiwan, ROC, October 31–November 3 2004. Springer-Verlag.

[PBG05] M.R. Prasad, A. Biere, and A. Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer*, January 2005.

[PDH99] C.S. Pasareanu, M.B. Dwyer, and M. Huth. Assume-guarantee model checking of software: a comparative case study. In *Proc. of the 5th and 6th Int. SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *LNCS*, pages 168–183. Springer-Verlag, 1999.

[Pel93] D. Peled. All from one, one for all: on model checking using representatives. In C. Courcoubetis, editor, *Proc. of the 5th Int. Conf. on Computer Aided Verification (CAV'93)*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423, Elounda, Greece, June 28–July 1 1993. Springer-Verlag.

[Pen03] M. Pena. *Reletive timing-based formal verification of complex timed systems.* PhD thesis, Technical University of Catalonia, 2003.

[Pet62] C.A. Petri. *Kommunikation mit automaten.* Ph.D. dissertaion. University of Bonn, 1962.

[Pet66] C.A. Petri. Communication with automata. Technical Report RADC-TR-65-377, Griffiss Air Force Base, New York, 1966.

[Pet81] J.L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.

[PG86] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293 – 304, 1986.

[PICW04] G. Parthasarathy, M.K. Iyer, K.-T. Cheng, and L.-C. Wang. Safety property verification using sequential SAT and bounded model checking. *IEEE Transactions on Design & Test of Computers*, 21(2):132–143, 2004.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. of the 8th Annual Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57, Providence, Rhode Island, U.S., October 31–November 2 1977. IEEE.

[PRCB94] E. Pastor, O. Roig, J. Cortadella, and R.M. Badia. Petri net analysis using boolean manipulation. In *Proc. of the 5th Int. Conf. on Application and Theory of Petri Nets*, volume 815 of *Lecture Notes in Computer Science*, pages 416–435, Zaragoza, Spain, June 20–24 1994. Springer-Verlag.

[PVK01] D. Peled, A. Valmari, and I. Kokkarinen. Relaxed visibility enhances partial order reduction. *Formal methods in system design*, 19(3):275–289, November 2001.

[PWZ02a] W. Penczek, B. Woźna, and A. Zbrzezny. Bounded model checking for the universal fragment of CTL. *Fundamenta Informaticae*, 50:1–22, 2002.

[PWZ02b] W. Penczek, B. Wozna, and A. Zbrzezny. Towards bounded model checking for the universal fragment of TCTL. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Proc. of the 7th Int. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 2002)*, volume 2469 of *LNCS*, pages 265–288, Oldenburg, Germany, September 2002. Springer-Verlag.

[QS82] J-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. of 5th Colloquium on International Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer, 1982.

[Rei67] C. H. Reinsch. Smoothing by spline functions. *Numerische Mathematik*, 10(3):177–183, 1967.

[Rei85] W. Reisig. Petri nets: an introduction. In *EATCS Monographs on Theoretical Computer Science*, volume 4, Berlin, 1985. Springer-Verlag.

[RMB04] Jarrod A. Roy, Igor L. Markov, and Valeria Bertacco. Restoring circuit structure from SAT instances. In *Proc. of the 6th Int. Workshop on Logic and Synthesis (IWLS 2004)*, Temecula, California, U.S., June 2004.

[Ros05] P.E. Ross. The exterminators. *IEEE Spectrum*, 42(9–INT):30–35, September 2005.

[Sa85] A. Sistla and E. Clarke and. The complexity of propositional linear temporal logics. *Journal of the Association for Computing Machinery (JACM)*, 32(3):733–749, July 1985.

[SAT] SATLIB. Benchmarks problems. Available at http://www.cs.ubc.ca/∼hoos/SATLIB/benchm.html.

[SB00] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In E.A. Emerson and A.P. Sistla, editors, *Proc. of the 12th Int. Conf. Computer Aided Verification (CAV'2000)*, volume 1855 of *LNCS*, pages 248–263, Chicago, IL, U.S., July 2000. Springer.

[SB05] V. Schuppan and A. Biere. Shortest vounterexamples for symbolic model checking of LTL with past. In N. Halbwachs and L.D. Zuck, editors, *Proc. of the 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 493–509, Edinburgh, U.K., April 4-8 2005. Springer-Verlag.

[SBD02] A. Stump, C. Barrett, and D. Dill. CVC: a cooperating validity checker. In E. Brinksma and K. G. Larsen, editors, *Proc. of the 14th Int. Conf. on Computer Aided Verification, (CAV 2002)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504, Copenhagen, Denmark., July 2002. Springer-Verlag.

[SBSV96] P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9):1167–1176, September 1996.

[Sch99] K. Schneider. Yet another look at LTL model checking. In L. Pierre and T. Kropf, editors, *Proc. of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *LNCS*, pages 321–325, Bad Herrenalb, Germany, September 27-29 1999. Springer.

[SdBF04] I. Skliarova and A. de Brito Ferrari. Reconfigurable hardware sat solvers: a survey of systems. *IEEE Transactions on Computers*, 53(11):1449–1461, November 2004.

[SG90] G. Shurek and O. Grumberg. The modular framework of computer-aided verification. In E.M. Clarke and R.P. Kurshan, editors, *Proc. of the 2th Int. Conf. on Computer Aided Verification (CAV'90)*, volume 531 of *Lecture Notes in Computer Science*, pages 186–196, New Brunswick, NJ, U.S., June 1990. Springer-Verlag.

[She04] D. Sheridan. The optimality of a fast CNF conversion and its use with SAT. In *Proc. of the 7th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT'05)*, Lecture Notes in Computer Science, Vancouver, Canada, 2004. Springer-Verlag.

[She05] D. Sheridan. Bounded model checking with SNF, alternating automata, and büchi automata. *Electr. Notes Theor. Comput. Sci.*, 119(2):83–101, 2005.

[Shn02] Ph. Shnoebelen. The complexity of temporal logic model checking. *Advances in Modal Logic*, 4:1–44, 2002.

[Sht00] O. Shtrichman. Tuning SAT checkers for bounded model checking. In *Proc. of the 12th Int. Conf. on Computer Aided Verification (CAV'2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 480–494. Springer-Verlag, 2000.

[Sin92] S. Singh. Expected connectivity and leader election in unreliable networks. *Information Processing Letters*, 42(5):283–285, 1992.

[Sip97] M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing company, 1997.

[SK91] S. Singh and J.F. Kurose. Electing leaders based upon performance: the delay model. In *Proc. of the 10th Int. Conf. on Distributed Computing Systems (ICDCS 1991)*, pages 464–471, Arlington, Texas, U.S., May 1991. IEEE Computer Society.

[Sor02] M. Sorea. Bounded model checking for timed automata. Technical Report SRI-CSL-02-03, SRI International, 2002.

[SP02] M. Solé and E. Pastor. Traversal techniques for concurrent systems. In *Proc. of Formal Methods in Computer-Aided Design (FMCAD 2002)*, volume 2517 of *LNCS*, pages 220–238, Portland, Oregon, U.S., November 2002. Springer-Verlag.

[SP05] S. Subbarayan and D. K. Pradhan. NiVER: non-increasing variable elimination resolution for preprocessing SAT instances. In E. Giunchiglia and A. Tacchella, editors, *Proc. of the 7th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT 2004), Revised Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*, pages 276–291, Santa Margherita Ligure, Italy,, May 2005. Springer-Verlag.

[SS99] J.P. Silva and K. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[SS00] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in System Design*, 16(1):23–58, 2000.

[SS05] H. Sheini and K. Sakallah. A scalable method for solving satisfiability of integer linear arithmetic logic. In F. Bacchus and T. Walsh, editors, *Proc. of the 8th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 241–256, St. Andrews, Scotland, 2005. Springer-Verlag.

[SS06] H.M Sheini and K.A. Sakallah. Pueblo: A hybrid pseudo-boolean SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2, 2006.

[SSA91] V. Sivaramakrishnan, S.C. Seth, and P. Agrawal. Parallel test pattern generation using Boolean satisfiability. In *Proc. of 4th CSI/IEEE Int. Symp. on VLSI Design*, pages 69–74, New Delhi, India, 1991.

[SSS00] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In W.A. Hunt Jr. and S.D. Johnson, editors, *Proc. of the 3rd Int. Conf. on Formal Methods in Computer-Aided Design, (FMCAD 2000)*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125, Austin, Texas, U.S., November 2000. Springer-Verlag.

[Ste93] B. Sterling. *The hacker crackdown: law and disorder on the electronic frontier*. Bantam, 1993.

[Str01] O. Strichman. Pruning techniques for the SAT-based bounded model checking problem. In T. Margaria and T. Melham, editors, *Proc. of the 11th Int. Conf. on Correct Hardware Design and Verification Methods (CHARME 2001)*, volume 2144 of *Lecture Notes in Computer Science*, pages 58–70, Livingston, Scotland, UK, September 2001. Springer-Verlag.

[Str04] O. Strichman. Accelerating bounded model checking of safety properties. *Formal Methods in System Design*, 24(1):5–24, 2004.

[SYSN01] T. Suyama, M. Yokoo, H. Sawada, and A. Nagoya. Solving satisfiability problems using reconfigurable computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(1):109–116, February 2001.

[SZ03] O. Shacham and E. Zarpas. Tuning the VSIDS decision heuristic for bounded model checking. In *Proc. of the 4th Int. Workshop on Microprocessor Test and Verification: Commomn Challenges and Solutions*, pages 75– 79, 2003.

[Tau03] H. Tauriainen. On translating linear temporal logic into alternating and nondeterministic automata. Research Report A83, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 2003.

[TBW04] C. Thiffault, F. Bacchus, and T. Wash. Solving non-clausal formulas with DPLL search. In *Proc. of the 7th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT'04)*, Vancouver, BC, Canada, May 2004. Springer-Verlag.

[TGH97] P. Tafertshofer, A. Ganz, and M. Henftling. A SAT-based implication engine for efficient ATPG, equivalencechecking, and optimization of netlists. In *Proc of Int. Conf. on Computer-Aided Design (ICCAD'97)*, pages 648–655, San Jose, CA, U.S., 1997.

[TH00] H. Tauriainen and K. Heljanko. Testing SPIN's LTL formula conversion into Büchi automata with randomly generated input. In K. Havelund, J. Penix, and W. Visser, editors, *Proc. of the 7th Int. SPIN Workshop*, volume 1885 of *LNCS*, pages 54–72, Stanford, CA, U.S., September 2000. Springer-Verlag.

[Tse83] G.S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 466–483. Springer-Verlag, 1983.

[Val90] A. Valmari. A stubborn attack on state explosion. In E.M. Clarke and R. P. Kurshan, editors, *Proc. of the 2th Int. Conf. on Computer Aided Verification (CAV'90)*, volume 531 of *Lecture Notes in Computer Science*, pages 156–65, New Brunswick, NJ, U.S., June 1990. Springer-Verlag.

[Var95] M. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller and G. Birtwistle, editors, *Proc. of 8th Banff Higher Order Workshop on Logics for Concurrency - Structure versus Automata*, volume 1043 of *LNCS*, pages 238–266. Springer, August 27–September 3 1995.

[Var06] M. Y. Vardi. Automata-theoretic techniques for temporal reasoning. In *Handbook of Modal Logic*, pages 971–989. Elsevier, 2006.

[VB01] M.N. Velev and R.E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. In *Proc. of Design Automation Conference*, pages 226–231, 2001.

[Vel] M. Velev. SAT benchmarks collection. Available at http://www.ece.cmu.edu/∼mvelev/sat_benchmarks.html.

[Vel04a] M. N. Velev. Exploiting signal unobservability for efficient translation to CNF in formal verification of microprocessors. In *Proc. of. the Conf. Design Automaton and Test in Europe (DATE'2004)*, pages 266–271, 2004.

[Vel04b] M.N. Velev. Efficient translation of boolean formulas to CNF in formal verification of microprocessors. In *Proc. of the 2004 conf. on Asia South*

Pacific design automation: electronic design and solution fair, pages 310–315, Yokohama, Japan,, 2004. IEEE Press.

[VHBP00] W. Visser, K. Havelund, G.P. Brat, and S. Park. Model checking programs. In Y. Ledru, P. Alexander, and P. Flener, editors, *Proc. of the 15th IEEE Int. Conf. on Automated Software Enginneering (ASE'2000)*, pages 3–12, Grenoble, France, September 2000. IEEE Computer Society.

[VW86] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. of the first Symp. on Logic in Computer Science (LICS)*, pages 332–344, Cambridge, Massachusetts, U.S., June 1986. IEEE Computer Society.

[Wan98] J. Wang. *Timed Petri nets, theory and application*. Kluwer Academic Publishers, 1998.

[WBCG00] P.F. Williams, A. Biere, E.M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Proc. of the 12th Int. Conf. on Computer Aided Verification (CAV'2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 124–138. Springer-Verlag, 2000.

[WIGG05] C. Wang, F. Ivancic, M. Ganai, and A. Gupta. Deciding separation logic formulae by SAT and incremental negative cycle elimination. In G. Sutcliffe and A. Voronkov, editors, *Proc. of the 12th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2005)*, volume 3835 of *Lecture Notes in Computer Science*, pages 322–336, Montego Bay, Jamaica, December 2005. Springer-Verlag.

[WJHS04] C. Wang, H. Jin, G. D. Hachtel, and F. Somenzi. Refining the SAT decision ordering for bounded model checking. In *Proc of th 41st annual Conf. on Design Automation*, pages 535–538. ACM Press, 2004.

[WKS01] J. Whittemore, J. Kim, and K. Sakallah. Satire: A new incremental satisfiability engine. In *Proc. of the 38th Int. Conf. on Design Automation (DAC'01)*, pages 542–545, Las Vegas, Nevada, U.S., June 2001. ACM.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In J. Sifakis, editor, *Proc. of the Int. Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 68–80, Grenoble, France, June 12-14 1989. Springer-Verlag.

[WLP04] B. Woźna, A. Lomuscio, and W. Penczek. Bounded model checking for reachability testing in timed Petri nets. In *Proc. of the Int. Workshop on Concurrency Specification and Programming (CS&P'04)*, volume 170(1), pages 124–135, Potsdam, September 2004.

[Wol02] P. Wolper. Constructing automata from temporal logic formulas: a tutorial. pages 261–277, 2002.

[WVS83] P. Wolper, M.Y. Vardi, and A.P. Sistla. Reasoning about infinite computation paths. In *Proc. of the 24th Annual Symp. on Foundations of Computer Science (FOCS)*, pages 185–194, Tucson, Arizona, U.S., November 1983. IEEE.

[WW96] B. Willems and P. Wolper. Partial-order methods for model checking: from linear time to branching time. In E.M. Clarke, editor, *Proc. of the 11th Annual IEEE Symp. on Logic in Computer Science (LICS) 1996*, pages 294–303, New Brunswick, NJ, U.S., July 1996. IEEE Computer Society Press.

[WZP03] B. Woźna, A. Zbrzezny, and W. Penczek. Checking reachability properties for timed automata via SAT. *Fundamenta Informaticae*, 55(2):223–241, 2003.

[YOM04] T. Yoneda, H. Onda, and C. J. Myers. Synthesis of speed independent circuits based on decomposition. In *10th Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2004)*, pages 135–145, Crete, Greece, April 19–23 2004.

[Yor94] K. Yorav. *Exploiting Syntactic Structure for Automatic Verification.* PhD thesis, Israel Institute of Technology, 1994.

[Zar04] E. Zarpas. Simple yet efficient improvements of SAT based bounded model checking. In A. J. Hu and A. K. Martin, editors, *Proc. of 5th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD 2004)*, volume 3312 of *Lecture Notes in Computer Science*, pages 174–185, Austin, Texas, U.S., November 2004. Springer-Verlag.

[Zar05] E. Zarpas. Benchmarking SAT solvers for bounded model checking. In F. Bacchus and T. Walsh, editors, *Proc. of the 8th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 340–354, St. Andrews, Scotland, 2005. Springer-Verlag.

[ZDR92] M. Zhou, F. DiCesare, and D. Rudolph. Design and implementation of a Petri net based supervisor for a flexible manufacturing system. *IFAC Journal Automatica*, 28(6):1199–1208, 1992.

[Zha97] H. Zhang. SATO: an efficient propositional prover. In *Proc. of the 14th Int. Conf. on Automated Deduction (CADE'97)*, volume 1249 of *LNCS*, pages 272–275. Springer-Verlag, July 1997.

[ZM03] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In *Proc. of the 6th Int. Conf. on Theory and Applications*

of Satisfiability Testing, SAT 2003, S. Margherita Ligure, Portofino, Italy, May 5–8 2003.

[ZPH04] L. Zhang, M.R. Prasad, and M.S. Hsiao. Incremental deductive & inductive reasoning for SAT-based bounded model checking. In *Proc. of the IEEE/ACM Int. Conf. on Computer Aided Design (ICCAD'2004)*, pages 502–509, November 2004.

[ZPHS05] L. Zhang, M.R. Prasad, M.S. Hsiao, and T. Sidle. Dynamic abstraction using SAT-based BMC. In *Proc. of the 36th ACM/IEEE conf. on Design Automation DAC*, pages 754–757, San Diego, California, U.S., 2005. ACM Press.

[Zub80] W. Zuberek. Timed Petri nets and preliminary performance evaluation. In *Proc. of the 7th annual symposium on Computer Architecture*, pages 88–96, La Baule, U.S., 1980. ACM Press.

Index

T

U