UNIVERSITY OF CALIFORNIA
RIVERSIDE

Bridging the Gap Between Logic and Probabilistic Model Checking

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Yang Zhao

August 2013

Dissertation Committee:

Professor Gianfranco Ciardo, Chairperson
Professor Zizhong Chen
Professor Christian Shelton
Professor Neal Young

The Dissertation of Yang Zhao is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

First and foremost, I would like to express my deepest appreciation to Dr. Ciardo, the best advisor I could imagine. His knowledge, patience, and advising make my thesis possible. I will never forget the nice experiences discussing with him and the spark of ideas we ignited together during the past five years.

My research also receives supports from Dr. Kristin Y. Rozier at NASA Ames Research Center and Dr. Radu Siminiceanu at National Institute of Aerospace. My summer interns in NASA and NIA greatly extend my prospective on formal methods.

I would also thank my friends here in UCR: Min Wan, Malcolm Mumme, Xiaoqing Jin, Diego Villasenor, Yousra Lembachar, Xin He, Lei Wang, Zhe Wu, and Li Yan. Without their help, my life in UCR would have been much more difficult.

Thank Amy and Madie from Department of CSE, you are the best and most helpful staffs.

Finally, the best gift I received while at UCR is the love from my wife, Xiaoquan (well, technically, Dr. Zhou :)).

To my little Yang Yang, and our homeland, China.

ABSTRACT OF THE DISSERTATION

Bridging the Gap Between Logic and Probabilistic Model Checking

by

Yang Zhao

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, August 2013
Professor Gianfranco Ciardo, Chairperson

Fast development of hardware/software design requires more versatile and powerful verification methods to help engineers understand, verify, and debug their systems. The scope of system verification now is not limited to finding functional errors at the logic level, but also includes analyzing and predicting the bottlenecks in performance and dependability. Model checking, which was originally proposed to verify discrete-state systems, has been further extended to the verification of probabilistic systems. Many techniques from other communities, such as Markov chain analysis, are involved in the model checking process. Traditional and new verification techniques must be integrated into a platform that can handle both the logic and probabilistic aspects of a given model.

Symbolic model checking using decision diagrams has achieved great success in verifying many practical software and hardware systems, and is still the primary approach to logic verification. Recent research shows that decision diagrams can be successfully employed in probabilistic model checking. This thesis is devoted to future improving the capability of decision diagram techniques in model checking. Specifically, this thesis explores

the application of a family of decision diagrams, including multi-way decision diagrams and edge-valued multi-way decision diagrams, to several topics in both classic and probabilistic model checking.

My thesis consists of two parts of work: In the first part, I extend the existing saturation algorithm, which was originally proposed for state-space generation, to CTL model checking, strongly-connected component enumeration, and shortest witness generation for various properties. The second part of the thesis focuses on probabilistic model checking using decision diagrams. I propose a new and more efficient algorithm to carry out the Gauss-Seidel iterative method, which is a key step in probabilistic model checking. This technique can be applied to both steady-state solution of continuous-time Markov chains and CSL model checking. Then, I introduce a new bounding semantics of CSL to tackle truncation errors in numerical analysis and correctly evaluate nested CSL formulas.

The proposed techniques have been integrated into the SMART tool developed in our lab. Experimental results demonstrate that SMART is a promising platform to handle both logic and quantitative verification for many practical systems.

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Model checking: logic and probabilistic . . . 2
1.2 Challenges . . . 3
1.3 Contributions . . . 5
1.4 Structure . . . 7

2 Preliminaries **8**
2.1 Decision diagrams . . . 8
2.1.1 Multi-way decision diagrams . . . 11
2.1.2 Edge-valued Multi-way decision diagrams . . . 14
2.2 Discrete-state systems . . . 18
2.2.1 Symbolic state-space generation and saturation . . . 23
2.2.2 Computation Tree Logic . . . 29
2.2.3 CTL model checking . . . 31
2.3 CTMCs . . . 32
2.3.1 Steady-state solution . . . 34
2.3.2 Transient analysis . . . 36
2.3.3 CSL . . . 38
2.3.4 CSL model checking . . . 40

I Logic model checking **44**

3 Constrained saturation and CTL model checking **45**
3.1 Constrained saturation for the EU operator . . . 47
3.2 Transitive closure and the EG operator . . . 52
3.3 Experimental results . . . 54
3.3.1 Results for the EU computation . . . 54

3.3.2 Results for the EG computation . . . 56
3.4 Summary . . . 59

4 SCC enumeration **60**
4.1 Previous work . . . 62
4.2 Using saturation in the XB algorithm . . . 66
4.3 Computing the TC with saturation . . . 67
4.4 Fair cycles . . . 72
4.5 Experimental results . . . 74
4.6 Summary . . . 77

5 Shortest EG witness generation **79**
5.1 Background and related work . . . 80
5.2 Overview . . . 82
5.2.1 Computing TCD . . . 84
5.3 Discussion . . . 88
5.3.1 Shortest witness generation beyond EG . . . 89
5.3.2 Shortest fair witness . . . 91
5.4 Experimental results . . . 92
5.5 Summary . . . 96

II Probabilistic model checking **99**

6 EVMDD-based two-phase Gauss-Seidel iteration **100**
6.1 Previous work . . . 102
6.2 Symbolic iterative methods . . . 105
6.2.1 Transition rate matrix and probability vector storage . . . 105
6.2.2 Symbolic Jacobi iterations . . . 108
6.2.3 Symbolic Gauss-Seidel iterations . . . 110
6.3 Speeding up the iteration . . . 114
6.4 Complexity and discussion . . . 118
6.5 Experimental results . . . 122
6.6 Summary . . . 125

7 A bounding semantics for CSL **127**
7.1 Bounding the probability in CSL model checking . . . 129
7.1.1 Time-bounded until . . . 129
7.1.2 Unbounded until . . . 131
7.1.3 Point-interval and general interval until . . . 133
7.2 Semantics for CSL formulas with bounds . . . 133
7.3 Case studies . . . 136
7.4 Summary . . . 143

8 Implementation: SMART tool **144**
8.1 Logic model checking 145
8.2 Probabilistic model checking 146

9 Conclusion **148**
9.1 Summary 148
9.2 Future work 150

Bibliography **152**

List of Figures

2.1 Quasi-reduced MDD (left), quasi-reduced sparse MDD (middle), and fully-reduced MDD (right) . . . 13
2.2 Union operation on fully-reduced BDDs. . . . 15
2.3 An EV$^+$MDD. . . . 18
2.4 Pseudocode for *Normalize*. . . . 18
2.5 Pseudocode for *Min* and *Sum*. . . . 19
2.6 An example of firing . . . 22
2.7 (a) The discrete-state model for a 2-bit counter; (b) MDDs encoding the next-state function. . . . 27
2.8 The relational product operator. . . . 28
2.9 State-space generation using breadth-first search. . . . 28
2.10 The (forward) saturation algorithm. . . . 29
2.11 Traditional CTL model checking algorithms. . . . 33
2.12 Encoding a CTMC with an EV$^+$MDD and an EV*MDD. . . . 39
2.13 Backward computation of the probability vector for $\phi \mathsf{U}^{[0,t]} \psi$. . . . 42

3.1 Saturation-based EU model checking algorithms. . . . 47
3.2 Saturation using a constrained next-state function (*EUsatConsNSF*). . . . 50
3.3 Constrained saturation (*EUconsSat*). . . . 51
3.4 EG computation based on BFS v.s. transitive closure. . . . 58

4.1 Lockstep for SCC computation. . . . 64
4.2 XB algorithm for terminal SCC computation. . . . 65
4.3 Improved XB algorithm to compute SCCs using saturation. . . . 68
4.4 Improved XB algorithm to compute terminal SCCs using saturation. . . . 68
4.5 Building the TC using saturation (continued on Figure 4.6). . . . 70
4.6 Building the TC using saturation. . . . 71
4.7 Computing recurrent states using TC. . . . 72
4.8 Computing fair cycles using TC. . . . 74

5.1 BFS-based algorithm to generate a witness for $\mathsf{EG}\phi$. . . . 83
5.2 Computing *TCD*. . . . 87

5.3 Building TCD_ϕ. 88
5.4 A witness for $\mathsf{EF}(r \wedge \mathsf{EG}\neg s)$. 92
5.5 Runtime and witness length of the BFS-based algorithm on *slot*, *cqn*, and *arbiter*. 97

6.1 EV*MDD (a) and Kronecker (b) representations of transition rate. 103
6.2 Jacobi iteration. 106
6.3 Gauss-Seidel iteration. 107
6.4 Two-phase iteration. 110
6.5 Top level Gauss Seidel iteration for unbounded U operator. 113
6.6 Algorithm to preprocess and build the cache array. 115
6.7 Structure of the cache. 117
6.8 Runtime (sec) per iteration: *Act-Cl*$_2$*-GSD* vs. our plain algorithm. 123
6.9 Experimental results comparing PRISM with the proposed algorithms. . . 124

7.1 Modified Fox-Glynn algorithm for lower bounds on Poisson probabilities. . . 130
7.2 An example of computing $\boldsymbol{\nu}(\mathcal{S}_1)$. 132
7.3 Model checking algorithm for U^I generating bounds. 134
7.4 Algorithm for the evaluation of nested CSL formulas. 137
7.5 Algorithm for refining the evaluation of nested CSL formulas. 138
7.6 Embedded system bounded until: size of lower and upper bound sets satisfying the formula (left) and probability bounds for the initial state (right). . 139
7.7 Embedded system unbounded until: size of lower and upper bound sets satisfying the formula (left) and probability bounds for the initial state (right). 140
7.8 High-level state transition of the AAC system. 141

8.1 The infrastructure of SMART. 144

List of Tables

2.1 Algorithms for model checking CSL [4]. 43

3.1 Results for the EU computation. 57
3.2 Results for the EG computation. 58

4.1 Results for SCC and terminal SCC computation. 78

5.1 Results for EG witness generation. 96

7.1 Results for Formula 7.4 using top-down refinement. 142
7.2 Results for Formula 7.5. 142

9.1 Functionality of decision diagrams. 149
9.2 The application of (improved) Saturation. 149

Chapter 1

Introduction

Verification is the process of checking whether an implementation, either using software or hardware, conforms to the given specification. When the system under verification is simple, a high confidence in the correctness of the system can be achieved by testing. As the complexity of software and hardware design grows, efforts on system testing and debugging keep increasing, and verification becomes the bottleneck in current software and hardware design flow.

Formal verification has long been a dream in both academia and industry. The objective of formal verification is to automatically prove or disprove the correctness of a system compared to the intended behavior. Unlike traditional system testing, formal verification does not require to generate test cases, and hopefully, once the system passes the verification, its correctness is ensured. There are several approaches in formal verification: static analysis, theorem proving, model checking, and so on. This thesis only focuses on model checking.

1.1 Model checking: logic and probabilistic

Model checking is an important and successful state-of-the-art approach in formal verification. Given a model of a real system, $\mathcal{M}$, and the specification Φ, described using a formal language, model checking is the process to exhaustively and automatically check whether $\mathcal{M} \models \Phi$ and identify the "counterexamples", which are system executions that violate Φ. Thus, research in model checking can be categorized according to the type of model $\mathcal{M}$ and the formal language used to express Φ. This thesis focuses on two types of model checking: logic and probabilistic model checking.

Logic model checking was originally proposed by Clarke and Emerson [27], and by Queille and Sifakis [67]. $\mathcal{M}$ is described as a (finite) discrete-state system, and Φ is described using temporal logic, like Computational Tree Logic (CTL) or Linear Tree Logic (LTL). A (logic) model checker identifies whether there is a possible path in $\mathcal{M}$ that proves or violates Φ. One simple yet typical model checking problem is the *reachability* of a system $\mathcal{M}$: given a set of "unsafe" states, check whether unsafe states are reachable from initial states in $\mathcal{M}$. If so, a path from an initial state to an unsafe state is returned as a counterexample. Otherwise, it is guaranteed that no such a path exists in $\mathcal{M}$, i.e., $\mathcal{M}$ will *never* reach unsafe states. In a nutshell, model checking focuses on the *existence* of some unexpected behaviors.

Inspired by dependability and performance analysis, probabilistic model checking addresses the probabilities of certain system behaviors. For example, hardware failures are inevitable in large computer systems, and we need to study the probability of these failures and their effects on service availability. This thesis considers probabilistic model checking problems in which $\mathcal{M}$ is described as a continuous-time Markov chain (CTMC),

and Φ is described using Continuous Stochastic Logic (CSL). Extended from logic model checking, probabilistic model checking combines techniques from logic verification, Markov chain analysis, and timed automata. Integrating logic and probabilistic model checking techniques into a new platform becomes a natural approach to handle both the logic and quantitative aspects of given models.

1.2 Challenges

The key step in logic model checking is to exhaustively search among all possible system executions. Thus, the complexity of a naïve approach is at least $O(E)$, where E is the number of edges in the state transition graph generated from the model. As the number of states in the model grows, the naïve approach becomes infeasible. "State-space explosion" describes the combinatorial blow up of the state space, which is the primary difficulties encountered by model checking techniques. The formidable memory consumption needed when storing and manipulating a potentially huge number of states is the main obstacle.

Two approaches have been extensively studied: explicit model checking and symbolic model checking, to overcome the state-space explosion and build practical model checkers. Explicit model checking is based on the explicit storage of states, and often employs state compression [42] and partial-order reduction [64] to reduce both the memory and runtime consumption in the traversal. Symbolic model checking [29, 56, 57] employs advanced algebra and represents states implicitly. For example, SAT-based model checking [2, 6, 79] converts a model checking problem to a boolean satisfiability (SAT) problem and solves it using a SAT solver, while BDD-based model checking encodes and manipu-

lates sets of states using Binary Decision Diagrams (BDDs). This thesis focuses on symbolic model checking utilizing decision diagrams.

Symbolic model checking has achieved great success by adopting Binary Decision Diagrams (BDDs) [8]. BDDs have been widely utilized as a time and space efficient data structure to perform operations such as union, intersection, and relational product over sets of states. Unlike explicit approaches, BDDs do not store or manipulate the states one by one, but represent sets of states using acyclic graphs whose size does not necessarily grow with the size of the set. BDDs are able to handle a large number of states using relatively small amount of memory. Hence, BDD-based techniques provide a feasible approach to verify complex systems that otherwise could not even be stored.

One difficulty that BDD-based model checking encounters is to verify asynchronous systems. Previous successes achieved by BDD-based model checking were mainly in hardware verification, like VLSI design or bus protocols, where system transitions are synchronized by a global clock. In asynchronous systems, such synchronization does not exist, and several events may occur in any arbitrary order. The performance of traditional BDD-based model checking deteriorates due to the complex transition relations of asynchronous systems. As many-core CPUs and multi-thread programs prevails, software debugging and testing become much more difficult, and the verification of asynchronous systems draws more attention in both academia and industry. This difficulty motivates us to find more efficient algorithms for asynchronous systems.

Turning to probabilistic model checking, the problem of state-space explosion becomes even worse because we need to handle both the large state space and also the proba-

bilities of different states. Decision diagrams were originally designed for discrete structures rather than structures containing continuous values, thus several extensions have been proposed to enable BDDs to encode real-value functions. However, as Chapter 6 shows, these extensions do not scale in some cases. Moreover, probabilistic model checking involves sophisticated numerical algorithms, like Gauss-Seidel iterations, and no existing implementation is available for decision diagrams. To fully exploit the advantages of decision diagrams in probabilistic model checking, new algorithms for numerical analysis based on decision diagrams must be developed.

Finally, practical system design requires versatile tools, which can not only find error at logic level, but also predict bottleneck in performance and dependability. It requires to integrate logic and probabilistic model checking into one single platform.

1.3 Contributions

This thesis focuses on improving the efficiency of both logic and probabilistic model checking. The work in this thesis is based on a family of decision diagram data structures, including Multi-way Decision Diagrams (MDDs) and Edge-valued MDDs (EVMDDs). This thesis shows that decision diagrams can be efficiently applied to both logic and probabilistic model checking for asynchronous systems, and thus have great potential as the key data structure in a unified model checking platform handling both logic and probabilistic properties.

The contributions of this thesis mainly lie along three aspects:

1. I extend the application of the existing saturation algorithm from state-space generate to CTL model checking and proposed *constrained saturation*. For model checking CTL operator EU, the new algorithm using constrained saturation is able to achieve a clear speedup and reduce the peek memory consumption over a mainstream BDD-based model checker. Furthermore, I employ constrained saturation to compute the transitive closure, which has long been considered infeasible for non-trivial models. The transitive closure can be employed to check the CTL EG operator, and to enumerate strongly-connected components in transition systems. Although, in general, building the transitive closure is still expensive, my experiments show that in some cases where traditional algorithms do not work, the transitive closure based algorithm provides a good substitutes.

2. I proposed a two-phase algorithm for stationary solution of continuous-time Markov chains. This algorithm carries out Gauss-Seidel iterations based on the decision diagram encoding the transition rate matrix. Experimental results show that the two-phase algorithm achieves comparable, if not better, speed with a mainstream tool and reduces the memory consumption at the same time. This thesis also introduces a bounding semantics for CSL formulas to tackle truncation errors in the numerical analysis. Then, we study the bounding semantics for a subset of nested CSL formulas.

3. All the above new algorithms have been implemented in our symbolic model checker, called SMART, which is built on decision diagram library "MDDL". This platform is now able to handle both logic and probabilistic model checking problems.

1.4 Structure

Chapter 2 reviews the required background knowledge, and the rest of the thesis is divided into two parts: on logic model checking and on probabilistic model checking, respectively.

The first part of my thesis covers three topics in logic model checking: first, I extend the saturation algorithm to CTL model checking and proposed constrained saturation (Chapter 3); second, I present two approaches to strongly-connected component (SCC) enumeration (Chapter 4), both of which employ the saturation algorithm; finally, I propose a new algorithm to build the transitive closure with distance, and show its application in finding EG and many other types of shortest witnesses (Chapter 5).

The second part of my thesis focuses on probabilistic model checking using decision diagram techniques. I introduce a new algorithm to carry out the Gauss-Seidel iteration, using EVMDDs to encode the CTMC (Chapter 6). This technique is a key step in numerical analysis and can be applied to both stationary solution and CSL model checking. Finally, I explore a new bounding CSL semantics (Chapter 7).

At the end of this thesis, I introduce the implementation of SMART and its new features (Chapter 8). Chapter 9 summarizes the whole thesis and points out future work.

Chapter 2

Preliminaries

This chapter reviews required background. Section 2.1 introduces the definition of three types of decision diagrams. Section 2.2 reviews discrete-state systems and assumptions for these systems. CTL and CTL model checking algorithms are also reviewed. Section 2.3 reviews the definition of continuous-time model checking, and CSL model checking algorithms.

2.1 Decision diagrams

This section reviews the key data structure, decision diagrams, in this thesis. While BDDs are the most widely used decision diagrams, we employ multi-way decision diagrams, which extend variable domains to arbitrary bounded integers. This choice is not a novelty by itself, but facilities the algorithms in following chapters.

Binary decision diagrams (BDDs) are the most widely used symbolic data structures to store and manipulate sets of states. The invention of BDDs [54] was motivated by

the design and analysis of digital circuits, where BDDs were proposed to encode general boolean functions. Bryant [8] proves that if the order of the variables in a BDD is fixed, there is a unique BDD structure corresponding to a given boolean function, and this property is called *canonicity*. BDD-based model checking has achieved great success [12, 11] in the verification of many real systems and is still a mainstream technique in formal verification.

Many different decision diagrams derived from BDDs have been proposed to tackle different problems. Definition 1 defines the most general rules shared by all decision diagrams in this thesis.

Definition 1 *A decision diagram is a directed, acyclic graph that contains a set of nodes N and satisfies the following rules:*

1. *Each node $p \in N$ is located on level $l \in \{0, 1, \cdots, L\}$, where L is the highest level in this decision diagram. Let $p.lvl$ denote the level of node p.*

2. *p is a terminal node iff $p.lvl = 0$. There is no outgoing edge from terminal node to any other nodes.*

3. *If p is not a terminal node, it may have outgoing edge with an index $i \in \mathbb{N}$, denoted by $p[i]$, pointing to a different node $q \in N$ (self-loop is not allowed). Moreover, $q.lvl < p.lvl$, which guarantees that there is not cycle in this graph.*

4. Canonicity rule*: given nodes $p, q \in N$ with $p.lvl = q.lvl > 0$, if $\forall i, p[i] = q[i]$, then $p = q$. In other words, decision diagrams do not allow* duplicated nodes*.*

In the application of decision diagrams, each level $l \in \{1, \cdots, L\}$ often corresponds to a variable v_l, so an order between variables $v_1, \cdots, v_L$ is defined. In this thesis, we assume

the variable order is fixed before decision diagrams are created. The size and structure of a decision diagram are often changed by applying a different variable order.

To completely define a type of the decision diagram, there are several other rules required besides the definition above:

- *Domain*: for all nodes on level k, the range of the indices of outgoing edges.
- *Range*: number of terminal nodes and their meanings.
- *Reduction rules*: it determines whether and when a node would not be stored. As an extreme case, *quasi-reduced* decision diagrams require that for any edge $p[i]$ pointing to node q, it always has $q.lvl = p.lvl - 1$, while there are other reduction rules.

For example, BDDs have the following rules besides those in Definition 1.

Definition 2 *Binary decision diagrams are decision diagrams satisfying:*

- *Domain: for all nodes, there are at most two outgoing edges with indices either* 0 *or* 1*.*
- *Range: there are only two terminal nodes, called* **0** *and* **1***.*
- *Reduction rules: BDDs are fully reduced, it means there is no* redundant *node q that $q[0] = q[1]$ stored in a BDD.*

Let $p[0]$ or $p[1]$ be the destination node of p's 0- or 1-edge. Due to the fully-reduced rule, it is possible that node $p[i] = q$ and $q.lvl < p.lvl - 1$, which means the nodes on levels $q.lvl + 1, \cdots, p.lvl - 1$ are skipped by this edge. The fully-reduced rule ensures that indices on those levels do not affect the destination terminal node of this path.

A BDD node p on level $k \leq L$ defines a function $f_p : \mathbb{B}^L \to \mathbb{B}$ that:

$$f_p(i_L, \cdots, i_1) = f_q(i_L, \cdots, i_1) \qquad \text{if } k > 0 \text{ and } p[i_k] = q,$$

and $f_{\mathbf{0}}(i_L, \cdots, i_1) = 0, f_{\mathbf{1}}(i_L, \cdots, i_1) = 1$, where $i_L, \cdots, i_1$ are boolean values. If we consider f_p as the characteristic function of a set $\mathcal{X} \subseteq \mathbb{B}^L$, node p can be considered as an encoding of $\mathcal{X}$ that $(i_L, \cdots, i_1) \in \mathcal{X}$ iff $f_p(i_L, \cdots, i_1) = 1$.

2.1.1 Multi-way decision diagrams

The definition of *Multi-way Decision Diagrams* (MDDs) [44] subsumes that of BDDs, and allows arbitrary finite integer values on each level.

Definition 3 *Multi-way Decision Diagrams (MDDs) are decision diagrams satisfying:*

- *Domain: for nodes on level k, there are at most n_k outgoing edges, with indices $0 \cdots n_k - 1$. Let $\mathcal{X}_k = \{0 \cdots n_k - 1\}$. Let $p[i] = q$ where q is the destination node for p's outgoing edge with index i*
- *Range: there are only two terminal nodes with values 0 and 1, denoted by* **0** *and* **1**, *respectively.*
- *Reduction rules: we consider both fully- and quasi- reduced MDDs, see definitions below.*

In the rest of this thesis, we use a, p, q or s to denote an MDD node. Compared with BDDs, MDDs just extend the domain from boolean to $\mathcal{X}_k$ on level k.

We first consider a quasi-reduced MDD, if $p[i] = q$, it is required that $q.lvl = p.lvl - 1$. Thus, the boolean function encoded by an MDD node p at level $k > 1$ is recursively defined as f_p where:

$$f_p(i_k, i_{k-1}, \cdots, i_1) = f_{p[i_k]}((i_{k-1}, \cdots, i_1)).$$

with terminal cases $f_{\mathbf{0}} = 0$ and $f_{\mathbf{1}} = 1$. Fig. 2.1 (left) shows an example of quasi-reduced MDDs. p on level L also encodes a set $\mathcal{B}(p) \subseteq \mathcal{X}_L \times \mathcal{X}_{L-1} \cdots \mathcal{X}_1$ s.t. $(i_L, \cdots, i_1) \in \mathcal{B}(p)$ iff $f_p(i_L, \cdots, i_1) = 1$

We often only care about paths leading to terminal **1**, thus we do not need to explicitly store terminal **0**, and nodes from which no path leads to **1**. In other words, every node p stored in an MDD must have at least one path leading to terminal **1**. If $p[k]$ is not stored, it is called a zero child, denoted by $p[k] = NULL$, meaning there is no path from p's child with index k leading to terminal **1**. This type of storage is called *sparse* storage. Fig. 2.1 (middle) shows an example of quasi-reduced sparse MDDs. In the remainder of this thesis, "quasi-reduced MDDs" always refers to sparse-stored quasi-reduced MDDs.

Similar with BDDs, fully-reduced MDDs eliminate all redundant nodes. A node p on level l is *redundant* if $\forall i_l, j_l \in \mathcal{X}_l$, $p[i_l] = p[j_l]$. The function that a non-terminal node p on level $k \leq L$ in a fully-reduced MDD encodes is:

$$f_p(i_L, \cdots, i_1) = f_{p[i_k]}(i_L, \cdots, i_1),$$

with terminal cases $f_{\mathbf{0}} = 0$ and $f_{\mathbf{1}} = 1$. Redundant nodes reflect the fact that the values of variables corresponding to the skipped levels do not affect the result, given the values of

variables on upper levels. Similarly, p on level $l \leq L$ encodes a set $\mathcal{B}(p) \subseteq \mathcal{X}_L \times \mathcal{X}_{L-1} \cdots \mathcal{X}_1$ s.t. $(i_L, \cdots, i_1) \in \mathcal{B}(p)$ iff $f_p(i_L, \cdots, i_1) = 1$.

As we mentioned in Definition 1, canonicity is a very important property, which guarantees that in an MDD (using either quasi- and fully-reduced rule), $p \neq q$ iff $\mathcal{B}(p) \neq \mathcal{B}(q)$.

Canonicity can be achieved recursively by fulfilling the following two rules:

- No two terminal nodes have the same value, i.e., $f_p \neq f_q$ for any terminal nodes $p \neq q$.
- Given non-terminal nodes p and q on level k, if there are $p[i_k] = q[i_k]$ for all $i_k \in \mathcal{X}_k$, $p = q$.

It is easy to prove that, if the above rules are enforced, canonicity in the MDD can be achieved. The first rule is trivial, and the second rule can be satisfied by using *unique table*, which is a hash table storing pointers to nodes. The hash value of a node n is determined by its level k and $n[i_k]$ for all $i_k \in \mathcal{X}_k$. When trying to create a new node p, all its child nodes must first be stored in the unique table, and the unique table checks whether there is an existing node q such that $q.lvl = p.lvl$ and $p[i] = q[i]$ for all $i \in \mathcal{X}_k$. If

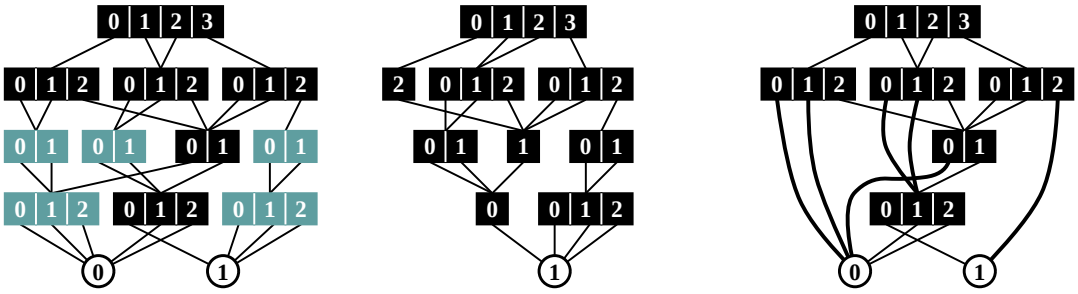


Figure 2.1: Quasi-reduced MDD (left), quasi-reduced sparse MDD (middle), and fully-reduced MDD (right)

so, no new node is created but the pointer pointing to q is returned. If not, we create a new node as p and store its pointer in the unique table.

Given MDD nodes p and q encoding sets $\mathcal{B}(p), \mathcal{B}(q) \subseteq \mathcal{X}_L \times \cdots \times \mathcal{X}_1$, set operations, like union, intersection, and minus, can be executed symbolically using operations on the MDD. Figure 2.2 shows the pseudo code for the union operation. Function *Union* returns an MDD node r where $\mathcal{B}(r) = \mathcal{B}(p) \cup \mathcal{B}(q)$. Similar algorithms are available for the intersection and minus operations.

An important strategy to reduce the complexity of MDD operations, like union, is to maintain an *operation cache*. Operation cache is a hash table storing the results obtained from previous computations. Before the real computation, a search in the operation cache is carried out (Line 4 in Figure 2.2) to find if the same operation has been computed before. If so, the previous result is returned directly, and thus the duplicated operations on the same nodes will be avoided. Assume we have an ideal operation cache, which keeps a complete history of previous computations, the complexity of union operation is $O(|p| \times |q|)$, where $|p|$ and $|q|$ are numbers of nodes in the MDD that is reachable from p and q, respectively.

2.1.2 Edge-valued Multi-way decision diagrams

Edge-Valued Multi-way Decision Diagrams (EVMDDs) are an extension of Multi-way Decision Diagrams (MDDs). An integer or real value is associated to each edge of the diagram, enabling them to encode integer or real functions over $\mathcal{X}_L \times \mathcal{X}_{L-1} \times \cdots \times \mathcal{X}_1$, instead of boolean functions like MDDs. Widely used Multi-terminal Binary Decision Diagrams (MTBDDs, and the similar Algebraic Decision Diagrams, ADDs) [53] or Multi-terminal

```
mdd Union(mdd p, mdd q) is
 1  if p = 0 or q = 1 then return q;
 2  if q = 0 or p = 1 then return p;
 3  if p = q then return p;
 4  if InCache_Union({p, q}, r) then return r;
 5  l_p ← p.lvl; l_q ← q.lvl;
 6  if l_p = l_q then foreach i ∈ X_{l_p} r[i] ← Union(p[i], q[i]); endfor
 7  else if l_p < l_q then foreach i ∈ X_{l_q} r[i] ← Union(p, q[i]); endfor
 8  else if l_p > l_q then foreach i ∈ X_{l_p} r[i] ← Union(p[i], q); endfor
 9  endif
10  r ← UniqueTableInsert(r);
11  CacheAdd_Union({p, q}, r);
12  return r;
```

Figure 2.2: Union operation on fully-reduced BDDs.

Multi-way Decision Diagrams (MTMDDs) are natural extensions of BDDs and MDDs to real values. In this thesis, however, we employ Edge-Valued Multi-way Decision Diagrams (EVMDDs) [21], which have the advantage of being more compact: [69] proves that an EVMDD never contains more nodes than the equivalent MTMDD under the same variable order. It is easy to find examples (like in Figure 2.3) where the EVMDD is exponentially more compact than the equivalent MTMDD for the same or even for any variable order.

Definition 4 *EVMDDs are decision diagrams where:*

- *Domain: for nodes on level k, there are at most n_k outgoing edges, with indices in $\{0 \cdots n_k - 1\}$.*
- *Range: the only terminal node is Ω.*
- Edge value*: for any non-terminal node p, each of its outgoing edge with index i pointing to q has an associated value ρ, we write $p[i_k] = \langle p[i_k].v, p[i_k].ch \rangle = \langle \rho, q \rangle$.*
- *Reduction rules: we first consider quasi-reduction.*

When referring to a node p in an EVMDD, we use $\langle\sigma,p\rangle$ to denote there is a "dangling edge" pointing to p with a value σ attached.

We use two versions of EVMDDs, multiplicative EV*MDDs and additive EV$^+$MDDs. The above properties are shared by both, but they differ in three aspects. First, EV$^+$MDD edges have value in $\mathbb{Z} \cup \{\infty\}$ (where "∞" is greater than any positive number and can be interpreted as "undefined"), while EV*MDD edges have value in $\mathbb{R}_{\geq 0}$.

Second, the edge values are combined in different ways: given a value σ and an EVMDD node p with $p.lvl = k$, in EV*MDDs, the pair $\langle\sigma,p\rangle$ encodes the function recursively defined by

$$f_{\langle\sigma,p\rangle}(i_L, ..., i_1) = \begin{cases} \sigma & \text{if } k{=}0, \text{ i.e., } p{=}\Omega \\ \sigma \cdot f_{p[i_k]}(i_L, ..., i_1) & \text{if } k{>}0, \text{ i.e., } p{\neq}\Omega, \end{cases}$$

while in EV$^+$MDD,

$$f_{\langle\sigma,p\rangle}(i_L, ..., i_1) = \begin{cases} \sigma & \text{if } k{=}0, \text{ i.e., } p{=}\Omega \\ \sigma + f_{p[i_k]}(i_L, ..., i_1) & \text{if } k{>}0, \text{ i.e., } p{\neq}\Omega. \end{cases}$$

Thus, for an EV*MDD node p at level L, $f_{\langle\sigma,p\rangle}(i_L, ..., i_1)$ equals σ times the product of the values encountered along the path from p to Ω corresponding to i_L, i_{L-1}, ..., i_1, in order, while, if p is an EV$^+$MDD node, it equals σ plus the sum of the values on this path.

The final difference is about *normalization* and *default edges*. An EV*MDD node is normalized if all edge values are in $[0, 1]$ and at least one is 1, while edges with value 0 are the default, thus do not need to be stored explicitly (the identity of the corresponding child node is irrelevant, since the function value will be 0 whenever a default edge is taken). Thus, a function $f : \mathcal{X} \to \mathbb{R}_{\geq 0}$ is (canonically) encoded by $\langle\sigma,p\rangle$ where $\sigma = \max\{f(\mathbf{i}) : \mathbf{i} \in \mathcal{X}\}$ and

p is a node at level L, with the exception of the constant function $f \equiv 0$, which we represent by $\langle 0,\Omega \rangle$. An EV$^+$MDD node is instead normalized if all edge values are non-negative and at least one is zero, while ∞ is the default value, so that a function $f : \mathcal{X} \to \mathbb{Z} \cup \{\infty\}$ is (canonically) encoded by $\langle \sigma,p \rangle$ where $\sigma = \min\{f(\mathbf{i}) : \mathbf{i} \in \mathcal{X}\}$ and p is a node at level L, with the exception of the constant function $f \equiv \infty$, which we represent by $\langle \infty,\Omega \rangle$. For both EV$^+$MDDs and EV*MDDs, if $p[i_k].v$ is the default value, we write $p[i_k] = \perp$ to indicate that i_k^{th} edge of p is default.

Figure 2.3 shows an example of EV$^+$MDD, which encodes a function:

$$f(v_4, v_3, v_2, v_1) = 18 \cdot v_4 + 6 \cdot v_3 + 3 \cdot v_2 + v_1.$$

Function *Normalize* in Figure 2.4 describes the procedure to normalize an EV$^+$MDD.

As examples, we show three EV$^+$MDD operations that are extensively used in Chapter 6:

- *Min*: given two EV$^+$MDDs $\langle \rho,a \rangle$ and $\langle \sigma,b \rangle$, return an EV$^+$MDD $\langle \mu,r \rangle$ encoding function $\min(f_{\langle \rho,a \rangle}, f_{\langle \sigma,b \rangle})$.

- *Sum*: given two EV$^+$MDDs $\langle \rho,a \rangle$ and $\langle \sigma,b \rangle$, return an EV$^+$MDD $\langle \mu,r \rangle$ encoding function $f_{\langle \rho,a \rangle} + f_{\langle \sigma,b \rangle}$.

- *MinState*: given an EV$^+$MDD $\langle \rho,a \rangle$, return a state $\mathbf{i}$ such that $f_{\langle \rho,a \rangle}(\mathbf{i})$ is the minimum value of function $f_{\langle \rho,a \rangle}$.

Figure 2.5 shows procedures *Min* and *Sum*. These algorithms run recursively on each level, without having to enumerate every state of a potentially huge state space. It

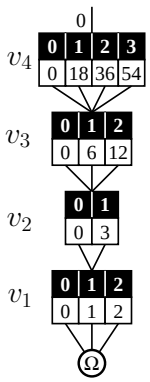


Figure 2.3: An EV$^+$MDD.

```
evmdd Normalize(evmdd.node a)
 1  if a = Ω then return ⟨0,Ω⟩;
 2  level   k ← a.lvl;
 3  b ← NewNode(k);                          • create a node at level k
 4  for i_k ∈ {0, ..., n_k − 1} do
 5    ⟨μ,s⟩ ← Normalize(a[i_k].node);
 6    b[i_k] ← ⟨μ + a[i_k].val,s⟩;
 7  endfor
 8  ρ ← min{b[i_k].val : i_k ∈ {0, ..., n_k − 1}}
 9  for i_k ∈ {0, ..., n_k − 1} do
10    b[i_k].val ← b[i_k].val − ρ        • b[i_k].val ≥ 0 and ∃i_k, b[i_k].val = 0
11  endfor
12  return ⟨ρ,b⟩;
```

Figure 2.4: Pseudocode for *Normalize*.

is known [53] that the number of recursive calls of the generic *Apply* operation, including *Min* and *Sum*, for EVMDDs at most equals those for the MTMDDs representing the same function. Procedure *MinState* is even simpler: since each non-terminal node must have at least one edge with an associated value of 0, we can simply follow any 0-value path from the root $\langle \rho^*, r^* \rangle$ to Ω. The function encoded by the EV$^+$MDD evaluates to the minimum possible value, ρ^*, for any state corresponding to such a path.

2.2 Discrete-state systems

The execution of almost all software/hardware systems can be discretized as a transitions graph (V, E), where vertex $v \in V$ represent a *state* of the system, and an edge represents called a transition between system states. For example, a 2-bit counter has four states $\{00, 01, 10, 11\}$. At any given time, the counter must be in one and only one of these four states. After a transition occurs, the state of the system may change. In our example of counter, there are four possible transitions: $00 \to 01, 01 \to 10, 10 \to 11$, and $11 \to 00$.

```
evmdd Min(evmdd ⟨ρ,p⟩, evmdd ⟨σ,q⟩)
 1  int   μ ← min{ρ, σ};     level   k ← p.lvl;
 2  if ρ = ∞ then return ⟨σ,q⟩;
 3  if σ = ∞ then return ⟨ρ,p⟩;
 4  if p = q then return ⟨μ,p⟩;                    • includes the case k = 0, i.e. p = q = Ω
 5  if InCache_Min(⟨p, q, ρ − σ, r⟩) then return ⟨μ,r⟩;  • Assume ρ ≥ σ, if not, swap two parameters.
 6  node  r ← NewNode(k);
 7  for i_k ∈ S_k do
 8    r[i_k] ← Min(⟨ρ − μ + p[i_k].val,p[i_k].node⟩, ⟨σ − μ + q[i_k].val,q[i_k].node⟩);
 9  endfor
10  ⟨γ,r⟩ ← Normalize(r);
11  InsertUT(r);     CacheAdd_Min(⟨p, q, ρ − σ, r⟩);
12  return ⟨μ + γ,r⟩;
```

```
evmdd Sum(evmdd ⟨ρ,p⟩, evmdd ⟨σ,q⟩)
 1  int   μ ← ρ + σ;     level   k ← p.lvl;
 2  if ρ = ∞ or σ = ∞ then return ⟨∞,Ω⟩;
 3  if InCache_Sum(⟨p, q, ⟨γ,r⟩⟩) then return ⟨μ + γ,r⟩;
 4  node   r ← NewNode(k);
 5  for i_k ∈ S_k do
 6    r[i_k] ← Sum(p[i_k], q[i_k]);
 7  endfor
 8  ⟨γ,r⟩ ← Normalize(r);
 9  InsertUT(r);     CacheAdd_Sum(⟨p, q, ⟨γ,r⟩⟩);
10  return ⟨μ + γ,r⟩;
```

Figure 2.5: Pseudocode for *Min* and *Sum*.

When modeling a real system, instead of building a low-level transition graph, which is often huge, we describe a high-level model that consists of several components and events. At any given time, each component has a local state within a local state space, and the (global) state of the system is given by the combination of the local states of all components. In the above example of 2-bit counter, the system can be partitioned into two components: the lower bit and the higher bit. The local state space of both components is $\{0, 1\}$. Note that not all possible combinations of the local states may be included in the state space, i.e., some of the "potential" global states may not in fact be reachable in the system.

In a high-level model, all possible transitions are described as a set of *events*, each of which consists of:

- *Guard*: the condition controlling when an event may occur
- *Effect*: a function mapping the source (current) state to a (non-empty and finite) set of possible destination states reached by taking this event

An event is *enabled* in a state iff its guard condition is satisfied in this state. In each transition, one and only one enabled event is *fired*. When there are multiple enabled events in a state, one of these event is fired nondeterministically.

We now give the formal definition of discrete-state model as follows:

Definition 5 *A discrete-state model $\mathcal{D}$ is defined as $(\widehat{\mathcal{S}}, \mathcal{S}_{init}, \mathcal{E}, \mathcal{N})$ where*

- *the* potential state space $\widehat{\mathcal{S}} = \mathcal{S}_L \times \mathcal{S}_{L-1} \times \cdots \times \mathcal{S}_1$. *$\mathcal{S}_k$ is the local state space for k^{th} submodel; Thus, each (global) state $\mathbf{i}$ is a tuple $(i_L, \cdots, i_1)$, where $i_k \in \mathcal{S}_k$, for $L \geq k \geq 1$;*
- *the set of initial states is $\mathcal{S}_{init} \subseteq \widehat{\mathcal{S}}$;*
- *the set of (asynchronous) events is $\mathcal{E}$;*
- *the next-state function $\mathcal{N} : \widehat{\mathcal{S}} \to 2^{\widehat{\mathcal{S}}}$ is described in disjunctively partitioned form as $\mathcal{N} = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e$, where $\mathcal{N}_e$ is the next-state function for event $e \in \mathcal{E}$. $\mathcal{N}(\mathbf{i})$ denotes the set of states that can be nondeterministically reached in one step from $\mathbf{i}$.*

In this thesis, we use $\mathcal{X}, \mathcal{Y},$ and $\mathcal{Z}$ to denote sets of states, $\mathbf{i}, \mathbf{j},$ and $\mathbf{k} \in \widehat{\mathcal{S}}$ to denote (global) states where $i_l, j_l,$ and k_l are the local state of the l^{th} submodels, respectively.

The next-state function $\mathcal{N}_e$ describes both the guard and effect for an event e. e is *enabled* in state $\mathbf{i}$ iff $\mathcal{N}_e(\mathbf{i}) \neq \emptyset$. $\mathcal{N}$ is the disjunction of $\mathcal{N}_e$ for all $e \in \mathcal{E}$. Let $\mathcal{N}_e(\mathcal{X}) = \bigcup_{\mathbf{i}\in\mathcal{X}} \mathcal{N}_e(\mathbf{i})$, and a similar notation also applies to $\mathcal{N}$.

Correspondingly, $\mathcal{P}$ and $\mathcal{P}_e$ denote the previous-state functions, e.g., $\mathcal{P}_e(\mathbf{i})$ is the set of states that can reach $\mathbf{i}$ in one step by firing event e. If $\mathbf{j} \in \mathcal{N}_e(\mathbf{i})$, then $\mathbf{i} \in \mathcal{P}_e(\mathbf{j})$. $\mathcal{P}$ denotes the previous-state function, e.g., $\mathcal{P}(\mathbf{i})$ is the set of states that can reach $\mathbf{i}$ in one step. Similarly, $\mathcal{P}_e(\mathcal{X})$ can also be defined.

As a typical example of discrete-state mode, Petri nets [65] are often used to model protocols and asynchronous systems.

Definition 6 *A Petri net as a tuple* $(\mathcal{P},\mathcal{T},\mathbf{D}^-,\mathbf{D}^+,\mathbf{s}^{init})$ *where:*

- $\mathcal{P}$ *is a set of* places*, drawn as circles, and* $\mathcal{T}$ *is a set of* transitions*, drawn as rectangles, satisfying* $\mathcal{P} \cap \mathcal{T} = \emptyset$*.*
- $\mathbf{D}^- : \mathcal{P} \times \mathcal{T} \to \mathbb{N}$ *and* $\mathbf{D}^+ : \mathcal{P} \times \mathcal{T} \to \mathbb{N}$ *are the* input arc *and the* output arc *cardinalities, respectively.*
- $\mathbf{s}^{init} \in \mathbb{N}^{|\mathcal{P}|}$ *is the* initial marking*, specifying a number of* tokens *initially present in each place.*

If the current marking is $\mathbf{i} \in \mathbb{N}^{|\mathcal{P}|}$*, we say that* $\alpha \in \mathcal{T}$ *is* enabled *in* $\mathbf{i}$*, written* $\alpha \in \mathcal{T}(\mathbf{i})$*, iff* $\forall p \in \mathcal{P}, \mathbf{D}^-_{p,\alpha} \leq i_p$*. Then,* $\alpha \in \mathcal{T}(\mathbf{i})$ *can* fire*, changing the marking to* $\mathbf{j}$*, written* $\mathbf{i} \xrightarrow{\alpha} \mathbf{j}$*, satisfying* $\forall p \in \mathcal{P}, j_p = i_p - \mathbf{D}^-_{p,\alpha} + \mathbf{D}^+_{p,\alpha}$*.*

The above definition is for standard Petri nets, while in the application, we also consider the following two extensions:

- $\mathbf{D}^{\circ} : \mathcal{P} \times \mathcal{T} \to \mathbb{N} \cup \{\infty\}$ are the inhibitor arc cardinalities, so that transition α is disabled in marking $\mathbf{i}$ if there is a place p such that $\mathbf{D}^{\circ}_{p,\alpha} \leq i_p$.

- $\mathbf{D}^{-}, \mathbf{D}^{+} : \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{|\mathcal{P}|} \to \mathbb{N}$ are the *marking-dependent* input and output arc cardinalities so that α is enabled in $\mathbf{i}$ iff, $\forall p \in \mathcal{P}$, $\mathbf{D}^{-}_{p,\alpha}(\mathbf{i}) \leq i_p$ and, if α fires, it leads to marking $\mathbf{j}$ satisfying $j_p = i_p - \mathbf{D}^{-}_{p,\alpha}(\mathbf{i}) + \mathbf{D}^{+}_{p,\alpha}(\mathbf{i})$. Both $\mathbf{D}^{-}_{p,\alpha}$ and $\mathbf{D}^{+}_{p,\alpha}$ are evaluated on the current, not the new, marking.

These two extensions can also be combined, allowing marking-dependent inhibitor arc cardinalities. Inhibitor arcs alone suffice to achieve Turing-equivalence, while self-modifying behavior may or may not, depending on the type of functions allowed to specify arc cardinalities.

Figure 2.6 shows markings of an extended Petri net before and after a transition is fired. Circles and the rectangle represent places and a transition respectively. Numbers in places indicate numbers of tokens in corresponding places, and numbers on arcs are their cardinalities (default cardinality is 1). The arrow with a circle represents an inhibitor arc. After the transition is fired, it is no longer enabled since it is inhibited by the inhibitor arc.

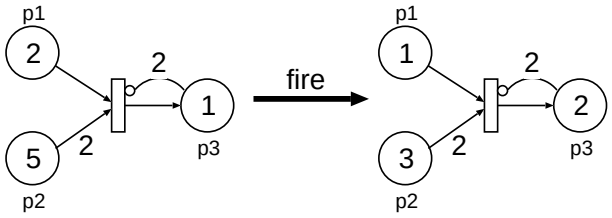


Figure 2.6: An example of firing

2.2.1 Symbolic state-space generation and saturation

Within the potential state space, we are often interested in the (real) state space, $\mathcal{S}$, which is the set of states that can be reached from the initial states. $\mathcal{S}$ can be recursively defined as follows:

1. $\mathcal{S}_{init} \subseteq \mathcal{S}$;

2. if $\mathbf{i} \in \mathcal{S}$, then $\mathcal{N}(\mathbf{i}) \subseteq \mathcal{S}$.

Thus, $\mathcal{S}$ can be computed as follows:

$$\mathcal{S}_{init} \cup \mathcal{N}(\mathcal{S}_{init}) \cup \mathcal{N}^2(\mathcal{S}_{init}) \cup \cdots,$$

where $\mathcal{N}^2(\mathcal{X}) = \mathcal{N}(\mathcal{N}(\mathcal{X}))$ and $\mathcal{N}^{k+1}(\mathcal{X}) = \mathcal{N}(\mathcal{N}^k(\mathcal{X}))$.

Considering the storage of $\mathcal{S}$ and $\mathcal{N}$ for a given system $\mathcal{D}$, the size of $\mathcal{S}$ could be potentially huge for many real systems, and the size of possible transitions in $\mathcal{N}$ could be as large as $|\mathcal{S}| \times |\mathcal{S}|$. Efficient data structures are required to encode $\mathcal{S}$ and $\mathcal{N}$ before we can carry out verification and analysis. Currently, there are two main approaches, explicit and symbolic. The explicit approach utilizes an explicit data structure to store each state. Although the data structure for each state may be efficient, as $\mathcal{S}$ grows, the scale of the model that can be handled using the explicit approach is always limited by the memory. The symbolic approach utilizes symbolic structure to encode a set of states. The merit of this approach is that the memory consumption does not necessarily grows with $\mathcal{S}$, so the symbolic approach could handle models with a huge state space (say, more than 10^{20} states [12]), which is not possible for the explicit approach. However, the memory and runtime consumption for the symbolic approach is not less predictable.

Encoding sets of states

To encode discrete-state system $\mathcal{D}$, we can simply set the domain $\mathcal{X}_k$ of level l to the local state space $\mathcal{S}_k$ in $\mathcal{D}$. An MDD node p can encode the characteristic function of a set of states $\mathcal{X}$ such that

$$(i_L, \cdots, i_1) \in \mathcal{X} \text{ iff } f_p(i_L, \cdots, i_1) = 1,$$

where $i_k \in \{0, \cdots, n_l - 1\}$. There are two ways to encode a set of states $\mathcal{Y} \subseteq \mathcal{S}_L \times \cdots \times \mathcal{S}_1$: using either quasi-reduced or fully-reduced L-level MDDs. As a guideline, we use a quasi-reduced MDD to encode sets of states within a real state space $\mathcal{S}$, and a fully-reduced MDD to encode properties defined on potential state space $\widehat{\mathcal{S}}$.

Fully-reduced MDDs allow edges to skip levels corresponding to "don't care" submodels. For example, when considering all global states where the k^{th} submodel is in local state i_k, the fully-reduced MDD contains only one node on level k. Fully-reduced MDDs can often compactly encode a set of states satisfying a property referring only some of the submodels in given discrete-state system, and the levels that corresponds to other submodels are all skipped.

Quasi-reduced MDDs are used to enumerate states in $\mathcal{S}$, since each path from a top-level (L) node to a terminal node only represent one specific global states. Given a quasi-reduced MDD node n on level L, we can compute the number of states, denoted by $|\mathcal{B}(n)|$, by counting the number of paths from node n to terminal **1**.

Encoding the next-state functions

The encoding of the next-state functions is in fact to encode a set of tuples $(i_L, \cdots, i_1, i'_L, \cdots, i'_1)$ such that $(i'_L, \cdots, i'_1) \in \mathcal{N}(i_L, \cdots, i_1)$ holds. Thus, next-state function $\mathcal{N}$ can be encoded using an MDD node on level $2L$, and in this thesis we always use the interleaved order: $i_L, i'_L, ..., i_1, i'_1$, where i_k in "from" states are placed on unprimed level k and i_k in "to" states on primed level k'. Let $\mathit{Unprimed}(k) = \mathit{Unprimed}(k') = k$. According to previous experience, interleaved order often results in a compact encoding of next-state functions.

A fundamental property enjoyed by most asynchronous systems, *locality*, can be exploited to obtain a compact symbolic expression. An event e is *independent* of the k^{th} submodel if its enabling does not depend on i_k and its firing does not change the value of i_k. A level k belongs to the *support* set of event e, denoted $\mathit{supp}(e)$, if e is not independent of k. We define $\mathit{Top}(e)$ to be the top level in $\mathit{supp}(e)$, and let $\mathcal{E}_k$ be the set of events $\{e \in \mathcal{E} : \mathit{Top}(e) = k\}$. Also, we let $\mathcal{N}_k$ be the next-state function corresponding to all events in $\mathcal{E}_k$, i.e., $\mathcal{N}_k = \bigcup_{e \in \mathcal{E}_k} \mathcal{N}_e$. If $e \in \mathcal{E}_k$, since e does not affect $\mathcal{S}_L, \ldots, \mathcal{S}_{k+1}$, $\mathcal{N}_e(i_L, \ldots, i_1)$ can be computed as $(i_L, \ldots, i_{k+1}) \times \mathcal{N}_e(i_k, \ldots, i_1)$. When the set of states is encoded with an MDD, firing $\mathcal{N}_e$ only modifies the subtree rooted at nodes on level k in the MDD.

We use the *quasi-identity-fully (QIF)* reduction rule [77] for MDDs encoding next-state functions. For an event e with $\mathit{Top}(e) = k$, $\mathcal{N}_e$ is encoded with a $2k$-level MDD since e does not affect states on levels $L, \ldots, k+1$ levels, which are skipped in this MDD. The advantages of the QIF reduction rule is that the application of $\mathcal{N}_e$ only needs to start at

level $Top(e)$, and not all the way up, at level L. We refer interested readers to [77] for more details about this encoding.

Figure 2.7(a) shows the discrete-state model for a 2-bit counter and Figure 2.7(b) shows the next-state functions encoded with MDDs. $\mathcal{N}_1$ represents the transition $\{i_2 = x, i_1 = 0\} \rightarrow \{i_2' = x, i_1' = 1\}$, where x is an arbitrary value. Thus this transition is independent of v_2. According to the QIF reduction rule, the top two levels are skipped in the MDD encoding $\mathcal{N}_1$, as shown in Figure 2.7(b).

The problem of state-space generation refers to computing $\mathcal{S}$, which is often performed before model checking and other analysis. For many models, even if the state space is finite, the size of each local state space is unknown a priori. After state-space generation, $\mathcal{S}$, the sets $\mathcal{S}_k$, and their sizes n_k are consequently known in the following discussion.

The relational product is a key operation in symbolic model checking. Given a set of state encoded by a quasi-reduced MDD node s, and MDD node r encoding the next-state function $\mathcal{N}$, the relational product computes a QFI-reduced MDD node t encoding the set $\mathcal{N}(\mathcal{B}(s))$. This process is also called image computation. The pseudo code for the relational produce is shown in Figure 2.8.

If MDD node r encodes the previous-state function $\mathcal{P}$, the relational produce $RelProd(s, r)$ computes the set $\mathcal{P}(\mathcal{B}(s))$, which is called preimage computation. Based on the image and preimage computations, symbolic state-space exploration can be carried out in either forward or backward direction. The forward reachable states from a set of states $\mathcal{X}$ can be computed by $ReachF(\mathcal{X}) = \mathcal{X} \cup \mathcal{N}(\mathcal{X}) \cup \mathcal{N}^2(\mathcal{X}) \cup \cdots$. Analogously, we let $ReachB(\mathcal{X}) = \mathcal{X} \cup \mathcal{P}(\mathcal{X}) \cup \mathcal{P}^2(\mathcal{X}) \cup \cdots$.

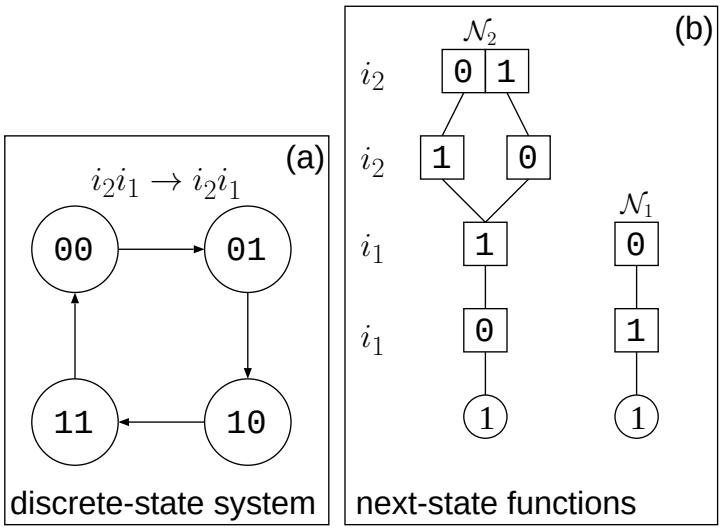


Figure 2.7: (a) The discrete-state model for a 2-bit counter; (b) MDDs encoding the next-state function.

A straight-forward algorithm for state-space generation is to compute $ReachF(\mathcal{S}_{init})$ using breadth-first search, as shown in Figure 2.9, where s encodes the set of initial states and r encodes $\mathcal{N}$.

Traditional symbolic state-space generation algorithms use some variant of symbolic image computation. The simplest approach is a breadth-first search (BFS) directly implementing the definition of $\mathcal{S}$ as $\mathcal{S}_{init} \cup \mathcal{N}(\mathcal{S}_{init}) \cup \mathcal{N}^2(\mathcal{S}_{init}) \cup \cdots$.

Saturation [18, 19, 20] employs a different approach, recursively computing *local fixpoints* and exploiting locality in the next-state functions. *Locality* is a fundamental property enjoyed by asynchronous systems, expresses the fact that most events affect only few systems components.

The key idea of saturation is to fire events in an order consistent with their *Top*: we attempt firing events in $\mathcal{E}_k$ on node a at level k only after having exhaustively fired

```
mdd  RelProd(mdd s, r)
 1  level l_s ← s.lvl; level l_r ← Unprimed(r.lvl)
 2  mdd t ← 0;
 3  if s = 1 and r = 1 then return 1; else create t s.t. ∀i ∈ S_{l_s}.t[i] = 0 endif;     • terminal case
 4  if InCache_RelProd(s, r, t) then return t; endif;
 5  if l_s = l_r then
 6    foreach i, i' ∈ S_{l_s} s.t. s[i], r[i][i'] ≠ 0 do
 7      t[i'] ← Union(t[i'], RelProd(s[i], r[i][i']));
 8    endfor;
 9  else
10    foreach i ∈ S_{l_s} s.t. s[i] ≠ 0 do
11      t[i] ← Union(t[i], RelProd(s[i], r));
12    endfor
13  endif
14  t ← InsertUT(t);
15  CacheAdd_RelProd(s, r, t);
16  return t;
```

Figure 2.8: The relational product operator.

```
mdd  SSGen(mdd s, r)
 1  mdd p_o ← 0; mdd p_n ← s;     • p_o is the set of old states, and p_n is the set of new states.
 2  while p_o ≠ p_n do
 3    p_o = p_n;
 4    p_n = Union(p_o, RelProd(p_o, r);
 5  endwhile
 6  return p_o;
```

Figure 2.9: State-space generation using breadth-first search.

events in $\mathcal{E}_h$, for all $h < k$, on nodes below a, until no new states are found. Then, a is *saturated* if it is a fixed point w.r.t. events independent of the levels above k: $\forall h, k \geq h \geq 1, \forall e \in \mathcal{E}_h, \mathcal{B}(a) \supseteq \mathcal{N}_e(\mathcal{B}(a))$.

Figure 2.10 shows the pseudocode of the saturation algorithm. Given a discrete-state model, $\mathcal{N}_L, ..., \mathcal{N}_1$ are globally available. The pseudocode shows a forward exploration, as it uses the next-state functions, but it can naturally be used for backward exploration by replacing $\mathcal{N}_k$ with $\mathcal{P}_k$. The input parameter s is the root node to be saturated, thus it is

```
mdd  SaturateF(mdd s)
 1  if InCache_SaturateF(s,t) then return t;                    • don't repeat work
 2  level k ← s.lvl;
 3  mdd t ← 0;
 4  foreach i ∈ S_k s.t. s[i] ≠ 0 do                            • first, saturate nodes below
 5    t[i] ← SaturateF(s[i]);
 6  endfor;
 7  repeat                                                      • keep firing N_k until reaching convergence
 8    foreach i, i' ∈ S_k s.t. N_k[i][i'] ≠ 0 do
 9      t[i'] ← Union(t[i'], RelProdSat(t[i], N_k[i][i']));
10    endfor;
11  until t does not change;
12  t ← InsertUT(t);                                            • Unique Table to avoid duplicate nodes
13  CacheAdd_SaturateF(s,t);                                    • to avoid repeating work later
14  return t;
```

```
mdd  RelProdSat(mdd s, r)
 1  level k ← s.lvl;
 2  mdd t ← 0;
 3  if s = 1 and r = 1 then return 1; endif;                    • terminal case
 4  if InCache_RelProdSat(s,r,t) then return t; endif;
 5  foreach i, i' ∈ S_k s.t. r[i][i'] ≠ 0 do
 6    t[i'] ← Union(t[i'], RelProdSat(s[i], r[i][i']));
 7  endfor;
 8  t ← InsertUT(t);                                            • analogous to RelProd until here...
 9  t ← SaturateF(t);                                           • ...but now we saturate t before returning
10  CacheAdd_RelProdSat(s,r,t);
11  return t;
```

Figure 2.10: The (forward) saturation algorithm.

initially set to the root of the MDD encoding $\mathcal{S}_{init}$. *SaturateF* saturates the nodes of MDD s in order, from the bottom level to the top level. Unlike the traditional relational product operation, *RelProdSat* always returns a saturated MDD.

2.2.2 Computation Tree Logic

Computation Tree Logic (CTL) [27] is a widely used temporal logic because of its simple yet expressive syntax. All CTL properties are state properties and can be checked by manipulating sets of states.

For a discrete-state system $\mathcal{D}$, we first define a set of atomic propositions $\mathcal{A}$ and in the remainder of this thesis we use $\phi, \psi, \varphi,$ and $\chi \in \mathcal{A}$ to denote atomic propositions. A proposition either holds or does not hold on a given state. Let labelling function $\mathcal{L}$ be $\mathcal{S} \to 2^{\mathcal{A}}$, mapping a state to the set of propositions that hold on this state, so that $\mathbf{i} \models \phi$ iff $\phi \in \mathcal{L}(\mathbf{i})$. Let $Sat(\phi) = \{\mathbf{i} \,|\, \mathbf{i} \models \phi\}$.

A *path* σ in $\mathcal{D}$ is an infinite sequence of states $\mathbf{i}_0\mathbf{i}_1\mathbf{i}_2\cdots$, where for any $k \geq 0$, $\mathbf{i}_{k+1} \in \mathcal{N}(\mathbf{i}_k)$. Let $\sigma[k] = \mathbf{i}_k$.

CTL defines four temporal operators $\mathsf{X}, \mathsf{U}, \mathsf{F}$, and G and two path quantifiers E and A. Temporal operators express properties for paths in the system:

- $\mathsf{X}\phi$: *neXt* operator, $\sigma \models \mathsf{X}\phi$ iff $\sigma[1] \models \phi$;
- $\phi\mathsf{U}\psi$: *Until* operator, $\sigma \models \phi\mathsf{U}\psi$ iff $\exists k \geq 0$ s.t. $\sigma[k] \models \psi$ and $\forall 0 \leq l < k, \sigma[l] \models \phi$;
- $\mathsf{F}\psi$: *Finally* operator, $\sigma \models \mathsf{F}\psi$ iff $\exists k \geq 0$ s.t. $\sigma[k] \models \psi$. $\mathsf{F}\psi$ is equivalent to $\mathit{True}\mathsf{U}\psi$;
- $\mathsf{G}\psi$: *Global* operator, $\sigma \models \mathsf{G}\phi$ iff $\forall k \geq 0, \sigma[k] \models \psi$.

The syntax of CTL formulas is defined recursively as follows:

- each atomic proposition ϕ is a CTL formula;
- if $\mathscr{F}_1$ and $\mathscr{F}_2$ are CTL formulas, so are
  - $\neg\mathscr{F}_1$, $\mathscr{F}_1 \vee \mathscr{F}_2$, $\mathscr{F}_1 \wedge \mathscr{F}_2$,
  - $\mathsf{EX}\mathscr{F}_1$, $\mathsf{E}[\mathscr{F}_1\mathsf{U}\mathscr{F}_2]$, $\mathsf{EG}\mathscr{F}_1$, $\mathsf{EF}\mathscr{F}_1$,
  - $\mathsf{AX}\mathscr{F}_1$, $\mathsf{A}[\mathscr{F}_1\mathsf{U}\mathscr{F}_2]$, $\mathsf{AG}\mathscr{F}_1$, $\mathsf{AF}\mathscr{F}_1$.

In other word, CTL requires temporal operators and path quantifiers to appear in pairs. CTL formulas are state properties and their semantics are given as follows:

- $\mathbf{i} \models \phi$ iff $\phi \in \mathcal{L}(\mathbf{i})$.
- $\mathbf{i} \models \neg\mathscr{F}$ iff $\mathbf{i} \not\models \mathscr{F}$
- $\mathbf{i} \models \mathscr{F}_1 \vee \mathscr{F}_2$ iff $\mathbf{i} \vDash \mathscr{F}_1 \vee \mathbf{i} \vDash \mathscr{F}_2$
- $\mathbf{i} \models \mathscr{F}_1 \wedge \mathscr{F}_2$ iff $\mathbf{i} \vDash \mathscr{F}_1 \wedge \mathbf{i} \vDash \mathscr{F}_2$
- $\mathbf{i} \models \mathsf{E}\Phi$ iff $\exists\sigma$ s.t. $\sigma[0] = \mathbf{i} \wedge \sigma \models \Phi$
- $\mathbf{i} \models \mathsf{A}\Phi$ iff $\forall\sigma$ with $\sigma[0] = \mathbf{i}. \sigma \models \Phi$,

where Φ is a path formula in one of the four forms: $\mathsf{X}\mathscr{F}_1$, $\mathscr{F}_1\mathsf{U}\mathscr{F}_2$, $\mathsf{G}\mathscr{F}_1$, or $\mathsf{F}\mathscr{F}_1$.

For a discrete-state system $\mathcal{D}$ and CTL formula $\mathscr{F}$, if $\forall\mathbf{i} \in \mathcal{S}_{init}, \mathbf{i} \models \mathscr{F}$, then $\mathcal{D} \models \mathscr{F}$. Thus, the key step in model checking a CTL formula $\mathscr{F}$ is to compute the set of states $\{\mathbf{i} \,|\, \mathbf{i} \models \mathscr{F}\}$.

2.2.3 CTL model checking

$\mathsf{AX}, \mathsf{AU}, \mathsf{AF},$ and AG operators can be equivalently converted to E operators:

- $\mathsf{AX}\phi = \neg\mathsf{EX}\neg\phi$
- $\mathsf{A}\phi\mathsf{U}\psi = \neg(\mathsf{E}[\neg\psi\mathsf{U}\neg(\phi \vee \psi)] \vee \mathsf{EG}(\neg\psi))$
- $\mathsf{AF}\psi = \neg\mathsf{EG}(\neg\psi)$
- $\mathsf{AG}\psi = \neg\mathsf{E}(\mathit{true}\mathsf{U}\neg\psi)$.

Thus $\{\mathsf{EX}, \mathsf{EU}, \mathsf{EG}\}$ is a complete set of operators for CTL, that is, it can be used to express any other CTL operator. The $\mathsf{EX}\phi$ operator can be easily computed as the relational product $\mathcal{P}(Sat(\phi))$, where $\mathcal{P}$ is the previous-state function. As a special case of EU, building the set of states satisfying $\mathsf{EF}\phi$ is instead essentially the same process as state-space generation, the only differences being that we start from $Sat(\phi)$ instead of $\mathcal{S}_{init}$ and that we go backwards instead of forwards, thus we use $\mathcal{P}$ (or $\mathcal{P}_\alpha$, or $\mathcal{P}_k$, as appropriate) instead of $\mathcal{N}$.

The traditional algorithm to obtain the set of states satisfying $\mathsf{E}\phi\mathsf{U}\psi$ computes a least fixpoint (see *EUtrad* in Figure 2.11; all sets of states and relations over states in our pseudocode are encoded using MDDs, of course). Starting from $Sat(\psi)$, it computes the intersection of the preimage of the explored states within the states in $Sat(\phi)$. The newly computed states are added to the explored states, for the next iteration. The number of iterations is thus equal to the maximum distance from states in $Sat(\phi) \setminus Sat(\psi)$ to states in $Sat(\psi)$.

The traditional EG algorithm (see *EGtrad* in Figure 2.11) computes a greatest fixpoint by iteratively eliminating states without successors in the working set $\mathcal{X}$.

2.3 CTMCs

Definition 7 *A (labeled) CTMC $\mathcal{M}$ is a tuple $(\mathcal{S}, \mathbf{R}, Init, \mathcal{A}, L)$ where:*

- *$\mathcal{S}$ is the* state space *(a set of states), which in our case is generated from a discrete-state model $(\widehat{\mathcal{S}}, \mathcal{S}_{init}, \mathcal{E}, \mathcal{N})$ and $\mathcal{S} \subseteq \widehat{\mathcal{S}} = \mathcal{S}_L \times \cdots \mathcal{S}_1$.*

$EUtrad(in\ Sat(\phi), Sat(\psi))$: set of state	$EGtrad(in\ Sat(\phi))$: set of state
1 declare $\mathcal{X}$: set of states	1 declare $\mathcal{X}$: set of states
2 $\mathcal{X} \leftarrow Sat(\psi)$;	2 $\mathcal{X} \leftarrow Sat(\phi)$;
3 repeat	3 repeat
4 $\mathcal{X} \leftarrow \mathcal{X} \cup (\mathcal{P}(\mathcal{X}) \cap Sat(\phi))$;	4 $\mathcal{X} \leftarrow \mathcal{X} \cap \mathcal{P}(\mathcal{X})$;
5 until $\mathcal{X}$ does not change;	5 until $\mathcal{X}$ does not change;
6 return $\mathcal{X}$;	6 return $\mathcal{X}$;

Figure 2.11: Traditional CTL model checking algorithms.

- $\mathbf{R} : \mathcal{S} \times \mathcal{S} \to \mathbb{R}^{\geq 0}$ *is the* transition rate matrix. *When in state* $\mathbf{i}$, $\mathcal{M}$ *remains in* $\mathbf{i}$ *for an exponentially distributed amount of time with rate* $\mathbf{E}(\mathbf{i}) = \sum_{\mathbf{j} \neq \mathbf{i}} \mathbf{R}[\mathbf{i}, \mathbf{j}]$, *then it moves to state* $\mathbf{j}$ *with probability* $\mathbf{R}[\mathbf{i}, \mathbf{j}]/\mathbf{E}(\mathbf{i})$.

- $Init : \mathcal{S} \to [0, 1]$ *is the* initial distribution. $Init(\mathbf{i})$ *is the probability that* $\mathcal{M}$ *is state* $\mathbf{i}$ *at time* 0, *thus* $\sum_{\mathbf{i} \in \mathcal{S}} Init(\mathbf{i}) = 1$.

- $\mathcal{A}$ *is a set of* atomic propositions.

- $L : \mathcal{S} \to 2^{\mathcal{A}}$ *is a* labeling. $L(\mathbf{i})$ *are the atomic propositions holding in state* $\mathbf{i}$. *We still use* ϕ *and* ψ *to denote propositions.*

- *An ordinary CTMC is similarly defined, without the* $\mathcal{A}$ *and* L *components.*

We consider homogeneous CTMCs, in which the transition rates are independent of the time. In CTMC analysis, we often use the *infinitesimal generator matrix* $\mathbf{Q}$, which satisfies $\mathbf{Q}[\mathbf{i}, \mathbf{j}] = \mathbf{R}[\mathbf{i}, \mathbf{j}]$ for $\mathbf{i} \neq \mathbf{j}$ and $\mathbf{Q}[\mathbf{i}, \mathbf{i}] = -\mathbf{E}[\mathbf{i}]$.

In the implementation, instead of $\mathbf{Q}$, we store $\mathbf{R}$ and a vector $\mathbf{h}$ where $\mathbf{h}[\mathbf{i}] = 1/\mathbf{E}[\mathbf{i}]$ is the *expected holding time* in state $\mathbf{i}$. Symbolic approaches (including Kronecker descriptors, see discussion in Chapter 6) might actually encode a supermatrix $\widehat{\mathbf{R}} \in \mathbb{R}_{\geq 0}^{\widehat{\mathcal{S}} \times \widehat{\mathcal{S}}}$

of $\mathbf{R}$, satisfying $\widehat{\mathbf{R}}[\mathbf{i},\mathbf{j}] = 0$ for reachable states $\mathbf{i} \in \mathcal{S}$ and unreachable states $\mathbf{j} \in \widehat{\mathcal{S}} \setminus \mathcal{S}$, that is, the "reachable rows" of $\widehat{\mathbf{R}}$ do not contain additional transitions. However, $\widehat{\mathbf{R}}[\mathbf{i},\mathbf{j}]$ is not required to be zero for unreachable states $\mathbf{i}$; in particular, $\widehat{\mathbf{R}}$ can contain nonzero entries from unreachable states to reachable states, and this can complicate the numerical iterations.

We then define vector $\boldsymbol{\pi}(t) : \mathcal{S} \to [0,1]$ such that $\boldsymbol{\pi}[\mathbf{i}](t)$ is the probability that $\mathcal{M}$ is in state $\mathbf{i}$ at time t, thus $\sum_{\mathbf{i}\in\mathcal{S}} \boldsymbol{\pi}[\mathbf{i}](t) = 1$. The dynamic characterization of a CTMC is given by *Kolmogorov equation*:

$$\frac{d\boldsymbol{\pi}(t)}{dt} = \boldsymbol{\pi}(t)\mathbf{Q}. \tag{2.1}$$

There are two important types of analysis on CTMCs: steady-state analysis studies the probability distribution $\lim_{t\to\infty} \boldsymbol{\pi}(t)$, while transient analysis studies $\boldsymbol{\pi}(t)$ for a given time t.

2.3.1 Steady-state solution

Given an ergodic CTMC, in which all states are included in a single strongly connected component (SCC), the steady-state solution is independent of the initial distribution, let $\boldsymbol{\pi} \in \mathbb{R}^{\mathcal{S}}_{\geq 0}$ be the steady-state probability vector, where $\boldsymbol{\pi}[\mathbf{i}]$ is the steady-state probability of state $\mathbf{i}$. $\boldsymbol{\pi}$ can be computed as the solution of the linear system

$$\boldsymbol{\pi}\mathbf{Q} = \mathbf{0} \qquad \text{subject to } \sum_{\mathbf{i}\in\mathcal{S}} \boldsymbol{\pi}[\mathbf{i}] = 1. \tag{2.2}$$

Chapter 6 focuses on solving this linear system.

For a non-ergodic CTMC, the states can be classified as *recurrent states*, denoted by $\mathcal{S}_r$, which are states in terminal SCCs, and *transient states*, denoted by $\mathcal{S}_t$, which are not in terminal SCCs. The set of states in a terminal SCC is called a recurrent class, and we use $\mathcal{R}_1, \cdots, \mathcal{R}_m$ to denote the states in each recurrent class. Given sufficiently long time, a CTMC will eventually be in its recurrent states, which means only recurrent states have non-zero steady-state probabilities.

The first step is to compute the probability to reach each recurrent class. We ignore the time spent on each state and only consider the transition probabilities from one state to the others. A CTMC can be converted to a discrete-time Markov chain (DTMC), called embedded Markov chain (EMC) whose transition probability matrix is denoted by $\mathbf{P}_e$ and:

$$\mathbf{P}_e[\mathbf{i}, \mathbf{j}] = \mathbf{R}[\mathbf{i}, \mathbf{j}] \cdot \mathbf{h}[\mathbf{i}].$$

We make all states in $\mathcal{S}_r$ absorbing. If $\mathbf{i} \in \mathcal{S}_r$, $\mathbf{P}_e[\mathbf{i}, \mathbf{i}] = 1$. $\mathbf{P}_e$ can be block-partitioned as

$$\mathbf{P}_e = \left[\begin{array}{c|c} \mathbf{P}_{\mathcal{S}_t\mathcal{S}_t} & \mathbf{P}_{\mathcal{S}_t\mathcal{S}_r} \\ \hline \mathbf{0} & \mathbf{I} \end{array}\right]$$

so that

$$\lim_{k\to\infty} \mathbf{P}_e^k = \left[\begin{array}{c|c} \mathbf{0} & \sum_{k=0}^{\infty}(\mathbf{P}_{\mathcal{S}_t\mathcal{S}_t}^k)\mathbf{P}_{\mathcal{S}_t\mathcal{S}_r} \\ \hline \mathbf{0} & \mathbf{I} \end{array}\right],$$

where $\mathbf{I}$ is the identity matrix. $\boldsymbol{\pi}(0)\sum_{k=0}^{\infty}(\mathbf{P}_{\mathcal{S}_t\mathcal{S}_t}^k)\mathbf{P}_{\mathcal{S}_t\mathcal{S}_r}$ give the probability of reaching each recurrent state, denoted by $Prob(\mathbf{i})$, where $\mathbf{i}$ is the destination recurrent state.

The steady-state solution for a non-ergodic CTMC is dependent to the initial distribution, because the initial distribution determines the probability to reach each recurrent

class. We first assume there is a single inital state $\mathbf{i}_0$ in the CTMC, and the steady-state solution can be computed by following steps:

1. Enumerate all recurrent classes $\mathcal{R}_1, \cdots, \mathcal{R}_m$.

2. Compute the steady-state solution in each recurrent class; let $\boldsymbol{\pi}_k[\mathbf{i}]$ be the steady-state solution on state $\mathbf{i}$ in $\mathcal{R}_k$;

3. Compute the probability of reaching each recurrent class $\mathcal{R}_k$, denoted by $Prob(\mathcal{R}_k)$. $Prob(\mathcal{R}_k) = \sum_{\mathbf{i} \in \mathcal{R}_k} Prob(\mathbf{i})$.

4. The steady-state probability on state $\mathbf{k} \in \mathcal{R}_k$, $\boldsymbol{\pi}[\mathbf{k}]$, is given by $\boldsymbol{\pi}_k[\mathbf{k}] \cdot Prob(\mathcal{R}_k)$.

Chapter 4 introduces the algorithm to enumerate terminal SCCs.

2.3.2 Transient analysis

Transient solution can be obtained by solving the differential equation 2.1. Alternatively, we consider a widely adopted method called *uniformization*.

Definition 8 *Given $\mathcal{M} = (\mathcal{S}, \mathbf{R}, Init)$ and a rate $q > max_{\mathbf{i} \in \mathcal{S}}\{\mathbf{E}(\mathbf{i})\}$, the* uniformization *of $\mathcal{M}$ is a discrete-time Markov chain (DTMC) $\mathcal{M}_{unif} = (\mathcal{S}, \mathbf{P}, Init)$, where $\mathbf{P}[\mathbf{i}, \mathbf{j}] = \mathbf{R}[\mathbf{i}, \mathbf{j}]/q$ for $\mathbf{i} \neq \mathbf{j}$ and $\mathbf{P}[\mathbf{i}, \mathbf{i}] = 1 - \sum_{\mathbf{j} \in \mathcal{S} \setminus \{\mathbf{i}\}} \mathbf{P}[\mathbf{i}, \mathbf{j}]$.*

$$\mathbf{P}_u = \mathbf{I} + \frac{1}{q}\mathbf{Q} \ .$$

Instead of solving Equation 2.1

$$\boldsymbol{\pi}(t) = \sum_{k=0}^{\infty} \boldsymbol{\pi}(0)\mathbf{P}_u^k \cdot Poisson[k],$$

where $Poisson[k] = e^{-qt}\frac{(qt)^k}{k!}$. Given an acceptable error ϵ, the infinite sum can be truncated and only computed on the range $[L_\epsilon, R_\epsilon]$. [38] introduced an algorithm for precisely computing L_ϵ, R_ϵ and Poisson probabilities within this range.

While $\boldsymbol{\pi}(t)$ describes the probability distribution of $\mathcal{M}$ at time t starting from a given initial distribution at time $t = 0$, model checking often requires us to "go backwards": given a "target" state $\mathbf{j}$, we need to compute vector $\boldsymbol{\nu}(\mathbf{j}, t) : \mathcal{S} \to [0, 1]$, where $\boldsymbol{\nu}[\mathbf{i}](\mathbf{j}, t)$ is the probability of reaching $\mathbf{j}$ at time t starting from state $\mathbf{i}$ at time 0, so that, if $Init(\mathbf{i}) = 1$, then $\boldsymbol{\nu}[\mathbf{i}](\mathbf{j}, t) = \pi[\mathbf{j}](t)$. Note that $\boldsymbol{\nu}(\mathbf{j}, t)$ is not a probability distribution but a vector of probabilities, i.e., its elements do not sum to 1 in general. Finally, define $\boldsymbol{\nu}(\mathcal{X}, t) = \sum_{\mathbf{j}\in\mathcal{X}} \boldsymbol{\nu}(\mathbf{j}, t)$.

For $\boldsymbol{\nu}(\mathcal{X}, t)$, there is instead the "backward solution" [47]:

$$\boldsymbol{\nu}(\mathcal{X}, t) = \sum_{k=0}^{\infty} \mathbf{P}^k \boldsymbol{\delta}^{\mathcal{X}} \cdot Poisson[k],$$

where $\boldsymbol{\delta}^{\mathcal{X}} : \mathcal{S} \to \{0, 1\}$ is the indicator vector satisfying $\delta^{\mathcal{X}}[\mathbf{i}] = 1$ iff $\mathbf{i} \in \mathcal{X}$.

To summarize this section, two matrices can be induced from transition rate matrix $\mathbf{R}$ in CTMC $\mathcal{M}$: $\mathbf{P}_e$ from the embedded Markov chain and $\mathbf{P}_u$ from the uniformization. In the following discussion, we omit the subscripts and refer to one of the matrices without causing confusions: when considering transient analysis, $\mathbf{P}$ refers to $\mathbf{P}_u$ and when considering steady-state analysis, $\mathbf{P}$ refers to $\mathbf{P}_e$.

In the implementation, an EV*MDD $\mathcal{M}$ encodes a transition rate matrix $\mathbf{R}$ while an EV+MDD $\mathcal{I}$ encodes a function mapping each state in $\mathcal{S}$ to a unique integer index in $\{0, ..., |\mathcal{S}| - 1\}$ and each state in $\widehat{\mathcal{S}} \setminus \mathcal{S}$ to ∞ (strictly speaking, $\mathcal{M}$ is $\langle \rho_{max}, r \rangle$ where ρ_{max} is

the largest entry in $\mathbf{R}$ and r is an EV*MDD node, while $\mathcal{I}$ is $\langle 0,o\rangle$ where o is an EV$^+$MDD node). Fig. 2.12 shows an example of CTMC and its $\mathcal{M}$ and $\mathcal{I}$.

2.3.3 CSL

Continuous stochastic logic (CSL) [4] extends CTL to probability and timing aspects. Instead of discrete-state systems, CSL formulas are defined on CTMCs. There are two temporal operators, U ("until") and X ("next"), in CSL, while the path quantifiers E and A in CTL are replaced by the probabilistic operators regarding the steady state and transient solution of the CTMC.

A run of $\mathcal{M}$, σ, is an infinite timed path: $(\mathbf{i}_0, t_0) \rightarrow (\mathbf{i}_1, t_1) \rightarrow \cdots$, where $\mathbf{i}_k$ is the state entered at time t_k and $t_0 = 0$. Let $\sigma[k]$ be $\mathbf{i}_k$, and $\tau[k] = t_{k+1} - t_k$ be the length of the k^{th} sojourn time, in $\sigma[k]$. The state of $\mathcal{M}$ at time t is $\sigma@t = \sigma[k]$ iff $t_k \leq t < t_{k+1}$.

We adopt the definition of CSL in [4]. Let $p \in [0,1]$ be a probability, $\bowtie$ be one of the $\{\leq, <, \geq, >\}$ operators, and $I \subset \mathbb{R}_{\geq 0}$ be a nonempty interval of the following types.

- time-bounded: $[0,t], t > 0$;
- unbounded: $[0,\infty)$, so that $\mathsf{X}^{[0,\infty)}$ and $\mathsf{U}^{[0,\infty)}$ are simply written as X and U;
- point-interval: $[t,t], t \geq 0$;
- general interval: $[t,t'], t' > t > 0$.

The syntax of CSL formulas over a set of atomic propositions $\mathcal{A}$ is defined inductively as follows:

- Each atomic proposition $\phi \in \mathcal{A}$ is a CSL formula.

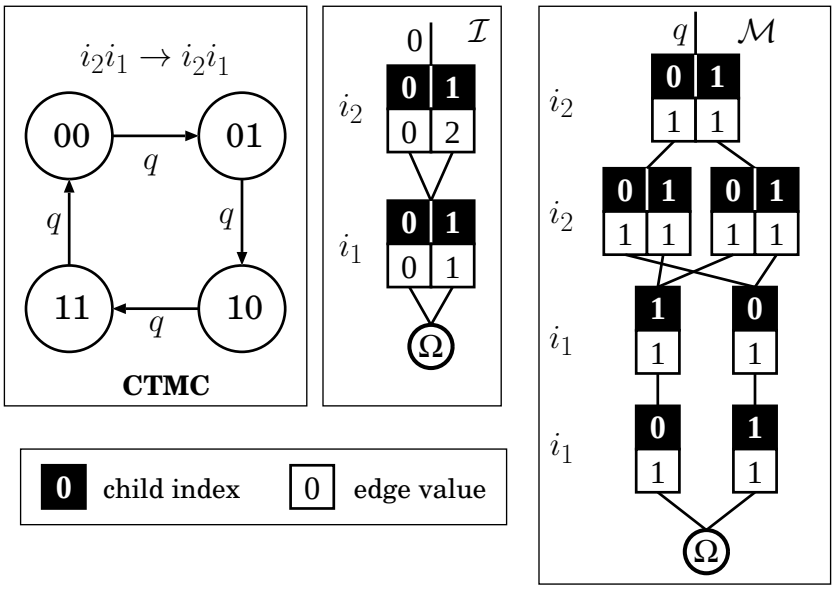


Figure 2.12: Encoding a CTMC with an EV$^+$MDD and an EV*MDD.

- If $\mathscr{F}_1$ and $\mathscr{F}_2$ are state formulas, so are $\neg\mathscr{F}_1$, $\mathscr{F}_1 \wedge \mathscr{F}_2$, $\mathbb{S}_{\bowtie p}(\mathscr{F}_1)$, $\mathbb{P}_{\bowtie p}(\mathsf{X}^I \mathscr{F}_2)$, and $\mathbb{P}_{\bowtie p}(\mathscr{F}_1 \mathsf{U}^I \mathscr{F}_2)$

In CSL, a path formula Φ is in one of the two forms: $\mathsf{X}^I\mathscr{F}$ or $\mathscr{F}_1\mathsf{U}^I\mathscr{F}_2$. $Prob(\mathbf{i}, \Phi)$ is the probability measure of all paths satisfying Φ starting from the state $\mathbf{i}$.

The semantics of $\mathbb{P}$ and $\mathbb{S}$ operators is defined as follows [4]:

- Given CSL formula $\mathscr{F}$, $\mathbf{i} \models \mathbb{S}_{\bowtie p}(\mathscr{F})$ iff from the initial distribution $\boldsymbol{\pi}(0)$ satisfying $\boldsymbol{\pi}[\mathbf{i}](0) = 1$ and $\boldsymbol{\pi}[\mathbf{j}](0) = 0$ for all $\mathbf{j} \neq \mathbf{i}$, the consequent steady-state solution $\boldsymbol{\pi}$ satisfies $\boldsymbol{\pi}(Sat(\phi)) \bowtie p$, where $\boldsymbol{\pi}(Sat(\phi)) = \sum_{\mathbf{i} \in Sat(\phi)} \boldsymbol{\pi}[\mathbf{i}]$.

- Given path formula Φ, $\mathbf{i} \models \mathbb{P}_{\bowtie p}(\Phi)$ iff $Prob(\mathbf{i}, \Phi) \bowtie p$.

The relation $\models$ for CSL path formulas in defined by:

- $\sigma \models \mathsf{X}^I\psi$ iff $\sigma[1] \models \psi \wedge \delta(\sigma, 0) \in I$.

- $\sigma \models \phi \mathsf{U}^I \psi$ iff $\exists t \in I.(\sigma@t \models \psi \wedge (\forall t' \in [0,t).\sigma@t' \models \phi)$.

2.3.4 CSL model checking

The $\mathbb{P}$ operator

This section reviews model checking algorithms for the until operator. The result of this formula is a set of states whose probabilities satisfy $\bowtie p$ in $\mathbb{P}_{\bowtie p}$. The key step is to calculate the vector $\mathbf{Prob}(\phi \mathsf{U}^I \psi)$.

Time-bounded and unbounded U.

We first introduce a conversion of $\mathcal{M} = (\mathcal{S}, \mathbf{R}, \mathit{Init})$ into $\mathcal{M}^{\mathcal{X}} = (\mathcal{S}, \mathbf{R}^{\mathcal{X}}, \mathit{Init})$, where states in $\mathcal{X}$ are absorbing, and then into the uniformized DTMC

$$\mathbf{R}^{\mathcal{X}}[\mathbf{i},\mathbf{j}] = \mathbf{R}[\mathbf{i},\mathbf{j}] \quad \text{and} \quad \mathbf{P}_u^{\mathcal{X}}[\mathbf{i},\mathbf{j}] = \mathbf{R}[\mathbf{i},\mathbf{j}]/q \text{ if } \mathbf{i} \neq \mathbf{j} \text{ else } 1 - \mathbf{E}[\mathbf{i}]/q \quad \text{if} \quad \mathbf{i} \notin \mathcal{X};$$

$$\mathbf{R}^{\mathcal{X}}[\mathbf{i},\mathbf{j}] = 0 \quad \text{and} \quad \mathbf{P}_u^{\mathcal{X}}[\mathbf{i},\mathbf{j}] = 0 \quad \text{if } \mathbf{i} \neq \mathbf{j} \text{ else } 0^1 \quad \text{if} \quad \mathbf{i} \in \mathcal{X}.$$

We simplify the notation $\mathbf{P}^{\mathcal{X}}$ to $\mathbf{P}$ when $\mathcal{X}$ is clear in the context.

We first discuss the time-bounded and unbounded until operator. Before numerical analysis, we partition $\mathcal{S}$ into three sets of states:

$$\mathcal{S}_1 = \{\mathbf{i} \in \mathcal{S} \mid \mathbf{i} \models \psi\}, \quad \mathcal{S}_0 = \{\mathbf{i} \in \mathcal{S} \mid \mathbf{i} \not\models \mathsf{E}\phi\mathsf{U}\psi\}, \quad \text{and } \mathcal{S}_? = \{\mathbf{i} \in \mathcal{S} \mid \mathbf{i} \models (\mathsf{E}\phi\mathsf{U}\psi) \setminus \mathcal{S}_1\}.$$

For states in $\mathcal{S}_0$, which do not satisfy $\mathsf{E}\phi\mathsf{U}\psi$, the probability is 0 for sure, while for states in $\mathcal{S}_1$, which satisfy ψ, this probability is 1. Thus, we have

$$\mathbf{i} \in \mathcal{S}_1 \Rightarrow \forall t > 0, \mathit{Prob}(\mathbf{i}, \phi \mathsf{U}^{[0,t]} \psi) = 1 \text{ and } \mathbf{i} \in \mathcal{S}_0 \Rightarrow \forall t > 0, \mathit{Prob}(\mathbf{i}, \phi \mathsf{U}^{[0,t]} \psi) = 0$$

[1] While, conceptually, $\mathbf{P}^{\mathcal{X}}[\mathbf{i},\mathbf{i}]$ should be 1 for absorbing state $\mathbf{i}$, we set it to 0 here to simplify Equation 2.3.

so we only need to calculate the probabilities for states in $\mathcal{S}_?$.

Computing the probability vector $\mathbf{Prob}(\phi \mathsf{U}^{[0,t]} \psi)$ on $\mathcal{M}$ is equivalent to computing $\boldsymbol{\nu}(\mathcal{S}_1, t)$ on $\mathcal{M}^{\mathcal{S}_0 \cup \mathcal{S}_1}$, which we can do using transient analysis and uniformization. A backward approach [47] is more desirable because it directly returns the vector $\boldsymbol{\nu}(\mathcal{S}_1, t)$, its pseudo-code is shown in Figure 2.13.

For unbouned until, we introduce the following transition matrix $\mathbf{P}_e$ for the embedded Markov chain:

$$\mathbf{R}^{\mathcal{X}}[\mathbf{i},\mathbf{j}] = \mathbf{R}[\mathbf{i},\mathbf{j}] \quad \text{and} \quad \mathbf{P}_e^{\mathcal{X}}[\mathbf{i},\mathbf{j}] = \mathbf{R}[\mathbf{i},\mathbf{j}]/\mathbf{E}[\mathbf{i}] \text{ if } \mathbf{i} \neq \mathbf{j} \text{ else } 0 \quad \text{if} \quad \mathbf{i} \notin \mathcal{X};$$

$$\mathbf{R}^{\mathcal{X}}[\mathbf{i},\mathbf{j}] = 0 \quad \text{and} \quad \mathbf{P}_e^{\mathcal{X}}[\mathbf{i},\mathbf{j}] = 0 \quad \text{if } \mathbf{i} \neq \mathbf{j} \text{ else } 0 \quad \text{if} \quad \mathbf{i} \in \mathcal{X}.$$

$\mathbf{Prob}(\phi \ \mathsf{U} \ \psi) = \boldsymbol{\nu}(\mathcal{S}_1)$ where $\boldsymbol{\nu}(\mathcal{S}_1) = \lim_{t\to\infty} \boldsymbol{\nu}(\mathcal{S}_1, t)$ is the solution of the linear system:

$$\mathbf{P}_e \cdot \boldsymbol{\nu} + \mathbf{I}^{\psi} = \boldsymbol{\nu}, \tag{2.3}$$

where $\mathbf{I}^{\psi}[\mathbf{i}] = 1$ iff $\mathbf{i} \in Sat(\psi)$ and $\mathbf{I}^{\psi}[\mathbf{i}] = 0$ otherwise.

Given the potentially huge state space, direct methods such as Gaussian elimination do not scale, thus iterative methods are normally employed. For example, the Gauss-Seidel iteration

$$\boldsymbol{\nu}^{(k+1)}[\mathbf{i}] = \sum_{\mathbf{j} \prec \mathbf{i}} \mathbf{P}_e[\mathbf{i},\mathbf{j}] \boldsymbol{\nu}^{(k+1)}[\mathbf{j}] + \sum_{\mathbf{j} \succ \mathbf{i}} \mathbf{P}_e[\mathbf{i},\mathbf{j}] \boldsymbol{\nu}^{(k)}[\mathbf{j}] + \mathbf{I}^{\psi}[\mathbf{i}] \tag{2.4}$$

can be used to converge to a (numerically close) answer.

Point-interval and general interval U.

The algorithms for both operators employ transient analysis, similar to that for time-bounded U. For $\mathbb{P}_{\bowtie p}(\phi \mathsf{U}^{[t,t]} \psi)$, we redefine ψ as $\psi \wedge \phi$, since the probability of moving

```
mdd  BoundedUntil(A φ, A ψ) is
 1  p, L_ε, R_ε ← FoxGlynn(qt, ε);        • Poisson probability, left and right bound w.r.t ε
 2  ν ← 0;     b ← I^ψ                    • I^ψ[i] = 1 iff i ∈ Sat(ψ)
 3  for k = 1 to R_ε − 1 do
 4     b ← P · b;                         • P: transition matrix for M^{φUψ}_{unif}
 5     if k ≥ L_ε then ν ← ν + p[k] · b;
 6  endfor;
 7  return ν;
```

Figure 2.13: Backward computation of the probability vector for $\phi \mathsf{U}^{[0,t]} \psi$.

from $Sat(\phi)$ to $Sat(\psi \wedge \neg\phi)$ exactly at time t is obviously 0. We can then generate $\mathcal{M}_{unif}^{Sat(\neg\phi)}$ and obtain the desired $\mathbf{Prob}(\phi \mathsf{U}^{[t,t]} \psi)$ as the vector $\boldsymbol{\nu}(Sat(\psi), t)$ computed for $\mathcal{M}_{unif}^{Sat(\neg\phi)}$.

$\mathbb{P}_{\bowtie p}(\phi \mathsf{U}^{[t,t']} \psi)$ requires two rounds of transient analysis: first we generate $\mathcal{M}_{unif}^{Sat(\neg\phi \vee \psi)}$ and compute $\boldsymbol{\nu}_{last}(Sat(\psi), t'-t)$, the probability of reaching ψ states within the last $t'-t$ time units starting from each state; then we generate $\mathcal{M}_{unif}^{Sat(\neg\phi)}$ with $\mathbf{P} = \mathbf{P}^{Sat(\neg\phi)}$ and compute:

$$\sum_{k=L_\varepsilon}^{R_\varepsilon} \mathbf{P}^k \boldsymbol{\nu}_{last}(Sat(\psi), t'-t) \cdot Poisson[k],$$

which is a backward transient analysis similar to that in Fig. 2.13 but starting from the probability vector $\boldsymbol{\nu}_{last}(Sat(\psi), t'-t)$ instead of $\boldsymbol{\delta}^\psi$.

The $\mathbb{S}$ operator.

The $\mathbb{S}$ operator requires a steady-state solution. If the CTMC is ergodic, thus the steady-state solution is independent of the initial distribution, either all states or no state satisfying $\mathbb{S}_{\bowtie p}(\phi)$, depending on whether $\boldsymbol{\pi}(Sat(\phi)) \bowtie p$ holds.

For reducible CTMCs, which contains more than one terminal SCC in the transition graph, we have to check whether $\mathbb{S}_{\bowtie p}(\phi)$ on each recurrent class one by one. The first

step is to enumerate recurrent classes $\mathcal{R}_1, \cdots, \mathcal{R}_m$. We then solve the steady-state solution for each recurrent class and let $\boldsymbol{\pi}_n$ be the steady-state solution for recurrent class n.

For $\mathbf{i} \in \mathcal{R}_k$, $\mathbf{i} \in \mathbb{S}_{\bowtie p}(\phi)$ iff

$$\boldsymbol{\pi}_n(Sat(\phi)) \bowtie p,$$

and for $\mathbf{i} \in \mathcal{S}_t$, $\mathbf{i} \in \mathbb{S}_{\bowtie p}(\phi)$ iff

$$\sum_n Prob(\mathbf{i}, \mathcal{R}_n)\boldsymbol{\pi}_n(Sat(\phi)) \bowtie p.$$

Operator	Algorithm
$\mathbb{S}_{\bowtie p}(\phi)$	(ergodic CTMCs) solve linear system $\boldsymbol{\pi}\mathbf{Q} = 0$
	(reducible CTMCs) solve linear system $\boldsymbol{\pi}\mathbf{Q} = 0$ on each terminal SCCs, and then solve the probabilistic reachability on transient states
$\mathbb{P}_{\bowtie p}(\phi \mathsf{U}^{[0,t]} \psi)$	use uniformization and then transient analysis in Figure 2.13
$\mathbb{P}_{\bowtie p}(\phi \mathsf{U} \psi)$	solve linear system $\mathbf{P}\boldsymbol{\nu} + \mathbf{B} = \boldsymbol{\nu}$
$\mathbb{P}_{\bowtie p}(\phi \mathsf{U}^{[t,t']} \psi)$	Single or multiple rounds of transient analysis

Table 2.1: Algorithms for model checking CSL [4].

Table 2.1 summarize the CSL model checking algorithms. As it shows, both the steady-state solution and the unbounded until operator require to solve linear systems, which will be addressed in Chapter 6. Chapter 7 will discuss truncation errors for operators $\mathsf{U}^{[0,t]}$ and $\mathsf{U}^{[t,t']}$.

Part I

Logic model checking

Chapter 3

Constrained saturation and CTL model checking

CTL model checking is an important state-of-the-art approach in formal verification. Paired with the use of BDDs [8], which provide a time and space efficient data structure to perform operations such as union, intersection, and relational product over sets of states, symbolic model checking [56] is one of the most successful techniques to verify industrial hardware and embedded software systems.

Mainstream symbolic model checkers, such as NuSMV [25], employ methods based on breath-first search (BFS). The saturation algorithm employs a very different philosophy, recursively computing "local fixpoints". A series of publications has proven the clear advantages of saturation for state-space generation over traditional symbolic approaches [17, 19, 20, 77], while extending its applicability to increasingly general settings.

A saturation-based EU computation algorithm was instead proposed in [22] (see *EUsat* in Figure 3.1). First, it partitions the set $\mathcal{E}$ of events into safe, $\mathcal{E}_S$, and unsafe, $\mathcal{E}_U = \mathcal{E} \setminus \mathcal{E}_S$, where $\alpha \in \mathcal{E}$ is safe iff $\mathcal{P}_\alpha(Sat(\phi) \cup Sat(\psi)) \subseteq Sat(\phi)$, i.e., it is such that, following its firing backwards, we can only find states in $Sat(\phi)$ (alternatively, we can restrict all sets by intersecting them with the reachable states $\mathcal{S}_{rch}$ in the above test). The algorithm iteratively (1) saturates the MDD encoding the set of explored states using only safe events, then (2) fires each unsafe event once using $\mathcal{P}_U = \bigcup_{\alpha \in \mathcal{E}_U} \mathcal{P}_\alpha$ in breadth-first fashion, then (3) intersects the result with $Sat(\phi)$, and finally (4) adds the result to the working set $\mathcal{X}$. However, this attempt have been only partially successful since it is not general enough. If there is no safe event in the system for a given EU property, then this algorithm degrades to normal BFS-based algorithm. [22] also attempts to compute set of states satisfying $\mathsf{EG}\phi$ using forward and backward EU saturation from a single state in $Sat(\phi)$. However, this approach is more efficient than the traditional algorithm only in very special cases.

This chapter addresses CTL model checking for asynchronous systems by proposing an extended *constrained saturation* algorithm. This algorithm constrains the saturation-based state-space exploration to a given set of states without explicitly executing the expensive intersection operations normally required to implement CTL operators. Furthermore, unlike the original approach [22] where the next-state function had to satisfy a Kronecker expression, the proposed algorithm is fully general, as it employs a disjunctive-then-conjunctive encoding that exploits the common characteristics of asynchronous systems. Constrained saturation can be used to compute the set of states satisfying an EU formula

```
EUsat(in Sat(φ), Sat(ψ)): set of state
 1  declare X, Y: set of state; E_U, E_S: set of event;
 2  ClassifyEvents(Sat(φ) ∪ Sat(ψ), E_U, E_S)
 3  X ← Sat(ψ);
 4  Saturate(X, E_S)
 5  repeat
 6    Y ← X;
 7    X ← X ∪ (P_U(X) ∩ (Sat(φ) ∪ Sat(ψ)))
 8    if X ≠ Y then
 9      Saturate(X, E_S)
10  until X = Y;
11  return X;
```

Figure 3.1: Saturation-based EU model checking algorithms.

as well as to efficiently compute the *backward transitive closure*, which we in turn use for a new algorithm to compute the set of states satisfying an EG formula.

The remainder of this chapter is organized as follows. Section 3.1 introduces constrained saturation and new EU computation algorithm. Section 3.2 proposes a new EG computation algorithm based on the backward transitive closure. We conclude this chapter and outline future work in the last section.

3.1 Constrained saturation for the EU operator

The set of states satisfying $\mathsf{E}\phi\mathsf{U}\psi$ is a least fixpoint, where the saturation algorithm could be efficiently employed. However, the challenge arises from the need to "filter out" the states not in $Sat(\phi)$ before exploring their predecessors. Failure to do so can results in paths which temporarily go out of $Sat(\phi)$, so that the result may include some states not satisfying $\mathsf{E}\phi\mathsf{U}\psi$. The saturation algorithm does not find states in breadth-first-search order, as the process of saturating a node often consists of firing a series of events. Performing

an expensive intersection operation *after each firing* would be enormously less time and memory efficient.

The advantage of Algorithm *EUsat* [22] over *EUtrad* depends on the structure of the model. If there are no safe events with respect to a given property ϕ, *EUsat* degrades to the simple breadth-first exploration of *EUtrad*. To overcome this difficulty, we propose two approaches, both aimed at exploring only states in $Sat(\phi)$ without requiring an expensive intersection operation after each firing.

1. Saturation with constrained next-state functions.

For each $\mathcal{P}_k$, we build a *constrained inverse next-state function* $\mathcal{P}_{k,Sat(\phi)}$ such that

$$\mathbf{j} \in \mathcal{P}_{k,Sat(\phi)}(\mathbf{i}) \iff (\mathbf{j} \in Sat(\phi)) \wedge \mathbf{j} \in \mathcal{P}_k(\mathbf{i}).$$

Algorithm *ConsNSF* in Figure 3.2 builds the MDD representation of $\mathcal{P}_{\alpha,Sat(\phi)}$.

2. Constrained saturation.

This is the main contribution of our chapter. We do not explicitly constrain the next-state functions, but perform instead a "check-and-fire" step when computing the constrained preimage (function *ConsRelProd* in the pseudocode of Figure 3.3), based on the following observation:

$$\mathcal{B}(t) = RelProd(s,r) \cap \mathcal{B}(a) \iff \mathcal{B}(t[i']) = \bigcup_{i \in \mathcal{S}_l} RelProd(s[i], r[i][i']) \cap \mathcal{B}(a[i']), \tag{3.1}$$

where t and s are L-level MDDs encoding sets of states, $l = s.lvl$, and r is a $2L$-level encoding a next-state function. This can be considered as a form of *ITE* operator [8], widely used

in BDD operations, but extended from boolean to integer variables. The overall process of EU computation based on constrained saturation is then shown in Figure 3.3. The key differences from the saturation algorithm in [20] are marked with a "$\star$".

The idea of the first approach is straightforward: all constrained next-state functions $\mathcal{P}_{\alpha,Sat(\phi)}$ are forced to be safe by definition. According to the saturation-based EU computation algorithm in Figure 2.11, the result is obtained in a single call to *Saturate*. The downside of this approach is a possible decrease in locality. A property ϕ is *dependent* on level k if the value of i_k affects the satisfiability of ϕ, i.e., if the (fully-reduced) encoding of ϕ has nodes at level k. After constraining a next-state function $\mathcal{N}_\alpha$ with ϕ, the levels on which ϕ depends become part of the support, thus, $Top(\mathcal{P}_{\alpha,Sat(\phi)}) = \max\{Top(\phi), Top(\mathcal{P}_\alpha)\}$. If $Sat(\phi)$ depends on level L, all events belong to $\mathcal{E}_L$ and the saturation algorithm degrades to BFS, losing its advantages.

The second approach, constrained saturation, does not modify the transition relation explicitly, but constrains the state exploration "on-the-fly" following the "check-and-fire" policy. This policy guarantees that the state exploration is constrained to set $Sat(\phi)$. At the same time, it retains the advantages of saturation due to exploiting event locality and employing recursive local fixpoint computations. In the pseudocode shown in Figure 3.3, assuming p is the MDD encoding the constraint, the "check-and-fire" policy can be summarized into two cases:

1. If $p[i] = \mathbf{0}$, $s[i]$ is kept unchanged without adding new states (line 4 in function *ConsSaturate*).

```
mdd  EUsatConsNSF(mdd Sat(φ), mdd Sat(ψ))
 1  foreach α ∈ ℰ do P_{α,Sat(φ)} ← ConsNSF(Sat(φ), P_α);
 2  mdd s ← Saturate(Sat(ψ));
 3  s ← intersection(s, S_rch);
 4  return s;
```

```
mdd  ConsNSF(mdd a, mdd r)
 1  if a = 1 and r = 1 then return 1;
 2  if InCache_ConsNSF(a,r,t) then return t;
 3  mdd t ← 0;   level lr ← r.lvl;   level la ← a.lvl;
 4  if lr < la then
 5    foreach i ∈ S_la s.t. a[i] ≠ 0 do t[i][i] ← ConsNSF(a[i], r);
 6  else if lr > la then
 7    foreach i, i' ∈ S_lr s.t. r[i][i'] ≠ 0 do t[i][i'] ← ConsNSF(a, r[i][i']);
 8  else                                                          • lr = la
 9    foreach i,i' ∈ S_lr s.t. r[i][i'] ≠ 0 and a[i'] ≠ 0 do t[i][i'] ← ConsNSF(a[i'],r[i][i']);
10  CacheAdd_ConsNSF(a, r, t);
11  return t;
```

Figure 3.2: Saturation using a constrained next-state function (*EUsatConsNSF*).

2. When computing the relational product, check whether the newly generated local state is included in p on each level (line 7 in function *ConsSaturate* and line 5 in function *ConsRelProd*). If instead $p[i'] = \mathbf{0}$ in formula (3.1), the relational product stops the recursive execution and returns $\mathbf{0}$.

Another tradeoff affecting efficiency is how to select the set of states $Sat(\phi)$ when checking $\mathsf{E}\phi\mathsf{U}\psi$. In high-level models, $Sat(\phi)$ is often associated with an atomic property, e.g., "place a of the Petri Net is empty" or a "localized" property dependent on just a few levels. There are then two reasonable choices to define $Sat(\phi)$:

- $Sat(\phi) = Sat(\phi)_{pot}$: include all states in the potential state space $\mathcal{S}$ that satisfy the given property, even if they are not reachable (recall that the potential state space is finite because the bound for each local state space $\mathcal{S}_k$ is known).

```
mdd  EUconsSat(mdd a, mdd b)                    • a: the constraint; b: the set being saturated
 1  mdd s ← ConsSaturate(a, b);
 2  s ← intersection(s, S_rch);
 3  return s;
```

```
mdd  ConsSaturate(mdd a, mdd s)                 • a: the constraint; s: the set being saturated
 1  if InCache_ConsSaturate(a, s, t) then return t;
 2  level l ← s.lvl;   mdd t ← NewNode(l);   mdd r ← P_l;
 3  foreach i ∈ S_l s.t. s[i] ≠ 0 do
 4 ⋆   if a[i'] ≠ 0 then t[i] ← ConsSaturate(a[i], s[i]);   else t[i] ← s[i];
 5  repeat
 6    foreach i, i' ∈ S_l s.t. r[i][i'] ≠ 0 do
 7 ⋆    if a[i'] ≠ 0 then
 8       mdd u ← ConsRelProd(a[i'],t[i],r[i][i']);   t[i'] ← Or(t[i'], u);
 9  until t does not change;
10  t ← UniqueTableInsert(t);   CacheAdd_ConsSaturate(a, s, t);
11  return t;
```

```
mdd  ConsRelProd(mdd a, mdd s, mdd r)
 1  if s = 1 and r = 1 then return 1;
 2  if InCache_ConsRelProd(a, s, r, t) then return t;
 3  level l ← s.lvl;   mdd t ← 0;
 4    foreach i, i' ∈ S_l s.t. r[i][i'] ≠ 0 do
 5 ⋆    if a[i'] ≠ 0 then
 6 ⋆      mdd u ← ConsRelProd(a[i'],s[i],r[i][i']);
 7 ⋆      if u ≠ 0 then
 8 ⋆        if t = 0 then t ← NewNode(l);
 9 ⋆        t[i'] ← Or(t[i'], u);
10  t ← ConsSaturate(a, UniqueTableInsert(t));   CacheAdd_ConsRelProd(a, s, r, t);
11  return t;
```

Figure 3.3: Constrained saturation ($EUconsSat$).

- $Sat(\phi) = Sat(\phi)_{rch}$: include in $Sat(\phi)$ only the *reachable* states that satisfy the given property, $Sat(\phi)_{rch} = Sat(\phi)_{pot} \cap \mathcal{S}_{rch}$.

We are normally only interested in reachable states and, of course, backward state exploration from unreachable states can only lead to more unreachable states; all these unreachable states can be filtered out *after* saturation, without affecting the correctness of the result (unlike the discussion at the beginning of this section, pertaining to filtering out

states not in $Sat(\phi)$). Exploration including the unreachable states might result in greater time and memory requirements, in which case using $Sat(\phi)_{rch}$ is preferable for algorithmic efficiency. On the other hand, $Sat(\phi)_{pot}$ is often dependent on very few levels, while, for most models, $Sat(\phi)_{rch}$ is a strict subset of $\mathcal{S}$, thus depends on many levels, and this increases the complexity of algorithm, especially for the first approach. In the ideal case, we can constrain the state exploration to $Sat(\phi)_{rch}$ with an acceptable overhead.

The experimental results in Section 3.3 demonstrate that constrained saturation using $Sat(\phi)_{rch}$ tends to perform much better than saturation with constrained next-state functions in both runtime and memory consumption. We select it as our main method to compute the EU operator, as well as the transition closure, which we introduce in the next section.

3.2 Transitive closure and the EG operator

The $\mathsf{EG}\phi$ property describes the existence of a path in $Sat(\phi)$ from a state leading to a nontrivial strongly-connected component (SCC), where ϕ holds in all states along the path and in the SCC. In this section, we propose a new $\mathsf{EG}\phi$ computation algorithm based on the transitive closure, built using constrained saturation.

The following defines the (backward) *transitive closure* of a set of states $\mathcal{X}$ within $Sat(\phi)$, denoted with $\mathcal{T}^b_{\mathcal{X},Sat(\phi)}$ (supper-script b means "backward").

Definition 9 *Given a state* $\mathbf{i} \in \mathcal{X}$, $\mathbf{j} \in \mathcal{T}^b_{\mathcal{X},Sat(\phi)}(\mathbf{i})$ *iff there exists a nontrivial (i.e., positive length) forward path* σ *from* $\mathbf{j}$ *to* $\mathbf{i}$ *and all states in* σ *belong to* $Sat(\phi)$.

If $\mathbf{j} \in \mathcal{T}^b_{\mathcal{X},Sat(\phi)}(\mathbf{i})$, we know that $\mathbf{j}$ is in $Sat(\phi)$. Since it is not always necessary to compute the transitive closure for all $\mathbf{i} \in \mathcal{S}$, we can build the transitive closure starting only from states in $\mathcal{X}$, to reduce time and memory consumption.

Claim 1: If $\mathbf{j} \in \mathcal{T}^b_{\mathcal{S},Sat(\phi)}(\mathbf{i})$, then $\exists \mathbf{i}' \in \mathcal{P}(\mathbf{i}) \cap Sat(\phi)$ s.t. $\mathbf{j} \in ConsSaturate(Sat(\phi), \{\mathbf{i}'\})$.

This claim comes from the definition of constrained saturation and suggests a way of building the transitive closure efficiently. Starting from the MDD encoding $Sat(\phi)$, appropriately restricted to $Sat(\phi)$, we compute the constrained saturation for states encoded at the primed levels. Analogous to constrained saturation, this process can be performed bottom-up recursively on each level.

Claim 2: $\mathbf{j} \models \mathsf{EG}\phi$ iff $\exists \mathbf{i} \in Sat(\phi)$ s.t. $\mathbf{i} \in \mathcal{T}^b_{Sat(\phi),Sat(\phi)}(\mathbf{i})$ and $\mathbf{j} \in \mathcal{T}^b_{Sat(\phi),Sat(\phi)}(\mathbf{i})$.

From this claim, we can obtain an algorithm to compute the set of states satisfying $\mathsf{EG}\phi$. Given a $2L$-level MDD encoding the transitive closure, it is easy to obtain the set of states $\mathcal{S}_{scc} = \{\mathbf{i} : \mathbf{i} \in \mathcal{T}^b_{Sat(\phi),Sat(\phi)}(\mathbf{i})\}$. These states belong to SCCs where property $Sat(\phi)$ holds continuously. Then, the result of $\mathsf{EG}\phi$ can be obtained computing $RelProd(\mathcal{S}_{scc}, \mathcal{T}^b_{Sat(\phi),Sat(\phi)})$.

Building the transitive closure is a time and memory intensive task, constituting the bottleneck for our new EG algorithm. On the other hand, the transitive closure contains more information than the basic EG property and has further applications. We discuss one of them: EG computation under a weak fairness constraint. Fairness is widely used in formal specification of protocols; in particular, weak fairness specifies that there is an infinite execution on which some states, say in $\mathcal{F}$, appear infinitely often. The difficulty lies in that the fact that executions in SCCs which do not contains states in $\mathcal{F}$ must be eliminated

to guarantee the fairness, and the traditional symbolic EG algorithm cannot handle this problem. However, this extension is easy in our framework, as discussed next.

Claim 3: $\mathsf{EG}\phi$ under a weak fairness constraint $\mathcal{F}$ holds in state $\mathbf{j}$ iff $\exists \mathbf{i} \in \mathcal{F}$ s.t. $\mathbf{i} \in \mathcal{T}^b_{\mathcal{F}\cap Sat(\phi),Sat(\phi)}(\mathbf{i})$ and $\mathbf{j} \in \mathcal{T}^b_{\mathcal{F}\cap Sat(\phi),Sat(\phi)}(\mathbf{i})$.

Since $\mathbf{i} \in \mathcal{F}$, the SCCs containing such a state satisfy the fairness constraint. We only need to build transitive closure on these states. An interesting point is that many fewer state pairs are in $\mathcal{T}^b_{\mathcal{F}\cap Sat(\phi),Sat(\phi)}$ than in $\mathcal{T}^b_{Sat(\phi),Sat(\phi)}$. Although, in symbolic approaches, fewer states do not always lead to smaller MDDs, thus to lower time and memory requirements, it is often the case in our framework that considering fairness will reduce the runtime, which is quite the opposite of what happens with traditional approaches.

3.3 Experimental results

We implemented the proposed approach in SMART [16] and report on experiments run on an Intel Xeon 3.0Ghz workstation with 3GB RAM under SuSE Linux 9.1. Detailed descriptions of the models we use in the experiments can be found in the SMART User Manual [15]. The state space size for each model is controlled by a parameter N. For comparison, we study each model in both SMART and NuSMV version 2.4.3 [1].

3.3.1 Results for the EU computation

Table 3.1 shows the results for each EU query. Runtime (seconds), final (mem-f) and peak memory (mem-p) consumption (Kbytes) required by NuSMV, by the old version

of SMART [22], and by our new approach are shown in the corresponding columns, for each model. We compare the following five approaches:

- *BFS*: the traditional EU algorithm implemented in SMART
- *ConNSFSat-*$\mathcal{P}_{pot}$: Saturation using constrained next-state functions, where the next-state functions are constrained using $\mathcal{P}_{pot}$
- *ConNSFSat-*$\mathcal{P}_{rch}$: Saturation using constrained next-state functions, where the next-state functions are constrained using $\mathcal{P}_{rch}$.
- *ConSat-*$Sat(\phi)_{pot}$: Constrained saturation with $Sat(\phi)_{pot}$.
- *ConSat-*$Sat(\phi)_{rch}$: Constrained saturation with $Sat(\phi)_{rch}$.

Our main method, constrained saturation using $\mathcal{P}_{rch}$, outperforms (sometimes by orders of magnitude) NuSMV and other methods in both time and memory. In comparison with NuSMV, the saturation-based methods excel because of the local fixpoint iteration scheme. The improvement of our new work over our old approach [22] can be attributed to both the MDD encoding of the next-state function and the more advanced saturation schemes.

Overall, *ConSat* requires less runtime as well as less memory than $ConNSFSat$, because the constrained next-state functions often impose overhead on relational product operations. The difference between the results of *ConNSFSat-*$\mathcal{P}_{pot}$ and *ConNSFSat-*$\mathcal{P}_{rch}$ shows the tradeoff discussed at the end of Section 3.1. *ConSat* constrained with $Sat(\phi)_{rch}$ is more advantageous than with $Sat(\phi)_{pot}$ because *ConSat* is not sensitive to the complexity

of the constraint set due to our lightweight "check-and-fire" policy. The reduction in state exploration becomes the dominant factor for efficiency.

3.3.2 Results for the EG computation

Table 3.2 compares the results of NuSMV, BFS (SMART-BFS) and the method in Section 3.2 (SMART-RchRel) for EG computation with or without fairness constraints. For BFS, the number of iterations is listed in column *itr*.

Without fairness, traditional BFS (in NuSMV or SMART-BFS) is often orders-of-magnitude faster than our algorithm based on the reachability relation. This result is not surprising due to the time and memory complexity of building the transitive closure, even if this is done using saturation.

Another experiment is provided to show the merit of our algorithm. We build a simple model with a long path from an SCC where $\mathsf{EG}\phi$ holds to a terminal state, with ϕ holding on every state on this path. We parameterize the length of the path to control the number of iterations which a traditional EG algorithm will require to reach the fixpoint. For different numbers of iterations, from 500 to 50,000, we compare the runtimes (in seconds) of traditional (BFS) search and our algorithm (SatTR) in Figure 3.4. As the number of iterations grows, the runtime of our algorithm grows much slower than that of the traditional algorithm, due to the efficient state exploration scheme in constrained saturation.

If we consider fairness, as discussed in Section 3.2, the time and memory complexity of building the transitive closure is often reduced, while that of the traditional algorithm

Model		EU query																			
		NuSMV [1]		OldSmart [22]		SMART															
						BFS			*ConsNSFSat-$\mathcal{P}_{pot}$*			*ConsNSFSat-$\mathcal{P}_{rch}$*			*ConsSat-Sat(ϕ)$_{pot}$*			*ConsSat-Sat(ϕ)$_{rch}$*		
		sec	KB(f)	sec	KB(f)	KB(p)	sec	KB(f)	KB(p)	sec	KB(f)	KB(p)	sec	KB(f)	KB(p)	sec	KB(f)	KB(p)	sec	KB(f)	KB(p)
leader		E($pref_1 = 0$)U($status_0 = leader$)																			
	5	44.1	62,057	9.8	163	6,998	1.6	3,112	6,539	28.8	632	809	1.1	1,351	2,375	17.6	544	776	1.2	602	728
	6	304.8	180,988	296.1	463	40,429	8.3	12,175	24,080	2,022.5	1,283	2,328	19.8	3,153	8,046	1,032.9	1,473	2,284	14.9	1,054	1,337
	7	1,791.5	666,779	–	–	–	39.8	44,551	91,466	–	–	–	347.3	8,726	23,332	–	–	–	114.9	2,078	2,616
	8	–	–	–	–	–	371.5	171,867	277,649	–	–	–	–	–	–	–	–	–	4,058.2	4,110	5,139
phil.		E($phil_1 \neq eat$)U($phil_0 = eat$)																			
	50	1,644.2	75,282	< 0.1	73	287	4.7	3,248	4,586	< 0.1	464	465	0.2	1,117	1,591	0.2	436	436	< 0.1	435	435
	100	–	–	0.2	147	872	60.6	7,141	16,446	0.2	861	864	0.8	3,119	4,907	0.4	817	843	0.3	816	841
	500	–	–	2.9	740	16,082	–	–	–	0.3	3,837	3,870	160.7	52,447	87,791	0.7	3,856	3,885	0.4	3,850	3,876
	1,000	–	–	4.1	1,036	30,707	–	–	–	0.6	5,262	5,269	1,228.2	100,516	164,137	4.8	7,589	7,697	0.8	5,253	5,271
robin		E($p_1 \neq load$)U($p_0 = send$)																			
	10	29.2	70,583	5.1	186	4,447	0.1	494	574	< 0.1	92	92	< 0.1	204	204	< 0.1	90	90	< 0.1	64	64
	20	–	–	–	–	–	1.4	3,088	3,738	< 0.1	283	317	< 0.1	598	659	0.1	312	312	< 0.1	166	166
	100	–	–	–	–	–	–	–	–	5.7	14,272	14,274	636.1	17,398	30,833	3.6	15,899	15,907	< 0.1	4,269	4,298
	200	–	–	–	–	–	–	–	–	163.1	105,787	105,788	–	–	–	111.6	119,988	119,988	0.4	28,781	28,790
fms		E($M_1 > 0$)U($P_1 s = P_2 s = P_3 s = N$)																			
	10	578.9	181,353	0.1	27	628	5.2	35,979	35,980	0.4	407	477	43.2	993	2,965	< 0.1	257	288	< 0.1	256	286
	25	–	–	1.6	155	8,223	–	–	–	31.5	638	1,362	–	–	–	1.8	981	1,107	1.9	977	1,104
	50	–	–	24.0	812	75,884	–	–	–	2,566.3	1,449	6,972	–	–	–	39.1	1,247	6,018	40.5	1,246	6,006
	100	–	–	–	–	–	–	–	–	–	–	–	–	–	–	1,128.0	4,302	40,588	1,200.9	4,299	40,504
queens		E($p[N-1][N-1]$U($p[N-1][N-1][N] = 0$)																			
	10	15.3	77,861	–	–	–	1.1	9,744	9,744	< 0.1	2,728	2,728	< 0.1	1,899	1,899	0.2	3,532	3,532	< 0.1	1,841	1,841
	11	84.0	327,907	–	–	–	7.2	41,252	41,252	0.5	12,445	12,445	0.1	7,289	7,312	1.6	15,877	15,877	< 0.1	7,043	7,062
	12	595.1	1,355,128	–	–	–	237.2	186,360	186,360	2.3	52,764	52,765	0.7	80,785	80,859	9.1	75,347	75,347	< 0.1	29,714	29,763
	13	–	–	–	–	–	–	–	–	10.9	236,501	236,506	6.1	166,711	166,837	86.9	337,265	337,265	< 0.1	133,121	133,121

Table 3.1: Results for the EU computation.

Model	EG query									Fairness				
	NuSMV		SMART-BFS				SMART-RchRel			NuSMV		SMART-RchRel		
	sec	KB(f)	*itr*	sec	KB(f)	KB(p)	sec	KB(f)	KB(p)	sec	KB	sec	KB(f)	KB(p)
leader	EG($status_0 \neq leader$)									$pref_0 = 1$				
3	0.2	9,308	14	0.1	139	196	4.9	934	1,115	0.6	11,428	0.02	266	268
4	3.0	50,187	18	< 0.1	400	436	791.6	5,999	7,225	11.2	49,655	207.4	5,271	6,394
phil.	EG($phil_0 \neq eat$)									$phil_0 = has\_left\_fork$				
10	0.1	7,193	4	< 0.1	95	95	0.1	170	170	0.2	8,447	< 0.1	113	113
50	3.0	50,187	4	< 0.1	220	241	0.2	682	682	1,244.5	75,274	< 0.1	393	399
100	–	–	4	0.1	444	562	1.1	1,180	1,191	–	–	0.1	704	705
robin	EG($true$)									$p1 = Ask$				
10	2.3	70,581	1	< 0.1	86	86	0.1	437	437	73.5	73,145	< 0.1	222	222
50	–	–	1	0.1	1,263	1,263	4.8	15,676	15,676	–	–	0.3	1,902	1,902
100	–	–	1	1.0	7,688	7,688	53.9	100,719	102,941	–	–	1.5	9,317	9,317
fms	EG$\neg(P_1s = P_2s = P_3s = N)$									$P_1s = N$				
5	0.8	18,474	1	< 0.1	61	135	1.9	1,022	1,024	2.5	35,238	0.27	419	475
10	16.5	60,559	1	< 0.1	128	220	1,062.4	4,338	6,231	191.8	62,188	77.7	607	1,050
kanban	EG($P_1out > 0 \vee P_2out > 0 \vee P_3out > 0 \vee P_4out > 0$)									$P_1out = N$				
8	2.1	42,925	1	< 0.1	279	415	1,131.1	1,949	2,714	2.2	43,511	6.2	1,303	1,486
10	4.4	58,693	1	< 0.1	529	930	–	–	–	4.6	58,705	27.0	2,507	2,939

Table 3.2: Results for the EG computation.

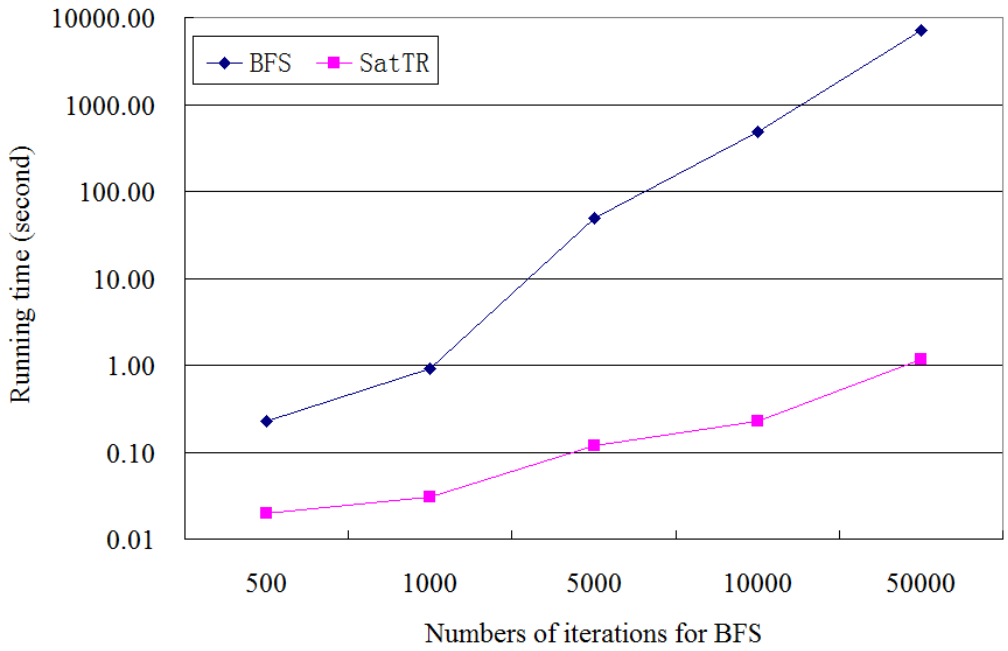


Figure 3.4: EG computation based on BFS v.s. transitive closure.

in NuSMV increases. The advantage of our algorithm is easily observable in this case, especially for some complex models which are not even manageable in NuSMV.

3.4 Summary

In this chapter, we focused on symbolic CTL model checking based on the idea of the saturation algorithm. To constrain state exploration to a given set of states, we present a constrained saturation algorithm. The "check-and-fire" policy filters out the states not in the given set when saturating MDD nodes recursively. For the EG operator, we first symbolically build the transitive closure, using constrained saturation, then compute the set of states satisfying $\mathsf{EG}\phi$. We discussed desirable properties of our new EU and EG algorithms and analyzed a set of experimental results.

Constrained saturation enables building the transitive closure for some complex systems. The application of the transitive closure could be further extended to SCC construction, a basic problem in emptiness checking for Büchi automata. Another future work is to reduce the cost of building the transitive closure. For SCC enumeration, $\mathcal{X}$ in $\mathcal{T}_{\mathcal{X},\mathcal{S}}$ can be refined to reduce the computation complexity. Next chapter presents these techniques.

Chapter 4

SCC enumeration

Finding strongly connected components (SCCs) is a basic problem in graph theory. For discrete-state models, some interesting properties, such as those expressible in Linear Time Logic (LTL) [48] and fair CTL [28], are correlated with the existence of SCCs in the state-transition graph. The same problem is also central to the language emptiness check for ω-automata [41, 48]. For large discrete-state models, it is impractical to find SCCs using explicit depth-first state-space search [76] since its complexity is *at least* linear in the size of the graph, motivating the study of symbolic SCC computation. In this chapter, the objective is to build the set of states in SCCs.

The structure of SCCs in a graph is captured by its *quotient graph*, obtained by collapsing each SCC into a single node. This graph is acyclic, thus defines a partial order on the SCCs. *Terminal SCCs* (or *bottom SCCs*) are nontrivial SCCs corresponding to leaf nodes in the quotient graph. For Markov chains [74] it is important to classify the reachable

states as *recurrent* (belonging to terminal SCCs) or *transient* (all other states), since the probability that a Markov chain returns to a given state equals 1 iff that state is recurrent.

The complexity of these problems arises from two aspects: having to explore a huge state space (almost always the case in real-life problems) and having to deal with a large number of SCCs or terminal SCCs (sometimes the case in real-life problems). The former, known as *state explosion* [28], is the main obstacle to formal verification due to the obvious burden it imposes on computational resources. Traditional BDD approaches cope with this problem by employing *image* and *preimage* computations for state-space exploration but, while successful for fully synchronous systems [12], they do not work as well for asynchronous systems [20]. The latter constitutes the bottleneck for one class of previous work [80, 81], which enumerates SCCs one by one. Section 4.1 analyzes this problem in more detail.

We address the computation of states in SCCs or terminal SCCs by improving two previous approaches: one proposed by Xie-Beerel (XB) [80, 81] and one based on computing the transitive closure (TC) of the transition relation [41]. We apply the saturation algorithm [20] to both to cope with the cost of state-space exploration. Our previous work demonstrated clear advantages for saturation in state-space generation [20] (summarized in Section 2.2.1) and CTL model checking [22, 82] over traditional symbolic approaches. In this chapter, we employ saturation for SCC analysis. Saturation greatly improve the XB algorithm over its original version using BFS. With regards to a potentially huge number of SCCs, the TC algorithm has the advantage of exploring all SCCs symbolically instead of enumerating them one by one. However, as previously proposed, computing the TC often

requires large amount of runtime and memory [41], to the point that [68] claims that the TC algorithm is "infeasible for large examples". To disprove this myth, we propose a new saturation algorithm to compute the TC, making it a practical method of SCC computation for complex systems. Furthermore, we present an algorithm to compute the recurrent states (i.e., states in terminal SCCs) based on the TC.

Then, we consider fair cycle detection, a problem related to SCC computation, but even more challenging. In Section 4.4, we discuss algorithms for detecting fair cycles satisfying Streett fairness (strong fairness) [70].

The remainder of this chapter is organized as follows. Section 5.1 introduces the relevant background on the data structures we use and saturation. Section 4.2 introduces an improved XB algorithm using saturation. Section 4.3 proposes our new algorithm to compute the TC using saturation and the corresponding algorithms for SCC and terminal SCC computation. Section 4.4 deals with the problem of finding fair cycles. Section 4.5 compares the performance of our algorithms and Lockstep. We discuss future work in the last section.

4.1 Previous work

Symbolic SCC analysis has been widely explored [33, 37, 39, 40, 68, 73], and much effort has been spent on computing the *SCC hull*, which includes states in SCCs, and along paths connection SCCs. A family of SCC hull algorithms [73] employing BFS iteration is available. Although closely related, there is crucial difference between SCC hull computation and our work, because an SCC hull contains not only states in nontrivial SCCs, but also

states on the paths between them, constituting a superset of our desired result. In non-probabilistic model checking, computing the SCC hull aims at detecting the *existence* of reachable fair cycles, which is critical to verify fair CTL, LTL and ω-automata properties. However, for Markov chains, the aim of SCC analysis is to collect the states belonging to (trivial or nontrivial) terminal SCCs [3, 24],. To this end, an SCC hull must be further decomposed into states that are in an SCC and those that are not. To the best of our knowledge, no existing efficient symbolic algorithm is available for this task. Hence, SCC hull computation algorithms are *not* applicable to our problem and we will not discuss them further. Instead, we review two categories of related previous work, which can compute the precise set of states in SCCs: the TC and the XB algorithms.

Hojati et al. [41] presented a symbolic algorithm to test ω-regular language containment by computing the TC $\mathcal{N} \cup \mathcal{N}^2 \cup \mathcal{N}^3 \cup \cdots$. Section 4.3 discusses the TC in more detail. Matsunaga et al. [55] proposed a symbolic procedure to compute the TC but the runtime is so poor that building the TC has long been considered impractical for complex systems.

Xie et al. [81] proposed an algorithm, referred to as the XB algorithm in this chapter, combining explicit state enumeration and symbolic state-space exploration. They explicitly pick a state as a "seed", compute the forward and backward reachable states from the seed, and find the SCC containing this seed as the intersection of these two sets of states. Bloem et al. [7] presented an improved algorithm called *Lockstep* (Figure 4.1) which, given a seed, instead of computing the forward and backward reachable states separately, it alternates BFS iterations between the two so that it can stop as soon as one of the two

```
mdd  Lockstep(mdd p)                                          • initially, B(p) = S
 1  if (p = 0) then return 0;
 2  mdd s ← PickState(p);                                     • pick a seed s from B(p)
 3  mdd a ← 0;                                                • a accumulates the answer
 4  mdd f ← 0;                                                • f accumulates the forward set from s
 5  mdd b ← 0;                                                • b accumulates the backward set from s
 6  mdd c ← 0;                                                • c will be the SCC containing s, if any
 7  mdd f' ← Intersect(ImageF(s), p);     mdd b' ← Intersect(ImageB(s), p);
 8  while f' ≠ 0 and b' ≠ 0 do
 9    f ← Union(f, f');       b ← Union(b, b');
10    f' ← Diff(Intersect(ImageF(f'), p), f);      b' ← Diff(Intersect(ImageB(b'), p), b);
11  endwhile;
12  if (f' = 0) then
13    mdd v ← f;                                              • v remembers the set that converged first, f
14    while Intersect(b', f) ≠ 0 do                           • continue exploring backward
15      b' ← Diff(Intersect(ImageB(b'), p), b);      b ← Union(b, b');
16    endwhile;
17  else
18    mdd v ← b;                                              • v remembers the set that converged first, b
19    while Intersect(f', b) ≠ 0 do                           • continue exploring forward
20      f' ← Diff(Intersect(ImageF(f'), p), f);      f ← Union(f, f');
21    endwhile;
22  endif;
23  if Intersect(f, b) ≠ 0 then
24    c ← Intersect(f, b);                                    • s belongs to the nontrivial SCC c
25  endif;
26  a←Union(c,Lockstep(Diff(v,c)),Lockstep(Diff(p,Union(v,s))));      • divide and conquer in p
27  return a;
```

Figure 4.1: Lockstep for SCC computation.

converges. This optimization achieves a $O(n \log n)$ complexity, compared to the $O(n^2)$ of the XB algorithm, computed in terms of number of image and preimage computations, where n is the number of reachable states. Our experiments show that Lockstep works very well when the number of SCCs is manageable. However, as the number of SCCs grows, the exhaustive enumeration of SCCs becomes a formidable problem for both the XB and the Lockstep algorithms.

```
mdd  XB_TSCC(mdd p)                                  • initially, B(p) = S
 1  mdd a←0;                                         • a collects the states in nontrivial terminal SCCs
 2  mdd s, f, b;
 3  while (p ≠ 0) do
 4    s ← PickState(p);                              • pick a seed s from B(p)
 5    f ← Intersect(ReachF(s), p);
 6    b ← Intersect(ReachB(s), p);
 7    if Diff(f, b) = 0 then                         • B(f) is a subset of B(b)
 8      a ← Union(a, f);                             • add the new terminal SCC to a
 9    endif;
10    p ← Diff(p, b);                                • don't consider the states in B(b) again
11  endwhile;
12  return a;
```

Figure 4.2: XB algorithm for terminal SCC computation.

Xie et al. [80] proposed a similar idea to compute the recurrent states in large Markov chains (*XB_TSCC* in Figure 4.2). From a randomly picked seed $\mathbf{i}$, if the set of forward reachable states $ReachF(\mathbf{i})$ is a subset of the set of backward reachable states $ReachB(\mathbf{i})$, then $ReachF(\mathbf{i})$ is a terminal SCC; otherwise, $ReachB(\mathbf{i})$ contains no terminal SCC and can be eliminated from further consideration. Functions *ReachF* and *ReachB* were implemented using BFS.

Our work is the first to employ saturation in SCC analysis. The two approaches we propose build upon the above two previous approaches. For the XB algorithm, we replace BFS state-space exploration with saturation. For the TC approach, we propose a new algorithm to compute TC using saturation.

4.2 Using saturation in the XB algorithm

A natural improvement to the XB algorithm is to employ saturation to explore the forward and backward reachable states. Figure 4.3 shows the pseudocode of our algorithms: the two versions of *XBSat* shown in the figure compute the states in SCCs, or just in the terminal SCCs, respectively. Unlike Lockstep, which uses the set that converges first to bound the other, our algorithm cannot interleave the two computations or even predict which one will converge first, since saturation does not run in a step-by-step manner. One obvious approach is then to run a forward and a backward saturation and intersect the results. However, we experimentally found that we can do better by bounding one of the two saturations with the other. In Figure 4.3, for example, we always "bet" on forward exploration and use it to bound backward exploration (Line 7). This bet is of course most beneficial when (unconstrained) backward exploration is much slower than forward exploration. Thus, there is a trade-off between using the slower BFS, which however allows us to interleave the two explorations, and the faster saturation, which does not. Performance is also affected by which seed is picked in each iteration and, for a fair comparison, we pick the same seed in both algorithms at each iteration.

Both our new algorithm and Lockstep improve the original XB algorithm, in different ways. Lockstep aims at reducing the number of steps by carefully scheduling the image and preimage computations, while our algorithm leverages event locality. Measuring complexity in number of BFS steps, $O(n \log n)$ for Lockstep [7], is not meaningful for saturation, which uses light-weight event firings instead of global image computations. Moreover, we argue that the number of steps is too rough a measure of complexity because the cost

of each symbolic step varies greatly, exponentially with the number of levels in the worst case. Instead, saturation aims at reducing the real cost in symbolic manipulation: it avoids building many unnecessary intermediate MDD nodes. Thus, while Lockstep ensures a tight bound on the number of steps, saturation often executes fewer node operations, thus lower runtime and memory requirements. Our experimental results show that, for most models we studied, the improved XB algorithm using saturation outperforms Lockstep, sometimes by orders of magnitude.

4.3 Computing the TC with saturation

We now define the forward and the backward TC, $\mathcal{T}$ and $\mathcal{T}^b$, of a discrete-state model.

Definition 10 *The transitive closure $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ contains all $(\mathbf{i}, \mathbf{j})$ such that there is a nontrivial (i.e., positive length) path from $\mathbf{i}$ to $\mathbf{j}$, denoted by $\mathbf{i} \to \mathbf{j}$. Analogously, we let $(\mathbf{i}, \mathbf{j}) \in \mathcal{T}^b$ iff $\mathbf{j} \to \mathbf{i}$.*

$\mathcal{T}^b$ represents a relation between states, and thus it can be encoded with a $2L$-level MDD with the same variable order as next-state functions. i_k is placed on level k (unprimed level) and j_k on level k' (primed level), where $Unprimed(k) = Unprimed(k') = k$. As $\mathcal{T}$ and $\mathcal{T}^b$ are symmetric, we focus on the algorithm to compute $\mathcal{T}^b$. After $\mathcal{T}^b$ has been built, $\mathcal{T}$ is obtained by simply swapping the unprimed and primed levels in the MDD encoding $\mathcal{T}^b$, written as $\mathcal{T} = Invert(\mathcal{T}^b)$.

```
mdd  XBSat(mdd p)                                          • initially, B(p) = S
 1  if p = 0 then return 0; endif;
 2  mdd a ← 0;
 3  mdd s ← PickState(p);                                  • pick a seed s from B(p)
 4  mdd f' ← Intersect(ImageF(s), p);
 5  mdd b' ← Intersect(ImageB(s), p);
 6  mdd f ← Intersect(SaturateF(f'), p);                   • forward reachable states from s in p
 7  mdd b ← Intersect(ConsSatB(f, b'), p);    • the intersection of backward and forward reachable
    states
 8  if b ≠ 0 then a ← b; endif;                            • b≠∅ is a set of states in an SCC
 9  a ← Union(a, XBSat(Diff(f, b)), XBSat(Diff(p, f)));
10  return a;
```

Figure 4.3: Improved XB algorithm to compute SCCs using saturation.

```
mdd  XBSat(mdd p)                                          • initially, B(p) = S
 1  if p = 0 then return 0; endif;
 2  mdd a ← 0;
 3  mdd s ← PickState(p);                                  • pick a seed s from B(p)
 4  mdd f' ← Intersect(ImageF(s), p);
 5  mdd b' ← Intersect(ImageB(s), p);
 6  mdd f ← Intersect(SaturateF(f'), p);                   • forward reachable states from s in p
 7  mdd b ← Intersect(SaturateB(b'), p);                   • backward reachable states from s in p
 8  if Diff(f, b) = 0 then a ← Union(a, f); endif;         • B(f) is a terminal SCC if B(f) ⊆ B(b)
 9  a ← Union(a, XBSat(Diff(p, b)));                       • recursion in the remaining state space
10  return a;
```

Figure 4.4: Improved XB algorithm to compute terminal SCCs using saturation.

We base our algorithm on the following observation:

$$(\mathbf{i}, \mathbf{j}) \in \mathcal{T}^b \Leftrightarrow \exists \mathbf{k} \in \mathcal{P}(\mathbf{i}) \wedge \mathbf{j} \in \mathcal{B}(\mathit{ConsSatB}(\mathcal{S}, s)),$$

where $\mathcal{B}(s) = \{\mathbf{k}\}$. Instead of running saturation on $\mathbf{j}$ for each pair $(\mathbf{i}, \mathbf{j})$, we propose an algorithm that executes on the $2L$-level MDD encoding $\mathcal{P}$. Line 1 in function $\mathit{SCC\_TC}$ of Figure 4.5 computes $\mathcal{T}^b$ by calling *TransClosSat*, which runs bottom-up recursively. As for the constrained saturation in Figure 3.3, this function runs node-wise on primed level and

fires lower level events exhaustively until the local fixed point is obtained. This procedure ensures the following properties.

Property 1: Given a k-level MDD a and $2k$-level MDD n, $\mathit{TransClosSat}(a, n)$ returns a $2k$-level MDD t s.t. $\forall(\mathbf{i}, \mathbf{j}) \in \mathcal{B}(n), \mathbf{k} \in \mathcal{B}(\mathit{ConsSatB}(a, \mathbf{j})) \Rightarrow (\mathbf{i}, \mathbf{k}) \in \mathcal{B}(t)$.

Property 2: $\mathit{TransClosSat}(\mathcal{S}, \mathcal{P}) = \mathcal{T}^b$.

The top-level pseudocode of the SCC computation using TC is shown as $\mathit{SCC\_TC}$ in Figure 4.5. Function $\mathit{TCtoSCC}$ extracts all states $\mathbf{i}$ such that $(\mathbf{i}, \mathbf{i}) \in \mathcal{T}^b$. Unlike SCC enumeration algorithms such as XB or Lockstep, a TC approach does not necessarily suffer from a large number of SCCs. Nevertheless, due to the complexity of building $\mathcal{T}^b$, this approach had been considered infeasible for complex systems. By employing saturation, instead, our algorithm to compute $\mathcal{T}^b$ completes on some large models, such as the dining philosopher problem with 1000 philosophers, while, for models containing many SCCs, it may succeed where other algorithms have no hope. Thus, while the TC approach is not as robust as Lockstep, it can be used as an alternative to it when Lockstep cannot enumerate all SCCs.

$\mathcal{T}^b$ can also be used to find recurrent states, i.e., states in terminal SCCs. State $\mathbf{j}$ is in a terminal SCC if, for any state $\mathbf{i}$, $\mathbf{j} \rightarrow \mathbf{i} \Rightarrow \mathbf{i} \rightarrow \mathbf{j}$. Given two states $\mathbf{i}, \mathbf{j}$, let $\mathbf{j} \mapsto \mathbf{i}$ mean $\mathbf{j} \rightarrow \mathbf{i}$ and $\mathbf{i} \not\rightarrow \mathbf{j}$. We can encode this relation with a $2L$-level MDD, obtained as $\mathcal{T}^b \setminus \mathcal{T}$. The pseudocode for this algorithm is shown as $\mathit{TSCC\_TC}$ in Figure 4.7. The set $\{(\mathbf{i}, \mathbf{j}) \,|\, \mathbf{j} \mapsto \mathbf{i}\}$ is encoded with a $2L$-level MDD r^*. Then, the set of states $\{\mathbf{j} \,|\, \exists \mathbf{i}, \mathbf{j} \mapsto \mathbf{i}\}$, which do *not* belong to terminal SCCs, is computed by existentially quantifying out the unprimed levels

```
mdd  SCC_TC(P)
 1  mdd T^b ← TransClosSat(S, P);
 2  mdd SCC ← TCtoSCC(T^b);
 3  return SCC;
```

```
mdd  TransClosSat(mdd a, mdd n)
 1  if n = 1 then return 1; endif;
 2  if InCache_TransClosSat(a, n, t) then return t; endif;
 3  level k ← n.lvl;                                          • L ≥ U(k) ≥ 1
 4  mdd t ← 0;
 5  mdd r ← P_U(k);
 6  foreach i, j ∈ S_k s.t. n[i][j] ≠ 0 do
 7    if a[j] ≠ 0 then                                        • first, saturate nodes below
 8      t[i][j] ← TransClosSat(a[j], n[i][j]);
 9    else                                                    • no new path in this sub-space since a[i] ≠ 0
10      t[i][j] ← n[i][j];
11    endif;
12  endfor;
13  foreach i ∈ S_U(k) s.t. n[i] ≠ 0 do
14    repeat                                                  • compute the local fixed point
15      foreach j,j' ∈ S_U(k) s.t. n[i][j] ≠ 0 ∧ r[j][j'] ≠ 0 ∧ a[j'] ≠ 0 do • r is applied to the primed levels
        of n
16        t[i][j'] ← Union(t[i][j'], TCRelProdSat(a[j'], t[i][j], r[j][j']));
17      endfor;
18    until t does not change;
19  endfor;
20  t ← UniqueTableInsert(t);
21  CacheAdd_TransClosSat(a, n, t);
22  return t;                                                 • t is a 2L-level MDD encoding TransClosSat(a, n)
```

Figure 4.5: Building the TC using saturation (continued on Figure 4.6).

```
mdd  TCRelProdSat(mdd a, mdd n, mdd r)
 1  if n = 1 and r = 1 then return 1; endif;
 2  if InCache_TCRelProdSat(a, n, r, t) then return t; endif;
 3  level k ← n.lvl;                                          • L ≥ U(k) ≥ 1
 4  mdd t ← 0;
 5  foreach i ∈ S_U(k) s.t. n[i] ≠ 0 do
 6    foreach j,j' ∈ S_U(k) s.t. n[i][j] ≠ 0 ∧ r[j][j'] ≠ 0 ∧ a[j'] ≠ 0 do • r is applied on primed levels in n
 7      t[i][j'] ← Union(t[i][j'], TCRelProdSat(a[j'], n[i][j], r[j][j']));
 8    endfor;
 9  endfor;
10  t ← TransClosSat(a, UniqueTableInsert(t));                • return a saturated result
11  CacheAdd_TCRelProdSat(a, n, r, t);
12  return t;
```

```
mdd  TCtoSCC(mdd n)
 1  if n = 1 return 1; endif;
 2  if InCache_TCtoSCC(n, t) then return t; endif;
 3  level k ← n.lvl;                                          • L ≥ U(k) ≥ 1
 4  mdd t ← 0;
 5  foreach i ∈ S_U(k) s.t. n[i][i] ≠ 0 do
 6    t[i] ← TCtoSCC(n[i][i]);
 7  endfor;
 8  t ← UniqueTableInsert(t);
 9  CacheAdd_TCtoSCC(n, t);
10  return t;                                                 • t is an L-level MDD enconding {i : (i, i) ∈ B(n)
```

Figure 4.6: Building the TC using saturation.

(Line 5), and stored in MDD *nontscc*. All other states are the recurrent states belonging to terminal SCCs.

```
mdd  TSCC_TC(P)
 1  mdd T^b ← TransClosSat(P);
 2  mdd T ← Inverse(T^b);                          • swap adjacent levels k and k'
 3  mdd scc ← TCtoSCC(T^b);
 4  mdd r* ← Diff(T^b, T);
 5  mdd nontscc ← QuantUnpr(r*); • quantify out unprimed levels in r* and get an L-level MDD
 6  mdd recurrent ← Diff(scc, nontscc);
 7  return recurrent;
```

Figure 4.7: Computing recurrent states using TC.

To the best of our knowledge, this is the first TC-based algorithm for terminal SCC computation. This algorithm is more costly in runtime and memory than SCC computation because of the need to obtain the $\mapsto$ relation. However, by employing *TransClosSat*, it works for most of the models we considered. Moreover, for models with a huge number of terminal SCCs, this algorithm is, again, the only feasible approach.

4.4 Fair cycles

One application of the SCC computation is to check the language emptiness of an ω-automaton. The language of an ω-automaton is nonempty if there is a nontrivial cycle which satisfies a certain fairness condition, i.e., a *fair cycle*. Thus, it is necessary to extend the SCC computation to finding fair cycles. Two widely used fairness conditions are Büchi fairness (weak fairness) and Streett fairness (strong fairness) [28].

Strong fairness is specified with a set of pairs of sets $\{(\mathcal{R}_1, \mathcal{C}_1), \ldots, (\mathcal{R}_n, \mathcal{C}_n)\}$ and a cycle satisfies it iff, for each $(\mathcal{R}_m, \mathcal{C}_m)$, either no state of $\mathcal{R}_m$ is in the cycle or some state of $\mathcal{C}_m$ is in the cycle. *Weak fairness* is specified with a set of sets of states $\{\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n\}$. Its semantics is expressed by the special case of strong fairness where, in each pair, $\mathcal{R}_m = \mathcal{S}$ and $\mathcal{C}_m = \mathcal{F}_m$, thus we focus on strong fairness.

Lockstep is also able to find strongly fair cycles [7], but might suffer from SCC *refinement* [7, Figure 4]. This occurs when an SCC intersects a $\mathcal{R}_i$ but not $\mathcal{C}_i$. Then, we need to to filter out all states in $\mathcal{R}_i$ and run Lockstep on the remaining states in this SCC. This process may enumerate all states in the worst case, as with SCC enumeration. Here, we present a TC approach that avoids enumeration. To the best of our knowledge, this is the first TC approach for fair cycle detection. The following defines the *constrained TC*.

Definition 11 *Given a set of states $\mathcal{X}$, the constrained backward TC is $\mathcal{T}^b_{\mathcal{X}} = TransClosSat(\mathcal{X}, \mathcal{P})$.*

$\mathcal{T}^b_{\mathcal{X}}$ specifies the relation between two states $\mathbf{i}$ and $\mathbf{j}$, such that $(\mathbf{i}, \mathbf{j}) \in \mathcal{T}^b_{\mathcal{X}} \Leftrightarrow \mathbf{j} \rightarrow \mathbf{t}_1 \rightarrow \mathbf{t}_2 \rightarrow \ldots \rightarrow \mathbf{i}$ and $\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{i} \in \mathcal{X}$. Thus, the set of states belonging to some (strongly) fair cycle is given by:

$$\bigcap_{m=1,\ldots,n} \Big(TCtoSCC(\mathcal{T}^b_{\mathcal{S}\setminus\mathcal{R}_m}) \ \cup \ \{\mathbf{i} \,|\, \exists \mathbf{c}_m \in \mathcal{C}_m, (\mathcal{T}(\mathbf{c}_m, \mathbf{i}) \cap \mathcal{T}^b(\mathbf{c}_m, \mathbf{i}))\}\Big).$$

If $(\mathbf{i},\mathbf{i}) \in \mathcal{T}^b_{\mathcal{S}\setminus\mathcal{R}_m}$, there is a nontrivial path from $\mathbf{i}$ to $\mathbf{i}$ that avoids $\mathcal{R}_m$. Thus, $TCtoSCC(\mathcal{T}^b_{\mathcal{S}\setminus\mathcal{R}_m})$ returns the states in cycles that contain no state in $\mathcal{R}_m$ and satisfy strong fairness. If $(\mathbf{i},\mathbf{c}_m) \in (\mathcal{T}(\mathbf{c}_m,\mathbf{i}) \cap \mathcal{T}^b(\mathbf{c}_m,\mathbf{i}))$, $\mathbf{i}$ belongs to a cycle which contains $\mathbf{c}_m$. The subformula in the second line corresponds to the states in a cycle containing at least one state in $\mathcal{C}_m$, satisfying strong fairness.

```
mdd FairSCC_TC(S, P, {(R_1, C_1), ..., (R_n, C_n)})
 1  mdd T^b ← TransClosSat(S, P);
 2  mdd T ← Inverse(T^b);
 3  mdd s ← S;
 4  foreach m = 1, ··· , n do
 5    mdd c ← TCtoSCC(TransClosSat(Diff(S, R_m), P));
 6    mdd d ← Intersect(T^b, T);
 7    d ← QuantUnpr(Intersect(d, Cross(C_m, S)));      • Cross(C_m,S) = {(c_m,i) : c_m ∈ R_m ∧ i ∈ S}
 8    s ← Intersect(s, Union(c, d));
 9  endfor;
10  return s;
```

Figure 4.8: Computing fair cycles using TC.

The pseudocode in Figure 4.8 computes the above formula. For each $(\mathcal{R}_m, \mathcal{C}_m)$, c encodes the states in cycles that do not intersect $\mathcal{R}_m$, d encodes the states in cycles that intersect $\mathcal{C}_m$, and $Cross(\mathcal{C}_m, \mathcal{S})$ returns a $2L$-level MDD encoding the cross-product of $\mathcal{C}_m$ and $\mathcal{S}$. The cost mainly lies in computing the intersection of $\mathcal{T}^b$ and $\mathcal{T}$, which is similar to computing the relation $\mapsto$. Moreover, the complexity grows linearly with the size n of the fairness condition.

4.5 Experimental results

We implemented our algorithms in SMART [16] and report experimental results running on an Intel Xeon 3.0Ghz workstation with 3GB RAM under SuSE Linux 9.1. The models, described as Petri nets and translated into the input language of SMART, include the closed queue network (*cqn*) of [81], two implementations of arbiters ($arbiter_1$, $arbiter_2$) [1], one which guarantees fairness and the other which does not, the queen placement problem (*queens*), the dining philosopher problem (*phil*), and the leader selection

protocol (*leader*) [15]. The size for each model is controlled by a parameter N. The number of SCCs (terminal SCCs) and states in SCCs (terminal SCCs) for each model is listed in column "SCC count" ("TSCC count") and column "SCC states" ("TSCC states"), respectively. We set an upper bound of 2hr for runtime and 1GB for the unique table (used to canonically store the MDD nodes). The main metrics of our comparison are runtime and peak memory consumption (for unique table plus operation cache, required for efficient dynamic programming implementation of MDD operation), which are measured in seconds and megabytes, respectively.

The top part of Table 4.1 compares three algorithms for SCC computation: the TC algorithm (column "TC") of Section 4.3, the improved XB algorithm (column "XBSAT") of Section 4.2, and Lockstep (column "Lockstep"). We can see that the improved XB algorithm using saturation is better than Lockstep for most models, in both runtime and memory. Compared with the SCC enumeration algorithms, the TC algorithm is often more expensive but, for *queens* and $arbiter_2$, it completes within the time limit while the other two algorithms fail. For $arbiter_2$ ("*" means the number of states in SCCs is computed from the model structure, i not using XB.), our TC algorithm explores over 10^{150} SCCs in a few seconds, while it is obviously not feasible to exhaustively enumerate all SCCs in reasonable time. To the best of our knowledge, this is the best result of SCC computation reported, confirming the main advantage of the TC algorithm: its insensitivity to the number of SCCs. With the help of our new algorithm, the TC can be built for some large systems, such as the dining philosopher problem with 1000 philosophers.

The bottom part of Table 4.1 compares three algorithms for terminal SCC computation: *XBSat* (column "XBSat") presented in Sections 4.2 , *TSCC_TC* (column "TC") presented in Sections 4.3, and the BFS XB algorithm *XB_TSCC* (column "XBBFS") shown in Figure 4.2. The basic trends are similar to those for SCC computation: algorithm $XBSat$ works consistently better than the traditional method, while *TSCC_TC* is less efficient for most models. In the framework of the XB algorithm, computing terminal SCCs is faster than computing SCC because a larger set of states is pruned at each recursion. On the contrary, *TSCC_TC* is more expensive than *SCC_TC* due to the computation of the $\mapsto$ relation. As a consequence, *TSCC_TC* suffers even more from large memory and runtime requirement. Nevertheless, for models with large numbers of terminal SCCs, such as *queens*, *TSCC_TC* outperforms the BFS XB algorithm.

Some conclusions can be drawn from the above results and discussion. First, saturation is effective in speeding up SCC and terminal SCC computation within the framework of the XB algorithm. Second, our new saturation algorithm makes TC computation feasible for some complex models containing up to 10^{150} states. Third, SCC computation based on TC is superior to SCC enumeration algorithms, which find SCCs one-by-one, for models with huge numbers of SCCs.

Although the TC approach is not as robust as Lockstep, especially when the number of SCCs in the model is manageable, we argue that it should be considered as a reasonable alternative worth of further research. Given a new model with an unknown number of existing SCCs, employing both of these approaches at the same time will be ideal. Current trend of multi-core computers provide a possible means of parallelizing these

two algorithms. Some of the common data structures, like MDDs encoding the state space and next-state functions, could then even be shared.

4.6 Summary

We focused on improving two previous approaches for SCC computation, the Xie-Beerel (XB) and the transitive closure (TC) algorithms, using saturation. For the asynchronous models we study, the improved XB algorithm using saturation achieves a clear speedup and our new algorithm to compute TC using saturation is experimentally shown to be capable of handling models with up to 10^{150} of SCCs. We argue that the TC-based approach is worth further research because of its ability to deal with models having huge numbers of SCCs.

SCC analysis can be an important step to simplify Markov chain analysis and stochastic model checking, as shown in [3, 24]. I will present the application of SCC enumeration in Part 2.

Results for the SCC computation									
Model		SCC count	SCC states	TC		XBSat		Lockstep	
name	N			mem	time	mem	time	mem	time
cqn	10	11	2.09e+10	34.2	13.6	3.4	< 0.1	4.0	3.9
	15	16	2.20e+15	64.4	73.8	5.0	0.2	89.1	44.5
	20	21	2.32e+20	72.7	687.8	25.8	0.5	118.7	275.0
phil	100	1	4.96e+62	5.0	0.5	3.2	< 0.1	52.0	4.5
	500	1	3.03e+316	33.0	4.0	24.5	0.1	–	to
	1000	1	9.18e+626	40.5	7.8	29.1	0.3	–	to
queens	10	3.22e+4	3.23e+4	8.2	1.6	64.4	14.5	63.9	12.4
	11	1.53e+5	1.53e+5	45.8	9.0	94.2	108.6	96.3	93.6
	12	7.95e+5	7.95e+5	184.8	60.6	170.2	1220.4	281.9	1663.9
	13	4.37e+6	4.37e+6	916.5	840.6	–	to	–	to
leader	3	4	6.78e+2	6.0	1.4	20.8	< 0.1	20.8	< 0.1
	4	11	9.50e+3	70.3	73.1	25.4	1.1	23.8	0.3
	5	26	1.25e+5	116.6	3830.4	35.6	40.8	49.4	6.4
	6	57	1.54e+6	–	to	41.6	1494.9	417.2	387.9
$arbiter_1$	10	1	2.05e+4	24.1	1.2	21.4	< 0.1	21.8	0.1
	15	1	9.83e+5	128.3	63.0	45.1	< 0.1	62.1	6.8
	20	1	4.19e+7	mo	–	709.7	< 0.1	mo	–
$arbiter_2$	10	1024	1.02e+4	20.3	< 0.1	26.2	0.7	31.1	1.1
	15	32768	4.91e+5	20.4	< 0.1	31.1	51.8	211.3	990.3
	20	1.05e+6	2.10e+7	20.4	< 0.1	31.2	2393.3	–	to
	500	**3.27e+150**	(**1.64e+151**)*	**41.0**	**4.0**	–	to	–	to
Results for the terminal SCC computation									
Model		TSCC count	TSCC states	TC		XBSat		XBBFS	
name	N			mem	time	mem	time	mem	time
cqn	10	10	2.09e+10	37.9	15.5	21.4	< 0.1	33.5	3.4
	15	15	2.18e+15	64.8	79.6	23.0	0.3	59.4	33.7
	20	20	2.31e+20	72.7	691.3	26.2	0.8	90.0	280.5
phil	100	2	2	26.5	0.5	20.9	< 0.1	39.2	8.7
	500	2	2	34.3	4.1	23.2	< 0.1	–	to
	1000	2	2	44.4	11.3	26.5	0.2	–	to
queens	10	1.28e+4	1.28e+4	36.2	3.0	46.7	2.8	62.3	35.1
	11	6.11e+4	6.11e+4	76.5	19.3	70.6	24.5	145.2	364.2
	12	3.14e+5	3.14e+5	244.1	205.4	98.8	179.4	mo	–
	13	1.72e+6	1.72e+6	mo	–	269.0	1940.81	mo	–
leader	3	3	3	26.6	1.5	20.7	< 0.1	21.4	0.1
	4	4	4	70.6	75.1	24.4	0.9	38.0	4.5
	5	5	5	119.3	3845.3	30.6	26.9	41.1	87.6
	6	6	6	–	to	39.0	492.9	44.8	1341.5
$arbiter_1$	10	1	2.05e+4	24.1	1.2	20.4	< 0.1	22.4	0.4
	15	1	9.83e+5	128.3	63.1	20.4	< 0.1	65.3	23.3
	20	1	4.19e+7	mo	–	20.5	< 0.1	–	to
$arbiter_2$	10	1	1	20.4	< 0.1	20.9	< 0.1	39.6	6.4
	15	1	1	20.5	< 0.1	40.6	4.6	–	to
	20	1	1	20.5	< 0.1	450.0	2897.8	–	to

Table 4.1: Results for SCC and terminal SCC computation.

Chapter 5

Shortest EG witness generation

As we review in the preliminaries, counterexample generation to universal path property can be converted to an equivalent witness generation to an existential property, in the context of CTL. Thus, we only focus on shortest witness generation. Many efforts [32][72][21][45] have been made on this topic. Unlike EX or EU witnesses, which are finite paths, EG witnesses are "lasso-shaped" [32], i.e., they contain a prefix leading to a cycle. Locating this cycle is the crucial, and difficult step in witness generation, and it is even more difficult to find a shortest (length of handle plus cycle) witness.

In this chapter, we utilize Edge-Valued Multi-way Decision Diagrams (EVMDDs) to encode the Transitive Closure with Distance (TCD), and employ an advanced algorithm for symbolic exploration, saturation [20], to efficiently build the EVMDD encoding the TCD.

The remainder of this chapter is structured as follows: Section 5.1 introduces the relevant background on CTL and the symbolic data structure we use. Section 5.2.1

presents our approach, stressing the computation of the TCD using the saturation algorithm. Section 5.3 extends our approach to a more general class of shortest witness generation problems. Section 5.4 offers some experimental results. We summarize this chapter and outline future work in the last section.

5.1 Background and related work

Recall that only a generator subset of CTL operators, e.g., $\{\mathsf{EX}, \mathsf{EU}, \mathsf{EG}\}$, needs to be implemented in a model checker, as the all other CTL operators can be expressed using the generator set and ordinary boolean operators. If a property with path quantifier A is violated, there must be an execution, i.e., a counterexample, which violates the temporal operator following the path quantifier. Conversely, if a property with path quantifier E holds, there must be an execution, i.e., a witness, which conforms to the temporal operator. The correspondence between counterexamples to universal "A" properties and witnesses to existential "E" properties is as follows:

a counterexample to	is the same as a witness to
$\mathsf{AX}\phi$	$\mathsf{EX}(\neg\phi)$
$\mathsf{AG}\phi$	$\mathsf{EF}(\neg\phi)$
$\mathsf{A}[\phi\mathsf{U}\psi]$	$\mathsf{E}[\neg\psi\mathsf{U}(\neg\phi \wedge \neg\psi)] \vee \mathsf{EG}(\neg\psi)$
$\mathsf{AF}\phi$	$\mathsf{EG}(\neg\phi)$

Clarke et al. [32] proposed the first symbolic approach to CTL witness generation. Using symbolic breath-first search, witness generations for EX and EU can naturally guarantee minimality, but the problem is much more difficult for the EG operator. A witness to

$\mathsf{EG}\phi$ is composed of a path from the initial state to a cycle, such that all states along that path and on the cycle satisfy ϕ. According to CTL semantics, if a state satisfies $\mathsf{EG}\phi$, it must have a successor that also satisfies $\mathsf{EG}\phi$. Thus, we can incrementally build a path of states satisfying $\mathsf{EG}\phi$, which must finally lead to a state already on the path, closing the cycle and resulting in a witness. Figure 5.1 shows the pseudocode of this algorithm, which can be easily implemented using BDDs or MDDs. A witness generation algorithm for weakly fair EG was also proposed in [32] based on this idea. Since there might be multiple successors satisfying $\mathsf{EG}\phi$, the algorithm is nondeterministic and the length of the witness depends in general on which state is chosen at each step. We stress that, while the pseudocode uses a symbolic encoding, this algorithm is largely explicit, as it follows a single specific path. Decision diagrams help very little, especially if $\mathcal{N}(\mathbf{i})$ is a very small set for each $\mathbf{i}$.

The work most related to ours was presented by Schuppan et al. in [71, 72], which proposed a framework to convert a liveness property to a reachability problem by performing a *state-recording translation*, so that a shortest witness for $\mathsf{EG}\phi$ can then be generated using breadth-first search (BFS). The bottleneck of this approach mainly lies in the BFS over the quadratic state space of the original system. Our previous work [21] has shown that saturation can effectively speed up shortest trace generation. This chapter extends the idea of [21] to the generation of shortest $\mathsf{EG}\phi$ witness. Instead of exploring a quadratic state space using BFS like [71], we employ saturation, usually more efficient than BFS for asynchronous systems, to build the TCD encoded as an EVMDD.

Orthogonal to symbolic algorithms, explicit-state model checkers adopt techniques such as heuristic-guided search [75] and crucial event identification [45] to shorten the

generated witnesses. These techniques aim at achieving both an efficient of state-space exploration and shorter witnesses. Still, none of these algorithms is guaranteed to find a shortest EG witness.

5.2 Overview

A witness to $\mathsf{EG}\phi$ in a finite-state system, often described as *lasso-shaped witness* [6], is an infinite path consisting of a finite prefix leading to a cycle, We provide the following definition to discuss EG witnesses in more detail:

Definition 12 *Given $\phi \in \mathcal{A}$, a finite acyclic path $\pi_s = \mathbf{s}_{init} \rightarrow \mathbf{i}_1 \rightarrow \cdots \rightarrow \mathbf{i}_n$ is a ϕ-stem of length $n \geq 0$ if $\mathbf{s}_{init}, \mathbf{i}_1, \ldots, \mathbf{i}_n \in Sat(\phi)$; a (cyclic) path $\pi_c : \mathbf{i}_1 \rightarrow \cdots \rightarrow \mathbf{i}_m \rightarrow \mathbf{i}_1$ is a ϕ-cycle of length $m \geq 1$ if $\mathbf{i}_1, \ldots, \mathbf{i}_m \in Sat(\phi)$. Let $stem(\phi, \mathbf{i})$ be the set of ϕ-stems that terminate in state $\mathbf{i}$ and $cycle(\phi, \mathbf{i})$ be the set of ϕ-cycles that start in state $\mathbf{i}$; when ϕ is understood, we simply write $stem(\mathbf{i})$ and $cycle(\mathbf{i})$.* □

An $\mathsf{EG}\phi$ witness is composed of a ϕ-stem leading to a ϕ-cycle. We say that state $\mathbf{k}$ which terminates a ϕ-stem and also starts a ϕ-cycle is a *knot*. The initial state is the knot in witnesses which have the 0-length stems. If $\mathsf{EG}\phi$ holds in the initial state, a state $\mathbf{k} \in Sat(\phi)$ induces a set of witnesses $witness(\mathbf{k}) = stem(\mathbf{k}) \times cycle(\mathbf{k})$, i.e., all combinations of ϕ-stems in $stem(\mathbf{k})$ and ϕ-cycles in $cycle(\mathbf{k})$. A shortest witness among $witness(\mathbf{k})$ consists of a shortest $stem(\mathbf{k})$ and a shortest $cycle(\mathbf{k})$. Hence, finding the shortest $\mathsf{EG}\phi$ witness can be seen as the minimization problem (let $|\pi|$ be the length of path π):

$$\min_{\mathbf{k} \in Sat(\phi)} \left(\min_{\pi_s \in stem(\mathbf{k})} (|\pi_s|) + \min_{\pi_c \in cycle(\mathbf{k})} (|\pi_c|) \right). \tag{5.1}$$

```
EGwitnessBFS(stateset P) is                    • P is the MDD encoding Sat(φ)
 1  stateset Q ← EG(P);                        • witness exists only if s_init ∈ Q
 2  stateset X ← ∅;                            • set of states in the witness
 3  stateset Y ← {s_init};                     • current frontier
 4  while (X ∩ Y = ∅) do
 5    state i ← pick(Y);      print i;
 6    X ← X ∪ {i};      Y ← N(i) ∩ Q;
 7  endwhile
 8  i ← pick(X ∩ Y);      print i;
```

Figure 5.1: BFS-based algorithm to generate a witness for $\mathsf{EG}\phi$.

Given a knot state $\mathbf{k}$, algorithms in [32, 21] can efficiently find the shortest $stem(\mathbf{k})$ and $cycle(\mathbf{k})$. The difficulty lies in finding a knot state $\mathbf{k}^*$ that induces a globally shortest witness in Equation 5.1. This difficulty can be attributed to the lack of algorithms able to compute the distance information between pairs of states in a huge state space. We attack the problem from this angle by computing the transitive closure with distance (TCD).

Definition 13 ***(Transitive Closure with Distance).*** *Function* $TCD_\phi : \mathcal{S} \times \mathcal{S} \to \mathbb{N}^+ \cup \{\infty\}$ *is such that* $TCD_\phi(\mathbf{i},\mathbf{j})$ *is the length of a non-zero shortest path* $\mathbf{i} \to \mathbf{s}_1 \to \mathbf{s}_2 \to \cdots \to \mathbf{j}$ *where* $\mathbf{i}, \mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{j} \in Sat(\phi)$*, and* $TCD_\phi(\mathbf{i},\mathbf{j}) = \infty$ *if no such path exists. Let* $TCD^{triv}_\phi(\mathbf{i},\mathbf{j})$ *be an extension of* $TCD_\phi(\mathbf{i},\mathbf{j})$ *that:*

$$TCD^{triv}_\phi(\mathbf{i},\mathbf{j}) = \begin{cases} 0 & \text{if } \mathbf{i}=\mathbf{j} \\ TCD_\phi(\mathbf{i},\mathbf{j}) & \text{otherwise} \end{cases}.$$

Also, let $TCD^{-1}_\phi(\mathbf{i},\mathbf{j}) = TCD_\phi(\mathbf{j},\mathbf{i})$. □

As the base case, $TCD_\phi(\mathbf{i},\mathbf{j}) = 1$ if $\mathbf{i}$ and $\mathbf{j}$ satisfy ϕ and $\mathbf{j} \in \mathcal{N}(\mathbf{i})$. Define

$$TCD^{stem}_\phi(\mathbf{i}) \triangleq TCD^{triv}_\phi(\mathbf{s}_{init}, \mathbf{i})$$

$$TCD^{cycle}_\phi(\mathbf{i}) \triangleq TCD_\phi(\mathbf{i},\mathbf{i}).$$

Formula 5.1 can then be rewritten as:

$$\min_{\mathbf{k} \in Sat(\phi)} (TCD_{\phi}^{stem}(\mathbf{k}) + TCD_{\phi}^{cycle}(\mathbf{k})).$$

TCD_{ϕ} is a two-parameter function, thus must be encoded with a $2L$-level EV^{+}MDD, while TCD_{ϕ}^{stem} and TCD_{ϕ}^{cycle} are single-parameter functions which can be encoded with L-level EV^{+}MDDs and are obtained from TCD_{ϕ} through symbolic manipulation. Thus, our algorithm for EG witness generation consists of the following steps:

1. Build the $2L$-level EV^{+}MDD encoding TCD_{ϕ}.

2. Build L-level EV^{+}MDDs encoding TCD_{ϕ}^{stem} and TCD_{ϕ}^{cycle}. Compute the sum of these two EV^{+}MDDs, which encodes length of the shortest witness induced by each state $\mathbf{k} \in Sat(\phi)$.

3. Extract a knot state that achieves the minimum value in the resulting function.

4. Find the shortest paths from the initial state to the knot (ϕ-stem) and from the knot to itself (ϕ-cycle). These two paths form a shortest witness for EGϕ.

As the computation of TCD_{ϕ} is the most time and memory intensive step in the overall procedure, we discuss it in detail in next section.

5.2.1 Computing TCD

Building TCD_{ϕ} is essentially the classic all-pair shortest path problem in a modified graph from the original discrete-state system where only states (vertices) in $Sat(\phi)$ and transitions (edges) between these states are retained. All edges have unit weight, so the

distance between $\mathbf{i}$ and $\mathbf{j}$ equals $TCD_\phi(\mathbf{i},\mathbf{j})$. Instead of using a distance matrix, however, we utilize a $2L$-level EV$^+$MDD $\langle\tau,t\rangle$ to encode the distance between each pair of states:

$$f_{\langle\tau,t\rangle}(i_L,j_L,i_{L-1},j_{L-1},\ldots,i_1,j_1) = TCD_\phi(\mathbf{i},\mathbf{j}),$$

where $\mathbf{i},\mathbf{j} \in Sat(\phi)$, and we interleave the levels for $\mathbf{i}$ and $\mathbf{j}$. To correlate local states with their submodels, the levels of i_k and j_k in $2L$-level EV$^+$MDD are referred to by k and k', respectively (unprimed and primed levels are interleaved in our implementation), and we let $Unprimed(k) = Unprimed(k') = k$.

The procedure to build $\langle\tau,t\rangle$ is analogous to a symbolic implementation of Dijkstra's algorithm. We start from the EV$^+$MDD $\langle\tau_1,t_1\rangle$ encoding

$$f_{\langle\tau_1,t_1\rangle}(\mathbf{i},\mathbf{j}) = \begin{cases} 1 & \text{if } \exists\alpha\in\mathcal{E},\ \mathbf{j}\in\mathcal{N}_\alpha(\mathbf{i}) \\ \infty & \text{otherwise} \end{cases}$$

(so that $\tau_1 = 1$ and all values associated with outgoing edges are either 0 or ∞), and use $\mathcal{N}_\alpha$ to build a new EV$^+$MDD $\mathcal{N}_\alpha(\langle\tau,t\rangle)$ satisfying

$$f_{\mathcal{N}_\alpha(\langle\tau,t\rangle)}(\mathbf{i},\mathbf{j}) = \min\left(\min_{\mathbf{k}\in pre(\mathbf{j})}\left(f_{\langle\tau,t\rangle}(\mathbf{i},\mathbf{k})\right),\infty\right),$$

where $pre(\mathbf{j}) = \{\mathbf{k}\in Sat(\phi) : \mathbf{j}\in\mathcal{N}_\alpha(\mathbf{k})\}$. Then, $\langle\tau,t\rangle$ can be updated:

$$\langle\tau,t\rangle \leftarrow Min(\langle\tau,t\rangle,\mathcal{N}_\alpha(\langle\tau,t\rangle)+1).$$

We can iteratively update $\langle\tau,t\rangle$ for any event $\alpha\in\mathcal{E}$, until achieving convergence. It is easy to prove that this procedure always terminates and that the fixpoint is the answer to the all-pair shortest path problem, regardless of the order of updates, as long as all next-state functions are considered often enough. However, different orders might lead to

huge variations in the size of the EV$^+$MDDs encoding the intermediate results, as well as the runtime. Saturation [20] has been shown to be an effective fixpoint iteration scheme that tends to minimize peak memory consumption and accelerate convergence. Recall the idea of exploiting the locality and event partition based on the top, $\mathcal{N}_k(\langle\tau,t\rangle)$ can also be computed locally and $\langle\tau,t\rangle$ can be updated solely considering nodes at or below level k, instead of recomputing the overall EV$^+$MDD. Moreover, we can repeatedly update $\langle\tau,t\rangle$ using $\mathcal{N}_k,\ldots,\mathcal{N}_1$ until convergence, at which point we say that the nodes are *saturated* on level k. $\langle\tau,t\rangle$ is *saturated* on level k iff

$$\forall j \leq k, \langle\tau,t\rangle \equiv \mathit{Min}(\langle\tau,t\rangle,\mathcal{N}_j(\langle\tau,t\rangle)+1).$$

We divide the iteration into L phases according to the saturation scheme. The k^{th} phase begins only after $\langle\tau,t\rangle$ is saturated up to level $k-1$ and completes when it is saturated up to level k. An important idea is that every time we compute $\mathcal{N}_k(\langle\tau,t\rangle)$, we expect to keep the results saturated up to level k. These L-phase local fixpoint iterations execute bottom-up, until reaching the global fixpoint.

Figure 5.2 and 5.3 shows the pseudocode to compute TCD_ϕ. This algorithm augments the transitive closure algorithm of [82] by computing distances between state pairs instead of a simple boolean reachability relation. MDD a encodes the set of states $\mathit{Sat}(\phi)$. Lines 5-7 and 12-14 in procedure *TCDSat*, and Lines 7-9 in procedure *TCDRelProdSat* constrain all paths between pairs of states to be along states in the set encoded by MDD a. We assume that the MDDs encoding $\{\mathcal{N}_L,\ldots,\mathcal{N}_1\}$ have been computed and are globally available. As we use the QFI-reduction rule [77], $\mathcal{N}_k$ is an MDD with the root at level k.

```
ComputeTCD(mdd a)                                          • a encodes Sat(φ)
 1  evmdd ⟨1,t_0⟩ ← EVMDDencode(𝒩);
 2  return TCDSat(a, ⟨1,t_0⟩);
```

Figure 5.2: Computing *TCD*.

The main procedure is a dual recursion between *TCDSat* and *TCDRelProdSat*. *TCDSat* computes the fixpoint in the k^{th} phase. In Line 13, it calls *TCDRelProdSat* on lower levels to compute $\mathcal{N}_k(\langle\tau,t\rangle)$. *TCDRelProdSat* computes $\mathcal{N}_k(\langle\tau,t\rangle)$ recursively and saturate the results at the end (Line 13) and thus returns a saturated result; this reflects the idea of aggressively computing local fixpoints on nodes as soon as they have been created. According to our experience [21, 20, 82], this scheme greatly speeds up convergence and reduces memory requirements in intermediate results for typical asynchronous systems.

While the saturation scheme exploits asynchronous event locality to speed up iterations, our algorithm does not impose any requirement on the system under verification. For systems where no natural asynchronous partition of the transition relation exists, such as fully synchronous systems, our algorithm is still applicable by letting $\mathcal{N}_L = \mathcal{N}$ and an empty $\mathcal{N}_k$ for $1 \leq k < L$. In this case, the algorithm degrades to a stepwise procedure always operating from the top level, but still benefits from the efficiency of EV$^+$MDDs. In our experience, using an asynchronous partition and saturation, when possible, results in much faster runtime than using the monolithic next-state function and the stepwise procedure. Thus, although Section 5.4 only discusses asynchronous systems, the algorithm in this chapter is fully general.

```
evmdd TCDSat(mdd a, evmdd ⟨μ,n⟩)
 1  if n = Ω then return ⟨μ,Ω⟩;
 2  if InCache_TCDSat(a, n, ⟨λ,r⟩) then return ⟨λ + μ,r⟩;
 3  level k ← n.lvl;    node t ← NewNode(k);    mdd r ← N_Unprimed(k)
 4  for i, j ∈ S_Unprimed(k) s.t. n[i][j].val ≠ ∞ do
 5    if a[j] ≠ 0 then                                  • constrain the path in a
 6      t[i][j] ← TCDSat(a[j], n[i][j]);
 7    else t[i][j] ← n[i][j]; endif
 8  endfor
 9  for i ∈ S_Unprimed(l) s.t. n[i].val ≠ ∞ do
10    repeat
11      for j,j' ∈ S_l s.t. n[i][j].val ≠ ∞ ∧ r[j][j'].node ≠ 0 do
12        if a[j'] ≠ 0 then                             • constrain the path in a
13          ⟨η,u⟩ ← TCDRelProdSat(a[j'], n[i][j], r[j][j']);
14          t[i][j'] ← Min(t[i][j'], ⟨η + 1,u⟩);       • incr. distance
15        endif
16      endfor
17    until ⟨λ,t⟩ does not change;
18  endfor
19  ⟨λ,t⟩ ← Normalize(t);    InsertUT(t);    CacheAdd_TCDSat(a, n, ⟨λ,t⟩);
20  return ⟨λ + μ,t⟩;
```

```
evmdd TCDRelProdSat(mdd a, evmdd ⟨μ,n⟩, mdd r)
 1  if n = Ω then return ⟨μ,Ω⟩;                        • r = 1 in this case
 2  if InCache_TCDRelProdSat(a, n, r, ⟨λ,t⟩) then
 3    return ⟨λ + μ,t⟩;
 4  level k ← n.lvl;    node t ← NewNode(k);
 5  for i ∈ S_Unprimed(k) s.t. n[i].val ≠ ∞ do
 6    for j, j' ∈ S_Unprimed(k) s.t. n[i][j].val ≠ ∞ ∧ r[j][j'] ≠ 0 do
 7      if a[j'] ≠ 0 then                               • constrain the path in a
 8        ⟨η,u⟩ ← TCDRelProdSat(a[j'], n[i][j], r[j][j']);
 9        t[i][j'] ← Min(t[i][j'], ⟨η,u⟩);
10      endif
11    endfor
12  endfor
13  ⟨λ,t⟩ ← TCDSat(a, Normalize(t));    InsertUT(t);    CacheAdd_TCDRelProdSat(a, n, r, ⟨λ,t⟩);
14  return ⟨λ + μ,t⟩;
```

Figure 5.3: Building TCD_ϕ.

5.3 Discussion

In this section, we discuss two extensions of our approach proposed above. First, Section 5.3.1 applies the idea of the above section to shortest witness generation for other properties. Then, we tackle fairness in EG.

5.3.1 Shortest witness generation beyond EG

We extend the approach of Section 5.2.1 to shortest witness generation (SWG) for more general properties of the form $\mathsf{E}\psi$, where ψ is a path formula and $\mathsf{E}\psi$ does not necessarily a CTL property. The resulting witnesses also constitute shortest counterexamples for $\mathsf{A}\neg\psi$.

Reviewing our TCD-based algorithm for shortest $\mathsf{EG}\phi$ witness generation, we can summarize the following steps:

1. Represent the length of a witness as a function, usually the sum of several *witness segments*. This is the objective of the minimization problem we need to solve and the minimal value is the shortest length of witnesses.

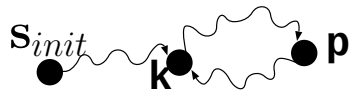
2. Encode the objective function with an EV$^+$MDD based on TCD, usually the sum of several EV$^+$MDDs, each of which corresponds to a witness segment.

3. Find a minimum solution using *MinState* from EV$^+$MDD encoding the objective function. The solution can be several states "inducing" the shortest witness.

4. Build each witness segment, which is a shortest path between two states, and connect these segments sequentially to obtain a shortest witness.

In SWG for EG, the objective function is Formula 5.1, the sum of stem and cycle. The central step is to find a minimal solution, knot $\mathbf{k}^*$, inducing a shortest witness. The extension of this approach to a complete framework able to handle all path formulas is non-trivial and beyond the scope of this chapter. Instead, we present the basic idea in an informal way, by discussing the following two widely used properties.

• **Witnesses for** $\mathsf{E}(\mathsf{GF}\phi)$. It descripts a witness as shown in Figure 5.3.1, which is a path from the initial state to a cycle containing a state satisfying ϕ.



We introduce function *CycleDist* based on *TCD*:

$$CycleDist(\mathbf{i},\mathbf{j}) = \begin{cases} TCD(\mathbf{i},\mathbf{i}) & \text{if } \mathbf{i}=\mathbf{j} \\ TCD(\mathbf{i},\mathbf{j}) + TCD^{-1}(\mathbf{i},\mathbf{j}) & \text{otherwise,} \end{cases}$$

where $TCD = TCD_{true}$. Then, we need to find a state pair $(\mathbf{k},\mathbf{p})$, where $\mathbf{k}$ is the knot, which connects the stem and the cycle, and $\mathbf{p} \in Sat(\phi)$, which belongs to the cycle. Each witnesses consists of three segments: paths from $\mathbf{s}_{init}$ to $\mathbf{k}$, from $\mathbf{k}$ to $\mathbf{p}$, and from $\mathbf{p}$ to $\mathbf{k}$. The objective function for the minimization problem is

$$\min_{\mathbf{k}\in\mathcal{S},\mathbf{p}\in Sat(\phi)} \big(TCD^{stem}(\mathbf{k}) + CycleDist(\mathbf{k},\mathbf{p})\big). \tag{5.2}$$

The minimal solution $(\mathbf{k}^*,\mathbf{p}^*)$ induces a shortest witness, consisting of three witness segments.

• **Witnesses for** $\mathsf{E}[\mathsf{F}(r \wedge \mathsf{G}\neg s)]$. These are counterexamples to CTL properties of the form $\mathsf{AG}(r \rightarrow \mathsf{AF}s)$, which describe liveness: e.g., once a process issues a request (r), it will be eventually satisfied (s). Witnesses for $\mathsf{E}[\mathsf{F}(r \wedge \mathsf{G}\neg s)]$ reflect possible starvation in the system. For notational consistency with the previous section, let $\phi = \neg s$.

There are two types of witnesses for this property, as shown in Figure 5.4, where each circle denotes a state and solid black circles denote states in $Sat(\phi)$. We can solve these two cases separately and then find a global minimal result. In the first case, Figure 5.4(a),

a witness consists of paths from $\mathbf{s}_{init}$ to a state $\mathbf{r} \in \mathcal{S}_r$ and from $\mathbf{r}$ to a knot $\mathbf{k}$, on a ϕ-cycle, and such that ϕ holds along the path between $\mathbf{r}$ and $\mathbf{k}$. In this case, there are three witness segments and the minimization objective is:

$$\min_{\mathbf{r},\mathbf{k} \in Sat(\phi)} \big(TCD^{stem}(\mathbf{r}) + TCD^{triv}_{\phi}(\mathbf{r},\mathbf{k}) + TCD^{cycle}_{\phi}(\mathbf{k}) \big).$$

In the second case, Figure 5.4(b), the stem leads to a knot $\mathbf{k}$ on a ϕ-cycle that contains a state $\mathbf{r} \in \mathcal{S}_r$. This case is similar to the SWG problem for $\mathsf{EGF}\phi$, except for replacing *TCD* with TCD_ϕ in *CycleDist*, to constrain the cycles to $Sat(\phi)$. A shortest witness for this case can then be generated accordingly.

5.3.2 Shortest fair witness

In this section, we consider *Büchi fairness*, which can be specified with $n > 0$ sets of states $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n$. A fair witness is a path leading to a *fair cycle*, which contains a state $\mathbf{i}_m \in \mathcal{F}_m$ for each $\mathcal{F}_m$. To simplify the discussion, we only explain how to generate shortest fair witness for $\mathsf{EG}true$, as the same idea can be extended to other properties. The complexity of this problem has been shown to be NP-complete in [32].

We employ the idea in [71] by adding a *fairness flag* $\mathcal{S}_f$ as a submodel in TCD. $\mathbf{f} \in \mathcal{S}_f$ can be considered as a n-bit array, where i^{th} bit indicates whether the i^{th} fairness constraint has been fulfilled on a path. Let $\perp \in \mathcal{S}_f$ be the initial state where all bits are 0, and $\top \in \mathcal{S}_f$ be the state where all bits are 1, as all constraints are fulfilled. Define the operation $Set(\mathbf{f}, m)$ to set the m^{th} bit to 1. The new TCD, denoted by TCD^f, can be expressed as an integer function on $(\mathbf{i}, \mathbf{j}, \mathbf{f})$, encoding the length of the shortest path that starts in $\mathbf{i}$, ends in $\mathbf{j}$, and satisfies the fair constraints indicated by $\mathbf{f}$. TCD^f can be build

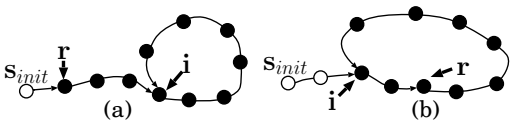


Figure 5.4: A witness for $\mathsf{EF}(r \wedge \mathsf{EG}\neg s)$.

recursively by the following rule, using a similar algorithm as the one discussed above:

$$\mathbf{j}\in\mathcal{N}(\mathbf{i}) \Rightarrow TCD^f(\mathbf{i},\mathbf{j},\bot) = 1$$

$$\wedge \min_{\mathbf{j}\in\mathcal{N}^{-1}(\mathbf{k})}(TCD^f(\mathbf{i},\mathbf{j},\mathbf{f}))=d \Rightarrow TCD^f(\mathbf{i},\mathbf{k},\mathbf{f})=d+1$$

$$\wedge TCD^f(\mathbf{i},\mathbf{j},\mathbf{f})=d \wedge \mathbf{j}\in\mathcal{F}_m \Rightarrow TCD^f(\mathbf{i},\mathbf{j},Set(\mathbf{f},m))=d+1.$$

Now the problem can be converted to witness generation without fair constraints. The minimization objective is:

$$\min_{\mathbf{k}\in\mathcal{S}}\big(TCD^{stem}(\mathbf{k}) + TCD^f(\mathbf{k},\mathbf{k},\top)\big).$$

The resulting witness can be mapped to the original system by eliminating n auxiliary steps that set fairness flag. This approach shares the same complexity with that in [71], and retains the benefits of using EV$^+$MDDs and saturation.

5.4 Experimental results

We implemented the proposed approach in SMART [16] and report experimental results running on an Intel Xeon 2.53GHz workstation with 36GB RAM under Linux 2.6.18. We also implemented the BFS-based algorithm of Figure 5.1 using MDD in SMART. We compare our results with those from the verification tool SAL [31]. Petri net models for SMART

were converted to models in the SAL input language. The results from these algorithms are in the following columns:

• **"SMART-TCD"**: the TCD-based algorithm we propose. If it completes in the time limit, it returns a shortest witness with length L^*, used as an oracle for the other algorithms.

• **"SAL-BMC"**: Bounded model checker in SAL. We have two sets of runtimes, by setting the bound B to L^* and $L^* - 1$ respectively, so that SAL-BMC tackles a satisfiable or unsatisfiable SAT problem, respectively.

• **"SMART-BFS"**: MDD-based witness generation implemented in SMART according to the algorithm in Figure 5.1. For the BFS-based algorithm, we run two sets of experiments. In Column "100 runs", we run the BFS-based algorithm 100 times and list the length of the shortest witness generated among these 100 runs, as well as the total runtime for the 100 runs, in subcolumn "L" and "time" respectively. In Column "runs till shortest", since we know L^* from SMART-TCD, we run the BFS-based algorithm repeatedly until it generates one of the shortest witnesses. The number of runs and runtimes required to generate the shortest witness using the BFS-based algorithm are listed in subcolumns "R" and "time", respectively. Also here we set a runtime limit of one hour. Since the BFS-based algorithm is randomized, the results in subcolumns "R" and "time" are the average over 100 experiments.

• **"SAL-WMC"**: BDD-based symbolic model checker in SAL. It generates a witness of length L without optimization.

The comparison metrics are runtime (columns "time", all measured in seconds) and length of the generated witness. Table 5.1 presents results on eight models, including

mutual exclusion protocols (*peterson* and *bakery* in [63]), leader election protocol (*leader*), the dining philosopher problem (*phil*), a closed queue network (*cqn*), an arbiter protocol (*arbiter*), a factory automation model (*kanban*), and the two *robin* and *slot* protocols [15]. The sizes of the state spaces of these models are parameterized by an integer N. The first three columns list the model names, the parameters, and the sizes of state spaces.

A bounded model checker can find the shortest witness using binary search; this requires running a SAT solver $O(\lceil log_2 L^* \rceil)$ times, to both generate a shortest counterexample and prove that no shorter counterexamples exist. Thus, the sum of the runtimes for $B = L^*$ and $B = L^* - 1$ is a reasonable lower bound for the runtime required by SAL-BMC to find a shortest witness. The results in Column "SMART-TCD" and "SAL-BMC" shows that SAL-BMC achieves obvious speedup over SMART-TCD only on *kanban*, but performs much worse in *robin*, *slot* and *cqn*. Even provided with L^* as the bound, SAL-BMC still requires much more time to find the witness in these three models. These results demonstrate the efficiency of our approach.

SAL-WMC and SMART-BFS are based on the same idea, but use different data structures, i.e., BDDs vs. MDDs. Neither of them can guarantee shortest witnesses. However, SMART-BFS runs much faster than SAL-WMC, due to the efficiency of our MDD library and our encoding of next-state functions. It is not surprising that SMART-BFS runs orders of magnitude faster than SMART-TCD because computing TCD is much more expensive than the image computations in SMART-BFS. On the other hand, SMART-TCD generates much shorter witness for *slot*, *arbiter* and *cqn* than SMART-BFS. Thanks to EV$^+$MDD and saturation, SMART-TCD completes on complex models with more than 10^{10} states and, on

cqn and *phils*, it runs even faster than SAL-WMC, which does not attempt to minimize the witness length.

For *cqn* and *arbiter*, SMART-TCD generates much shorter witnesses than SAL, while SMART-BFS fails to find a shortest witness within the time limit. Figure 5.5 illustrates how the runtime increases and the shortest length of witnesses found decreases as the BFS-based algorithm runs repeatedly and cumulatively in model *slot*5, *cqn*20, and *arbiter*10. Similar results can be observed in *arbiter*. The x-axis (in logarithmic scale) indicates the total number of runs, the solid line (associated with the left y-axis) shows the total runtime, and the dotted line (associated with the right y-axis) shows the shortest length of witnesses found. For comparison, the thin solid line and the dotted line mark the runtime SMART-TCD and L^*, respectively. We can see that runtime grows almost linearly, and many runs (recall that the x-axis is in logarithmic scale) are needed to find a short witness. Within the given runtime, the SMART-BFS produces much longer witnesses than the SMART-TCD. This is analogous to simulation-based verification, which, while it provides good coverage for simple designs, requires unacceptable runtimes to reach corner cases in complex designs. SMART-BFS randomly chooses the next step at each iteration, just as in unguided simulation. If there are only a few witnesses, as in *phils* and *kanban*, SMART-BFS can find a shortest witness in few runs with high probability, although, even in these cases, it cannot prove that there is no shorter witness without exhaustively searching all possible witnesses. If there are many witnesses, SMART-BFS might instead only be able to generate very long witnesses even after a long runtime, as Figure 5.5 illustrates. In this case the SMART-TCD becomes a better choice to generate a short witness, indeed a guaranteed shortest witness.

Model	N	SS	SMART-TCD		SAL-BMC		SAL-WMC		SMART-BFS			
					time	time			100 runs		runs till shortest	
			L^*	time	$B=L^*$	$B=L^*-1$	L	time	L	time	R	time
kanban	6	1.12×10^7	3	7.93	0.0	0.1	18	10.37	3	< 0.01	2.96	< 0.01
	8	1.33×10^8	3	67.86	0.1	0.1	10	16.85	3	< 0.01	3.00	< 0.01
	10	1.00×10^9	3	441.28	0.1	0.1	10	62.09	3	< 0.01	2.97	< 0.01
leader	3	8.49×10^2	15	0.47	0.1	8.56	15	0.03	15	0.03	3.36	< 0.01
	4	1.15×10^4	20	34.60	0.47	1017.33	59	0.80	20	0.18	11.75	0.03
	5	1.50×10^5	25	4746.63	4.14	TO	149	14.30	25	0.52	100.67	0.58
phils	10	1.86×10^6	4	0.05	0.08	0.04	38	0.32	4	0.04	21.91	0.01
	20	3.46×10^{12}	4	0.26	0.05	0.25	47	42.07	4	0.06	24.90	0.02
	100	4.96×10^{62}	4	45.58	0.57	35.98	–	TO	4	0.13	26.85	0.06
robin	10	2.30×10^4	40	0.07	5.35	TO	43	0.35	40	0.04	1.08	< 0.01
	20	4.71×10^7	80	0.30	321.24	TO	83	8.42	80	0.27	1.06	0.01
	30	7.24×10^{10}	120	0.78	3957.47	TO	120	152.07	120	0.65	1.04	0.03
slot	5	5.38×10^4	17	2.26	11.95	5.18	131	0.17	21	0.10	3620.16	4.16
	6	5.75×10^5	20	11.08	0.84	142.49	182	0.78	25	0.24	93411.96	243.99
	7	6.22×10^6	23	46.00	4197.17	611.84	483	2.43	46	0.51	–	TO
arbiter	10	2.04×10^4	10	0.26	1.27	2.45	37	0.1	20	0.05	–	TO
	15	9.83×10^5	15	19.31	33.01	49.71	74	0.44	39	0.11	–	TO
	20	4.19×10^7	20	2625.28	1597.63	1025.43	107	0.91	58	0.32	–	TO
cqn	20	1.93×10^{11}	40	9.76	5682.55	2542.93	–	TO	150	2.51	–	TO
	30	1.66×10^{17}	60	183.20	TO	TO	–	TO	322	10.17	–	TO
	40	1.51×10^{23}	80	3322.41	TO	TO	–	TO	625	25.97	–	TO
peterson	2	2.28×10^2	11	0.11	0.02	0.08	12	0.06	12	< 0.01	866.04	0.05
	3	1.47×10^4	24	477.29	7.84	244.54	37	0.61	37	0.32	–	TO
bakery	2	1.11×10^3	11	0.27	0.04	0.10	11	0.1	22	0.04	1100.66	0.10
	3	1.39×10^5	–	TO	N/A	N/A	161	2.71	65	2.21	N/A	N/A

Table 5.1: Results for EG witness generation.

5.5 Summary

We presented a saturation-based algorithm for shortest EG witness generation. We proposed a symbolic techniques using EV$^+$MDDs to compute the Transitive Closure with Distance (TCD), which compactly represents distances between each pair of states. Then, the shortest EG witness can be identified symbolically. We also extended this approach to tackle shortest witness generation for other properties and shortest fair witness generation.

Computing TCD is the bottleneck in our approach. Techniques to speed up this computation should be investigated, including dynamic variable ordering. Coupled with EV$^+$MDDs, the transitive closure provides an elegant way of analyzing quantitative proper-

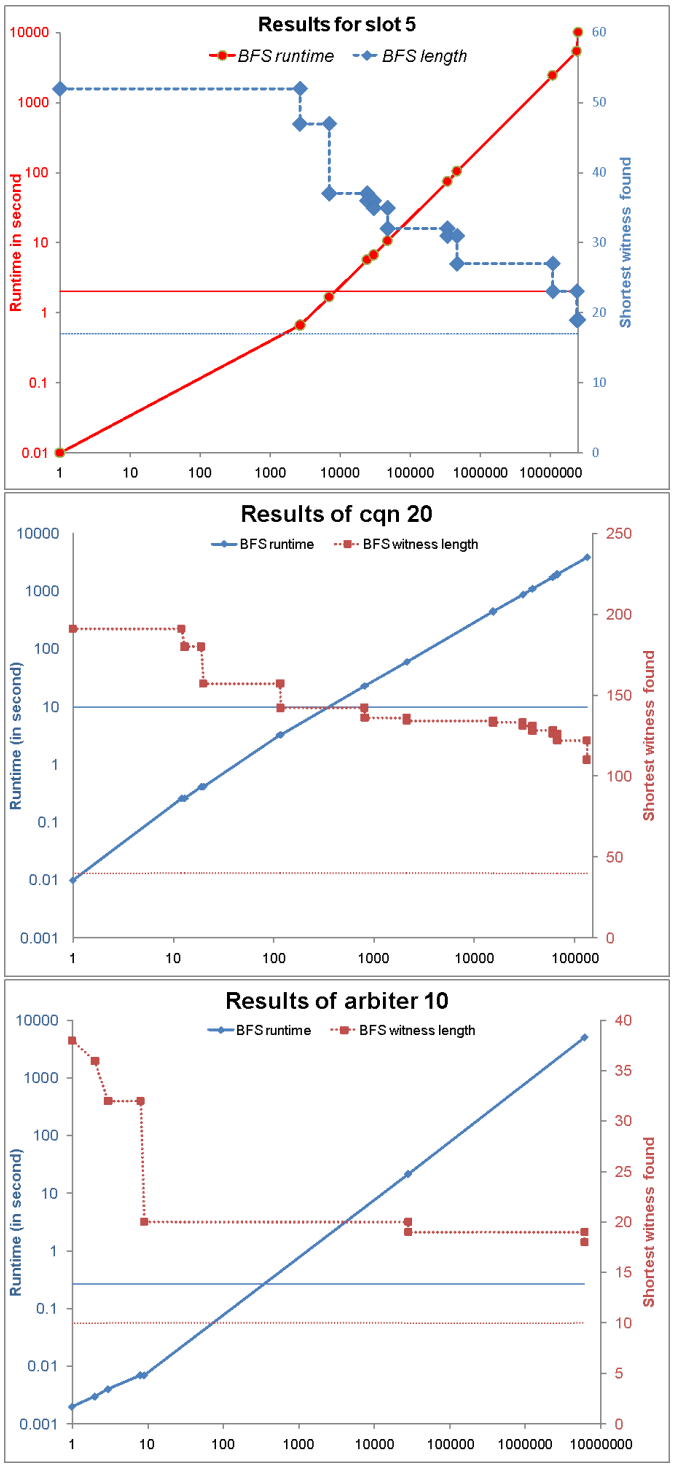


Figure 5.5: Runtime and witness length of the BFS-based algorithm on *slot*, *cqn*, and *arbiter*.

ties of traces in complex asynchronous systems, such as probabilistic model checking, which we intend to investigate in future work.

Part II

Probabilistic model checking

Chapter 6

EVMDD-based two-phase Gauss-Seidel iteration

In this chapter, we address two problems: the steady-state solution of ergodic CTMCs and the unbounded until, both of which require to solve linear systems. We present a new symbolic approach for these two problems using iterative methods, based on EVMDDs to store an indexing function for the structured states and the transition rate matrix. The approach is memory efficient for general structured CTMCs, and supports both Jacobi and Gauss-Seidel iterations. In particular, our main contribution is a new two-phase algorithm to perform Gauss-Seidel iterations with a reduced overhead for the traversal of the decision diagram (a cost also encountered by Kronecker-based approaches). Then, we show how even better speedup can be achieved through a caching scheme. The complexity of our algorithm is linear in the number of nonzero entries in the transition rate matrix, and, even more importantly, it is independent of the number L of submodels in which the CTMC

is decomposed under most common conditions. This is an improvement over previous structured methods, which are plagued by this L factor in practice. The advantages of our algorithm are supported by experimental results and a comparison with the tool PRISM.

Assuming the CTMC is ergodic, let $\boldsymbol{\pi} \in \mathbb{R}^{\mathcal{S}}_{\geq 0}$ be the steady-state probability vector, where $\boldsymbol{\pi}[\mathbf{i}]$ is the steady-state probability of state $\mathbf{i}$. $\boldsymbol{\pi}$ can be computed as the solution of the linear system $\boldsymbol{\pi}\mathbf{Q} = \mathbf{0}$ subject to $\sum_{\mathbf{i}\in\mathcal{S}} \boldsymbol{\pi}[\mathbf{i}] = 1$.

Iterative methods (where a new approximation $\boldsymbol{\pi}^{new}$ of $\boldsymbol{\pi}$ is computed from the current approximation $\boldsymbol{\pi}^{old}$), are usually employed to solve linear systems. For steady-state solution, we consider the Jacobi iteration,

$$\boldsymbol{\pi}^{new}[\mathbf{j}] = \mathbf{h}[\mathbf{j}] \cdot \sum_{\mathbf{i}\neq\mathbf{j}} \boldsymbol{\pi}^{old}[\mathbf{i}] \cdot \mathbf{R}[\mathbf{i},\mathbf{j}], \tag{6.1}$$

and the Gauss-Seidel iteration, which has usually faster convergence and requires a total order on $\mathcal{S}$, i.e., for any two states $\mathbf{i} \neq \mathbf{j}$, either $\mathbf{i} \succ \mathbf{j}$ or $\mathbf{i} \prec \mathbf{j}$,

$$\boldsymbol{\pi}^{new}[\mathbf{j}] = \mathbf{h}[\mathbf{j}] \cdot \left(\sum_{\mathbf{i}\prec\mathbf{j}} \boldsymbol{\pi}^{new}[\mathbf{i}] \cdot \mathbf{R}[\mathbf{i},\mathbf{j}] + \sum_{\mathbf{i}\succ\mathbf{j}} \boldsymbol{\pi}^{old}[\mathbf{i}] \cdot \mathbf{R}[\mathbf{i},\mathbf{j}] \right). \tag{6.2}$$

As we discussed in Chapter 2, given $\mathbb{P}_{\bowtie p}(\phi \mathsf{U} \psi)$, the state space can be partitioned into $\mathcal{S}_0, \mathcal{S}_1,$ and $\mathcal{S}_?$, and we only need to compute the probabilities for states in $\mathcal{S}_?$. Let $\boldsymbol{\nu}$ be the solution where for all $\mathbf{i} \in \mathcal{S}_?$, $\boldsymbol{\nu}[\mathbf{i}]$ is the probability $Prob(\mathbf{i}, \phi \mathsf{U} \psi)$. $\boldsymbol{\nu}$ is the solution of the linear system $\mathbf{P}_e\boldsymbol{\nu} + \mathbf{B} = \boldsymbol{\nu}$ where $\mathbf{B}[\mathbf{i}] = \sum_{\mathbf{j}\in\mathcal{S}_1} \mathbf{P}_e[\mathbf{i},\mathbf{j}]$. The Gauss-Seidel iteration for this linear system is:

$$\boldsymbol{\nu}^{(k+1)}[\mathbf{i}] = \sum_{\mathbf{j}\prec\mathbf{i}} \boldsymbol{\nu}^{(k+1)}[\mathbf{j}] \cdot \mathbf{P}_e[\mathbf{i},\mathbf{j}] + \sum_{\mathbf{j}\succ\mathbf{i}} \boldsymbol{\nu}^{(k)}[\mathbf{j}] \cdot \mathbf{P}_e[\mathbf{i},\mathbf{j}] + \mathbf{B}[\mathbf{i}].$$

We do not need to create a separate $\mathbf{P}_e$ just for unbounded until. With $\mathbf{Q}$ and $\mathbf{h}$ in place, we utilize decision diagram exploration to carry out the above computation only for

states in $\mathcal{S}_?$. The algorithm for unbounded until is pretty similar to that for steady-state solution of an ergodic CTMC, thus we focus first on the steady-state solution and then apply it to the unbounded until.

6.1 Previous work

Plateau [66] presented the first algorithm to solve structured Markov chains encoded by a *Kronecker descriptor*, i.e., a sum (over the set of *events* $\mathcal{E}$) of Kronecker products (over the L submodels) of local matrices: $\widehat{\mathbf{R}} = \sum_{e \in \mathcal{E}} \bigotimes_{k=1}^{L} \mathbf{R}_{k,e}$, where $\mathbf{R}_{k,e} \in \mathbb{R}^{\mathcal{X}_k \times \mathcal{X}_k}$. The numerical solution used the *shuffle* algorithm to multiply a full vector by a Kronecker descriptor, whose apparently excellent complexity is studied in [9, 36]; its generalization is presented in [30]. However, [10] points out that a better complexity than explicit vector-by-sparse-matrix multiplication is achieved only when the matrices $\mathbf{R}_{k,e}$ are not very sparse, that is, when

$$\eta(\widehat{\mathbf{R}})/|\widehat{\mathcal{S}}| > L^{\frac{1}{L-1}}. \tag{6.3}$$

and this is rarely the case, as $\widehat{\mathbf{R}}$ contains at most $|\mathcal{E}|$ nonzero entries per row for practical systems.

To make things worse, Kronecker-based algorithms suffer from three drawbacks. First, as initially proposed, a "potential" probability vector $\widehat{\boldsymbol{\pi}}$ of size $|\widehat{\mathcal{S}}|$ is employed; in practical models, $|\widehat{\mathcal{S}}| \gg |\mathcal{S}|$, thus much memory is wasted. A modified version operating on the "actual" probability vector $\boldsymbol{\pi}$ can be used, but at the cost of additional indexing overhead [10]. Second, not all nonzero entries defined by a Kronecker descriptor correspond to possible transitions in the CTMC, if $\widehat{\mathcal{S}} \supset \mathcal{S}$: some describe transitions from unreachable

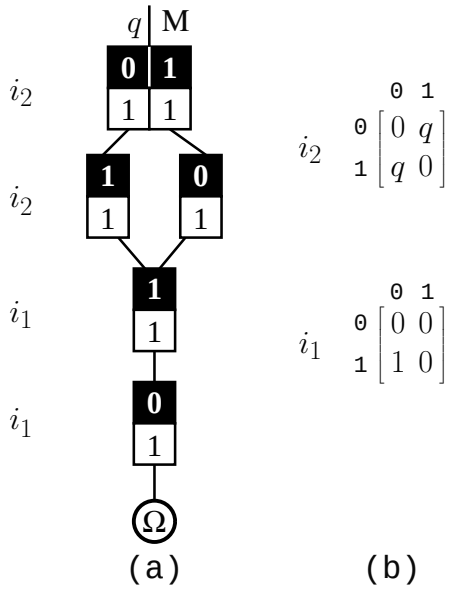


Figure 6.1: EV*MDD (a) and Kronecker (b) representations of transition rate.

states to reachable states. In each Gauss-Seidel iteration, this requires a test to avoid unnecessary computations (when using $\widehat{\boldsymbol{\pi}}$) or array indexing errors (when using $\boldsymbol{\pi}$). Finally, a Kronecker descriptor can always be defined, but sometimes at the cost of splitting events (resulting in a set $\mathcal{E}$ of exponential size) or merging state variables (resulting in sets $\mathcal{X}_k$ of exponential size), either way making the approach unfeasible in such cases.

Iterative methods perform a sequence of multiplications of the probability (row) vector and the column vectors of the rate matrix. Unlike Jacobi, Gauss-Seidel requires the column vectors to be accessed in order, which increases the complexity when the matrix is stored with Kronecker descriptors or decision diagrams. With a Kronecker descriptor, each matrix must be repeatedly traversed to derive different columns, this is the main overhead in the previous algorithms. In the worst case, a Kronecker approach has an overhead factor

L when $\widehat{\mathcal{S}} = \mathcal{S}$ (i.e., when there are no spurious entries from unreachable states to reachable states); however, if $\widehat{\mathcal{S}} \supset \mathcal{S}$, the overhead is even higher, $\eta(\widehat{\mathbf{R}}[\widehat{\mathcal{S}}, \mathcal{S}])/\eta(\mathbf{R}) \cdot L$.

More recent work has then employed decision diagrams of various types to store $\mathbf{R}$. PRISM [52], perhaps the best known probabilistic model checker, uses multi-terminal binary decision diagrams (MTBDDs) [26] to store $\mathbf{R}$. A *hybrid approach* is normally used, which stores $\mathbf{R}$ as a single MTBDD and $\boldsymbol{\pi}$ as a full probability vector of size $|\mathcal{S}|$. Furthermore, [50] even explores this approach when the probability vector (which is then the main storage limitation) is kept on disk. While MTBDDs are general (they can encode any matrix), they do not help mitigate the overhead in accessing the columns of $\mathbf{R}$. Hence, prior to performing Gauss-Seidel iterations, PRISM converts the MTBDD storing $\mathbf{R}$ into a two-level data structure [58], equivalent to splitting $\mathbf{R}$ into many submatrices and using sparse storage for them. This scheme often results in compact encoding and fast runtime, as fast as with a traditional sparse method, in spite of the overhead to derive the submatrices; however, it has potentially large matrix storage requirements, especially when few submatrices can be reused in the storage scheme, as shown in Section 6.5.

Our algorithm can also be categorized as hybrid, as it stores $\boldsymbol{\pi}$ in full. The difference is that we store $\mathbf{R}$ using EV*MDDs, which can be more compact than MTBDDs, and, more fundamentally, it uses a two-phase scheme to carry out Gauss-Seidel iterations, resulting in better complexity.

6.2 Symbolic iterative methods

We now describe how we store $\mathbf{R}$ and $\mathcal{I}$ using EVMDDs. Then, we introduce our implementation of Jacobi in Section 6.2.2 and our main contribution, a two-phase algorithm for Gauss-Seidel, in Section 6.2.3.

6.2.1 Transition rate matrix and probability vector storage

Compared with MTBDDs using the same variable order, EVMDDs have been shown to be at least as compact, and possibly exponentially more compact [69]. In particular, we use an EV*MDD $\mathcal{M}$ to encode $\widehat{\mathbf{R}}$, since this often requires less memory than encoding $\mathbf{R}$. However, encoding $\mathbf{R}$ does result in tighter upper-bound complexity, as we will see. All algorithms we introduce work with either approach.

As usual when encoding transition relations or matrices with decision diagrams, we employ an *interleaved order* for the $2L$ levels of the EV*MDD $\mathcal{M}$, i.e., we interleave the "from" state variables with the "to" state variables. Thus, $\mathcal{M}$ has a path labeled with $(i_L, j_L, ..., i_1, j_1)$ from the root to Ω having total value $r > 0$ iff the CTMC contains a transition from $\mathbf{i} = (i_L, ..., i_1)$ to $\mathbf{j} = (j_L, ..., j_1)$ with rate r. To simplify the notation, from now on we call level $2l$ in $\mathcal{M}$ the (*unprimed*) level l and level $2l-1$ the *primed* level l, or l'. We also write the $2L$-tuple $(i_L, j_L, ..., i_1, j_1)$ as $\mathbf{i}||\mathbf{j}$, so that the rate $\mathbf{R}[\mathbf{i},\mathbf{j}]$ is retrieved in $\mathcal{M}$ as $\mathcal{M}[\mathbf{i}||\mathbf{j}]$.

We use instead an EV$^+$MDD $\mathcal{I}$ to encode the indexing function [60] that maps a potential state $\mathbf{i}$ to ∞ if it is unreachable, or to its lexicographic position in $\mathcal{S}$, i.e., its

```
void JacobiIteration(EV*MDD M, EV+MDD I)                    • M = ⟨ρ_max,r⟩; I = ⟨0,o⟩
 1  pi_old ← "initial guess";
 2  num_iter ← 0;
 3  repeat
 4     pi_new ← "zero vector";
 5     JacobiRecur(M, I, I);
 6     foreach i ∈ {0, ..., |S| − 1} do pi_new[i] ← pi_new[i] · h[i];
 7     swap(pi_old, pi_new);
 8     num_iter ← num_iter + 1;
 9  until num_iter > MAX_ITER or converged(pi_old, pi_new);
```

```
void JacobiRecur(EV*MDD ⟨rate,m⟩, EV+MDD ⟨src_idx,src⟩, EV+MDD ⟨des_idx,des⟩)  • src.lvl =
des.lvl
 1  k ← src.lvl;
 2  if (src = des = Ω) do
 3    pi_new[des_idx] ← pi_new[des_idx] + pi_old[src_idx] ∗ rate;
 4    return;
 5  endif
 6  for i from 0 to n_k − 1 s.t. m[i] ≠ ⊥ and src[i] ≠ ⊥ do
 7    for j from 0 to n_k − 1 s.t. m[i][j] ≠ ⊥ and des[j] ≠ ⊥ do
 8      r_offset ← src[i].v;    c_offset ← des[j].v;
 9      s ← src_idx + r_offset;    d ← des_idx + c_offset;
10      JacobiRecur( ⟨rate · m[i][j].v, m[i][j].ch⟩, ⟨s, src[i].ch⟩, ⟨d, des[j].ch⟩);
11    endforeach
12  endforeach
```

Figure 6.2: Jacobi iteration.

index i in $\boldsymbol{\pi}$, otherwise (i is called *offset* in [10]). This indexing also determines the order of computations in Gauss-Seidel iterations.

Definition 14 *(Order on local states) In the local state space $\mathcal{S}_k$, we define a total order $0 \prec 1 \prec \cdots \prec n_k - 1$. Let $i \preceq j$ iff $i = j$ or $i \prec j$, $i \succ j$ iff $j \prec i$, and $i \succeq j$ iff $j \preceq i$.*

Definition 15 *(Lexicographic order on states and interleaved state pairs) In the state space $\mathcal{S}$, we define a total order such that $\mathbf{i} = (i_L, ..., i_1) \succ \mathbf{j} = (j_L, ..., j_1)$ iff there is an m such that $i_k = j_k$ for $L \geq k > m$ and $i_m \succ j_m$. Let $\mathbf{i} \preceq \mathbf{j}$ iff $\mathbf{i} = \mathbf{j}$ or $\mathbf{i} \prec \mathbf{j}$, $\mathbf{i} \prec \mathbf{j}$ iff $\mathbf{j} \succ \mathbf{i}$, and*

```
void GaussSeidelIteration(EV*MDD M, EV+MDD I)
 1  Initialize(prob_vector);      num_iter = 0;
 2  repeat
 3     GSForward(M, I, I);
 4     foreach i ∈ {0, ..., |S| − 1} do prob_vector[i] ∗ h[i];
 5     GSBackward(M, I, I);
 6     num_iter ← num_iter + 1;
 7  until num_iter > MAX_ITER or converged(...);        • see text for possible convergence tests
```

```
void GSForward(EV*MDD ⟨rate,m⟩, EV+MDD ⟨src_idx,src⟩, ⟨des_idx,des⟩)
 1 ⋆assert src_idx ⪰ des_idx;                                          • Constraint for Forward phase
 2  if (src = des = Ω) then
 3 ⋆  if (first_time_to_write(des_idx)) then π[des_idx] ← π[src_idx] ∗ rate;          • see text
 4 ⋆  else π[des_idx] ← π[des_idx] + π[src_idx] ∗ rate; endif
 5     return;
 6  endif
 7  for i from 0 to n_k−1 s.t. m[i] ≠ ⊥ and src[i] ≠ ⊥ do
 8     for j from 0 to n_k−1 s.t. m[i][j] ≠ ⊥ and des[j] ≠ ⊥ do
 9 ⋆     if (src_idx ≻ des_idx or i ⪰ j) then
10           r_offset ← src[i].v;      c_offset ← des[j].v;
11           s ← src_idx+r_offset;     d ← des_idx+c_offset;
12 ⋆         GSForward(⟨rate · m[i][j].v,m[i][j].ch⟩, ⟨s,src[i].ch⟩, ⟨d,des[j].ch⟩);
13       endif
14     endfor
15  endfor
```

```
void GSBackward(EV*MDD ⟨rate,m⟩, EV+MDD ⟨src_idx,src⟩, ⟨des_idx,des⟩)
 1 ⋆assert src_idx ⪯ des_idx;                                          • Constraint for Backward phase
 2  if (src = des = Ω) then
 3     π[des_idx] ← π[des_idx] + π[src_idx] ∗ rate ∗ h[src_idx]
 4     return;
 5  endif
 6  for i from 0 to n_k−1 s.t. m[i] ≠ ⊥ and src[i] ≠ ⊥ do
 7     for j from 0 to n_k−1 s.t. m[i][j] ≠ ⊥ and des[j] ≠ ⊥ do
 8 ⋆     if (src_idx ≺ des_idx or i ⪯ j) then
 9           r_offset ← src[i].v;      c_offset ← des[j].v;
10           s ← src_idx+r_offset;     d ← des_idx+c_offset;
11 ⋆         GSBackward(⟨rate · m[i][j].v,m[i][j].ch⟩, ⟨s,src[i].ch⟩, ⟨d,des[j].ch⟩);
12       endif
13     endfor
14  endfor
```

Figure 6.3: Gauss-Seidel iteration.

$\mathbf{i} \preceq \mathbf{j}$ *iff* $\mathbf{j} \succeq \mathbf{i}$. *An exactly analogous total order can be defined on interleaved state pairs, i.e.,* $2L$*-tuples of the form* $\mathbf{i}||\mathbf{j}$.

For example, the state lexicographic order in Fig. 2.12 is $00 \prec 01 \prec 10 \prec 11$, and is encoded by EV$^+$MDD $\mathcal{I}$. States $00, 01, 10, 11$ are mapped to indices from 0 to 3. The indexing function encoding we use was introduced in [60], although there it was not formalized as an EV$^+$MDD, and it is optimal, in the sense that, for given a variable order, it corresponds to a minimum size EV$^+$MDD among all the possible total orders for $\mathcal{S}$ (the proof for this property is based on the fact that this EV$^+$MDD is isomorphic to the MDD encoding $\mathcal{S}$). Considering instead interleaved state pairs, we have, for example, $01||10 \prec 10||11$, since $(01||10) = (0110)$, $(10||11) = (1101)$ and $(0110) \prec (1101)$.

The following Lemma is immediately proven:

Lemma 16 *If* $\mathbf{i} \prec \mathbf{i}'$ *and* $\mathbf{j} \prec \mathbf{j}'$*, then* $(\mathbf{i}||\mathbf{j}) \prec (\mathbf{i}'||\mathbf{j}')$.

Note that Lemma 16 does not hold in the reverse direction, i.e., $(\mathbf{i}||\mathbf{j}) \prec (\mathbf{i}'||\mathbf{j}')$ implies that at least one of $\mathbf{i} \prec \mathbf{i}'$ and $\mathbf{j} \prec \mathbf{j}'$ holds, but not necessarily both.

6.2.2 Symbolic Jacobi iterations

Function *JacobiIteration* in Fig. 6.2 shows our Jacobi iteration. To simplify notation, we let

$$m[i][j].ch \triangleq (m[i].ch)[j].ch$$

$$m[i][j].v \triangleq m[i].v \cdot (m[i].ch)[j].v.$$

Starting from their root nodes, *JacobiRecur* simultaneously traverses $\mathcal{M}$ and two copies of $\mathcal{I}$ in a depth-first fashion. Each recursive call to *JacobiRecur* executes on an EV*MDD node m in $\mathcal{M}$ and two EV$^+$MDD nodes *src* and *des* in $\mathcal{I}$. We call tuple (m, src, des) a *snapshot*, indicating the current position of the traversal. When reaching the terminal nodes, a path $\mathbf{i}||\mathbf{j}$ has been traversed in $\mathcal{M}$, thus Line 3 multiplies $\boldsymbol{\pi}^{old}[\mathbf{i}]$ by $\mathbf{R}[\mathbf{i},\mathbf{j}]$ and adds the result to $\boldsymbol{\pi}^{new}[\mathbf{j}]$. The pseudocode uses *pi_old* and *pi_new* instead of the semantically equivalent $\boldsymbol{\pi}_{old}$ and $\boldsymbol{\pi}^{new}$ to stress that the former are accessed using state indices, which, for $\mathbf{i}$ and $\mathbf{j}$, are given by $\mathcal{I}[\mathbf{i}]$ and $\mathcal{I}[\mathbf{j}]$, respectively. Analogously, the rate $\mathbf{R}[\mathbf{i},\mathbf{j}]$ is given by $\mathcal{M}[\mathbf{i}||\mathbf{j}]$. Even if $\mathcal{M}$ stores $\widehat{\mathbf{R}}$ and not just $\mathbf{R}$, *JacobiRecur* never traverses $\mathbf{i}||\mathbf{j}$ in $\mathcal{M}$ if $\mathbf{i}$ or $\mathbf{j}$ is unreachable, because this procedure ensures that $(i_L, ..., i_1)$ and $(j_L, ..., j_1)$ are indexed in $\mathcal{I}$ so they are in $\mathcal{S}$.

We now focus on the order in which the path $\mathbf{i}||\mathbf{j}$ is traversed in $\mathcal{M}$. *JacobiRecur* implements Equation 6.1 by traversing all reachable pairs $\mathbf{i}||\mathbf{j}$ in lexicographic order, as defined above, since in each step, local lexicographic order is followed in Lines 6 and 7. Thus, the following Lemma holds.

Lemma 17 *Given* $\mathbf{i}, \mathbf{i}', \mathbf{j}, \mathbf{j}' \in \mathcal{S}$*, JacobiRecur traverses path* $\mathbf{i}||\mathbf{j}$ *in* $\mathcal{M}$ *before traversing* $\mathbf{i}'||\mathbf{j}'$ *iff* $\mathbf{i}||\mathbf{j} \prec \mathbf{i}'||\mathbf{j}'$.

Lemma 17 reveals the relationship between lexicographic order and the order of EVMDD-based computation. This relationship is exploited to carry out the Gauss-Seidel iteration in the next subsection.

An important point about performance should be made: the cost of performing the "for i" and "for j" loops in *JacobiRecur* is proportional to the number of nonzero $m[i][j]$,

not $(n_k)^2$, since our decision diagrams library employs a sparse representation for nodes, so that only the nonzero children of a node are stored, in order.

6.2.3 Symbolic Gauss-Seidel iterations

We divide each Gauss-Seidel iteration into two phases. The *forward phase* considers only transitions from states with *higher index* to states with *lower index*, or *forward transitions*, and generates an intermediate result $\boldsymbol{\pi}^{int}$, a vector of size $|\mathcal{S}|$:

$$\boldsymbol{\pi}^{int}[\mathbf{j}] = \mathbf{h}[\mathbf{j}] \cdot \sum_{\mathbf{i} \succ \mathbf{j}} \boldsymbol{\pi}^{old}[\mathbf{i}] \cdot \mathbf{R}[\mathbf{i}, \mathbf{j}].$$

The subsequent *backward phase* considers instead transitions from states with *lower index* to states with *higher index*, or *backward transitions*, and generates the result $\boldsymbol{\pi}^{new}$, also a vector of size $|\mathcal{S}|$:

$$\boldsymbol{\pi}^{new}[\mathbf{j}] = \boldsymbol{\pi}^{int}[\mathbf{j}] + \mathbf{h}[\mathbf{j}] \cdot \sum_{\mathbf{i} \prec \mathbf{j}} \boldsymbol{\pi}^{int}[\mathbf{i}] \cdot \mathbf{R}[\mathbf{i}, \mathbf{j}].$$

Fig. 6.4 illustrates this two-phase iteration. Squares stand for entries in the probability vector in ascending lexicographic order from left to right, i.e., $\mathbf{i} \prec \mathbf{j} \prec \mathbf{k}$, and edges stand for transitions. It can be shown that this two-phase computation implements Equation 6.2 with the same number of additions and multiplications. However, its memory

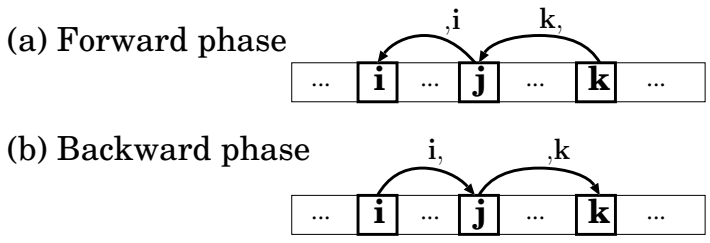


Figure 6.4: Two-phase iteration.

requirements appear higher, as it uses and additional vector $\boldsymbol{\pi}^{int}$. Thus, it would be desirable if, in both phases, we could directly overwrite the old values in the same array, without having to allocate $\boldsymbol{\pi}^{int}$. To achieve this goal, we must enforce some constraints on the order in which the probability vector entries are updated:

- In the forward phase, when rewriting the entry for $\mathbf{j}$, all computation corresponding to transitions from $\mathbf{j}$ to $\mathbf{i}$, where $\mathbf{i} \prec \mathbf{j}$, must have been completed, i.e., the old value of the entry for $\mathbf{j}$ will not be used in the remaining portion of the forward iteration.

- In the backward phase, when using the value of the entry for $\mathbf{j}$ to rewrite the value of the entry for $\mathbf{k}$, all computation corresponding to the transition from $\mathbf{i}$ to $\mathbf{j}$, where $\mathbf{i} \succ \mathbf{j}$, must have been completed, i.e., at that point, the entry for $\mathbf{j}$ has its new value and can be used to compute the value for $\mathbf{k}$.

The pseudo-code *GaussSeidelIteration* in Fig. 6.3 depicts the proposed algorithm for Gauss-Seidel iteration. In each iteration, first *GSForward*, then *GSBackward*, are invoked to execute the forward and backward phases in sequence. The differences between *GSForward* or *GSBackward* and *JacobiRecur* are indicated with $\star$ and are needed to constrain the traversal only to forward or backward transitions, where $src\_idx > des\_idx$ or $src\_idx < des\_idx$, respectively. The traversals executed by *GSForward* or *GSBackward* are a subset of those executed by *JacobiRecur*, following the same order of *JacobiRecur*, i.e., if $\mathbf{i}||\mathbf{j}$ is traversed before $\mathbf{i}'||\mathbf{j}'$ in *GSForward* or *GSBackward*, the same holds in *JacobiRecur*. What we need to prove now is that *GSForward* and *GSBackward* satisfy the above constraints, thus, together, produce the same result as one Gauss-Seidel iteration.

Lemma 18 *Assume* $\mathbf{i} \prec \mathbf{j} \prec \mathbf{k}$. *Then:*

- *(Forward phase) In GSForward, when* $\mathbf{k}||\mathbf{j}$ *is traversed, all paths* $\mathbf{j}||\mathbf{i}$ *corresponding to a positive rate have been traversed.*
- *(Backward phase) In GSBackward, when* $\mathbf{j}||\mathbf{k}$ *is traversed, all paths* $\mathbf{i}||\mathbf{j}$ *corresponding to a positive rate have been traversed.*

In the forward phase, given any two paths $\mathbf{j}||\mathbf{i}$ and $\mathbf{k}||\mathbf{j}$, Lemma 1 ensures that $\mathbf{j}||\mathbf{i} \prec \mathbf{k}||\mathbf{j}$ since $\mathbf{j} \prec \mathbf{k}$ and $\mathbf{i} \prec \mathbf{j}$. Because *GSForward* executes the same order as *JacobiRecur*, Lemma 2 holds and $\mathbf{j}||\mathbf{i}$ should be traversed before $\mathbf{k}||\mathbf{j}$, Similarly, we can prove the case in the backward phase.

Given the lemmas we proved, the next theorem follows immediately.

Theorem: Procedure *GaussSeidelIteration* in Fig. 6.3 correctly implements the update of $\boldsymbol{\pi}$ specified by Equation 6.2.

In *GSForward*, when updating an entry in the probability vector, we need to first identify whether the value is the old value from previous iteration or the intermediate result in current forward iteration. In the former case, we overwrite the old value with the newly computed result, which is safe as we proved above; in the latter case, we add the new results to this entry. We can check which case applies using a simple "trick": before *GSForward*, we negate each value in the probability vector and we store a positive value for an entry after updating this entry in *GSForward*. When updating an entry, we check if this entry is negative, in which case we know that it is the first time we update it in the current iteration.

Another implementation issue is convergence checking. Since we overwrite the old values before the new values are available in the probability vector, we either have to choose a convergence checking method which does not depend on the difference between old and new vectors (e.g., residual computation), or accept to store two vectors ($\boldsymbol{\pi}^{new}$ and either $\boldsymbol{\pi}^{old}$ or the vector $\boldsymbol{\pi}^{new}$ some number of iterations, e.g., 10, before, which, as some authors have suggested, tends to reduce the possibility of erroneously mistaking slow convergence for achieved convergence). In our implementation, we store $\boldsymbol{\pi}^{old}$ in each iteration, but we stress that this is only for convergence checking, not for the iteration.

Symbolic Gauss-Seidel iteration for unbounded until

Figure 6.5 shows the top-level pseudo code for this algorithm. $\mathcal{P}$ is the EV*MDD encoding the transition probability matrix. The difference with Figure 6.3 is that *GSBackward* performs first and then *GSForward*.

```
void GSIterationUnboundedU(EV*MDD P, EV+MDD I)
 1  Initialize(prob_vector);
 2  num_iter = 0;
 3  repeat
 4     GSBackward(P, I, I);
 5     foreach i ∈ {0, ..., |S| − 1} do prob_vector[i] + B[i];
 6     GSForward(P, I, I);
 7     num_iter ← num_iter + 1;
 8  until num_iter > MAX_ITER or converged(...);      • see text for possible convergence tests
```

Figure 6.5: Top level Gauss Seidel iteration for unbounded U operator.

If we initialize $\boldsymbol{\nu}^{(0)}$ to be a zero vector, the following theorem from [5] holds.

Theorem 19 *In a Gauss-Seidel iteration, if* $\boldsymbol{\nu}^{(0)} = \mathbf{0}$*, then* $\boldsymbol{\nu}^{(k+1)} \geq \boldsymbol{\nu}^{(k)}$.

This means that, as the iteration proceeds, $\boldsymbol{\nu}^{(k)}$ grows monotonically. We can judge whether convergence is reached by checking the sum of all probabilities in $\boldsymbol{\nu}$. If $\sum \boldsymbol{\nu}^{(k+1)} - \sum \boldsymbol{\nu}^{(k)} < \varepsilon$, where ε is an acceptable error, we can terminate the Gauss-Seidel iteration. Thus, we do not need to store the old vector $\boldsymbol{\nu}^{(k)}$ when computing $\boldsymbol{\nu}^{(k+1)}$.

6.3 Speeding up the iteration

In the following discussion, we refer to the algorithm in last section as the "plain" algorithm. The main performance issue in the plain algorithm is multiple traversals of the same snapshot. In one iteration, there can be multiple paths leading to the same snapshot, thus multiple recursions on this snapshot will be repeated in one iteration. Meanwhile, each iteration repeats the same procedure. Caching some intermediate results can avoid duplicate computation and substantially speed up the iteration. In this section, we introduce a cache scheme to speed-up the iteration, especially our two-phase iteration. [52] proposed a similar idea, storing some intermediate results in submatrices attached to MTBDD nodes, but our scheme has two important differences. First, instead of storing the rate values in the submatrices, each rate value is factored into two parts, based on the EV*MDD, and stored in a data structure consisting of two levels of arrays. More submatrices can be shared with this scheme than with that of [52]. Second, to facilitate our two-phase iteration, the cache scheme preserves the order of decision diagram traversal. More sophisticated cache schemes might be explored in the future, but the experimental results of Section 6.5 show that even this simple scheme is able to balance memory and runtime well.

```
void Preprocess(EV*MDD ⟨rate,m⟩, EV+MDD ⟨src_idx,src⟩, ⟨des_idx,des⟩)
 1  if num_path(m) < THRESHOLD then
 2    entry_array a ← empty_array();      CacheRecur(⟨1.0,m⟩, ⟨0,src⟩, ⟨0,des⟩, a);
 3    Push(CacheArray, (cache_entry(rate, src_idx, des_idx), a));
 4    return;
 5  endif
 6  for i from 0 to n_k−1 s.t. m[i] ≠ ⊥ and src[i] ≠ ⊥ do
 7    for j from 0 to n_k−1 s.t. m[i][j] ≠ ⊥ and des[j] ≠ ⊥ do
 8      r_offset ← src[i].v; c_offset ← des[j].v;
 9      Preprocess( ⟨rate · m[i][j].v,m[i][j].ch⟩, ⟨src_idx+r_offset,src[i].ch⟩, ⟨des_idx+c_offset,des[j].ch⟩);
10    endforeach
11  endforeach
```

```
void CacheRecur(EV*MDD ⟨rate,m⟩, EV+MDD ⟨src_idx,src⟩,⟨des_idx,des⟩, entry_array a)
 1  if src = des = Ω then
 2    Push(a, cache_entry(rate, src_idx, des_idx));      return;
 3  endif
 4  for i from 0 to n_k−1 s.t. m[i] ≠ ⊥ and src[i] ≠ ⊥ do
 5    for j from 0 to n_k−1 s.t. m[i][j] ≠ ⊥ and des[j] ≠ ⊥ do
 6      r_offset ← src[i].v; c_offset ← des[j].v;
 7      CacheRecur(⟨rate · m[i][j].v,m[i][j].ch⟩, ⟨src_idx+r_offset,src[i].ch⟩, ⟨des_idx+c_offset,des[j].ch⟩);
 8    endforeach
 9  endforeach
```

```
void GSForwardCache()
 1  foreach a in CacheArray do
 2    rate_A ← a.rate;    src_idx_A ← a.src_idx;    des_idx_A ← a.des_idx;
 3    if src_idx_A ⪰ des_idx_A then
 4      foreach b in a.cache_array do
 5        rate_B ← b.rate;    src_idx_B ← b.src_idx;    des_idx_B ← b.des_idx;
 6        if (src_idx_B ≻ des_idx_B or src_idx_A ≻ des_idx_A) then
 7          rate ← rate_A * rate_B;    src_idx ← src_idx_A + src_idx_B;    des_idx ← des_idx_A + des_idx_B;
 8          if (first_time_to_write[des_idx]) then π[des_idx] ← π[src_idx] * rate;
 9          else π[des_idx] ← π[des_idx] + π[src_idx] * rate;
10        endif
11      endfor
12    endif
13  endfor
```

Figure 6.6: Algorithm to preprocess and build the cache array.

First, we select a set SP of *split point* nodes on unprimed levels in $\mathcal{M}$ forming a cut in the decision diagram, so that each path from the root of $\mathcal{M}$ to Ω is split into two parts: from the root to a node $m \in SP$ and from m to Ω. Each iteration reaches the nodes of

SP in the same order, thus results in the same computation of ($rate_A$, src_idx_A, des_idx_A), representing the partial transition rate and source and destination indices calculated within the recursion up to that point (subscript A stands for "above-path"). On the other hand, the recursion started on snapshot (m, src, des), regardless of how this snapshot is reached, generates the same sequence of ($rate_B$, src_idx_B, des_idx_B) triplets, representing the factor for the resulting transition rate and the source and destination index additions based on the "below-path" (subscript B stands for "below-path"). The indices and value of a nonzero entry in $\mathbf{R}$ can be derived by first finding the above-path and below-path corresponding to this entry and then multiplying $rate_A$ and $rate_B$ for the rate, or adding src_idx_A to src_idx_B and des_idx_A to des_idx_B for the source and destination indices.

As the base of our cache structure, a *cache entry* is defined as a tuple (*rate*, *src_idx*, *des_idx*). The structure of the cache illustrated in Fig. 6.7 is a two-level array, with a top-level storing above-path cache entries and linking to a bottom-level array of storing below-path cache entries. Since several different traversals can lead to the same snapshot (m, src, des), the corresponding bottom-level array can be reused, as shown in Fig. 6.7, where there are two pointers to the same bottom-level array from the top-level array.

Fig. 6.6 shows how to build the cache. *Preprocess*$(\mathcal{M}, \mathcal{I}, \mathcal{I})$ is called prior to beginning the steady-state solution computation. *Preprocess* and *CacheRecur* traverse the decision diagrams just as *JacobiRecur* does. When *Preprocess* reaches a node m from which the number of below-paths emanating from it is less than a threshold, it identifies m as a split point and invokes *CacheRecur* to create the bottom-level array stored in a. When *CacheRecur* reaches the terminal level, it pushes the obtained cache entry

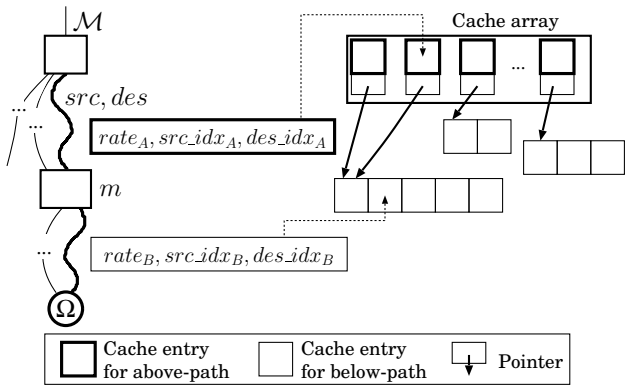


Figure 6.7: Structure of the cache.

$(rate, src\_idx, des\_idx)$ to the end of a. After that, the pointer to a is stored in the top-level cache array, *CacheArray*, associated with the cache entry storing the above-path when invoking *CacheRecur*. After the cache array is built, we can execute Jacobi or Gauss-Seidel by traversing the top-level array in order and following each link to the bottom-level arrays as they are encountered. Function *GSForwardCache* in Fig. 6.7 shows the pseudo-code for forward Gauss-Seidel iteration using the cache.

The number of cache entries stored is given by:

$$\sum_{m \in SP} (\mathscr{A}(m) + \mathscr{P}(m))$$

where $\mathscr{A}(m)$ is the number of paths from the root to the node m in SP and $\mathscr{P}(m)$ is the number of below-paths, from m to Ω. The two limiting cases, the cut consisting of just the root or just Ω, correspond to storing $\mathbf{R}$ as a sparse matrix in a single bottom-level array or in the top-level array, respectively. There exist optimal sets SP that minimize memory consumption, but our algorithm *Preprocess* based on the numbers of below-paths from a split point is just a heuristic to find a hopefully good SP. As future work, we intend

to explore multiple-level cache structures and investigate the trade-off between memory consumption and runtime on different cache structures.

6.4 Complexity and discussion

Since Gauss-Seidel imposes a restriction on the order in which entries of the probability vector are computed, thus entries of the transition rate matrix are accessed, while Jacobi does not, it is obvious that the complexity of one Gauss-Seidel iteration is at least that of one Jacobi iteration (the advantage of Gauss-Seidel stems from the potentially better convergence rate, thus lower number of iterations). Using our approach, the complexity of a Gauss-Seidel iteration is at most twice that of a Jacobi iteration, since both *GSFoward* and *GSBackward* traverse a subset of the entries *JacobiRecur* does. Thus, we can focus on the complexity of a Jacobi iteration as specified in Fig. 6.2, and compare it with alternative approaches.

Explicit storage algorithms can achieve an essentially optimal complexity when we have "access by column", meaning that we can access the columns of $\mathbf{R}$ in order and traverse only the nonzero entries in a given column, all of which is easy to provide in an explicit sparse storage setting. Thus, one Gauss-Seidel iteration requires traversing the matrix once and has complexity $O(\eta(\mathbf{R}))$ which, to the best of our knowledge, can only be surpassed by the shuffle algorithm in the very special cases where Equation 6.3 holds.

If a cache array is built and exploited, the complexity of each iteration is still $O(\eta(\mathbf{R}))$, but the multiplicative constant is worse than for sparse storage because more computation is required to calculate indices and rates.

Considering the plain algorithm introduced in Section 6.2.2, the number of "real" computations executed in Line 3 of *JacobiRecur* is $O(\eta(\mathbf{R}))$. The following discussion focuses on the overhead for decision diagram traversal. Before each recursion, two new indices and a new rate are computed, costing two additions and one multiplication.

The overhead of traversal can be measured by the number of recursive calls to *JacobiRecur*. As mentioned, the complexity of nonterminal recursion is determined by the number of nonzero $m[i][j]$, denoted by $\mathscr{C}(m)$. Thus, each node m in $\mathcal{M}$ contributes a cost $\mathscr{A}(m) \cdot \mathscr{C}(m)$, where $\mathscr{A}(m)$ counts the paths from the root of $\mathcal{M}$ to m. In the worst case, all these paths are traversed and the overall complexity of traversal is

$$\sum_{k=1}^{L} \sum_{m.lvl=k} \mathscr{A}(m) \cdot \mathscr{C}(m).$$

For $k = 1$, we have

$$\sum_{m.lvl=1} \mathscr{A}(m) \cdot \mathscr{C}(m) = \mathscr{P}(\mathcal{M}) = \eta(\mathbf{R}), \tag{6.4}$$

where $\mathscr{P}(\mathcal{M})$ denotes the number of (nonzero) paths in $\mathcal{M}$.

Moreover, considering adjacent levels, we have

$$\sum_{m.lvl=k+1} \mathscr{A}(m) \cdot \mathscr{C}(m) = \sum_{q.lvl=k} \mathscr{A}(q).$$

Letting $\sum_{m.lvl=k} \mathscr{A}(m) \cdot \mathscr{C}(m) = (\sum_{m.lvl=k} \mathscr{A}(m)) \cdot \mathscr{C}_k$, where $\mathscr{C}_k$ represents the average value of $\mathscr{C}(m)$ for the nodes m at level k, the overall complexity can be written as:

$$\eta(\mathbf{R}) + \frac{\eta(\mathbf{R})}{\mathscr{C}_1} + \frac{\eta(\mathbf{R})}{\mathscr{C}_1 \cdot \mathscr{C}_2} + \cdots + \frac{\eta(\mathbf{R})}{\mathscr{C}_1 \cdot \cdots \cdot \mathscr{C}_{L-1}}. \tag{6.5}$$

The worst-case complexity $O(L \cdot \eta(\mathbf{R}))$ is reached when $\mathscr{C}_k \approx 1$ for all $1 \leq k \leq L-1$. If $\min\{\mathscr{C}_k : 1 \leq k \leq L-1\} = \mathscr{C}_{min} > 1$, the complexity becomes $O(\eta(\mathbf{R}))$, since the sum in

Equation 6.5 is bounded by the sum of a geometric sequence with ratio $1/\mathscr{C}_{min}$ starting with $\eta(\mathbf{R})$. Moreover, the larger $\mathscr{C}_{min}$ is, the closer the performance of our algorithm is to that of a sparse matrix approach. The lowest overhead is achieved when $\mathscr{C}_k = (n_k)^2$, corresponding to CTMCs with full transition rate matrices. For a given $\eta(\mathbf{R})$, a sparser $\mathbf{R}$ increases the complexity of our algorithm, but no more than L times the best-case complexity.

A previous Kronecker-based algorithm for Gauss-Seidel iterations using actual state space, called *Act-Cl$_2$-GSD* in [10], is based on vector multiplication and the central step is to obtain a column of $\mathbf{R}$ from the Kronecker descriptor. A similar idea can be implemented in our setting and, in this case, the overall complexity becomes

$$\sum_{k=1}^{L} \sum_{m.lvl=k} \mathscr{A}(m) \cdot \mathscr{P}(m),$$

where $\mathscr{P}(m)$ is the number of paths below m. This complexity is always worse than that of our new approach, substantially so if $\mathbf{R}$ is very dense since, then, $\mathscr{P}(m) \gg \mathscr{C}(m)$ especially for m at higher levels. The improvement comes from our two-phase iteration, which traverses any path in $\mathscr{A}(m)$ only twice for any m, while *Act-Cl$_2$-GSD* could traverse such path many (up to $|\mathcal{S}|$) times. To the best of our knowledge, this is the best complexity for a symbolic algorithm implementing Gauss-Seidel iterations on the actual state space.

Equation 6.4 applies if $\mathcal{M}$ stores $\mathbf{R}$. If instead $\mathcal{M}$ stores $\widehat{\mathbf{R}}$, we must replace $\mathbf{R}$ with $\widehat{\mathbf{R}}$ in the above results. In some (perhaps many, or even most) models, $\eta(\widehat{\mathbf{R}}) \gg \eta(\mathbf{R})$, and that is the reason we choose to store $\mathbf{R}$. Thanks to EV*MDDs, we can often store $\mathbf{R}$ with reasonable memory consumption. The complexity of our proposed algorithms can then be summarized as follows:

- If a cache array is employed, the complexity of a Jacobi or Gauss-Seidel iteration is $O(\eta(\mathbf{R}))$.

- Using the plain algorithms of Sections 6.2.2 and 6.2.3, the upper-bound complexity of a Jacobi or Gauss-Seidel iteration is $O(L \cdot \eta(\mathbf{R}))$. However, if $\mathscr{C}_{min} > 1$, this complexity is reduced to $O(\eta(\mathbf{R}))$.

- If $\mathcal{M}$ encodes $\eta(\widehat{\mathbf{R}})$, the complexity of a Jacobi or Gauss-Seidel iteration is $O(L \cdot \eta(\widehat{\mathbf{R}}))$. However, if $\mathscr{C}_{min} > 1$, this complexity is again reduced to $O(\eta(\widehat{\mathbf{R}}))$.

Since the hybrid engine in PRISM stores split submatrices of $\mathbf{R}$, its complexity is also $O(\eta(\mathbf{R}))$. Compared with our algorithm, the hybrid engine in PRISM also saves computation because it does not need to multiply edge values as it descends the diagram. However, we argue that the main problem of the hybrid engine in PRISM is the potentially formidable memory consumption when the scheme of storing split submatrices fails to scale, as we will see in the experimental results. It is worth noting that the proposed algorithm and the above complexity are applicable if $\mathbf{R}$ is encoded using MTBDD. The only difference in the algorithm, if using MTBDD, is that rate values are not calculated by multiplying the edge values during the traversal, but obtained directly from terminal nodes, and the same complexity holds. However, the size of the MTBDD encoding $\mathbf{R}$ may be larger than that of EV*MDD.

6.5 Experimental results

We implemented Gauss-Seidel using the plain algorithm of Section 6.2 and the cache scheme enhancement of Section 6.3 in our stochastic analysis tool SMART [16] and run experiments on an Intel Xeon 2.53GHz workstation with 36GB RAM running Linux 2.6.18. We use four CTMC models distributed with PRISM as benchmarks. The most important criteria in our comparison are runtime per iteration and memory consumption for the transition rate matrix storage (including any cache).

First, we compare the runtime of our plain algorithm with *Act-Cl*$_2$*-GSD* in [10], a symbolic algorithm for Gauss-Seidel iteration on actual state space. Although the original *Act-Cl*$_2$*-GSD* employs Kronecker-based storage for $\widetilde{\mathbf{R}}$, we implemented it in SMART using EV*MDD storage of $\mathbf{R}$ (experimentally, we observed that this did not substantially affected performance or memory requirements of the method). The runtime per iteration is listed in Fig. 6.8. These results clearly show that our algorithm greatly outperforms the previous algorithm, consistent with the theoretical results of the last section.

Then, we compare our algorithm with PRISM using both its sparse and hybrid engines. Fig. 6.9 lists both runtime and memory experimental results, where "MO" means "out of memory", "TO" means the total runtime is more than 8 hours, and "NA" means "not available". The second and third column in the table report the size of the state space and the average number of nonzero entries in each row of $\mathbf{R}$, indicating how sparse $\mathbf{R}$ is. In PRISM, the sparse engine employs sparse storage for $\mathbf{R}$, while the hybrid engine uses MTBDD and sparse submatrices. PRISM memory consumption is listed in Columns "sparse" and "hybrid", the latter using the format $M + S$, where M is the memory for

N	$\lvert\mathcal{S}\rvert$	*Act-Cl*$_2$*-GSD*[10]	Plain algorithm
cluster $L = 16$			
16	1.01×10^4	0.07	<0.001
32	3.86×10^4	0.40	0.002
64	1.51×10^5	2.41	0.006
fms $L = 8$			
3	6.52×10^3	0.03	<0.001
4	3.59×10^4	0.26	0.003
5	1.52×10^5	1.70	0.015
kanban $L = 16$			
2	4.63×10^3	0.02	<0.001
3	5.84×10^4	0.36	0.003
4	4.54×10^5	3.36	0.027
polling $L = 2N + 1$			
13	1.59×10^5	1.05	0.069
14	3.44×10^5	2.54	0.015
15	7.37×10^5	6.08	0.037

Figure 6.8: Runtime (sec) per iteration: *Act-Cl*$_2$*-GSD* vs. our plain algorithm.

MTBDDs and S is the memory for sparse submatrices. Our plain algorithm uses EV*MDDs and our cache algorithm uses a cache list to store $\mathbf{R}$. Column "$\widehat{\mathbf{R}}$+Cache" lists the memory to encode $\widehat{\mathbf{R}}$ plus the memory for the cache list, and Column "$\mathbf{R}$" lists the memory to encode $\mathbf{R}$. We report the number of iterations (Column "Iter.") and the average runtime per iteration, in seconds. The main comparison is between the proposed algorithm with cache and the hybrid engine in PRISM, and for these two, we list the runtime per iteration and total runtime, including both the runtime for iterations and building the auxiliary data structures, in two columns. EV*MDDs are much more compact than sparse storage and, except for model "cluster", use less memory than MTBDDs do in PRISM. The inefficiency of EV*MDDs in "cluster" is due to some large nodes with N outgoing edges, and it could be solved by splitting a single level into several levels in the EV*MDD. Our cache scheme, although simple, often uses less memory than PRISM, while achieving comparable runtime,

N	$\lvert\mathcal{S}\rvert$	$\frac{\eta(\mathbf{R})}{\lvert\mathcal{S}\rvert}$	Memory for $\mathbf{R}$ (Mb)				Runtime (sec)							
			PRISM		Proposed		PRISM				Proposed			
			Sparse	Hybrid	$\mathbf{R}$	$\bar{\mathbf{R}}$+Cache	Iter.	Sparse	H(iter|total)		Iter.	Plain	C(iter|total)	
cluster $L = 16$														
128	5.97×10^5	2.56	11.7	0.03+0.12	0.40	0.12+0.48	306	0.02	0.03	9	297	0.44	0.03	9
256	2.37×10^6	2.56	46.5	0.03+0.28	0.79	0.24+0.95	329	0.07	0.19	62	574	1.85	0.17	99
512	9.46×10^6	2.56	538.2	0.03+0.42	1.57	0.47+1.90	358	0.21	0.81	293	1104	7.81	0.81	940
1024	3.78×10^7	2.55	740.8	0.03+0.85	3.14	0.94+3.80	397	2.39	3.20	1276	1104	31.75	3.50	7805
fms $L = 8$														
7	1.63×10^6	1.15	53.3	5.8+6.7	1.30	0.76+8.81	144	0.07	0.16	29	188	2.32	0.21	41
8	4.45×10^6	1.09	151.2	10.3+15.3	1.93	1.14+16.91	151	0.20	0.47	87	205	6.55	0.70	145
9	1.10×10^7	1.06	388.5	14.7+35.0	3.05	1.69+34.39	159	0.51	1.66	306	221	16.96	1.87	415
10	2.53×10^7	1.03	918.9	19.3+71.8	4.25	2.36+62.50	166	1.20	4.23	798	238	40.04	4.14	989
kanban $L = 16$														
6	1.12×10^7	1.88	452.1	0.4+2.9	0.13	0.02+3.02	232	0.53	0.99	233	226	15.49	1.03	232
7	4.16×10^7	1.91	1740.8	0.4+6.3	0.17	0.02+6.37	296	2.54	4.06	1215	289	57.46	4.34	1255
8	1.33×10^8	1.92	5836.8	0.7+17.5	0.22	0.02+12.34	365	6.66	13.15	4847	358	TO	14.69	5261
9	3.84×10^8	1.94	MO	0.7+37.1	0.27	0.02+7.65	493	MO	38.52	17044	432	TO	40.58	17533
polling $L = 2N + 1$														
20	3.14×10^7	2.18	1331.2	0.1+18.1	0.06	0.10+5.13	36	1.61	2.29	87	36	61.75	2.00	72
21	6.60×10^7	2.19	2867.2	0.2+40.1	0.07	0.11+6.64	47	3.35	4.90	241	36	137.50	4.11	148
22	1.38×10^8	2.20	6246.4	0.2+40.1	0.07	0.13+10.41	60	7.56	10.94	681	37	297.52	8.97	332
23	2.89×10^8	2.20	MO	0.2+88.1	0.08	0.14+13.45	72	MO	22.48	1679	38	646.69	19.62	746
flip $L = N$														
14	1.64×10^4	13.99	1.0	7.0+1.0	<0.01	(<0.01)+0.08	48	0.15	<0.01	42	48	0.03	<0.01	<1
15	3.28×10^4	14.99	2.2	13.6+2.5	<0.01	(<0.01)+0.13	52	0.32	<0.01	284	52	0.06	<0.01	<1
16	6.55×10^4	16.00	12.1	26.7+4.1	<0.01	(<0.01)+0.19	55	0.71	0.04	1400	55	0.13	<0.01	<1
20	1.04×10^6	20.16	NA	NA	<0.01	(<0.01)+1.01	NA	NA	NA	NA	70	2.50	0.21	15

Figure 6.9: Experimental results comparing PRISM with the proposed algorithms.

even if, as we pointed out at the end of Section 6.4, our algorithm using EV*MDDs needs to multiply edge values, unlike the hybrid engine in PRISM. The sparse storage in PRISM and our plain algorithm, not surprisingly, are the fastest and slowest, respectively. The hybrid engine in PRISM and our algorithm with cache perform almost equally well.

For the first four models, the submatrix storage for $\mathbf{R}$ in PRISM's hybrid engine scales well because many submatrices in $\mathbf{R}$ turn out to be the same, substantially reducing storage requirements. We consider a model called "flip"[1], modeling random flips on N bits. The rates of flips are determined by multiplying functions of each local state, thus the

[1]The PRISM and SMART files for flip are available at http://www.cs.ucr.edu/∼zhaoy/QEST2012.html

number of different rate values grow exponentially with N, and no submatrix can be shared. We can see from the table that the memory for sparse submatrices in the hybrid engine is close to that for sparse storage of $\mathbf{R}$, and the memory for MTBDDs is much larger than that for the corresponding EV*MDDs. The hybrid engine also requires large amount of runtime to build sparse submatrices, making it much slower than our algorithm. We cannot report results from PRISM for $N > 16$ because PRISM crashes unexpectedly in those cases, so we put "NA" in the corresponding cells.

We can reach the following conclusions based on our experimental results. First, our algorithm achieves an obvious speedup compared with previous symbolic algorithm for Gauss-Seidel iterations on the actual state space. Second, thanks to EV*MDDs, our storage of the transition rate matrix is often more compact than using MTBDDs, and able to tackle some models, like flip, on which MTBDDs do not scale. Third, enhanced with our cache scheme, our algorithm achieves a runtime comparable with PRISM.

6.6 Summary

We presented a symbolic algorithm to compute the stationary solution of ergodic structured CTMCs using Jacobi or Gauss-Seidel. EV*MDDs are used to store the transition rate matrix on the actual state space. By exploiting a lexicographic order on the state space, the Gauss-Seidel iteration can be implemented using a two-phase traversal that allows it to achieve a complexity of $O(\eta(\mathbf{R}))$, which has never been achieved in previous work with structured representations.

Our algorithm raises some interesting questions. First, the variable order not only affects the size of decision diagrams, but also the lexicographic order of the state space and, consequently, the convergence rate of Gauss-Seidel. This relationship ought to be studied in future work. Second, the proposed cache scheme affects the efficiency of our algorithm and the trade-off between memory and runtime when building the cache needs to be studied in greater detail. Finally, while we only addressed Jacobi and Gauss-Seidel, it is worth exploring the use of our algorithm for more sophisticated iterative methods, such as the multi-level algorithm [43], which could achieve faster convergence.

Chapter 7

A bounding semantics for CSL

This chapters tackles the truncation errors in iterative methods, which are widely used in probabilistic model checking. Unlike CTL, where the result is obtained by exploring the state space, the set of states corresponding to a CSL formula is generated by comparing real values obtained from a numerical analysis against some given threshold. Errors and approximations in the numerical analysis steps are inevitable, and may propagate to the resulting set of states. For example, if we seek the set of states with probability ≤ 0.5 (with respect to some condition) and the computed probability on a state is between 0.49 and 0.51, it is not clear whether this state belongs to the result. No matter how high we set the required precision in the numerical analysis, such cases can never be ruled out. Thus, the correctness of CSL results may not be guaranteed due to numerical errors, especially under resource constraints (runtime, memory). In other words, the exact semantics of CSL, defined as a resulting set of states, is not generally achievable in practice.

Worse yet, due to the inability to compute an exact result for a given CSL formula, nested CSL formulas are even more difficult to handle in the current CSL model-checking framework. While a nested CTL formula can be evaluated by simply following its syntax tree from leaf nodes to the root, the corresponding approach for nested CSL formulas can only be proposed after we are able to handle errors in CSL model checking.

We propose a solution to the above problems by defining the result of a CSL formula using bounds (sets). The lower bound set gives the states that "must" satisfy the threshold on the probability and the higher bound set gives the set of states that "might" satisfy the threshold, based on the given precision of the numerical analysis. We focus on the error introduced in the truncation of the iteration and modify the CSL model checking algorithm to provide and support our new CSL semantics with bounds. The proposed algorithms handle nested CSL formulas by taking into account uncertainty in the subformulas.

Using lower and upper bounds to handle model checking uncertainty is not a new idea. In the probabilistic setting, [35] and [46] discuss the application of three-value logic, which reflects a similar idea as our lower and upper bounds. The main difference between our chapter and these previous works is the source of uncertainty. In [35] and [46], uncertainty comes from abstracting models, while in this chapter we discuss the inherent uncertainty arising from the numerical analysis employed in CSL model checking itself. The proposed bounding semantics and techniques for nested CSL formulas also apply to the problem settings in [35] and [46], where abstraction on Markov chains is carried out.

The rest of the chapter is structured as follows. Section 7.1 presents new algorithms to generate bounds on the exact result. Section 7.2 introduces an approach to handle nested CSL formulas. Section 7.3 shows the application of our new techniques to two nontrivial cases. The last section concludes the chapter and points out future work.

7.1 Bounding the probability in CSL model checking

We now introduce our numerical technique to tackle truncation errors and generate bounds when computing the U^I operator. Sections 7.1.1 and 7.1.2 focus on the time-bounded and unbounded U operators, respectively, and exploit the monotonic property of the probability vector being computed. Section 7.1.3 briefly introduces how to handle the point-interval and general interval U operators.

7.1.1 Time-bounded until

Given $\mathcal{M}$ and formula $\mathbb{P}_{\bowtie p}(\phi \mathsf{U}^{[0,t]} \psi)$, we first compute the partition $\{\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_?\}$, then apply the conversion of Section 2.3.3 to obtain CTMC $\mathcal{M}^{\mathcal{S}_0 \cup \mathcal{S}_1}$. Consequently, $\mathbf{Prob}(\phi \mathsf{U}^{[0,t]} \psi)$ for $\mathcal{M}$ equals $\boldsymbol{\nu}(\mathcal{S}_1, t)$ for $\mathcal{M}^{\mathcal{S}_0 \cup \mathcal{S}_1}$. The following equality

$$\boldsymbol{\nu}(\mathcal{S}_?, t) + \boldsymbol{\nu}(\mathcal{S}_1, t) + \boldsymbol{\nu}(\mathcal{S}_0, t) = \mathbf{1} \qquad \text{where } \mathbf{1} \text{ is a vector of 1's}$$

holds at any time $t \geq 0$, since the state of $\mathcal{M}^{\mathcal{S}_0 \cup \mathcal{S}_1}$ is in one of $\mathcal{S}_1$, $\mathcal{S}_0$, or $\mathcal{S}_?$. From now on, let the l and u superscripts indicate lower and upper bounds on the corresponding quantities, respectively. Then, if we can compute $\boldsymbol{\nu}^l(\mathcal{S}_1, t)$, $\boldsymbol{\nu}^l(\mathcal{S}_?, t)$, and $\boldsymbol{\nu}^l(\mathcal{S}_0, t)$, we can let $\boldsymbol{\nu}^u(\mathcal{S}_1, t) = \mathbf{1} - \boldsymbol{\nu}^l(\mathcal{S}_?, t) - \boldsymbol{\nu}^l(\mathcal{S}_0, t)$.

In the traditional use of uniformization, $\boldsymbol{\nu}(\mathcal{S}_1, t)$ is calculated by truncating an infinite sum:

$$\sum_{k=0}^{\infty} \mathbf{P}^k \boldsymbol{\delta}^{\psi} \cdot Poisson[k] \approx \sum_{k=L_\varepsilon}^{R_\varepsilon} \mathbf{P}^k \boldsymbol{\delta}^{\psi} \cdot p[k].$$

We compute the approximate Poisson probability $p[k] \approx Poisson[k]$ for $L_\varepsilon \leq k \leq R_\varepsilon$ using the Fox-Glynn algorithm [38], which first finds the left and right truncation points L_ε and R_ε that ensure $\sum_{k=L_\varepsilon}^{R_\varepsilon} Poisson[k] \geq 1 - \varepsilon$, where ε is the specified acceptable truncation error, then computes $p[k]$ based on the recurrence

$$p[k-1] \cdot qt = p[k] \cdot k \qquad \text{with normalization} \qquad \sum_{k=L_\varepsilon}^{R_\varepsilon} p[k] = 1, \tag{7.1}$$

which results in $p[k] > Poisson[k]$. To calculate a lower bound $\boldsymbol{\nu}^l(\mathcal{S}_1, t)$, we first need to obtain a lower bound $p^l[k]$ for the Poisson probability $Poisson[k]$, thus we substitute the normalization in Equation 7.1 with

$$\sum_{k=L_\varepsilon}^{R_\varepsilon} p^l[k] = 1 - \varepsilon. \tag{7.2}$$

Fig. 7.1 shows the pseudo-code of our modified Fox-Glynn algorithm generating lower bounds for the Poisson probabilities. We then have

$$\boldsymbol{\nu}^l(\mathcal{S}_1, t) = \sum_{k=L_\varepsilon}^{R_\varepsilon} \mathbf{P}^k \boldsymbol{\delta}^{\psi} \cdot p^l[k] < \sum_{k=0}^{\infty} \mathbf{P}^k \boldsymbol{\delta}^{\psi} \cdot Poisson[k] = \boldsymbol{\nu}(\mathcal{S}_1, t).$$

```
{L_ε, R_ε, p^l[L_ε, ..., R_ε]}  FoxGlynnLB(qt, ε) is
 1  m ← ⌊qt⌋;
 2  L_ε, R_ε, w[m] ← Finder(λ, ε);
 3  w[L_ε, ..., R_ε] ← ComputeWeight(L_ε, R_ε, w[m]);
 4  W ← sumup(w[L_ε, ..., R_ε]);
 5  W ← W/(1 − ε);                          • Upper bound on overall weight
 6  p^l[L_ε, ..., R_ε] ← w[L_ε, ..., R_ε]/W;   • Lower bound on Poisson probabilities
 7  return L_ε, R_ε, p^l[L_ε, ..., R_ε];
```

Figure 7.1: Modified Fox-Glynn algorithm for lower bounds on Poisson probabilities.

We could compute $\boldsymbol{\nu}^l(\mathcal{S}_?, t)$ and $\boldsymbol{\nu}^l(\mathcal{S}_0, t)$ with the same technique, but for $\boldsymbol{\nu}^u(\mathcal{S}_1, t)$ we do not need to do that. Considering Eq. 7.2, we have

$$\boldsymbol{\nu}^l(\mathcal{S}_?, t) + \boldsymbol{\nu}^l(\mathcal{S}_1, t) + \boldsymbol{\nu}^l(\mathcal{S}_0, t) = (1-\varepsilon)\cdot\mathbf{1} \quad \text{and} \quad \boldsymbol{\nu}^l(\mathcal{S}_?, t) + \boldsymbol{\nu}(\mathcal{S}_1, t) + \boldsymbol{\nu}^l(\mathcal{S}_0, t) \leq \mathbf{1},$$

which means that we can define the upper bound

$$\boldsymbol{\nu}^u(\mathcal{S}_1, t) = \boldsymbol{\nu}^l(\mathcal{S}_1, t) + \varepsilon\cdot\mathbf{1} = \mathbf{1} - \boldsymbol{\nu}^l(\mathcal{S}_?, t) - \boldsymbol{\nu}^l(\mathcal{S}_0, t) \geq \boldsymbol{\nu}(\mathcal{S}_1, t).$$

By adjusting ε, we can obtain arbitrarily tight bounds.

7.1.2 Unbounded until

Again, we first build $\mathcal{M}^{\mathcal{S}_0 \cup \mathcal{S}_1}$ so that $\mathbf{Prob}(\phi \mathsf{U} \psi)$ in $\mathcal{M}$ equals $\boldsymbol{\nu}(\mathcal{S}_1)$ (recall that $\boldsymbol{\nu}(\mathcal{S}_1) = \lim_{t\to\infty} \boldsymbol{\nu}(\mathcal{S}_1, t)$) in $\mathcal{M}^{\mathcal{S}_0 \cup \mathcal{S}_1}$. Eventually, $\mathcal{M}^{\mathcal{S}_0 \cup \mathcal{S}_1}$ will be absorbed in $\mathcal{S}_0$ or $\mathcal{S}_1$, thus

$$\boldsymbol{\nu}(\mathcal{S}_1) + \boldsymbol{\nu}(\mathcal{S}_0) = \mathbf{1}.$$

Hence, as long as we obtain the lower bounds $\boldsymbol{\nu}^l(\mathcal{S}_1)$ and $\boldsymbol{\nu}^l(\mathcal{S}_0)$, we can define the upper bound $\boldsymbol{\nu}^u(\mathcal{S}_1) = \mathbf{1} - \boldsymbol{\nu}^l(\mathcal{S}_0)$.

$\boldsymbol{\nu}(\mathcal{S}_1)$ is the solution $\boldsymbol{\nu}$ of the linear system $\mathbf{P}\boldsymbol{\nu} + \boldsymbol{\delta}^\psi = \boldsymbol{\nu}$, which can be solved using Gauss-Seidel iterations

$$\nu^{(k+1)}[\mathbf{i}] = \sum_{\mathbf{j}\prec\mathbf{i}} \mathbf{P}[\mathbf{i},\mathbf{j}]\nu^{(k+1)}[\mathbf{j}] + \sum_{\mathbf{j}\succ\mathbf{i}} \mathbf{P}[\mathbf{i},\mathbf{j}]\nu^{(k)}[\mathbf{j}] + \delta^\psi[\mathbf{i}].$$

A practical criterion to terminate the iteration is $||\boldsymbol{\nu}^{(k+1)} - \boldsymbol{\nu}^{(k)}|| < \varepsilon$, where ε is again a parameter expressing the error tolerance. This criterion does *not* guarantee that the result is close to the actual solution, thus the error is *not* predictable from ε. Fig. 7.2 shows a simple

example, $\mathcal{S}_?$ contains two states and we compute $\boldsymbol{\nu}[\mathbf{i}](\mathcal{S}_1)$. It is clear that $\boldsymbol{\nu}[\mathbf{i}](\mathcal{S}_1)=0.5$ for any p, since the two top states are bisimilar. However, if $p \ll \varepsilon$, a näive convergence test stops the iteration at $\boldsymbol{\nu}[\mathbf{i}](\mathcal{S}_1)=p/2$, which is far from the actual result.

If we initialize $\boldsymbol{\nu}^{(0)}$ to be the zero vector, the following theorem holds (Theorem 5.8 in [5]).

Theorem: If $\boldsymbol{\nu}^{(0)} = \mathbf{0}$, $\boldsymbol{\nu}^{(k+1)} \geq \boldsymbol{\nu}^{(k)}$ for all $k \geq 0$.

This theorem holds for the most widely used iterative methods such as Power, Jacobi, Gauss-Seidel, and SOR. It guarantees that the computed $\boldsymbol{\nu}$ using one of these iterative methods naturally gives a lower bound $\boldsymbol{\nu}^l(\mathcal{S}_1)$. We can similarly compute the lower bound $\boldsymbol{\nu}^l(\mathcal{S}_0)$ with a second numerical solution, and let $\boldsymbol{\nu}^u(\mathcal{S}_1) = \mathbf{1} - \boldsymbol{\nu}^l(\mathcal{S}_0)$.

Unlike the case of time-bounded until, it is difficult to predict the distance between $\boldsymbol{\nu}^l(\mathcal{S}_1)$ and $\boldsymbol{\nu}^u(\mathcal{S}_1)$, since there is no simple relation between ε and $||\boldsymbol{\nu}^u - \boldsymbol{\nu}^l||$. However, $||\boldsymbol{\nu}^u - \boldsymbol{\nu}^l||$ provides a much better criterion for convergence check. Still, in the example of Fig. 7.2 with $p \ll \varepsilon$, a näive convergence test stops the iteration at $\boldsymbol{\nu}[\mathbf{i}](\mathcal{S}_1) = p/2$, and we obtain the bounds $\nu^l[\mathbf{i}](\mathcal{S}_1) = p/2$ and $\nu^u[\mathbf{i}](\mathcal{S}_1) = 1 - p/2$, which alert us that there is still a huge uncertainty in the result, thus the iteration should proceed further (with a smaller ε).

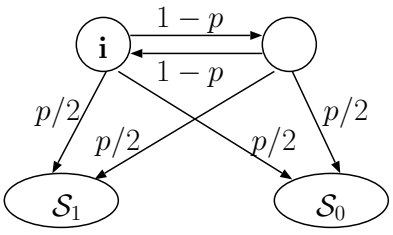


Figure 7.2: An example of computing $\boldsymbol{\nu}(\mathcal{S}_1)$.

7.1.3 Point-interval and general interval until

In both cases, we employ the analysis of Section 7.1.1. For $\mathbb{P}_{\bowtie p}(\phi \mathsf{U}^{[t,t]} \psi)$, where ψ implies ϕ, we generate $\mathcal{M}_{unif}^{Sat(\neg\phi)}$ and compute $[\boldsymbol{\nu}^l(Sat(\psi),t),\boldsymbol{\nu}^u(Sat(\psi),t)]$ as the bounds for $\mathbf{Prob}(\phi \mathsf{U}^{[t,t]} \psi)$.

Bounding the probability for $\phi \mathsf{U}^{[t,t']} \psi$ requires multiple transient analysis rounds. We first compute $[\boldsymbol{\nu}^l_{last}(Sat(\psi),t'-t),\boldsymbol{\nu}^u_{last}(Sat(\psi),t'-t)]$ in $\mathcal{M}_{unif}^{Sat(\neg\phi\vee\psi)}$. Then in $\mathcal{M}_{unif}^{Sat(\neg\phi)}$,

$$\sum_{k=L_\varepsilon}^{R_\varepsilon} \mathbf{P}^k \boldsymbol{\nu}^l_{last}(Sat(\psi),t'-t) \cdot p^l[k]$$

gives the bound $\mathbf{Prob}^l(\phi \mathsf{U}^{[t,t']} \psi)$. The upper bound can be computed by considering that any path is described by exactly one of these path formulas:

$$\phi \mathsf{U}^{[0,t]}(\neg\phi) \qquad \phi \mathsf{U}^{[t,t']} \psi \qquad \phi \mathsf{U}^{[t,t']}(\neg\psi \wedge \neg\phi) \qquad \phi \mathsf{U}^{[t',t']}(\phi \wedge \neg\psi)$$

so that we can obtain $\mathbf{Prob}^u(\phi \mathsf{U}^{[t,t']} \psi)$ by

$$\mathbf{1} - (\mathbf{Prob}^l(\phi \mathsf{U}^{[0,t]}(\neg\phi)) + \mathbf{Prob}^l(\phi \mathsf{U}^{[t,t']}(\neg\psi \wedge \neg\phi)) + \mathbf{Prob}^l(\phi \mathsf{U}^{[t',t']}(\phi \wedge \neg\psi))).$$

To summarize the above discussion, Fig. 7.3 depicts the algorithm for U^I, which returns a pair of "lower and upper sets" $[\mathcal{X}^l,\mathcal{X}^u]$ satisfying $\mathcal{X}^l \subseteq \mathcal{X} \subseteq \mathcal{X}^u$, where $\mathcal{X}$ is the set satisfying the until formula.

7.2 Semantics for CSL formulas with bounds

This section addresses nested CSL formulas under our semantics with bounds. To accommodate the bounds generated by the above algorithms, we need to redefine the

```
bound UntilBounds(interval I, StateSet 𝒴, StateSet 𝒵, float ε) is
 1  if I = [0,t] then
 2     Build P for M^{S0∪S1}_unif for uniformizing rate q;        • S1 = 𝒵, S0 = S\E(𝒴U𝒵)
 3     ν^l = BoundedUntil(qt, 𝒴, 𝒵, ε);        • Replace FoxGlynn by FoxGlynnLB in Fig. 7.1
 4     ν^u = ν^l + ε · 1;
 5  elseif I = [0,∞) then
 6     Build P for M^{S0∪S1}_unif for uniformizing rate q;
 7     ν^l ← GaussSeidel(P, δ^{S1}, ε);    ν^u ← 1 − GaussSeidel(P, δ^{S0}, ε);
 8  elseif I = [t,t] then
 9     Build P for M^{S\𝒴}_unif for uniformizing rate q;
10     ν^l = BoundedUntil(qt, 𝒴, 𝒵, ε);   ν^u = ν^l + ε · 1;
11  elseif I = [t,t'] then
12     ...                                                  • See Section 7.1.3
13  endif
14  return P_{⋈p}[ν^l, ν^u];
```

Figure 7.3: Model checking algorithm for U^I generating bounds.

semantics of CSL. Specifically, we replace the exact set of states with the lower and upper bounds of this set.

Definition 7.2.1 The evaluation of a CSL state formula $\mathscr{F}$ returns a pair of state sets $[eval^l(\mathscr{F}),eval^u(\mathscr{F})]$ satisfying $eval^l(\mathscr{F}) \subseteq eval(\mathscr{F}) \subseteq eval^u(\mathscr{F})$.

- $eval(\psi) = [Sat(\psi),Sat(\psi)]$
- $eval(\neg\mathscr{F}) = [\mathcal{S}\backslash eval^u(\mathscr{F}),\mathcal{S}\backslash eval^l(\mathscr{F})]$
- $eval(\mathscr{F}_1 \wedge \mathscr{F}_2) = [eval^l(\mathscr{F}_1) \cap eval^l(\mathscr{F}_2), eval^u(\mathscr{F}_1) \cap eval^u(\mathscr{F}_2)]$
- $eval(\mathscr{F}_1 \vee \mathscr{F}_2) = [eval^l(\mathscr{F}_1) \cup eval^l(\mathscr{F}_2), eval^u(\mathscr{F}_1) \cup eval^u(\mathscr{F}_2)]$
- $eval(\mathbb{P}_{\geq p}(\mathsf{X}^I\mathscr{F})) = [\mathbb{P}_{\geq p}\mathbf{Prob}(\mathsf{X}^I eval^l(\mathscr{F})),\mathbb{P}_{\geq p}\mathbf{Prob}(\mathsf{X}^I eval^u(\mathscr{F}))]$
- $eval(\mathbb{P}_{\leq p}(\mathsf{X}^I\mathscr{F})) = [\mathbb{P}_{\leq p}\mathbf{Prob}[\mathsf{X}^I eval^u(\mathscr{F})],\mathbb{P}_{\leq p}\mathbf{Prob}(\mathsf{X}^I eval^l(\mathscr{F}))]$

- $eval(\mathbb{P}_{\geq p}(\mathscr{F}_1 \mathsf{U}^I \mathscr{F}_2)) =$

  $[\mathbb{P}_{\geq p}\mathbf{Prob}^l(eval^l(\mathscr{F}_1)\mathsf{U}^I eval^l(\mathscr{F}_2)),\mathbb{P}_{\geq p}\mathbf{Prob}^u(eval^u(\mathscr{F}_1)\mathsf{U}^I eval^u(\mathscr{F}_2))]$

- $eval(\mathbb{P}_{\leq p}(\mathscr{F}_1 \mathsf{U}^I \mathscr{F}_2)) =$

  $[\mathbb{P}_{\leq p}\mathbf{Prob}^u(eval^u(\mathscr{F}_1)\mathsf{U}^I eval^u(\mathscr{F}_2)),\mathbb{P}_{\leq p}\mathbf{Prob}^l(eval^l(\mathscr{F}_1)\mathsf{U}^I eval^l(\mathscr{F}_2))]$.

For the evaluation of nested formulas, we can still follow the syntax tree order from leaves to the root, and always enforce a high precision (small ε) when evaluating each subformula. The drawback of this approach is the potentially high and unnecessary cost. Instead, we can start from a low precision, and, if the resulting bounds are not tight enough, refine by incrementally increasing the precision for evaluating each level of subformula. The merit of this approach is the possibility of focusing the expensive numerical analysis efforts on the important subformulas. However, depending on the order in which we refine subformulas, this may result in unnecessarily going back and forth between the evaluation of outer and inner subformulas.

We propose the framework in Fig. 7.4 and 7.5 to evaluate nested CSL formulas and we only consider the evaluation of $\mathbb{P}_{\bowtie p}(\mathscr{F}_1 \mathsf{U}^I \mathscr{F}_2)$ since the evaluation of $\mathbb{P}_{\bowtie p}(\mathsf{X}^I \mathscr{F})$ can be simply reduced to that of $\mathscr{F}$. *Eval* is the top-level function to evaluate the U^I operator, and all subformulas are assumed to be of the form $\mathbb{P}_{\bowtie p}(\mathscr{F}_1 \mathsf{U}^I \mathscr{F}_2)$. We first evaluate with an initial precision and invoke Function *EvalError*, which evaluates the whole formula with error parameter ε. For the refinement, we give two different orders: *RefineTopDown* and *RefineBottomUp*. The former first increases the precision for evaluating the top-level formula, then, when it does not further tighten the resulting bounds, it refines the results from subformulas; the latter instead first refines subformulas, then the top-level formula. As

the following case studies show, these two orders of refinement generally result in different costs to eventually reach sufficiently tight bounds.

7.3 Case studies

We consider two models, one for an embedded system and the other for the Advanced Airspace Concept (AAC) system. The embedded system model is described in [61] and also released as an example with PRISM [51]. The AAC system model describes the reliability in an airspace control protocol which was recently proposed; [83] provides the details about this protocol and also discusses the model-checking scheme for this system. We demonstrate the algorithms in Section 7.1 for non-nested CSL formulas, then show preliminary results on nested CSL formulas.

Embedded system

This system consists of a main processor, an input processor, an output processor, two actuators, and three sensors, each associated with a failure rate. The main processor maintains a count of the number of retries when receiving data from sensors or sending data to actuators and, if the count exceeds a threshold, the entire system fails. More details about this model are available in [61].

First, we check the time-bounded until property "the system runs without failure until the count reaches threshold 3 within 12 hours", written as:

$$[\mathcal{X}^l,\mathcal{X}^u] := \mathbb{P}_{\leq 0.005}[\mathit{Normal}\ \mathsf{U}^{[0,12\times 3600]}\ \mathit{InOutFail}].$$

```
bound  Eval(P_{⋈p}(F1 U^I F2)) is
 1  ε ← InitialValue;
 2  [X^l,X^u] ← EvalError(P_{⋈p}(F1 U^I F2), ε);
 3  while |X^u| − |X^l| > acceptable do
 4     [X^l,X^u] ← RefineTopDown/RefineBottomUp(F1 U^I F2);
 5  endwhile;
 6  return [X^l,X^u];
```

```
bound  EvalError(F, ε) is
 1  if F in form P_{⋈p}(F1 U^I F2);
 2     [Y^l,Y^u] ← EvalError(F1, ε);     [Z^l,Z^u] ← EvalError(F2, ε);
 3     [X^l,X^u] ← bounds from UntilBounds(I, Y^l, Z^l, ε), UntilBounds(I, Y^u, Z^u, ε);
 4  else if F in form φ or ¬φ                          • base case, atomic proposition
 5     X^l, X^u ← Sat(φ) or S\Sat(φ)
 6  else if F in form F1 ∨ F2 or F1 ∧ F2
 7     [Y^l,Y^u] ← EvalError(F1, ε);     [Z^l,Z^u] ← EvalError(F2, ε);
 8     [X^l,X^u] ← [Y^l ∨ Z^l,Y^u ∨ Z^u] or [Y^l ∧ Z^l,Y^u ∧ Z^u];
 9  endif
10  Set [X^l,X^u] and ε as CurrentResult and CurrentPrecision for F;
11  return [X^l,X^u];
```

Figure 7.4: Algorithm for the evaluation of nested CSL formulas.

For initial state $\mathbf{s}_0$, the calculated probability bounds are $[\nu^l[\mathbf{s}_0],\nu^u[\mathbf{s}_0]]$, In Fig. 7.6, the horizontal axis represents the error parameter ε. The left of Fig. 7.6 shows the size of $\mathcal{X}^l$ and $\mathcal{X}^u$ while the right shows $\nu^l[\mathbf{s}_0]$ and $\nu^u[\mathbf{s}_0]$; as we discussed, $\nu^u[\mathbf{s}_0] - \nu^l[\mathbf{s}_0] = \varepsilon$. When $\varepsilon \leq 0.0013$, we obtain an exact result for this formula, i.e., $\mathcal{X}^l = \mathcal{X}^u$.

We then check the unbounded until formula "the embedded system eventually fails with probability ≤ 0.2", written as:

$$[\mathcal{X}^l,\mathcal{X}^u] := \mathbb{P}_{\leq 0.2}[\mathit{Normal}\ \mathsf{U}\ \mathit{InOutFail}].$$

Again, the left of Fig. 7.7 shows the size of $\mathcal{X}^l$ and $\mathcal{X}^u$ while the right shows $\nu^l[\mathbf{s}_0]$ and $\nu^u[\mathbf{s}_0]$. In general $\nu^u[\mathbf{s}_0] - \nu^l[\mathbf{s}_0]$ does not have a simple relation with ε, but, in this case, it decreases almost linearly as ε decreases, which is desirable.

```
bound RefineTopDown(ℙ_{⋈p}(ℱ1 U^I ℱ2)) is
 1  ε ← CurrentPrecision for ℙ_{⋈p}(ℱ1 U^I ℱ2);
 2  Reduce ε;
 3  [𝒴^l,𝒴^u] ← CurrentResult for (ℱ1);
 4  [𝒵^l,𝒵^u] ← CurrentResult for (ℱ2);
 5  oldbound ← CurrentResult for ℙ_{⋈p}(ℱ1 U^I ℱ2);
 6  repeat
 7     [𝒳^l,𝒳^u] ← bounds from UntilBounds(I, 𝒴^l, 𝒵^l, ε), UntilBounds(I, 𝒴^u, 𝒵^u, ε);
 8     if oldbound = [𝒳^l,𝒳^u] then
 9        [𝒴^l,𝒴^u] ← RefineTopDown(ℱ1);    [𝒵^l,𝒵^u] ← RefineTopDown(ℱ2);
10     else
11        oldbound ← [𝒳^l,𝒳^u];
12        Reduce ε;
13     endif
14  until |𝒳^u| − |𝒳^l| ≤ acceptable;
15  Set [𝒳^l,𝒳^u] and ε as CurrentResult and CurrentPrecision for ℱ;
16  return [𝒳^l,𝒳^u];
```

```
bound RefineBottomUp(ℙ_{⋈p}(ℱ1 U^I ℱ2)) is
 1  ε ← CurrentPrecision for ℙ_{⋈p}(ℱ1 U^I ℱ2);
 2  [𝒴^l,𝒴^u] ← RefineBottomUp(ℱ1);    [𝒵^l,𝒵^u] ← RefineBottomUp(ℱ2);
 3  oldbound ← CurrentResult for ℙ_{⋈p}(ℱ1 U^I ℱ2);
 4  repeat
 5     [𝒳^l,𝒳^u] ← bounds from UntilBounds(I, 𝒴^l, 𝒵^l, ε), UntilBounds(I, 𝒴^u, 𝒵^u, ε);
 6     if oldbound = [𝒳^l,𝒳^u] then
 7        Reduce ε;
 8     else
 9        oldbound ← [𝒳^l,𝒳^u];
10        [𝒴^l,𝒴^u] ← RefineBottomUp(ℱ1);    [𝒵^l,𝒵^u] ← RefineBottomUp(ℱ2);
11     endif
12  until |𝒳^u| − |𝒳^l| ≤ acceptable;
13  Set [𝒳^l,𝒳^u] and ε as CurrentResult and CurrentPrecision for ℱ;
14  return [𝒳^l,𝒳^u];
```

Figure 7.5: Algorithm for refining the evaluation of nested CSL formulas.

Advanced Airspace Concept (AAC)

This model depicts conflict detection and resolution between a pair of aircraft. To ensure safe separation between aircraft, the AAC system provides three subsystems: AutoResolver, TSAFE, and TCAS, which can detect and resolve potential future conflicts. To resolve a detected conflict, a resolution is calculated automatically and sent to the pilot of

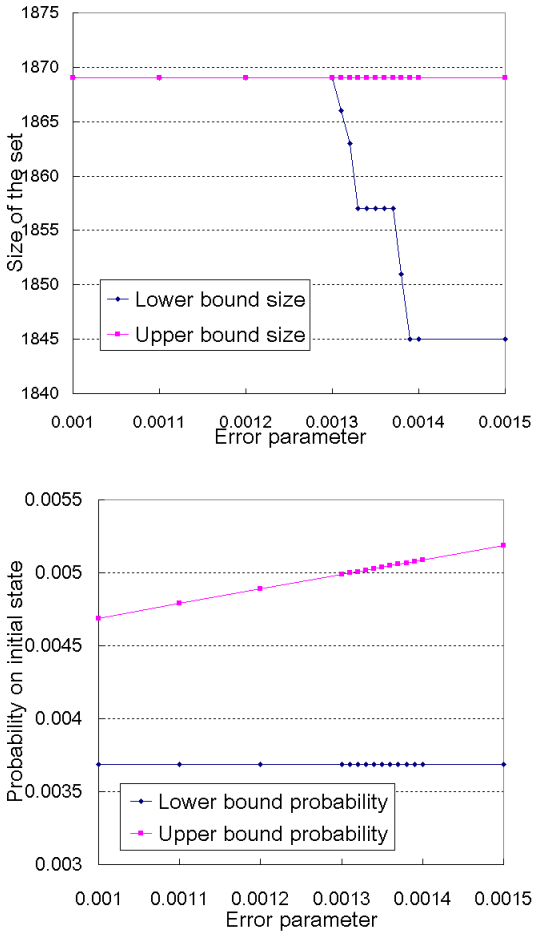


Figure 7.6: Embedded system bounded until: size of lower and upper bound sets satisfying the formula (left) and probability bounds for the initial state (right).

the involved aircraft. The pilot is then responsible for executing the most urgent resolution first. Fig. 7.8 shows the high-level state transition in the AAC system for a pair of aircraft.

First, we define "dangerous states" to be those where a TSAFE alert rises and, with probability greater than 0.05 it will not be resolved within 3 minutes (180 seconds). Dangerous states can be described using the CSL formula:

$$[\mathcal{D}^l,\mathcal{D}^u] := TSAFEalert \wedge \mathbb{P}_{\leq 0.95}[TSAFEalert \ \mathsf{U}^{[0,180]} \ \neg TSAFEalert]. \tag{7.3}$$

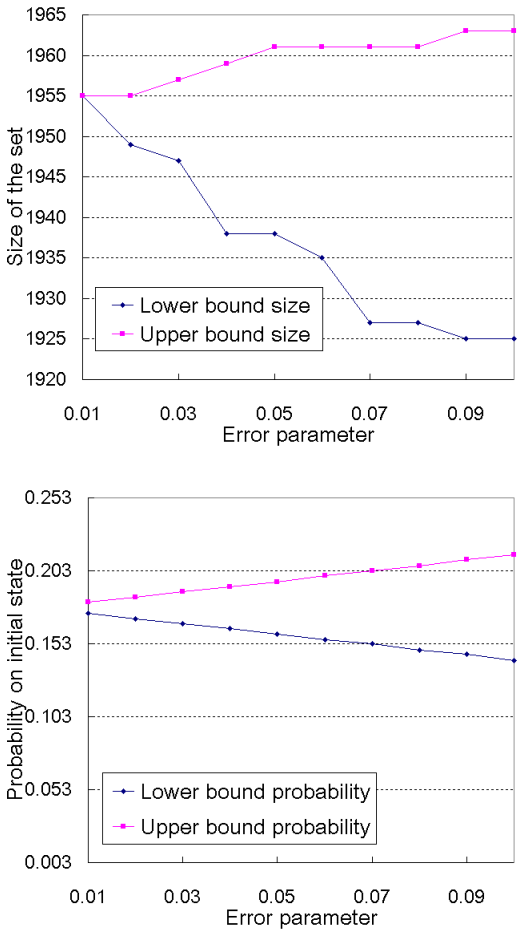


Figure 7.7: Embedded system unbounded until: size of lower and upper bound sets satisfying the formula (left) and probability bounds for the initial state (right).

Then, we study the probability that, from a state without TSAFE alert on, the system reaches dangerous states within 5 minutes (300 seconds).

$$[\mathcal{X}^l,\mathcal{X}^u] := \mathbb{P}_{\leq 0.01}[\neg \mathit{TSAFEalert}\ \mathsf{U}^{[0,300]}\ [\mathcal{D}^l,\mathcal{D}^u]]. \tag{7.4}$$

We use ε_d and ε_n to denote the error parameters for Formula 7.3 and 7.4, respectively. For this experiment, we keep refining the bound $[\mathcal{X}^l,\mathcal{X}^u]$ until $\mathcal{X}^l = \mathcal{X}^u$ (of course, in practice this might be neither achievable nor necessary). We first try to evaluate For-

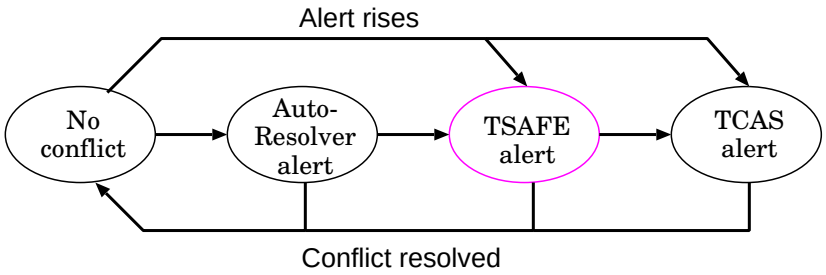


Figure 7.8: High-level state transition of the AAC system.

mula 7.4 employing top-down refinement, and obtain the results in Table 7.1. For each row, we also tried to refine by just reducing ε_n, but this did not generate tighter bounds $[\mathcal{X}^l,\mathcal{X}^u]$ than those listed. In the first two rows, $[\mathcal{D}^l,\mathcal{D}^u]$ is too loose to obtain an exact result for the outer formula; from the third row, instead, while there is still uncertainty in the result of the inner subformula, $[\mathcal{D}^l,\mathcal{D}^u]$ is tight enough to generate an exact result for the nested formula. Thus, the model checking procedure could stop at $\varepsilon_d = \varepsilon_n = 0.001$.

We also study the formula using a larger probability threshold:

$$[\mathcal{X}^l,\mathcal{X}^u] := \mathbb{P}_{\leq 0.1}[\neg \mathit{TSAFEalert}\ \mathsf{U}^{[0,300]}\ [\mathcal{D}^l,\mathcal{D}^u]]. \tag{7.5}$$

We start from $\varepsilon_d = \varepsilon_n = 0.1$ and refine. Using the bottom-up approach, we should first refine the subformula to tighter bounds, then go back to the top level. From Fig. 7.1, we know that $\varepsilon_d = 0.0001$ ensures an exact result for the subformula. However, Table 7.2 shows, refining the top-level formula directly produces an exact result, so it is in fact unnecessary to refine the subformula. This is because the probability threshold $\mathbb{P}_{\leq 0.1}$ is so slack that a precise evaluation on the top-level is sufficient to get an exact result even with very loose bounds on the subformula.

$\varepsilon_d = \varepsilon_n$	$\lvert\mathcal{D}^l\rvert$	$\lvert\mathcal{D}^u\rvert$	$\lvert\mathcal{X}^l\rvert$	$\lvert\mathcal{X}^u\rvert$	$\nu^l[\mathbf{s}_0]$	$\nu^u[\mathbf{s}_0]$
0.01	480	2160	224	1532	0.0439	0.0539
0.005	480	1392	224	1532	0.0441	0.0491
0.001	480	576	1532	1532	0.0443	0.0453
0.0005	480	576	1532	1532	0.0443	0.0448
0.0001	480	480	1532	1532	0.0443	0.0444

Table 7.1: Results for Formula 7.4 using top-down refinement.

ε_d	ε_n	$\lvert\mathcal{D}^l\rvert$	$\lvert\mathcal{D}^u\rvert$	$\lvert\mathcal{X}^l\rvert$	$\lvert\mathcal{X}^u\rvert$	$\nu^l[\mathbf{s}_0]$	$\nu^u[\mathbf{s}_0]$
0.1	0.1	432	2160	224	5453	0.000	0.1400
	0.01	432	2160	5453	5453	0.000	0.0539

Table 7.2: Results for Formula 7.5.

We can see that finding a scheme for nested formula requires us to identify the "bottleneck" of the precision for the final result. For Formula 7.4, the bottleneck lies in the inner formula, while for Formula 7.5 the bottleneck lies in the outer formula. However, it is difficult to come up with the best general scheme without several trials, thus finding good heuristics for efficient evaluation of nested CSL formulas is an interesting future work.

7.4 Summary

Since iterative methods are widely utilized in CSL model checking, truncation errors must be considered to ensure correctness of the results. In this paper, we investigated a bounding semantics of CSL formulas with the U^I operator. We first improved the CSL model checking algorithm by providing lower and upper bounds, to support the bounding semantics. Then, we applied the bounding semantics to nested CSL formulas and studied approaches for their evaluations. We demonstrated the new algorithms on two case studies. The results show that, for nested CSL formulas, appropriately scheduling the precision on different subformulas could achieve tight bounds and even exact results, with less computational cost. However, finding an optimal scheme is nontrivial. Thus we believe that finding good heuristics to guide the evaluation of nested CSL formulas is an important line of future investigation.

Chapter 8

Implementation: SMART tool

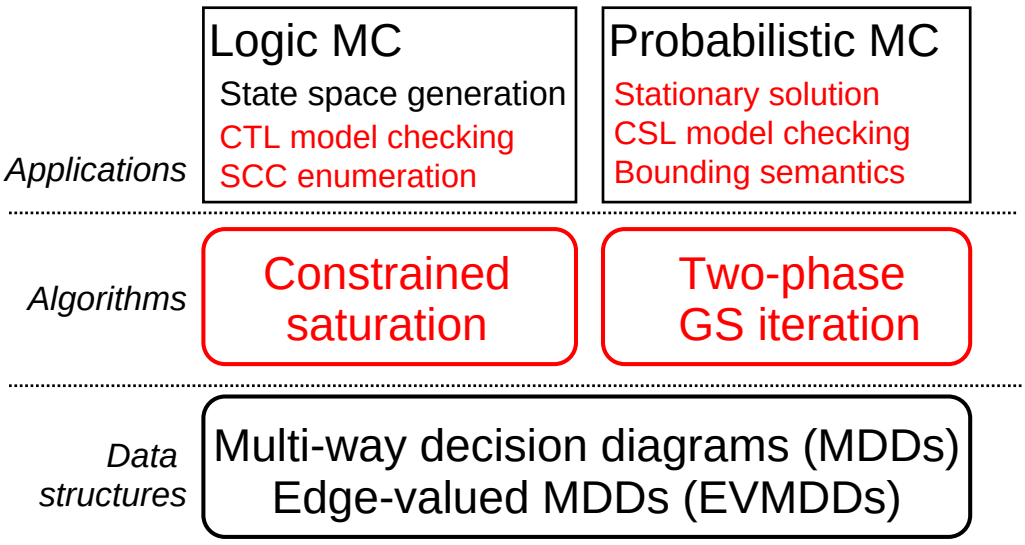


Figure 8.1: The infrastructure of SMART.

All the proposed techniques have been implemented in our tool SMART (the Stochastic Model checking Analyzer for Reliability and Timing). Figure 8.1 shows the structure of SMART. The important features discussed in this thesis are highlighed in red. Currently,

SMART takes in high-level description (a stochastic Petri net) and generates the underlying CTMC. It is able to answer a set of logic and quantitative queries. The user manual for SMART is available at [15].

8.1 Logic model checking

CTL model checking

SMART now supports EX, EU, and EG queries, each of which returns a resulting set encoded with an MDD nonde (key word `mdd`). The follow code snippet shows these queries:

```
mdd psi := ...
mdd phi := ...
mdd ex  := EX(psi);
mdd eu  := EU(phi, psi);
#EGMethod {EG_BFS, EG_TC}
mdd eg  := EG(psi);
```

There are two options for EG algorithm: BFS (default) and transitive closure, which are introduced in Chapter 3.

SCC enumeration

SMART supports SCC and terminal SCC computation and returns a set of states that belong to SCC or terminal SCC. The following code snippet shows these two queries:

```
#SCCMethod {FMSD06_SAT, SAT_TRANSCLOSURE, Lockstep};
mdd scc := scc();
#TSCCMethod {XB_SAT, SAT_TRANSCLOSURE, XB_BFS};
mdd rec := tscc();
```

There are three algorithms available for SCC computation query `scc`: improved Lockstep using saturation algorithm (`FMSD06_SAT`, default), new algorithm based on transitive clo-

sure (`SAT_TRANSCLOSURE`), and the previous Lockstep implemented with BFS (`Lockstep`). Also, there are three algorithms available for terminal SCC computation query `tscc`: Xie-Beerel algorithm using Saturation (`XB_SAT`, default), new algorithm based on transitive closure (`SAT_TRANSCLOSURE`), and the previous Xie-Beerel algorithm implemented with BFS (`XB_BFS`).

Shortest EG witness generation

As Chapter 5 shows, SMART is able to generate a shortest EG witness. The query is in the following form:

```
mdd eg  := EGtrace(psi);
```

The above query returns the set of state $\mathsf{EG}\psi$ and prints out one of the shortest witness for $\mathsf{EG}\psi$.

8.2 Probabilistic model checking

As an objective of our project, we will eventually implement *four* approaches to handle the stochastic properties: exact solution, bounding solution, simulation, and approximation. Approximation of steady-state solution using EVMDDs is implemented based on a previous idea in [78].

First, SMART now supports steady-state solution of ergodic CTMCs using the symbolic Jacobi and Gauss-Seidel iteration introduced in Chapter 6. Given a state formula `F`, SMART is able to compute the stationary reward, which can be expressed as:

$$\sum_{\mathbf{i}\in\mathcal{S}} \pi[\mathbf{i}] \cdot f(\mathbf{i}),$$

where $f(\mathbf{i})$ is a map $f : \mathcal{S} \rightarrow \mathbb{R}$ indicating the weight of $\mathbf{i}$ and the reward is the weighted sum of the stationary probabilities for all states. SMART provides the following two queries:

$$\text{real prob} := prob\_ss(\psi); \quad // \sum_{\mathbf{i} \models \psi} \pi[\mathbf{i}]$$

$$\text{real prob} := avg\_ss(f); \quad // \sum_{\mathbf{i} \in \mathcal{S}} \pi[\mathbf{i}] \cdot f(\mathbf{i})$$

As a new feature, SMART now supports CSL formulas using bounding semantics: $\mathbb{P}_{\leq p}(\phi \mathsf{U}^{[t1,t2]} \psi)$ and $\mathbb{P}_{\leq p}(\phi \mathsf{U} \psi)$. It employs the technique introduced in Chapter 7. The following code snippet shows a CSL query:

```
mdd phi, psi := ...
real varepsilon :=
mdd s := PU(t1, t2, phi, psi, p, {false, true}, varepsilon);
```

where parameter `false` or `true` determines whether it returns a lower or an upper bound for the resulting set of states. If `t1=t2=0`, it will compute the query $\mathbb{P}_{\leq p}(\phi \mathsf{U} \psi)$.

Chapter 9

Conclusion

9.1 Summary

In this thesis, we explored one topic in each chapter, covering both logic and probabilistic model checking. In this subsection, we provide a big picture of our framework, in which three components: decision diagrams, saturation, and two-phase Gauss-Seidel iteration, constitute the cornerstones. First, we summarize the functionality of each type of decision diagrams in this thesis in Table 9.1. Secondly, we summarize the application of (improved) Saturation in Table 9.2.

Finally, there are two applications for the proposed symbolic Gauss-Seidel iteration: stationary solution for ergodic CTMCs (Section 6.2) and unbounded U in probabilistic model checking (Section 6.2.3). From another dimension, we proposed a bounding semantics for U operators in Chapter 7 together with a way to compute it.

DD Type	Encoding	Section and Citation
MDD	Sets of states, next- or previous- state function	Section 2.1.1
	Transitive closure	Section 4.3
EV$^+$MDD	Distance function	[21]
	Transitive closure with distance	Section 5.2.1
	Indexing function	Section 6.2
EV*MDD	Transition rate matrix $\mathbf{R}$ or probability matrix $\mathbf{P}$	Section 6.2.1

Table 9.1: Functionality of decision diagrams.

Within this "trilogy", decision diagrams provide support for both saturation and symbolic Gauss-Seidel iteration, which implement functionality in logic and probabilistic model checking, respectively. Because of their compactness, decision diagrams have long been considered as an excellent data structure for the storage of discrete-state systems. My work in the thesis further show that coupled with Saturation and two-phase Gauss-Seidel

Application	Section and Citation
State-space generation	[18]
Constrained-saturation in EU model checking	Section 3.1
Transitive closure for EG model checking and SCC enumeration	Section 3.2 and 4.3
Transitive closure with distance for shortest EG witness generation	Section 5.2.1

Table 9.2: The application of (improved) Saturation.

iterations, decision diagrams have great potential to perform efficiently in both logic and probabilistic model checking engine.

9.2 Future work

Parallelization has become a trend in software engineering in recent years. Since the frequence of CPUs might not be further improved, more efforts are devoted to algorithms utilizing multi-thread and multi-core programming paradigm for better runtime performance on runtime. Decision diagrams are known to be extremely difficult to parallelize since many operations in decision diagrams are inherently serial.

There have been at least two approaches to the parallelization of DD-based model checking: the first is to parallelize the low-level operations in a decision diagram library, like in [49, 62, 59]. The second is to parallelize the high-level algorithm such as Saturation, like in [13, 14, 34]. However, none of these attempts has shown obvious improvements on runtime in general. [23] compares the parallelizing explicit algorithms, which is much more straightforward than (single thread) symbolic algorithms. The argument between symbolic algorithms and explicit algorithms, especially when taking parallelization into consideration, will still continue, and finding good parallelization schemes for DD-based algorithms is worth further research.

The newly proposed two-phase Gauss-Seidel iteration shows great parallelization potential. In a nutshell, it executes a DFS on EV*MDD. While the elements in the probability vector must be updated following DFS order, the traversal could be in any order and thus could be parallelized. Moreover, the EV*MDD encoding the transition matrix is

often compact (a few Mbytes for many non-trivial models) and, while the computation is intensive, some hardware speedup scheme, like GPU, might be applicable.

Bibliography

[1] NuSMV: a new symbolic model checker. Available at http://nusmv.irst.itc.it/.

[2] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *Proc. TACAS*, volume 1785 of *LNCS*, pages 411–425. Springer, 2000.

[3] E. Ábrahám, N. Jansen, R. Wimmer, J.-P. Katoen, and B. Becker. DTMC model checking by SCC reduction. In *QEST*, pages 37–46, 2010.

[4] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model checking algorithms for continuous-time Markov chains. *IEEE Trans. Softw. Eng.*, 29(6):524–541, June 2003.

[5] A. Berman and R. Plemmons. *Nonnegative Matrices*. SIAM, 1979.

[6] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. TACAS*, pages 193–207. Springer, 1999.

[7] R. Bloem, H. N. Gabow, and F. Somenzi. An Algorithm for Strongly Connected Component Analysis in $n \log n$ Symbolic Steps. In *Formal Methods in Computer Aided Design*, pages 37–54. Springer-Verlag, 2000.

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

[9] P. Buchholz. A class of hierarchical queueing networks and their analysis. *Queueing Systems.*, 15:59–80, 1994.

[10] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS J. Comp.*, 12(3):203–222, 2000.

[11] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Trans. CAD of Integr. Circ. and Syst.*, 13(4):401–424, Apr. 1994.

[12] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

[13] M.-Y. Chung and G. Ciardo. Saturation NOW. In *Proc. Quantitative Evaluation of SysTems (QEST)*, pages 272–281. IEEE Comp. Soc. Press, 2004.

[14] M.-Y. Chung and G. Ciardo. A dynamic firing speculation to speedup distributed symbolic state-space generation. In *Proc. International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE Comp. Soc. Press, 2006. Electronic proceedings.

[15] G. Ciardo et al. SMART: Stochastic Model checking Analyzer for Reliability and Timing, User Manual. Available at http://www.cs.ucr.edu/∼ciardo/SMART/.

[16] G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. Logical and stochastic modeling with SMART. *Perf. Eval.*, 63:578–608, 2006.

[17] G. Ciardo, G. Lüttgen, and A. S. Miner. Exploiting interleaving semantics in symbolic state-space generation. *Formal Methods in System Design*, 31:63–100, 2007.

[18] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In *Proc. TACAS*, LNCS 2031, pages 328–342. Springer, 2001.

[19] G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In *Proc. TACAS*, LNCS 2619, pages 379–393. Springer, 2003.

[20] G. Ciardo, R. Marmorstein, and R. Siminiceanu. The saturation algorithm for symbolic state space exploration. *Software Tools for Technology Transfer*, 8(1):4–25, 2006.

[21] G. Ciardo and R. Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In *Proc. FMCAD*, LNCS 2517, pages 256–273. Springer, 2002.

[22] G. Ciardo and R. Siminiceanu. Structural symbolic CTL model checking of asynchronous systems. In *Proc. CAV*, LNCS 2725, pages 40–53. Springer, 2003.

[23] G. Ciardo, Y. Zhao, and X. Jin. Parallel symbolic state-space exploration is difficult, but what is the alternative? *EPTCS*, 14:1–17, 2009.

[24] F. Ciesinski, C. Baier, M. Grösser, and J. Klein. Reduction techniques for model checking Markov decision processes. In *Proceedings of the 2008 Fifth International Conference on Quantitative Evaluation of Systems*, pages 45–54. IEEE Computer Society, 2008.

[25] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: an open source tool for symbolic model checking. In *Proc. CAV*, LNCS 2404. Springer, July 2002.

[26] E. Clarke, M. Fujita, P. C. McGeer, J. C.-Y. Yang, and X. Zhao. Multi-terminal binary decision diagrams: an efficient data structure for matrix representation. In *IWLS '93 International Workshop on Logic Synthesis*, May 1993.

[27] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Progr. Lang. and Syst.*, 8(2):244–263, Apr. 1986.

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

[29] E. M. Clarke, K. L. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *Proc. CAV*, LNCS 1102, pages 419–422, 1996.

[30] R. M. Czekster, P. Fernandes, J.-M. Vincent, and T. Webber. Split: a flexible and efficient algorithm to vector-descriptor product. In *Proc. ValueTools*, pages 83:1–83:8. ICST, 2007.

[31] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In *Proc. CAV*, LNCS 3114, pages 496–500. Springer, July 2004.

[32] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *32nd Design Automation Conference (DAC 95)*, pages 427–432, 1995.

[33] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-Calculus). In *Proceedings, Symposium on Logic in Computer Science, 16-18 June 1986, Cambridge, Massachusetts, USA*, pages 267–278. IEEE Computer Society, 1986.

[34] J. Ezekiel, G. Lüttgen, and G. Ciardo. Parallelising symbolic state-space generators. In *Proc. CAV*, LNCS 4590, pages 268–280. Springer, 2007.

[35] H. Fecher, M. Leucker, and V. Wolf. Don't know in probabilistic systems. In *Model Checking Software*, LNCS 3925, pages 71–88. Springer, 2006.

[36] P. Fernandes, B. Plateau, and W. J. Stewart. Efficient descriptor-vector multiplication in stochastic automata networks. *J. ACM*, 45(3):381–414, 1998.

[37] K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Proc. TACAS, pages 420–434. Springer, 2001.

[38] B. L. Fox and P. W. Glynn. Computing Poisson Probabilities. *Comm. ACM*, 31(4):440–445, Apr. 1988.

[39] G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Markovian analysis of large finite state machines. *IEEE Trans. CAD of Integr. Circ. and Syst.*, 15(12):1479–1493, Dec. 1996.

[40] R. H. Hardin, R. P. Kurshan, S. K. Shukla, and M. Y. Vardi. A new heuristic for bad cycle detection using bdds. *Formal Methods in System Design*, 18(2):131–140, 2001.

[41] R. Hojati, H. J. Touati, R. P. Kurshan, and R. K. Brayton. Efficient ω-regular language containment. In *CAV*, pages 396–409, 1992.

[42] G. J. Holzmann. State compression in spin: Recursive indexing and compression training runs. In *Proceedings of Third International SPIN Workshop*, 1997.

[43] G. Horton and S. T. Leutenegger. A multi-level solution algorithm for steady state Markov chains. In *Proc. ACM SIGMETRICS*, pages 191–200, May 1994.

[44] T. Kam, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.

[45] S. Kashyap and V. K. Garg. Producing short counterexamples using "crucial events". In *Proc. CAV*, CAV '08, pages 491–503. Springer, 2008.

[46] J.-P. Katoen, D. Klink, M. Leucker, and V. Wolf. Three-Valued abstraction for continuous-time Markov chains. In *Proc. CAV*, LNCS 4590, pages 311–324. Springer, July 2007.

[47] J.-P. Katoen, M. Kwiatkowska, G. Norman, and D. Parker. Faster and symbolic CTMC model checking. In *Process Algebra and Probabilistic Methods. Performance Modelling and Verification*, LNCS 2165, pages 23–38. Springer, 2001. 10.1007/3-540-44804-7_2.

[48] Y. Kesten, A. Pnueli, and L.-o. Raviv. Algorithmic verification of linear temporal logic specifications. In *ICALP '98: Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, pages 1–16, London, UK, 1998. Springer-Verlag.

[49] S. Kimura and E. M. Clarke. A parallel algorithm for constructing binary decision diagrams. In *Proc. Int. Conf. on Computer Design (ICCD)*, pages 220–223. IEEE Comp. Soc. Press, Sept. 1990.

[50] M. Kwiatkowska, R. Mehmood, G. Norman, and D. Parker. A symbolic out-of-core solution method for Markov models. *Electronic Notes in Theoretical Computer Science*, 68(4):589 – 604, 2002.

[51] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. CAV*, LNCS 6806, pages 585–591. Springer, 2011.

[52] M. Z. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: a hybrid approach. *Software Tools for Technology Transfer*, 6(2):128–142, 2004.

[53] Y.-T. Lai and S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proceedings of the 29th Conference on Design Automation*, pages 608–613. IEEE Computer Society Press, June 1992.

[54] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Syst. Techn. J.*, 38(4):985–999, July 1959.

[55] Y. Matsunaga, P. McGeer, and R. Brayton. On computing the transitive closure of a state transition relation. In *Design Automation, 1993. 30th Conference on*, pages 260–265, June 1993.

[56] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1992. CMU-CS-92-131.

[57] K. L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[58] R. Mehmood, D. Parker, and M. Kwiatkowska. An efficient BDD-based implementation of Gauss-Seidel for CTMC analysis. Technical report, University of Birmingham, 2003.

[59] K. Milvang-Jensen and A. J. Hu. BDDNOW: A parallel bdd package. In *Proc. FMCAD*, LNCS 1522, pages 501–507. Springer, Nov. 1998.

[60] A. S. Miner. *Data structures for the analysis of large structured Markov models*. PhD thesis, College of William and Mary, 2000.

[61] J. K. Muppala, G. Ciardo, and K. S. Trivedi. Stochastic reward nets for reliability prediction. *Communications in Reliability, Maintenability and Serviceability*, 1(2):9–20, 1994.

[62] Y. Parasuram, E. Stabler, and S.-K. Chin. Parallel implementation of BDD algorithms using a distributed shared memory. In *The 27th Hawaii International Conference on System Sciences (HICSS'94)*, volume 1, pages 16–25. IEEE Comp. Soc. Press, 1994.

[63] R. Pelánek. BEEM: benchmarks for explicit model checkers. In *Proceedings of the 14th international SPIN conference on Model checking software*, pages 263–267. Springer, 2007.

[64] D. Peled. Ten years of partial order reduction. In A. Hu and M. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 17–28. Springer, 1998.

[65] C. Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, 1962.

[66] B. Plateau. On the stochastic structure of parallelism and synchronisation models for distributed algorithms. In *Proc. ACM SIGMETRICS*, pages 147–153, May 1985.

[67] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *5th International Symposium in Programming*, LNCS 137, pages 337–351. Springer, April 1982.

[68] K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In *FMCAD '00: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 143–160, London, UK, 2000. Springer-Verlag.

[69] P. Roux and R. Siminiceanu. Model Checking with Edge-valued Decision Diagrams. In *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, pages 222–226. NASA, April 2010.

[70] S. Safra and M. Y. Vardi. On omega-automata and temporal logic. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing, 15-17 May 1989, Seattle, Washington, USA*, pages 127–137. ACM, 1989.

[71] V. Schuppan and A. Biere. Efficient reduction of finite state model checking to reachability analysis. *Software Tools for Technology Transfer*, 5(2):185–204, March 2004.

[72] V. Schuppan and A. Biere. Shortest counterexamples for symbolic model checking of LTL with past. In *Proc. TACAS*, LNCS 3440, pages 493–509. Springer, 2005.

[73] F. Somenzi, K. Ravi, and R. Bloem. Analysis of symbolic SCC hull algorithms. In *FMCAD '02: Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, pages 88–105, London, UK, 2002. Springer-Verlag.

[74] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[75] J. Tan, G. S. Avrunin, L. A. Clarke, S. Zilberstein, and S. Leue. Heuristic-guided counterexample search in FLAVERS. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, SIGSOFT '04/FSE-12, pages 201–210. ACM, 2004.

[76] R. Tarjan. Depth-first search and linear graph algorithms. In *Proceedings of the 12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pages 114–121, Washington, DC, USA, 1971. IEEE Computer Society.

[77] M. Wan and G. Ciardo. Symbolic state-space generation of asynchronous systems using extensible decision diagrams. In *Proc. SOFSEM*, LNCS 5404, pages 582–594. Springer, 2009.

[78] M. Wan, G. Ciardo, and A. S. Miner. Approximate steady-state analysis of large Markov models based on the structure of their decision diagram encoding. *Perf. Eval.*, 68:463–486, 2011.

[79] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Proc. CAV*, pages 124–138, 2000.

[80] A. Xie and P. A. Beerel. Efficient state classification of finite-state Markov chains. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 17(12):1334–1339, 1998.

[81] A. Xie and P. A. Beerel. Implicit enumeration of strongly connected components and an application to formal verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 19(10):1225–1230, 2000.

[82] Y. Zhao and G. Ciardo. Symbolic CTL model checking of asynchronous systems using constrained saturation. In *Proc. ATVA*, LNCS 5799, pages 368–381. Springer, 2009.

[83] Y. Zhao and K. Y. Rozier. Formal specification and verification of a coordination protocol for an automated air traffic control system. In *Proc. AVoCS*, volume 53 of *Electronic Communications of the EASST*. European Association of Software Science and Technology, 2012.