

2004

## Structural model checking

Radu Siminiceanu

*College of William & Mary - Arts & Sciences*

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

---

### Recommended Citation

Siminiceanu, Radu, "Structural model checking" (2004). *Dissertations, Theses, and Masters Projects*. Paper 1539623441.

<https://dx.doi.org/doi:10.21220/s2-8tcm-na84>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact [scholarworks@wm.edu](mailto:scholarworks@wm.edu).

# STRUCTURAL MODEL CHECKING

---

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William and Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

---

by

Radu Siminiceanu

2003

# APPROVAL SHEET

This dissertation is submitted in partial fulfillment of  
the requirements for the degree of

Doctor of Philosophy

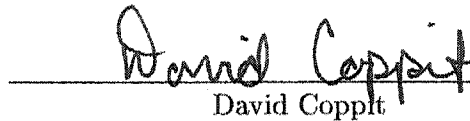


Radu Siminiceanu

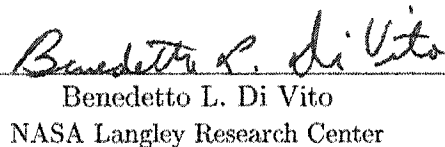
Approved by the Committee, December 2003

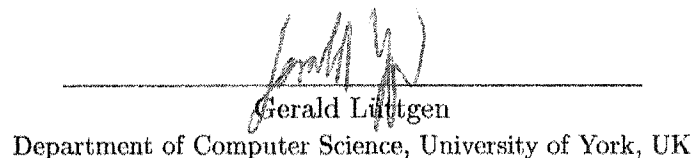
  
Gianfranco Ciardo, Chair

  
Nikos Chrisochoides

  
David Coppi

  
Weizhen Mao

  
Benedetto L. Di Vito  
NASA Langley Research Center

  
Gerald Lüttgen  
Department of Computer Science, University of York, UK

*To my parents*



# Table of Contents

<b>Acknowledgments</b>	<b>ix</b>
<b>List of Tables</b>	<b>x</b>
<b>List of Figures</b>	<b>xiii</b>
<b>Abstract</b>	<b>xiv</b>
<b>1 Introduction</b>	<b>2</b>
1.1 Overview of Formal Methods . . . . .	5
1.1.1 Specification . . . . .	5
1.1.2 Verification . . . . .	6
1.1.2.1 Simulation . . . . .	6
1.1.2.2 Theorem Proving . . . . .	7
1.1.2.3 Model Checking . . . . .	8
1.2 Contribution . . . . .	10
1.3 Organization . . . . .	11
<b>2 State Space Construction</b>	<b>12</b>

2.1	Discrete state systems . . . . .	13
2.1.1	Modeling with Petri Nets . . . . .	14
2.2	State spaces . . . . .	16
2.3	Explicit state-space generation . . . . .	17
2.4	The state space explosion problem . . . . .	18
2.5	Explicit methods . . . . .	19
2.6	Symbolic methods . . . . .	22
2.6.1	Symbolic encoding of states . . . . .	24
2.6.1.1	Binary decision diagrams . . . . .	24
2.6.1.2	Multi-way decision diagrams . . . . .	30
2.6.2	MDDs and structured systems . . . . .	33
2.6.3	Symbolic encoding of next-state functions . . . . .	35
<b>3</b>	<b>The Road to Saturation</b>	<b>38</b>
3.1	Breadth-first symbolic state-space generation . . . . .	38
3.2	Kronecker encoding of the next-state function . . . . .	43
3.3	An identity crisis . . . . .	46
3.4	Event locality . . . . .	48
3.5	In-place updates . . . . .	50
3.6	The saturation strategy . . . . .	53
3.7	Correctness . . . . .	59
3.8	Implementation issues . . . . .	61
3.9	Results . . . . .	62

<b>4</b>	<b>Saturation Unbound</b>	<b>69</b>
4.1	Local state spaces with unknown bounds . . . . .	72
4.2	Example . . . . .	75
4.3	Implementation issues . . . . .	82
4.3.1	Data structures for the next-state function . . . . .	82
4.3.2	Unconfirmed vs. confirmed states . . . . .	83
4.3.3	Storing MDD nodes . . . . .	85
4.3.4	MDD nodes of variable size . . . . .	86
4.3.5	Garbage collection . . . . .	88
4.3.6	Overflow of potential local state spaces . . . . .	90
4.4	Results . . . . .	91
<b>5</b>	<b>Structural CTL Model Checking</b>	<b>93</b>
5.1	Temporal logic . . . . .	94
5.1.1	Linear time, branching time . . . . .	95
5.1.2	The CTL*, CTL, and LTL temporal logics . . . . .	97
5.1.3	Computing CTL operators . . . . .	102
5.2	Saturation-based CTL Model Checking . . . . .	104
5.2.1	The <b>EX</b> operator . . . . .	104
5.2.2	The <b>EF</b> operator . . . . .	105
5.2.3	The <b>EU</b> operator . . . . .	105
5.2.4	The <b>EG</b> operator . . . . .	109
5.3	Results . . . . .	111

<b>6</b>	<b>Counterexamples and Witnesses</b>	<b>115</b>
6.1	The distance function . . . . .	116
6.1.1	Explicit decision diagram encoding of state distances . . . . .	117
6.1.2	Symbolic encoding of state distances . . . . .	119
6.2	A new approach . . . . .	121
6.2.1	Edge-valued MDDs . . . . .	121
6.2.2	Canonicity of EV <sup>+</sup> MDDs . . . . .	123
6.3	Operations with EV <sup>+</sup> MDDs . . . . .	125
6.3.1	State-space and distance generation using EV <sup>+</sup> MDDs . . . . .	128
6.3.2	Trace generation using EV <sup>+</sup> MDDs . . . . .	131
6.4	Results . . . . .	133
<b>7</b>	<b>Application: the Runway Safety Monitor</b>	<b>137</b>
7.1	The Runway Incursion Prevention System . . . . .	137
7.2	The SMART model of RSM . . . . .	140
7.2.1	State variables . . . . .	140
7.2.2	Modeling the state transitions . . . . .	143
7.2.2.1	The 3-D motion of targets . . . . .	144
7.2.2.2	Status definitions . . . . .	145
7.2.3	State-space measurements . . . . .	147
7.2.4	Setting the alarm . . . . .	148
7.3	Model Checking RSM . . . . .	153
7.3.1	A “memory-less” property . . . . .	154

7.3.2	Analysis of the transition that causes loss of separation . . . . .	155
7.3.3	A stronger safety property . . . . .	157
7.3.3.1	Counterexamples for strong safety . . . . .	157
7.3.4	Conclusions . . . . .	161
<b>8</b>	<b>Future Research</b>	<b>164</b>
<b>A</b>	<b>Overview of SMART</b>	<b>169</b>
<b>B</b>	<b>Benchmark of Examples</b>	<b>177</b>
B.1	Dining philosophers . . . . .	177
B.2	A flexible manufacturing system . . . . .	179
B.3	Slotted ring . . . . .	181
B.4	A Kanban system . . . . .	182
B.5	Randomized leader election protocol . . . . .	184
B.6	A round-robin mutual exclusion protocol . . . . .	189
	<b>Bibliography</b>	<b>192</b>
	<b>Vita</b>	<b>203</b>

## ACKNOWLEDGMENTS

I owe my thanks to many people who have supported me during my studies. From the very beginning, this work would not have been possible without the guidance of my adviser, Dr. Gianfranco Ciardo. His constant strive for excellence, impeccable work ethic and brilliant ideas have made it easier for me to achieve my goals. His undeniable mentoring skills are reflected in the way every student that worked with him grew from a just good student into a good researcher.

I am grateful to Dr. Gerald Lüttgen for my initiation in the field of Formal Methods and for the internship that he entrusted me with so early in my studies. I truly admire his research attitude and writing skills. I am equally grateful to the people at NASA Langley Research Center, especially to Dr. Ben Di Vito, for fostering our research ideas and providing us with many opportunities, most importantly through their financial support.

I would like to thank my committee, for supporting my work and providing valuable comments, and the colleagues that I collaborated with: Andrew Miner, for jump-starting this whole body of work and whose design and implementation of SMART have made it easy to put my ideas in practice, and Robert Marmorstein, who provided figures and comments for one of the chapters in this thesis. Special thanks go to Zvezdan Petković, who was always there for me with a good advice on everything from technical support to the myriad of every day life problems that I faced. I also am grateful to Carol Conner, Sandra, David, and the entire Wagoner family for their magnanimous help in making a home away from home in America.

On a personal note, my endeavor would not have been possible without the affectionate support of my whole family. My deepest gratitude goes to my parents, Lizeta and Ilie, whom I simply owe everything I am, and to my sister, Laura, who endured an ocean apart of distance between us.

Last, but not least, my thoughts and love go to my sweet and generous wife, Valentina, who brings out the best in me and makes life all worth it.

# List of Tables

3.1	Time for state-space generation algorithms in SMART. . . . .	64
3.2	Memory consumption for state-space generation algorithms in SMART. . . .	65
3.3	Results for the saturation algorithm. . . . .	67
4.1	Generation of the state space: On-the-fly vs. pregeneration vs. NuSMV . .	91
5.1	Experimental results for computing EF and EU: SMART vs. NuSMV. . . .	112
5.2	Experimental results for computing EG: SMART vs. NuSMV. . . . .	113
6.1	Experimental results for computing the distance function. . . . .	134
7.1	The state-to-state transition matrix for RSM. . . . .	147
7.2	State space measurements for RSM: number of states. . . . .	148
7.3	State-space construction time for RSM (in seconds). . . . .	149
7.4	State-space generation memory consumption for RSM (in Megabytes). . . .	149
7.5	RSM protocol: alarm setting criteria. . . . .	150
A.1	Model checking functions in SMART. . . . .	175

# List of Figures

1.1	The “map” of Formal Methods . . . . .	5
2.1	A small Petri net example. . . . .	15
2.2	The underlying discrete state model of the Petri net in Figure 2.1 . . . . .	16
2.3	Algorithm for explicit state space exploration. . . . .	18
2.4	A BDD encoding of a boolean function . . . . .	24
2.5	The effect of variable ordering over the BDD size . . . . .	26
2.6	The algorithm for binary operations on BDDs. . . . .	28
2.7	An example MDD. . . . .	32
2.8	The algorithm for the union of MDDs. . . . .	34
2.9	Image computation with MDDs. . . . .	36
3.1	Two symbolic breadth-first generation algorithms. . . . .	39
3.2	Algorithm for symbolic breadth-first search with chaining. . . . .	40
3.3	A producer-consumer model and its submodels. . . . .	41
3.4	The local states of submodels in the producer-consumer model. . . . .	42
3.5	Kronecker encoding of $\mathcal{N}$ for the producer-consumer model. . . . .	45



3.6	Satisfying Kronecker-consistency requirements. . . . .	45
3.7	Kronecker versus MDD encoding of the next state function. . . . .	47
3.8	Algorithm for firing events with in-place updates. . . . .	51
3.9	The pseudocode for the <i>Generate</i> , <i>Saturate</i> algorithms. . . . .	57
3.10	The pseudocode for <i>SatFire</i> and <i>SatRecFire</i> . . . . .	58
4.1	Local state spaces built in isolation. . . . .	72
4.2	The <i>Confirm</i> procedure for saturation on-the-fly. . . . .	74
4.3	Saturation by example (part 1). . . . .	76
4.4	Saturation by example (part 2). . . . .	77
4.5	Saturation by example (part 3). . . . .	78
4.6	Storage for the matrix nodes. . . . .	83
4.7	Node and arc storage for MDDs. . . . .	86
4.8	The implicit index 1 in reduced vs. quasi-reduced MDDs. . . . .	87
4.9	Example of potential, but not actual, overflow of a local state space. . . . .	90
5.1	The eight basic CTL operators . . . . .	100
5.2	The four basic LTL operators . . . . .	101
5.3	The traditional algorithm to compute <b>EU</b> . . . . .	106
5.4	The saturation-based algorithm to compute <b>EU</b> . . . . .	107
5.5	Comparing BFS and saturation order: distance vs. unsafe distance. . . . .	109
5.6	Traditional and saturation-based <b>EG</b> algorithms. . . . .	110
6.1	Storing the distance function: ADD vs. forests of MDDs. . . . .	118
6.2	Canonical and non-canonical EVBDDs. . . . .	120

6.3	Storing total and partial arithmetic functions with EV <sup>+</sup> MDDs. . . . .	123
6.4	The <i>UnionMin</i> algorithm for EV <sup>+</sup> MDDs. . . . .	125
6.5	An example of the <i>UnionMin</i> operator for EV <sup>+</sup> MDDs. . . . .	127
6.6	The pseudocode for <i>BuildDistance</i> , <i>EVSaturate</i> and <i>EVFire</i> . . . . .	129
6.7	The pseudo-code for <i>EVRecFire</i> . . . . .	130
7.1	The Runway Safety Monitor flow chart. . . . .	139
7.2	RSM: missed alarm scenario 1, at ground level. . . . .	154
7.3	RSM: missed alarm scenario 2, airborne. . . . .	156
7.4	RSM: missed alarm scenario 3, at ground level. . . . .	158
7.5	RSM: missed alarm scenario 4, at ground level. . . . .	159
7.6	RSM: missed alarm scenario 5, aircraft vs vehicle. . . . .	161
B.1	The dining philosophers, $i^{\text{th}}$ subnet. . . . .	177
B.2	A flexible manufacturing system. . . . .	179
B.3	Slotted ring model, $i^{\text{th}}$ subnet. . . . .	181
B.4	A kanban system. . . . .	183
B.5	Statechart for the leader election protocol. . . . .	185
B.6	The round-robin mutual exclusion protocol. . . . .	190

## ABSTRACT

The introduction of symbolic approaches, based on Binary Decision Diagrams (BDD), to Model Checking has led to significant improvements in Formal Verification, by allowing the analysis of very large systems, such as complex circuit designs. These were previously beyond the reach of traditional, explicit methods, due to the *state space explosion* phenomenon. However, after the initial success, the BDD technology has peaked, due to a similar problem, the BDD explosion.

We present a new approach to symbolic Model Checking that is based on exploiting the system structure. This technique is characterized by several unique features, including an encoding of states with Multiway Decision Diagrams (MDD) and of transitions with boolean Kronecker matrices. This approach naturally captures the property of event locality, inherently present in the class of globally asynchronous/locally synchronous systems.

The most important contribution of our work is the *saturation* algorithm for state space construction. Using saturation, the peak size of the MDD during the exploration is drastically reduced, often to sizes equal or comparable to the final MDD size, which makes it optimal in these terms. Subsequently, saturation can achieve similar reductions in run-times. When compared to the leading state-of-the art tools based on traditional symbolic approaches, saturation is up to 100,000 times faster and uses up to 1,000 times less memory. This enables our approach to study much larger systems than ever considered. Following the success in state space exploration, we extend the applicability of the saturation algorithm to CTL Model Checking, and also to efficient generation of shortest length counterexamples for safety properties, with similar results.

This approach to automatic verification is implemented in the tool SMART. We test the new model checker on a real life, industrial size application: the NASA Runway Safety Monitor (RSM). The analysis exposes a number of potential problems with the decision procedure designed to signal all hazardous situations during takeoff and landing procedures on runways. Attempts to verify RSM with other model checkers (NuSMV, SPIN) fail due to excessive memory consumption, showing that our structural method is superior to existing symbolic approaches.

## STRUCTURAL MODEL CHECKING

# Chapter 1

## Introduction

*“A clever man commits no minor blunders”*

Johann Wolfgang von Goethe (1749-1832)

Today’s hardware and software systems grow in size and complexity at a very fast pace. Unfortunately, the rapid advance in technology sophistication is not matched by the advance in the degree of certification of the deployed devices. Therefore, with every leap forward in technology, the risk for small and subtle errors to escape undetected in the process of design and verification with low-cost empirical methods increases substantially. At the same time, an exhaustive verification process takes considerable amounts of resources, in terms of time, human expertise, and money. The major obstacle in using computerized tools to verifying modern protocols is usually the *state-space explosion phenomenon*. As the complexity of a system increases, the memory and time required to store its combinatorially expanding state space can easily become excessive. On the other hand, if appropriate measures are not taken, the cost of errors eventually occurring during operation can be enormous. There is a number of (in)famous examples already on record:

- The Pentium I processor division bug [57], caused by a circuit design error, costed Intel half a billion dollars in 1994. The discovery of the error was credited to Prof. Thomas Nicely, a mathematics professor at the nearby Lynchburg College in Virginia;

- The explosion of the Ariane 5 rocket during liftoff, on June 4 1996, due to a floating point conversion error in a software control module [55], caused the European Space Agency an estimated loss of seven billion dollars and ten years of development efforts;
- The loss of the Mars Polar Lander exploration probe in December 1999, due in part to an alleged object initialization error in the touchdown sensing software, which was not correctly tested [86], had a price tag of 120 million dollars in equipment and set back an entire research program at the Jet Propulsion Laboratory;
- The Y2K-bug scare [71], which can be blamed on the computer software industry as a collective, whose prevention costs worldwide are yet to be quantified, but are estimated between 500 billion and 5 trillion dollars;

But most critical is the prospect of even more catastrophic events, where human life is at stake, such as aircraft collisions or malfunctioning of nuclear plant surveillance. Safety-critical systems, such as flight guidance, collision avoidance, and early warning systems, require a *high degree of assurance* of design correctness [23]. Existing techniques for verifying safety properties still rely heavily on testing and simulation, which examine only a portion of the behaviors of the system. Since flaws can hide in the unexamined portions of the system, engineers need reliable means to design and operate increasingly large systems, despite their vast complexity.

The field of Formal Methods is one way to achieve this goal, by using rigorous mathematical techniques for the *specification* and *verification* of hardware and software systems. *Model Checking* is one direction in Formal Methods, concerned with the tasks of representing a system as an automaton, usually finite-state, and then showing that the initial state

of this automaton satisfies a *temporal logic* statement [116]. Model Checking has gained increasing attention since the development of *symbolic* techniques, based on *binary decision diagrams (BDDs)* [17]. *Symbolic Model Checking* [21] is known to be effective for a number of simple temporal logics, such as CTL (*Computation Tree Logic*) and LTL (*Linear Temporal Logic*) [37], as it allows for the efficient storage and manipulation of the large sets of states corresponding to temporal logic formulae. However, practical limitations to symbolic Model Checking still exist.

- First, memory and time requirements might be excessive when tackling real systems. This is especially true since the size (in number of nodes) of the BDD encoding the set of states corresponding to a CTL formula is usually much larger *during* the fixed-point iterations than upon convergence. This phenomenon can be viewed as the reincarnation of the state space explosion in the form of *BDD explosion*. This has spurred work in two major directions. Distributed and parallel algorithms for BDD manipulation [106, 137] attempt to alleviate the problem by corralling the resources of multiple workstations. However, the parallelization of BDD algorithms is notoriously ineffective, and the resulting algorithms have very little if any speed-up. Another direction has concentrated on verification techniques that try to reduce the number of expensive BDD computations in an attempt to avoid the risk of BDD expansion. This includes the work in [24, 111, 126] that reduce the number of iteration steps needed to explore the state space symbolically, and also the heuristics to better guide the exploration of states [12, 121].
- Second, symbolic Model Checking has been quite successful for hardware verification

but software, in particular distributed software, has so far been considered beyond reach. This is because the state space of software is much larger, but also because of the widely-held belief that symbolic techniques work well only in synchronous settings.

In this work, we attempt to dispel these myths by showing that symbolic Model Checking based on the model *structure* copes well with asynchronous behavior and even benefits from it. Furthermore, the techniques we introduce excel at reducing the *peak memory consumption* in the fixed-point iterations, thus directly addressing the issue of BDD explosion.

## 1.1 Overview of Formal Methods

As mentioned before, the field of Formal Methods includes two areas, *specification* and *verification*, which we briefly overview next. The map of Formal Methods is sketched in the diagram of Figure 1.1 as a tree and our work is situated along the highlighted path.

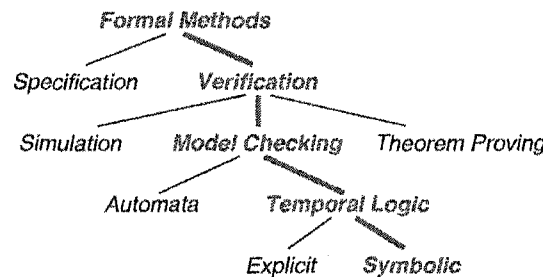


Figure 1.1: The “map” of Formal Methods

### 1.1.1 Specification

Specification (and/or modeling) is the process of describing a real-life system and its desired properties in a formal way, by using a specification *language*. The properties that need to



be expressed may range from functional behavior, timing behavior, internal structure, to performance characteristics.

Z [128] and VDM [85] are the most popular languages for specifying the *sequential* behavior of state-based systems. States are described by means of rich structures such as sets, relations, and functions. The transitions between states are given in terms of pre-conditions and post-conditions logic formulae.

Other methods, like CSP [78], CCS [105], state-charts [74], and I/O automata [102] focus on specifying the behavior of *concurrent* systems.

### 1.1.2 Verification

Verification goes one step beyond modeling, by *analyzing* the specification of the system for desired properties. The verification can be partial or exhaustive. Simulation and test case generation fall in the former category, while Model Checking and Theorem Proving are in the latter category.

#### 1.1.2.1 Simulation

Excluded nowadays by purists, but widely used in practice, simulation and test case generation were the only means of testing and validating hardware and software in the early days of system design. The two methods are similar in that they attempt to cover, with a set of inputs, as many execution paths in the system as possible, mostly the ones that presumably contain errors. The difference lies in the fact that simulation runs on a theoretical model of the system, while testing runs on a physical prototype of the system itself.

The main disadvantage of both remains the inability to cover all the possible cases,

hence the risk of missing unwanted events that occur with very small probability. Good simulations are difficult to build and time consuming. Their effectiveness sharply drops in the late stages of the design, when the number of errors is considerably diminished. The time necessary to discover bugs in the late phases increases dramatically.

The sheer size of modern systems is gradually making the traditional methods obsolete, as at the end of a simulation session one is never fully sure of the validity of the model. A suggestive example is the TCAS II system [60] for which the Federal Aviation Administration has contracted the Rannoch company to build scenarios for the aircraft collision avoidance protocol. The end-product of Rannoch comprised 300 encounter scenarios. Put in the light of the actual state-space size (that the tool SMV fails to completely build within 1 Gigabyte of memory, after constructing over  $10^{20}$  possible states), the number of tested scenarios is alarmingly small. The more so as the system is *already* installed on all US commercial aircraft as of January 1994 (and a worldwide extension, ACAS II, is promoted for equipping all aircraft by January 2005).

#### 1.1.2.2 Theorem Proving

Theorem Proving is the most powerful formal verification approach, in which both the system and its desired properties are expressed as formulae in some mathematical logic, given as a set of axioms and a set of inference rules to make deductions. The process of finding a proof for a property starts from the axioms and then uses the inference rules (and possibly derived definitions and intermediate lemmas) to validate the initial assertions. Proofs can be constructed by hand, but machine-assisted proving has become more and more powerful, although not fully automated, remaining mainly interactive.

In contrast to Model Checking, Theorem Proving can deal with infinite systems, since it relies on more powerful techniques, like structural induction. However, the interaction with a user results in a slow and sometimes error-prone process. Depending on the degree of automation, theorem provers can be classified in three groups:

- User-guided automatic deduction tools: Nqhtm [14], Reve [99], RRL [88], Eves [49], LP [65], and ACL2 [89] are guided by user input, which is a sequence of lemmas, but each theorem is proven automatically, using built-in heuristics. The milestone in this category has been the proof of Gödel’s first incompleteness theorem in Nqhtm by Boyer and Moore [14];
- Proof checkers: LCF [70], Nuprl [46], HOL [69], LEGO [101], and Coq [48] are used to formalize and verify problems in Mathematics and Program Verification;
- Combination provers: PVS [110], Analytica [43], and STeP [10] combine Theorem Proving with symbolic algebra, Model Checking, and interactive proof to verify a variety of problems ranging from number theory to hardware design and fault-tolerant algorithms.

### 1.1.2.3 Model Checking

Model Checking is the formal verification approach that relies on building a finite *model* of a system and exhaustively searching its state space to check the desired properties. In principle, the systematic search is guaranteed to end, as the model is finite. The practical limitations are, obviously, the amount of available memory (on one or multiple processors) and the running times of the required algorithms. Therefore, the challenge in Model Check-

ing is devising algorithms and data structures that enable handling of very large state-spaces in a reasonable amount of time.

In practice there are two main approaches in Model Checking: temporal and automata-based. The *temporal Model Checking* was developed independently in the 1980s by Clarke and Emerson [37] in the U.S. and Quielle and Sifakis [119] in France. In this approach, systems are modeled as finite state transition systems, and specifications are expressed in a temporal logic.

In the second approach, the specification is given as an *automaton*. Then, the system which is modeled as an automaton itself is compared to the specification to determine whether it satisfies the required properties. The conclusion can be drawn via language inclusion, refinement orderings, or observational equivalence. Vardi and Wolper showed how to translate temporal logic Model Checking in terms of automata, thus making the connection between the two approaches [133].

Model Checking is fully automated and fast. Its main asset is the delivery of witnesses (and counterexamples) that show how states where the desired property does (or does not) hold can be reached. Its major challenge is represented by the state-space explosion problem. This task has been addressed by the introduction of BDDs, partial order reductions, and semantic minimizations. All of these methods either reduce the size of the model by eliminating unnecessary states, or compress the state-space representation itself.

There are numerous Model Checking tools in use, the best known are listed below:

- Temporal logic model checkers: EMC [37] and Caesar [119], SMV [104] – the first to use symbolic techniques (BDDs), SPIN [81] – based on partial order reductions, Murphy [54], the Concurrency Workbench [45] – based on  $\mu$ -calculus, SVE [61], FORMAT

[50], and CV [53] – for hardware verification, HyTech [5] – for hybrid systems, and Kronos [77] – for real-time systems.

- Combined checkers: Berkeley’s HSIS [80] combines model checking with language inclusion, Stanford’s STeP [10] uses deductive methods, VIS [15] employs logic synthesis, PVS [110] is a theorem prover with Model Checking abilities for  $\mu$ -calculus, while the METAFame environment [129] supports Model Checking in the entire software development process.

## 1.2 Contribution

This work contributes with a series of novel techniques to the area of temporal logic Model Checking. We have designed a new model checker, integrated in the software tool SMART [25] at the College of William and Mary, which features:

- *Multiway Decision Diagrams* (MDD) [87] for compressed state storage of *structured* discrete-state systems;
- *Kronecker matrix* [6] storage for efficient representation of the transition relation between states, based on a decomposition of the model into submodels;
- *Saturation* [27], a revolutionary iteration strategy for exploring state spaces and computing fixed-point model checking operators;
- An *on-the-fly* algorithm that performs an exact computation of local state spaces of the decomposed model for each state variable of the system [28];

- *Edge-valued* decision diagrams [30], an advanced storage scheme to encode information that helps building shortest paths in reachability graphs, with direct implications in providing shortest length counterexamples for safety properties in Model Checking.

We call this new approach *Structural Model Checking* [31].

### 1.3 Organization

The remainder of this document is organized as follows. Chapter 2 introduces the basic concepts in discrete state modeling and presents various alternatives to addressing the problem of efficient state space construction. Chapter 3 describes our contributions and improvements to the problem of state space construction, leading to our most advanced algorithm, based on the idea of *saturation*. An extension of the algorithm in a less restrictive context is presented in Chapter 4, where we also compare the performance of our algorithm implemented in SMART with another state-of-the-art model checker, NuSMV. Chapters 5 and 6 further expand the scope of our saturation strategy to other areas of Model Checking, such as the efficient computation of temporal logic operators and the generation of counterexamples, along with the corresponding experimental results. Chapter 7 presents our findings regarding the performance and applicability of our model checker to a real-life example: the Runway Safety Monitor. We conclude by discussing future research plans in Chapter 8.

## Chapter 2

# State Space Construction

In this chapter, we first look at the models most frequently used in practice and examine their common characteristics. We then proceed with defining the *state space* of a model, discussing the various existing techniques to construct it, and the major challenges in undergoing this task.

Since the introduction of automated verification tools in industry, the most popular systems to be analyzed were digital circuits, concurrent programs, distributed protocols, and reactive systems. When talking about behavior, one should capture in one way or another the notion of time and evolution. Therefore, the basic entities that all the models have to be built on are *states* and *transitions* (between states). The states should be seen as snapshots of the parameters of the system at specific moments in time, while the transitions are a set of rules that describe the changes in the system state when an event occurs.

Several frameworks have been proposed for modeling systems. We adopt the following terminology.

## 2.1 Discrete state systems

**Definition 2.1.1** A discrete-state model is a triple  $M = (\widehat{S}, \mathbf{s}, \mathcal{N})$ , where

- $\widehat{S}$  is the set of potential states of the model, specifying the “type” of the states.

We indicate states with lowercase boldface letters,  $\mathbf{i}, \mathbf{j}, \dots$

- $\mathbf{s} \in \widehat{S}$  is the initial state of the model.
- $\mathcal{N} : \widehat{S} \rightarrow 2^{\widehat{S}}$  is the next-state function specifying the states reachable from each state in a single step.

There exist a number of variations to the above definition. One of them assumes a more general case of a set of initial states,  $S_0$ . This can be easily integrated in our definition by adding a unique initial state  $\mathbf{s}$  and defining  $\mathcal{N}(\mathbf{s}) = S_0$ .

The next-state function  $\mathcal{N}$  is sometimes defined as a binary relation  $\mathcal{N} \subseteq \widehat{S} \times \widehat{S}$ , such that  $(\mathbf{i}, \mathbf{j}) \in \mathcal{N}$  if and only if  $\mathbf{i}$  can transition to  $\mathbf{j}$ . In this case,  $\mathcal{N}$  is called the *transition relation*.

Another variation is the partitioning of  $\mathcal{N}$  into a union of next-state functions [20]:  $\mathcal{N}(\mathbf{i}) = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e(\mathbf{i})$ , where  $\mathcal{E}$  is the (finite) set of *events* in the model. Then,  $\mathcal{N}_e$  is the next-state function associated with event  $e$ , i.e.,  $\mathcal{N}_e(\mathbf{i})$  is the set of states the system can enter when  $e$  occurs, or *fires*, in state  $\mathbf{i}$ . An event  $e$  is said to be *disabled* in  $\mathbf{i}$  if  $\mathcal{N}_e(\mathbf{i}) = \emptyset$ ; otherwise, it is *enabled*.

Discrete-state systems are usually described in a *formalism* that allows the specification of its states and transitions. *Low-level formalisms* explicitly define the entire set of states that the system can be in, together with the complete transition relation. A more



convenient approach is to use a *high-level formalism* that recurs to more sophisticated mathematical objects to define the behavior of the system. This is done by describing rules that characterize *groups* of states and events together.

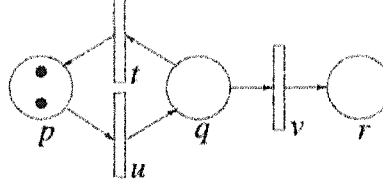
### 2.1.1 Modeling with Petri Nets

Introduced by C.A.Petri in 1962 [113], Petri nets are an elegant high-level mathematical formalism that has been well studied and has since become popular due to some of its practical properties.

**Definition 2.1.2** *A Petri net  $N$  is a quintuple  $(P, T, D^-, D^+, \mu_0)$ , where*

- *$P$  is a finite set of places; places can hold a non-negative number of tokens; an assignment  $\mu : P \rightarrow \mathbb{N}$  of places with tokens is called a marking;*
- *$T$  is a finite set of transitions (or events), which update the number of tokens in places;*
- *$\mu_0 : P \rightarrow \mathbb{N}$  is an initial marking of places with tokens;*
- *$D^+, D^- : P \times T \rightarrow \mathbb{N}$ , describe the rules to add and remove tokens from places by firing transitions;*

Note that a Petri net defines a bipartite directed graph, where the sets of nodes are  $P$  and  $T$ , and the set of arcs  $A \subseteq (P \times T) \cup (T \times P)$  is uniquely determined by the two functions  $D^+$  and  $D^-$ . There is an arc from place  $p$  to transition  $t$  if and only if  $D^-(p, t) > 0$  and respectively from  $t$  to  $p$  if and only if  $D^+(p, t) > 0$ .



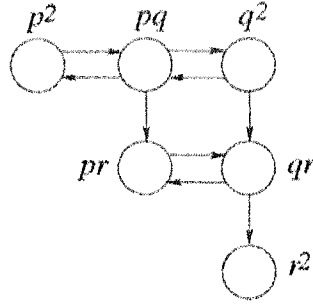
**Figure 2.1:** A small Petri net example.

A transition  $t$  is said to be enabled in a marking  $\mu$  if and only if its input places contain at least as many tokens as the corresponding arc cardinalities,  $\forall p \in P : \mu(p) \geq D^-(p, t)$ . Enabled transitions may fire by removing the number of tokens indicated by the input arc cardinality, and adding to each of the output places the number of tokens indicated by the output arc cardinality. Hence, the firing of an enabled transition  $t$  changes a marking  $\mu$  to a marking  $\mu'$ , where  $\mu'(p) = \mu(p) + D^+(p, t) - D^-(p, t)$ . The firing is sometimes written as  $\mu \xrightarrow{t} \mu'$ .

Figure 2.1 displays a small Petri net, with places  $P = \{p, q, r\}$ , transitions  $T = \{t, u, v\}$ , and arcs  $A = \{(q, t), (t, p), (p, u), (u, q), (q, v), (v, r)\}$ .  $D^-$  and  $D^+$  take the value 1 for all existing arcs, and  $\mu_0(p) = 2$ ,  $\mu_0(q) = 0$ ,  $\mu_0(r) = 0$ . For a more concise representation, we will also use the superscript notation for markings:  $\mu_0 = p^2q^0r^0$ .

Although there is a connection between the places and transitions of a Petri net, and the states and events in a model, it is now clear that the discrete state model that a Petri net defines does not coincide with the net itself. In Figure 2.1 for example, the Petri net has three places and three transitions, while its discrete state model has six possible states and nine transitions, as shown in Figure 2.2.

In the underlying state model, the states are represented by a set of markings,  $\hat{S} \subset \mathbb{N}^P$ , and the transition relation is defined between markings:  $(\mu_1, \mu_2) \in \mathcal{N}$  if and only if there



**Figure 2.2:** The underlying discrete state model of the Petri net in Figure 2.1

exists an enabled transition  $t$  in  $\mu_1$ , and  $\mu_1 \xrightarrow{t} \mu_2$ .

There are several extensions of ordinary Petri nets, such as inhibitor arcs, guards, marking dependent arc cardinalities (self modifying nets), firing times (Stochastic Petri Nets, or SPNs), Colored Petri nets, and Hierarchical Petri Nets [109].

Besides providing a modeling formalism, Petri nets have also been studied as a *computation model*, by means of their associated *language* [112]. The class of standard Petri net languages is a strict superset of regular languages and a subset of context-sensitive languages. Inhibitor arcs, marking dependent arc cardinalities and firing times add expressive power to ordinary Petri nets. In fact these generalizations have the same expressive power as Turing machines [112] (they are Turing-equivalent).

## 2.2 State spaces

Given a discrete state model  $M = (\hat{S}, \mathbf{s}, \mathcal{N})$ , the *state space*  $\mathcal{S} \subseteq \hat{S}$  is defined as the smallest set containing  $\mathbf{s}$  and closed with respect to  $\mathcal{N}$ :

$$\mathcal{S} = \{\mathbf{s}\} \cup \mathcal{N}(\mathbf{s}) \cup \mathcal{N}(\mathcal{N}(\mathbf{s})) \cup \dots = \mathcal{N}^*(\mathbf{s}),$$

where “ $*$ ” denotes the reflexive and transitive closure of applying the function, and the

application of  $\mathcal{N}$  is extended to sets of states:  $\mathcal{N}(\mathcal{X}) = \bigcup_{s \in \mathcal{X}} \mathcal{N}(s)$ .

An equivalent way to define  $\mathcal{S}$  is as the smallest fixed point of the equation

$$\mathcal{S} = \{\mathbf{s}\} \cup \mathcal{N}(\mathcal{S}).$$

Starting from these definitions, there are several algorithms to construct the state space of a system. They are distinguished by two aspects: the *iteration strategy* and the *data structures* used to store states.

The iteration strategy specifies a systematic mechanism to discover reachable states, starting from the initial state and firing events  $\mathcal{N}_e$  in a specific order. A fundamental result was established in [66], by showing that the state space can be constructed by firing the individual events in an *arbitrary* order, as long as each event is considered often enough.

From the state storage point of view, state space generation algorithms can be classified in two categories: *explicit* and *implicit* (or symbolic). Explicit techniques store each state individually, hence the space needed to store some representation of the state-space is at least linear in the number of constructed states. Symbolic techniques instead represent states implicitly, using advanced data structures to compactly encode and manipulate entire sets of states at once.

## 2.3 Explicit state-space generation

In traditional explicit state-space generation approaches, states are discovered one by one.

Figure 2.3 shows a general algorithm that stores the unexplored states in a set  $\mathcal{U}$ . If  $\mathcal{U}$  is managed as a *queue*, the exploration is breadth-first. If  $\mathcal{U}$  is managed as a *stack*, the

exploration is depth-first. Statement 11 is performed as many times as there are state-to-state transitions between reachable states, thus an efficient search data structure, such as a hash table or a search tree, must be used to organize the states in  $\mathcal{S}$ .

Each state requires a number of bytes for its storage, so it is efficient to *index* states with a function  $\psi : \widehat{\mathcal{S}} \rightarrow \mathbb{N} \cup \{\text{null}\}$  such that  $\psi(\mathbf{i}) = i \in \{0, \dots, |\mathcal{S}|-1\}$  if  $\mathbf{i} \in \mathcal{S}$  and  $\psi(\mathbf{i}) = \text{null}$  if  $\mathbf{i} \notin \mathcal{S}$ . A natural definition for  $\psi$  is to assign increasing indices to new states in the order they are found, starting from  $\psi(\mathbf{s}) = 0$ , as shown in Figure 2.3.

<i>ExplicitSsGen</i> ( $\mathbf{s} : \text{state}, \mathcal{N} : \text{stateset} \rightarrow 2^{\text{stateset}}$ ) : <i>stateset</i>	
Compute $\mathcal{S} \subseteq \widehat{\mathcal{S}}$ starting from $\mathbf{s}$ by repeatedly applying $\mathcal{N}$ .	
1. declare $\mathcal{S}, \mathcal{U} : \text{stateset}, \psi : \text{stateset} \rightarrow \mathbb{N} \cup \{\text{null}\}, \mathbf{i}, \mathbf{j} : \text{state};$	
2. $\mathcal{S} \leftarrow \{\mathbf{s}\};$	• known states
3. $\mathcal{U} \leftarrow \{\mathbf{s}\};$	• unexplored known states
4. $\psi(\mathbf{s}) \leftarrow 0;$	• 0 is the index of the first state
5. while $\mathcal{U} \neq \emptyset$ do	• still states to explore
6.   choose a state $\mathbf{i}$ in $\mathcal{U};$	
7.   if $\mathcal{N}(\mathbf{i})$ has been exhausted	
8. $\mathcal{U} \leftarrow \mathcal{U} \setminus \{\mathbf{i}\}$	• remove $\mathbf{i}$ from $\mathcal{U}$
9.   else	
10. $\mathbf{j} \leftarrow$ next element in $\mathcal{N}(\mathbf{i})$	
11.    if $\mathbf{j} \notin \mathcal{S}$ then	• if $\mathbf{j}$ is a new state...
12. $\psi(\mathbf{j}) \leftarrow  \mathcal{S} ;$	• ...assign the next index to it,...
13. $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathbf{j}\};$	• ...add it to $\mathcal{S}$ , and...
14. $\mathcal{U} \leftarrow \mathcal{U} \cup \{\mathbf{j}\};$	• ...remember to explore it later
15. return $\mathcal{S};$	

Figure 2.3: Algorithm for explicit state space exploration.

## 2.4 The state space explosion problem

The size of the state space  $\mathcal{S}$  is a factor that influences in a radical measure not only the efficiency of verification, but also the very *feasibility* of the whole process.

For example, if the components of a system are “loosely coupled” — i.e., they can evolve independently in an asynchronous fashion — the number of possible global states is given by the number of possible combinations of local states for each component. If there are  $K$  subcomponents, each taking up to  $n_i$  values, for  $1 \leq i \leq K$ , there could be as many as  $n_1 \cdot n_2 \cdot \dots \cdot n_K$  different combinations of values. This number can easily become too large to handle. The state-space growth is therefore exponential, as for a system that is regular (i.e.,  $n_i = n$ , for all  $i$ ) and scalable by  $K$ , the upper bound of  $|\mathcal{S}|$  is  $n^K$ . It is then utterly justified to call this phenomenon state-space *explosion*.

Using unsophisticated algorithms that represent the model with explicit storage for states and transitions, each state consuming a number of bits, one will only manage to deal with very small systems. As the models cannot exceed in size the memory capacity of a computer/network of workstations, this usually means the maximum state space size that can be handled is of the order of millions of states ( $10^7 - 10^8$ ). Therefore, only systems with just a handful of (not very complex) subcomponents can be analyzed.

The question that naturally arises is how to deal with larger and more complex systems? In the following we enumerate the most important methods devised to cope with state space explosion [41].

## 2.5 Explicit methods

**Partial order reduction.** The method consists of constructing a reduced version of the state space [82]. It is most suitable for concurrent systems where events may be independent, hence the order in which these events are considered may be irrelevant to the evaluation of

properties. For example, if a sequence of  $n$  transitions has the same effect when executed in any order, the total number of possible interleavings that have to be considered is  $n!$ , while the number of intermediate states is  $2^n$ . If it were possible to choose one representative of all interleavings, then we have to consider just 1 permutation and  $n + 1$  states. The conditions under which the reduction is applicable are that the transitions involved commute and do not disable each other [4, 67, 82, 132, 134], and this is sometimes quite restrictive.

**Structural equivalences.** This technique is designed to replace large portions of the state-space with smaller structures that have the exact same properties with respect to formulae that are checked. To achieve this goal, a notion of equivalence between two structures is introduced, namely the *bisimulation* relation [45]. The most challenging task in exploiting the structural equivalences is searching for those structures that may be bisimulated by smaller structures. In particular, it is often desirable to find a *minimal* equivalent of a (sub)model. The algorithm that checks for the largest bisimulation relation between two structures is inspired from checking the equivalence between languages. In many cases, however, the maximal equivalence is found to be the identity, hence the original model cannot be reduced.

**Symmetry.** A similar approach is targeted at exploiting the internal symmetry of certain models. If there is some symmetry in a model, an equivalence relation can be introduced between groups of states. The rest of the verification is performed on the quotient model induced by that equivalence [38]. As in the previous cases, the quotienting is not allowed to change the behavior of the original system. A typical class of examples are the token-ring-like systems. These consist of a set of behaviorally equivalent “cells” arranged

in a circle and numbered modulo  $n$ . Then, one state in each cell orbits through the whole ring, hence the quotient model may be reduced by a factor of  $n$  in the number of states.

**Compositional reasoning.** The strategy is to check *local* properties using only the part of the system they actually affect. Then, a composition step follows in which the local properties are put together. At any time, only a portion of the global system is considered. The assembling must lead to an equivalent specification of the original model [42, 75, 97].

A related approach is compositional minimization. The state-space of a concurrent system is built stepwise, one parallel component at a time, and the space is minimized at every intermediate step according to behavioral congruences [72].

**Abstraction.** This approach is effective at the system description level, hence before the model is even constructed. Therefore, it avoids the generation of large, unreduced models from the very beginning. There are two well-known techniques based on abstraction: the cone of influence reduction and data abstraction [7, 47].

The *cone of influence reduction* attempts to reduce the number of variables that are taken into consideration from the original model. Starting from the variables that are directly referred to in the specification and going backwards through a chain of dependencies, all the variables that remain outside the chain can be discarded from the model, as they have no influence on the outcome of the verification part. This approach is straightforward, but in many cases it may find very few or even no redundant variables at all, hence little reduction is possible.

*Data abstraction* targets the *domains* of the variables. The size of a domain may be reduced by mapping it to a smaller (abstract) domain that contains only those values that



are significant to establish a property. For example, an integer variable  $v$  may take any value from the domain  $\mathcal{D}_v = \mathbb{Z} = \{\dots, -1, 0, 1, \dots\}$ . But, if the specification cares only about the sign of  $v$ , the domain  $\mathcal{D}_v$  can be simply changed to the abstract domain  $\mathcal{D}_v^A = \{v_-, v_0, v_+\}$ , via the mapping  $a : \mathcal{D}_v \rightarrow \mathcal{D}_v^A$ :

$$a(x) = \begin{cases} v_-, & \text{if } x < 0 \\ v_0, & \text{if } x = 0 \\ v_+, & \text{if } x > 0 \end{cases}$$

The original model is then collapsed into a reduced, abstract model that simulates the original one, and is much smaller. As a result, whatever properties can be established for the original model they will also be valid in the abstract model. Note that by using this method, only formulae over the abstract set of propositions can be checked, so the user has to make sure in advance that it is possible to express properties of the original model in the abstract form [51].

**Induction.** The induction technique allows for reasoning on infinite families of finite-state systems. The most typical examples are the systems parameterized with a “free” variable. Finding an *invariant* reduces the verification process to one, arbitrary, member of the infinite family. To extend the validity to all the systems, an inductive argument is considered acceptable.

## 2.6 Symbolic methods

The common characteristic of all explicit approaches is they attempt to reduce the **number** of states that need to be explored for verifying or invalidating a certain property. An alternative is offered by symbolic methods, which, in contrast to the explicit methods,

reduce the **size** of the representation of the states. By using a compact, sub-linear encoding of large sets of states with advanced data-structures, such as Binary Decision Diagrams, symbolic techniques make it affordable to store the *entire* state space very efficiently.

Binary Decision Diagrams (BDD) have taken Model Checking by storm, ever since their rediscovery in the mid 1980s, thanks to a paper by R. Bryant [16]. BDDs have been previously used in boolean logic to efficiently store functions [3, 98]. But so important was the impact of BDDs in Model Checking, that they have now been “borrowed” by dozens other fields. A simple search for the keyword BDD on several on-line reference sites produced more than 2000 titles, in over 50 different journals and conferences, in the last decade only. At the same time, Bryant’s paper on BDDs is now the most cited paper in all Computer Science, with more than 1300 references, according to the website *Citeseer* (<http://citeseer.nj.nec.com/cs>).

What made BDDs so successful is that they work at a low level, on the storage scheme of states themselves. They exploit the internal structure, symmetries and common properties of the components of the system by default, without additional analysis or prior knowledge of the system. Hence, no heuristic is needed to reduce the **number** of states and in some cases sets of the order of  $10^{1000}$  states can be stored efficiently in as little as a few megabytes of memory. Moreover, since partial order reduction, equivalences, and abstraction are completely orthogonal to the actual representation of the states, any of the above may still be applied to an already efficiently stored state-space.

To symbolically represent a model, one needs a scheme to store its states, and another scheme to store its transition relation.

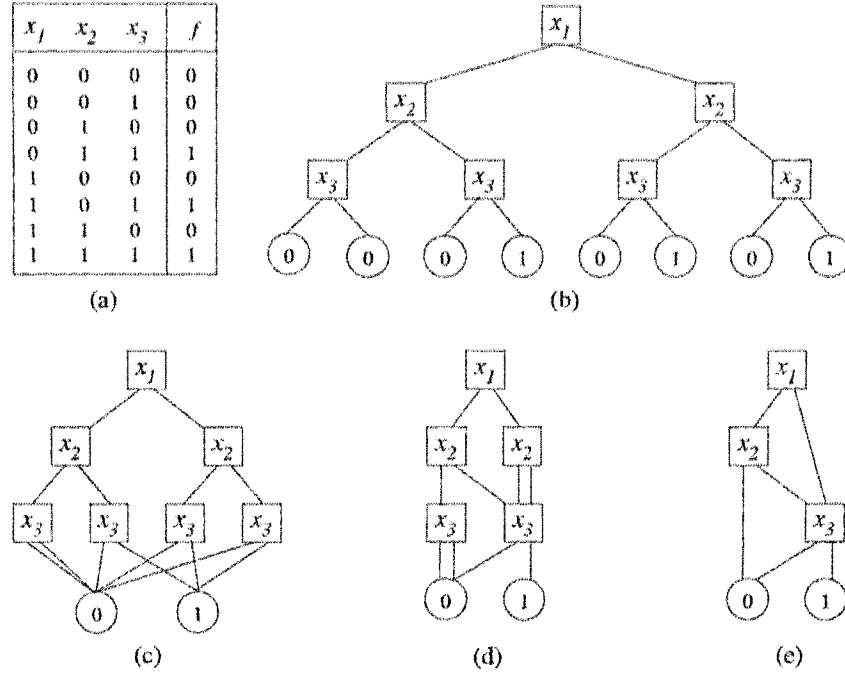


Figure 2.4: A BDD encoding of a boolean function

## 2.6.1 Symbolic encoding of states

### 2.6.1.1 Binary decision diagrams

**Description.** A binary decision diagram represents a boolean function of  $n$  binary variables,  $f : \{0, 1\}^n \rightarrow \{false, true\}$ , as a directed acyclic graph (DAG), with a unique root node and two terminal nodes corresponding to the boolean values *false* and *true* (or 0 and 1 for simplicity). Non-terminal nodes are labeled with variable names  $x_i$ ,  $1 \leq i \leq n$ , and have two outgoing arcs, left and right, corresponding to the two possible values of the boolean variable, 0 and 1. To evaluate a function for a given input, one follows the path from the root to a terminal node corresponding to the truth assignment of variables to 0 and 1. The terminal node indicates the value of the function for the given truth assignment.

A non-terminal node whose two successors are equal is called *redundant*; two nodes

with identical children are called *duplicate* or *non-unique*. The compactness of the BDD representation is achieved by merging duplicate nodes into a single copy, and eliminating redundant nodes.

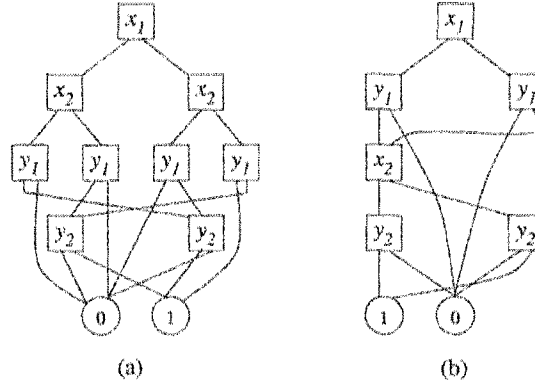
Figure 2.4 shows the transformation of the function  $f(x_1, x_2, x_3) = (\overline{x_1}x_2 + x_1)x_3$  given as a truth table into several intermediate forms leading to a reduced ordered BDD, by a series of reductions: (b) is the initial ordered binary tree, the non-unique terminals and non-terminals are merged in (c) and (d) respectively, and redundant nodes are eliminated in (e).

There is a clear relationship between the set of BDDs with nodes labeled with identifiers  $x_i, 1 \leq i \leq n$  (call it  $BDD_n$ ) and the set of  $n$ -ary boolean functions,  $\{0, 1\}^n \rightarrow \{0, 1\}$ . An arbitrary DAG from  $BDD_n$  truly represents a function only if there are no two distinct paths labeled with the same truth assignment of variables and leading to different terminal nodes.

To make the relationship biunivocal, a canonical representation is needed. That is, two boolean functions that are logically equivalent should be represented by isomorphic BDDs.

This can be achieved by imposing three restrictions on a BDD:

- (BDD1) There is a total order over the set of identifiers  $x_1 < x_2 < \dots < x_n$ , such that the labels along every path are conforming to this order;
- (BDD2) There are no duplicate nodes (terminal or non-terminal); duplicates are merged into one single copy of all such nodes;
- (BDD3) There are no redundant nodes;



**Figure 2.5:** The effect of variable ordering over the BDD size

The above restrictions also give a simple algorithm that transforms an initial binary tree into a reduced and ordered binary decision diagram (ROBDD), which is a canonical form for representing functions. In Figure 2.4, the final stage (e) is a ROBDD.

If restriction (BDD3) is dropped from the requirement list and replaced with the constraint (BDD3') that all arcs connect nodes on consecutive levels, the decision diagrams are called *quasi-reduced* (QOBDD) [98]. In our example this corresponds to the intermediate step (d). It is easy to see that this form is also canonical. QOBDDs may contain redundant nodes (which are additional nodes compared to the ROBDD form). On the other hand QOBDDs possess a handy property: every node has its successors labeled with the same variable name, hence the whole diagram is structured in  $n$  “levels”, with arcs connecting only nodes on adjacent levels. In the reduced version, arcs may skip levels.

The step from storing functions to storing sets is straightforward, as the characteristic function of a set  $\mathcal{U} \subseteq \{0, 1\}^n$  is an  $n$ -ary boolean function:  $\chi_{\mathcal{U}}(x) = 1$  if and only if  $x \in \mathcal{U}$ . Most of our discussion on decision diagrams will be concerned with representing sets.

The **variable ordering** is a critical factor in reducing the size of a BDD. A compelling case is the class of  $n$ -bit-comparators. For  $n = 2$ , the analytical form is  $f(x_1, x_2, y_1, y_2) =$

$(\overline{x_1} \cdot \overline{y_1} + x_1 \cdot y_1)(\overline{x_2} \cdot \overline{y_2} + x_2 \cdot y_2)$ . As shown in Figure 2.5, for the ordering  $x_1 < x_2 < y_1 < y_2$  (a) the BDD has nine non-terminal nodes, while for the ordering  $x_1 < y_1 < x_2 < y_2$  (b) the number of non-terminals is just six. In fact, for the general case with  $n$  bits, the first ordering produces  $3(2^n - 1)$  non-terminals and the second  $3n$ , a logarithmic reduction in size.

Finding an optimal variable ordering is an intractable problem. Even checking that a particular ordering is optimal is NP-complete [3]. Furthermore, certain classes of functions have exponential size for *any* variable ordering [17].

The only solution in such situations is to look for a “good” ordering. The intuition behind most heuristics is that variables that depend on each other should be placed in close vicinity [62, 63]. When no obvious rule can be assessed about a good ordering, a *dynamic reordering* technique [124] which periodically rearranges the variables to reduce the number of nodes can give reasonably good results.

**Operations with BDDs.** We next explain how to perform various logical operations using BDDs. The basis for combining two BDDs is the *restriction* operator: in an  $n$ -ary boolean function  $f$ , if the variable  $x_i$  is bound (restricted) to the constant  $b \in \{0, 1\}$ , we obtain

$$f|_{x_i \leftarrow b}(x_1, \dots, x_n) = f(x_1, \dots, x_{i-1}, b, x_{i+1}, \dots, x_n).$$

Given the ROBDD for  $f$ , the ROBDD for the restriction can be obtained by redirecting all the arcs that point to nodes labeled  $x_i$  to the  $b$ -descendant of that node. Then, step (BDD3) of the definition is performed to eliminate any newly introduced redundant nodes.

The resulting BDD has one variable less than the original, as  $x_i$  was eliminated.

The reverse operation is called (Shannon) *expansion*:

$$f = (\bar{x} \cdot f|_{x \leftarrow 0}) + (x \cdot f|_{x \leftarrow 1}).$$

Based on this divide-and-conquer-like technique, one can write a recursive algorithm to build the BDD for any binary operation  $f * g$  [16], as listed in Figure 2.6. Note that, for QOBDDs, the test  $x = y$  in line 7 holds in every recursive call for BDDs of the same height defined over the same ordered set of variables.

<i>BDDapply</i> (* : operator, <i>f</i> : bdd, <i>g</i> : bdd) : bdd
The recursive algorithm to compute a generic $f * g$
<ol style="list-style-type: none"> <li>1. let <math>x</math> and <math>y</math> be the variables corresponding to the roots of <math>f</math> and <math>g</math></li> <li>2. if both <math>x</math> and <math>y</math> represent terminal nodes</li> <li>3. return <math>val(x) * val(y)</math></li> <li>4. else</li> <li>5. <math>f_0 \leftarrow f _{x \leftarrow 0}; f_1 \leftarrow f _{x \leftarrow 1};</math></li> <li>6. <math>g_0 \leftarrow g _{y \leftarrow 0}; g_1 \leftarrow g _{y \leftarrow 1};</math></li> <li>7. if <math>x = y</math> then</li> <li>8. return <math>\bar{x} \cdot BDDapply(*, f_0, g_0) + x \cdot BDDapply(*, f_1, g_1);</math></li> <li>9. else if <math>x &lt; y</math></li> <li>10. return <math>\bar{x} \cdot BDDapply(*, f_0, g) + x \cdot BDDapply(*, f_1, g);</math></li> <li>11. else</li> <li>12. return <math>\bar{y} \cdot BDDapply(*, f, g_0) + y \cdot BDDapply(*, f, g_1);</math></li> </ol>

**Figure 2.6:** The algorithm for binary operations on BDDs.

Since each call to *BDDapply* may generate two other recursive calls, the algorithm as shown has exponential time complexity. By using an auxiliary data structure, called *cache* to store all the answers to already computed subproblems, the complexity is reduced to polynomial time. Indeed, the number of distinct pairs that can be formed with roots of

subdiagrams from  $f$  and  $g$  is the product of the number of nodes in each BDD [17].

**Extensions.** There exists a multitude of BDD generalizations [56]:

- By using decomposition rules other than Shannon's to obtain Ordered Functional Decision Diagrams (OFDDs): examples are the positive and negative Davio decomposition [52]. All the three rules (the two Davio and Shannon) may then be combined in the Kronecker functional decision diagrams (OKFDDs), which allow different decompositions at different nodes. A canonical form of OKFDDs can be obtained by fixing the rule for each variable.
- By relaxing the ordering condition: these are the Free BDDs.
- By representing arithmetic functions instead of boolean functions:

Multi-terminal BDDs (MTBDDs) [8, 39], are extensions of BDDs used to represent binary functions of the type  $f : \{0, 1\}^K \rightarrow \{0, 1, \dots, m - 1\}$ . MTBDDs have  $m$  terminal nodes, corresponding to the  $m$  possible values of the function. This definition can be further extended to arithmetic functions, where each variable may take a finite set of integer values, instead of binary values:  $f : \{0, \dots, n_1 - 1\} \times \{0, \dots, n_2 - 1\} \dots \times \{0, \dots, n_K - 1\} \rightarrow \{0, \dots, m - 1\}$ . The resulting form is called Multi-way Decision Diagrams (MDDs). They are presented in detail in the next section.

- By introducing edge labeling, to obtain the Edge Valued BDDs (EVBDDs) [94]: in this category also fall the Binary Moment Diagrams (BMD), Multiplicative BMDs (\*BMDs) [18], and Zero Supressed BDDs (ZBDDs) [83].



### 2.6.1.2 Multi-way decision diagrams

The very first symbolic model checker, called SMV [104], was successfully employed in the verification of digital circuits, as BDDs are suitable for modeling this type of systems. Signals are naturally represented by boolean variables and the combination of gates in the circuit can be encoded as a large boolean function.

However, for arbitrary asynchronous systems, BDDs have many difficulties especially in representing integer functions. A variable  $x_k$  that takes  $n_k$  values, can be encoded using  $\lceil \log_2 n_k \rceil$  binary variables. However, the height of the diagram may become very large, and therefore hampers the efficiency of diagram manipulation, as the complexity of BDD operations depends on the number of BDD nodes.

One way to cope with this problem is to use Multi-way Decision Diagrams to efficiently encode functions of the type

$$f : \mathcal{S}_K \times \dots \mathcal{S}_2 \times \mathcal{S}_1 \rightarrow \{0, 1\}$$

where  $\mathcal{S}_k = \{0, \dots, n_k - 1\}$ , for  $1 \leq k \leq K$ .

Originally introduced in [87], MDDs are natural extensions of BDDs, more suitable for modeling complex asynchronous systems, since they allow discrete variables to take values over arbitrary finite sets instead of requiring boolean variables. The quasi-reduced version of MDDs we now define is a natural combination of the multi-way extension with the quasi-reduced canonical format introduced for BDDs in [90].

Formally:

**Definition 2.6.1** *A  $K$ -variable quasi-reduced ordered multi-way decision diagram, or MDD*

for short, is a directed acyclic edge-labeled multi-graph where:

- Nodes are organized into  $K + 1$  levels. We write  $\langle k|p \rangle$  to denote a node at level  $k$ , where  $p$  is a unique index for that level.
- Level  $K$  contains a single non-terminal node  $\langle K|r \rangle$ , the root, whereas levels  $K - 1$  through 1 contain one or more non-terminal nodes.
- Level 0 consists of two terminal nodes,  $\langle 0|0 \rangle$  and  $\langle 0|1 \rangle$ .
- A non-terminal node  $\langle k|p \rangle$  has  $n_k$  arcs pointing to nodes at level  $k - 1$ . If the  $i^{\text{th}}$  arc, for  $i \in S_k$ , is to node  $\langle k - 1|q \rangle$ , we write  $\langle k|p \rangle[i] = q$ . Duplicate nodes are not allowed but, unlike reduced ordered MDDs, redundant nodes where all arcs point to the same node are allowed (both versions are canonical).

Notice the inverse indexing of levels, from  $K$  down to 1. This is justified by the convention that the terminal nodes should always be at level 0.

Given the above definition, we can extend our arc notation to paths. The index of the node at level  $l - 1$  reached from a node  $\langle k|p \rangle$  through a sequence  $(i_k, \dots, i_l) \in S_k \times \dots \times S_l$ , for  $K \geq k > l \geq 1$ , is recursively defined as

$$\langle k|p \rangle[i_k, i_{k-1}, \dots, i_l] = \langle k - 1 | \langle k|p \rangle[i_k] \rangle[i_{k-1}, \dots, i_l].$$

The substates encoded by, or “below”,  $\langle k|p \rangle$  are then

$$\mathcal{B}(\langle k|p \rangle) = \{\beta \in S_k \times \dots \times S_1 : \langle k|p \rangle[\beta] = 1\}$$

and those reaching, or “above”,  $\langle k|p \rangle$  are

$$\mathcal{A}(\langle k|p \rangle) = \{\alpha \in \mathcal{S}_K \times \cdots \times \mathcal{S}_{k+1} : \langle K|r \rangle[\alpha] = p\}.$$

For any node  $\langle k|p \rangle$  on a path from the root  $\langle K|r \rangle$  to  $\langle 0|1 \rangle$ , the set of (global) states  $\mathcal{B}(\langle k|p \rangle) \times \mathcal{A}(\langle k|p \rangle)$  is a subset of the set of (global) states encoded by the MDD, which we can write as  $\mathcal{B}(\langle K|r \rangle)$  or  $\mathcal{A}(\langle 0|1 \rangle)$ .

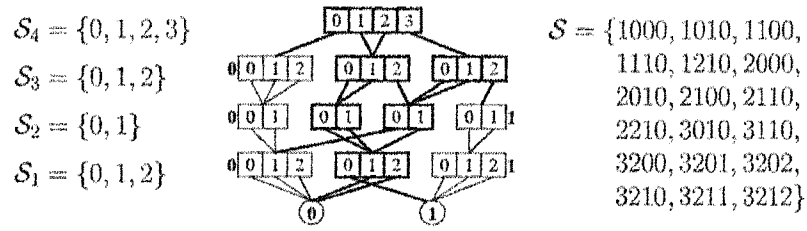


Figure 2.7: An example MDD.

We reserve the indices 0 and 1 at each level  $k$  to encode the sets  $\emptyset$  and  $\mathcal{S}_k \times \cdots \times \mathcal{S}_1$ , respectively. Such nodes do not need to be explicitly stored; this saves not only a small amount of memory, but also substantial execution time in the recursive manipulation algorithms. Figure 2.7 shows a four-variable MDD and the set  $\mathcal{S}$  it encodes (the node indices are in boldface and have been chosen arbitrarily, except for those with value 0 and 1). To see whether state  $(1,2,1,0)$  is encoded by this MDD, begin with the root node  $\langle 4|5 \rangle$  and follow the path:  $\langle 4|5 \rangle[1] = 4$ ,  $\langle 3|4 \rangle[2] = 7$ ,  $\langle 2|7 \rangle[1] = 4$ , and  $\langle 1|4 \rangle[0] = 1$ : since we reach the terminal node 1, the state is indeed encoded, i.e.,  $(1,2,1,0) \in \mathcal{B}(\langle 4|5 \rangle)$ . On the other hand, we can determine in a single operation that no state of the form  $(0, i_3, i_2, i_1)$  is encoded by the MDD, since  $\langle 4|5 \rangle[0] = 0$  and, by convention, node  $\langle 3|0 \rangle$  is known to encode the empty set.

To illustrate how using quasi-reduced MDDs simplifies the manipulation algorithms, and how our convention on the meaning of the indices 0 and 1 improves efficiency, we show the pseudocode for performing a union operation on two MDDs, in Figure 2.8. We use the types *level*, *index*, and *local* for MDD levels, MDD node indices within a level, and local state indices, respectively. Function *NewNode*( $k$ ) allocates a new MDD node  $\langle k|x \rangle$  at level  $k$  with all arcs  $\langle k|x \rangle[i] = 0$ , for all  $i \in \mathcal{S}_k$ . *SetArc*( $k, p, i, y$ ) sets  $\langle k|p \rangle[i]$  to  $y$ ; *CheckIn*( $k, p$ ) searches a *unique table* at level  $k$  for a node  $\langle k|q \rangle$  with the same  $n_k$  arc values as  $\langle k|p \rangle$ ; if it finds such a node, it deletes the duplicate node  $\langle k|p \rangle$  and returns  $q$ , otherwise it inserts  $\langle k|p \rangle$  in the table and returns  $p$ . *Cached*(*UNION*,  $k, p, q, u$ ) searches an *operation cache* at level  $k$  using the hash key “*UNION* (an integer code),  $p, q$ ”; if it finds the key, it sets  $u$  to the associated cached value and returns *true*, otherwise it returns *false*. *PutInCache*(*UNION*,  $k, p, q, u$ ) associates the value  $u$  to the key “*UNION*,  $k, p, q$ ” in the level- $k$  operation cache.

## 2.6.2 MDDs and structured systems

In an unstructured system, the states are opaque and no internal information is given about their properties, only their inter-dependencies via the transition relation. Structured systems, on the other hand, give more detailed information about the system states and their internal properties, such as variables types, ranges, and other functional dependencies between them. This information can be exploited in devising techniques to better store and manipulate large sets of states [108].

A common characteristic of the high-level models we consider is that they can be partitioned, or *structured*, into interacting *submodels*. For a model composed of  $K$  submodels, a *global* system state  $\mathbf{i}$  can be written as a  $K$ -tuple  $(\mathbf{i}_K, \dots, \mathbf{i}_1)$ , where  $\mathbf{i}_k$  is the *local* state of

<i>Union</i> ( <i>k</i> : <i>level</i> , <i>p</i> : <i>index</i> , <i>q</i> : <i>index</i> ) : <i>index</i>	
Return an index <i>u</i> such that $\mathcal{B}(\langle k u \rangle) = \mathcal{B}(\langle k p \rangle) \cup \mathcal{B}(\langle k q \rangle)$ .	
1. declare <i>u</i> : <i>index</i> , <i>i</i> : <i>local</i> ;	
2. if <i>p</i> = <i>q</i>	• $\mathcal{B}(\langle k p \rangle) \cup \mathcal{B}(\langle k p \rangle) = \mathcal{B}(\langle k p \rangle)$
3. return <i>p</i> ;	
4. else if <i>p</i> = 1 or <i>q</i> = 1	• exploit special meaning of $\langle k 1 \rangle$
5. return 1;	
6. else if <i>p</i> = 0	• exploit special meaning of $\langle k 0 \rangle$
7. return <i>q</i> ;	
8. else if <i>q</i> = 0	• exploit special meaning of $\langle k 0 \rangle$
9. return <i>p</i> ;	
10. else if <i>k</i> = 0	• terminal case, <i>p</i> and <i>q</i> are booleans
11. return <i>p</i> ∨ <i>q</i> ;	
12. if <i>Cached</i> ( <i>UNION</i> , <i>k</i> , <i>p</i> , <i>q</i> , <i>u</i> )	• don't redo work
13. return <i>u</i> ;	
14. <i>u</i> ← <i>SingleNode</i> ( <i>k</i> );	• create a new node at level <i>k</i>
15. for <i>i</i> = 0 to <i>n<sub>k</sub></i> − 1	
16. <i>SetArc</i> ( <i>k</i> , <i>u</i> , <i>i</i> , <i>Union</i> ( <i>k</i> − 1, $\langle k p \rangle[i]$ , $\langle k q \rangle[i]$ ));	
17. <i>u</i> ← <i>CheckIn</i> ( <i>k</i> , <i>u</i> );	• insert $\langle k u \rangle$ in unique table
18. <i>PutInCache</i> ( <i>UNION</i> , <i>k</i> , <i>p</i> , <i>q</i> , <i>u</i> );	• remember result
19. return <i>u</i> ;	

Figure 2.8: The algorithm for the union of MDDs.

submodel *k*, for  $K \geq k \geq 1$ . Thus, the potential state space  $\hat{\mathcal{S}}$  is given by the cross-product  $\mathcal{S}_K \times \cdots \times \mathcal{S}_1$  of *K* local state spaces, where  $\mathcal{S}_k$  contains (at least) all the possible local states for submodel *k*. For example, the places of a Petri net can be partitioned into *K* subsets, so that the marking can be written as a vector of the *K* corresponding submarkings. In software modeling, a variable can be considered a component, and its range is one of the sets  $\mathcal{S}_k$ . In both circumstances, the one-to-one correspondence between submodels and MDD levels is straightforward, and the partitioning of the system allows us to exploit the system structure by means of new and improved algorithms.

Returning to the issue of indexing, just as each reachable global state *i* is mapped to a natural number *i* in the explicit approach, one can map local states as well. Assuming for

now that each local state space  $\mathcal{S}_k$  is known a priori, i.e., can be built considering the  $k^{\text{th}}$  submodel in isolation, we define  $K$  mappings  $\psi_k : \mathcal{S}_k \rightarrow \{0, 1, \dots, n_k - 1\}$ , where  $n_k$  is the size of the local state space  $\mathcal{S}_k$ . Then, a global structured state  $(\mathbf{i}_K, \dots, \mathbf{i}_1)$  can be mapped to a vector  $(i_K, \dots, i_1)$  of indices.

### 2.6.3 Symbolic encoding of next-state functions

The next-state function  $\mathcal{N}$  of a model can also be encoded using a decision diagram, since a binary relation can be viewed as a *set of pairs* of states. Therefore, for representing transitions between states, twice as many variables as the decision diagram encoding the states are needed. In the case of MDDs, this means encoding the characteristic function of the transition relation with a  $2K$ -variable MDD. Rather than numbering the variables from  $2K$  down to 1, we distinguish them as “from” and “to” variables, and use unprimed and primed integers for “from” and “to” levels, respectively. Also, we prefer a double-bracket notation to distinguish these nodes from those of  $K$ -variable MDDs encoding sets of states. Thus,  $\langle\langle 5'|p \rangle\rangle$  is a “to” node associated with the  $5^{\text{th}}$  submodel, and  $\langle\langle 5'|p \rangle\rangle[j_5] = q$  means that the arc labeled with local state index  $j_5$  from this node points to  $\langle\langle 4|q \rangle\rangle$ , a “from” node associated with the  $4^{\text{th}}$  submodel.

If the MDD encoding  $\mathcal{N}$  is rooted at  $\langle\langle K|t \rangle\rangle$ , we have

$$(\mathbf{i}_K, \mathbf{i}'_K, \dots, \mathbf{i}_1, \mathbf{i}'_1) \in \mathcal{B}(\langle\langle K|t \rangle\rangle) \Leftrightarrow (\mathbf{i}'_K, \dots, \mathbf{i}'_1) \in \mathcal{N}(\mathbf{i}_K, \dots, \mathbf{i}_1).$$

In principle, any permutation of the  $2K$  variables could be used, but most common is that the variables are *interleaved*, so that the “from” and “to” variables for a given level

<i>Image</i> ( $k : \text{level}, p : \text{index}, q : \text{index}$ ) : <i>index</i>	
Computes $\mathcal{N}_{\langle\langle k q \rangle\rangle}(B(\langle k p \rangle))$	
1. declare $r, s, t : \text{index}, i : \text{local}$	
2. if $p = 0 \vee q = 0$	• states not encoded or no event enabled
3. return 0	• return empty set
4. if $k = 0$	• reached terminal node
5. return $q$	
6. $r \leftarrow \text{NewNode}(k)$ ;	
7. foreach $i \in \{0 \dots n_k - 1\}$	
8. $s \leftarrow \langle\langle k q \rangle\rangle[i]$	• follow the “from” pattern
9. foreach $j \in \{0 \dots n_k - 1\}$	
10. $t \leftarrow \text{Image}(k - 1, \langle k p \rangle[i], \langle\langle k' s \rangle\rangle[j])$	• recursive call downstream
11. $\text{SetArc}(k, r, j, \text{Union}(\langle k r \rangle[j], t))$	• accumulate new states in $\langle k r \rangle$
12. $r \leftarrow \text{CheckIn}(K, r)$ ;	
13. return $r$	

Figure 2.9: Image computation with MDDs.

are next to each other<sup>1</sup>. There are at least two good reasons for this. First, it is widely acknowledged that interleaving is often the most efficient order in terms of nodes required to store the next-state function. Second, the symbolic state-space generation algorithms we consider next can be easily expressed using recursion if interleaving is used. Indeed, a very important third reason related to the above is that traditional symbolic approaches to represent  $\mathcal{N}$  fail to exploit the large number of identity transformations (i.e.,  $i_k = i'_k$ ) present in the class of systems we address.

In order to construct a complete BDD-based approach for state-space construction, the only missing piece is an algorithm to symbolically compute the next state function on sets of states. This is traditionally referred to in the literature as *image computation*.

The recursive algorithm in Figure 2.9 takes at input a node at level  $k$  in the  $K$ -level

<sup>1</sup>We assume that level  $k$  is above level  $k'$ , since we store the *forward* next-state function  $\mathcal{N}$ . However, symbolic model-checking [21] also makes use of the *backward* or *previous-state* function  $\mathcal{N}^{-1}$  and, in this case, we would have level  $k'$  above level  $k$ .

MDD encoding the “from” states,  $\langle k|p\rangle$ , a corresponding node  $\langle\langle k|q\rangle\rangle$  at level  $2k$  in the  $2K$ -level MDD encoding the next state function, and combines them to extract from the  $2K$ -level MDD the corresponding “to” states. The function is recursive and at each step it descends one level in the “from” MDD synchronized with a descent of two levels in the  $2K$ -level MDD. During a call, the “from” states pattern is matched against the interleaved encoding of  $\mathcal{N}$ , and all the results of executing the transitions are collected in the node  $\langle k|r\rangle$ . An  $Image(K, p, q)$  call at level  $K$  applies the global next-state function to  $\mathcal{B}(\langle K|p\rangle)$ , while a call at a lower level,  $k < K$ , computes the restriction of the next-state function to the subdomain encoded by the sub-diagram rooted at  $\langle\langle k|q\rangle\rangle$  (for brevity, this is denoted as  $\mathcal{N}_{\langle\langle k|q\rangle\rangle}$ ).

For asynchronous systems, it is often advantageous to store the next-state function as a *disjunction* [20]. If the events  $\mathcal{E}$  are implied by the high-level model, as is clearly the case for Petri nets, where each *transition*  $e$  in the net corresponds to exactly one event in  $\mathcal{E}$ , we simply need to store each  $\mathcal{N}_e$  as a separate  $2K$ -variable MDD (of course, an MDD *forest* should be used, so that these MDDs can share common nodes [117]).



## Chapter 3

# The Road to Saturation

In this chapter we introduce our new state-space exploration strategy, *saturation*. This came at the end of a series of improvements on the traditional breadth-first algorithm, which is still used in most modern model checkers. We present these improvements in chronological order, starting with already existing heuristics in Section 3.1 and continuing with our contributions in the remainder of the chapter.

### 3.1 Breadth-first symbolic state-space generation

The simplest and most widely used symbolic state-space generation algorithm is shown at the top of Figure 3.1. An alternative algorithm, shown at the bottom of the same figure, is conceptually simpler and closer to the definition of the state space  $\mathcal{S}$  as a fixed-point. Both versions will halt after exactly as many steps as the maximum distance of any state from the initial state  $s$ . The main computational difference is the cost of applying the next-state function; the traditional algorithm applies  $\mathcal{N}$  *only to the unexplored states*  $\mathcal{U}$  which, at iteration  $d$ , consists of all states at distance *exactly*  $d$  from  $s$ , while the alternative algorithm applies  $\mathcal{N}$  to *all known states*, that is, all states at distance *at most*  $d$  from  $s$ .

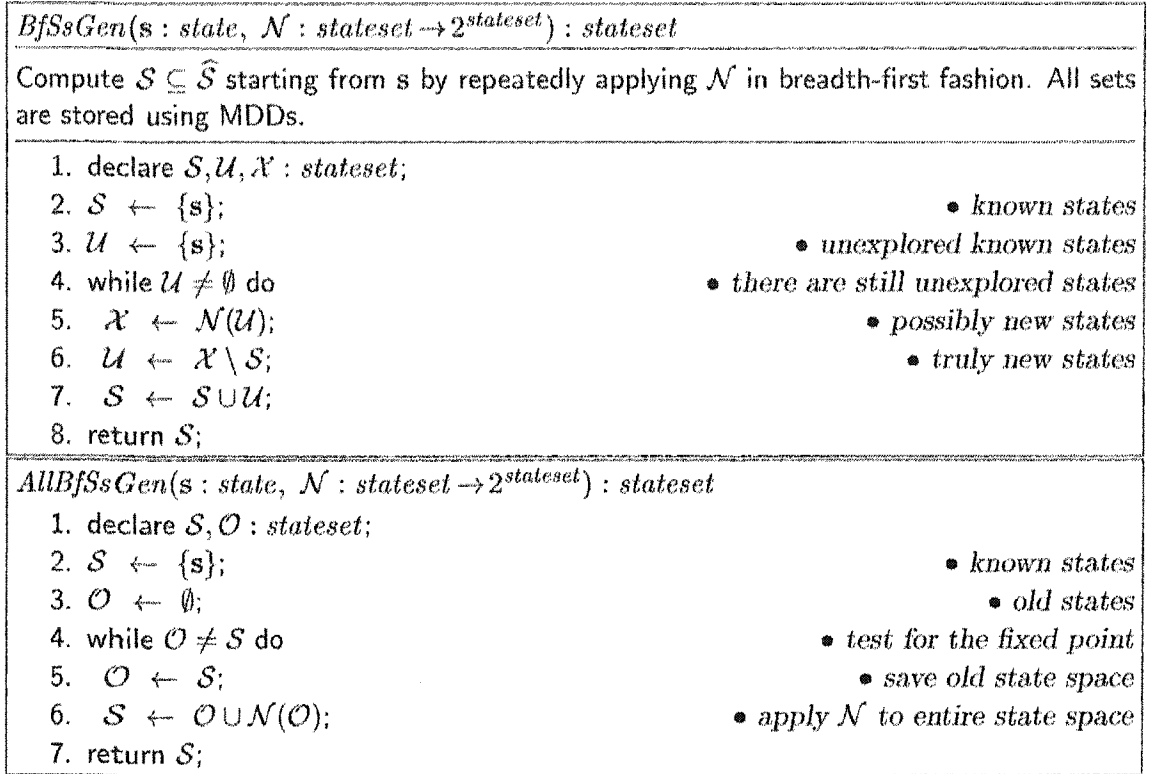


Figure 3.1: Two symbolic breadth-first generation algorithms.

While it would be wasteful to apply the next-state function to the same state more than once in an explicit setting, the cost of applying  $\mathcal{N}$  to a set of states encoded as an MDD in a symbolic setting depends on the *number of nodes in the MDD*, not on the *number of states encoded by the MDD*, so it is not clear that the traditional algorithm is better, nor why the alternative version appears to have been neglected in the literature. Indeed, Section 3.9 reports experimental evidence suggesting that the set  $\mathcal{U}$  of states having exactly distance  $d$  is often not a “nice” set to be encoded as an MDD. Furthermore, the traditional version of the breadth-first generation algorithm incurs the additional cost of the set-difference operation required to compute the “truly new states”  $\mathcal{X} \setminus \mathcal{S}$ . However, our main motivation for discussing *AllBfSsGen* is that it introduces a different way of applying the next-state function, i.e., one that does not distinguish between new and old states. This is one of the

<i>ChSsGen</i> ( $s : \text{state}, \mathcal{N}_{a \in \mathcal{E}} : \text{stateset} \rightarrow 2^{\text{stateset}} : \text{stateset}$ )	
Compute $\mathcal{S} \subseteq \hat{\mathcal{S}}$ starting from $s$ by incrementally applying each $\mathcal{N}_e$ . All sets are stored using MDDs.	
1. declare $\mathcal{S}, \mathcal{U} : \text{stateset};$	
2. $\mathcal{S} \leftarrow \{s\};$	• known states
3. $\mathcal{U} \leftarrow \{s\};$	• unexplored known states
4. while $\mathcal{U} \neq \emptyset$ do	• there are still unexplored states
5. for each $e \in \mathcal{E}$ do	
6. $\mathcal{U} \leftarrow \mathcal{U} \cup \mathcal{N}_e(\mathcal{U});$	• add to possibly new states
7. $\mathcal{U} \leftarrow \mathcal{U} \setminus \mathcal{S};$	• truly new states
8. $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{U};$	
9. return $\mathcal{S};$	
<i>AllChSsGen</i> ( $s : \text{state}, \mathcal{N}_{a \in \mathcal{E}} : \text{stateset} \rightarrow 2^{\text{stateset}} : \text{stateset}$ )	
1. declare $\mathcal{S}, \mathcal{O} : \text{stateset};$	
2. $\mathcal{S} \leftarrow \{s\};$	• known states
3. $\mathcal{O} \leftarrow \emptyset;$	• old states
4. while $\mathcal{O} \neq \mathcal{S}$ do	• test for the fixed point
5. $\mathcal{O} \leftarrow \mathcal{S};$	• save old state space
6. for each $e \in \mathcal{E}$ do	
7. $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{N}_e(\mathcal{S});$	• apply $\mathcal{N}_e$ to entire state space
8. return $\mathcal{S};$	

Figure 3.2: Algorithm for symbolic breadth-first search with chaining.

characteristics of the algorithms we introduce in Section 3.6.

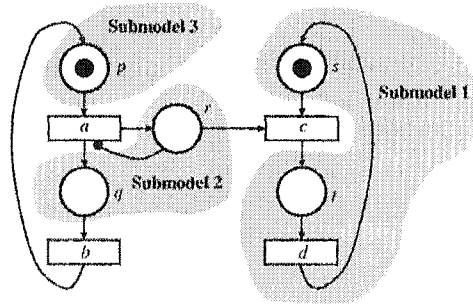
A second variation of symbolic breadth-first state-space generation, which introduces the idea of *chaining* [123], is shown at the top of Figure 3.2. Its key advantage lies in that the states in  $\mathcal{N}_e(\mathcal{U})$ , i.e., the (possibly) new states reachable in one firing of event  $e$  from the currently unexplored states  $\mathcal{U}$ , are added to  $\mathcal{U}$  right away, instead of only after having explored each event in isolation from the same original  $\mathcal{U}$ . In other words, if  $\mathbf{j}^{(1)} \in \mathcal{N}_{e^{(1)}}(\mathbf{i})$ ,  $\mathbf{j}^{(2)} \in \mathcal{N}_{e^{(2)}}(\mathbf{j}^{(1)})$ ,  $\dots$ ,  $\mathbf{j}^{(d)} \in \mathcal{N}_{e^{(d)}}(\mathbf{j}^{(d-1)})$ , and if the for-loop considers the events in the order  $e^{(1)}, e^{(2)}, \dots, e^{(d)}$ , *ChSsGen* finds all the states in this path in a single iteration, while both *BfSsGen* and *AllBfSsGen* would require  $d$  iterations. Note that statement 6 in *ChSsGen*, just like statement 6 in *AllBfSsGen*, applies the next-state function to the same states more

than once. However, it does so only to states in the set  $\mathcal{U}$  being built. The difference between explored and unexplored states is still kept in the outer while-loop. An algorithm merging the ideas in *AllBfSsGen* and *ChSsGen* is shown at the bottom of Figure 3.2. We believe that this is the most straightforward state-space generation algorithm when  $\mathcal{N}$  is disjunctively partitioned into events, and the results of Section 3.9 show that its performance is competitive with respect to that of *BfSsGen*, *AllBfSsGen*, and *ChSsGen*.

*ChSsGen* and *AllChSsGen* are fundamentally different from *BfSsGen* and *AllBfSsGen* in one way: *they do not find states in strict breadth-first order*. This is a second characteristic of the algorithms we introduce in Section 3.6, where we also discuss the importance of the order in which events are considered.

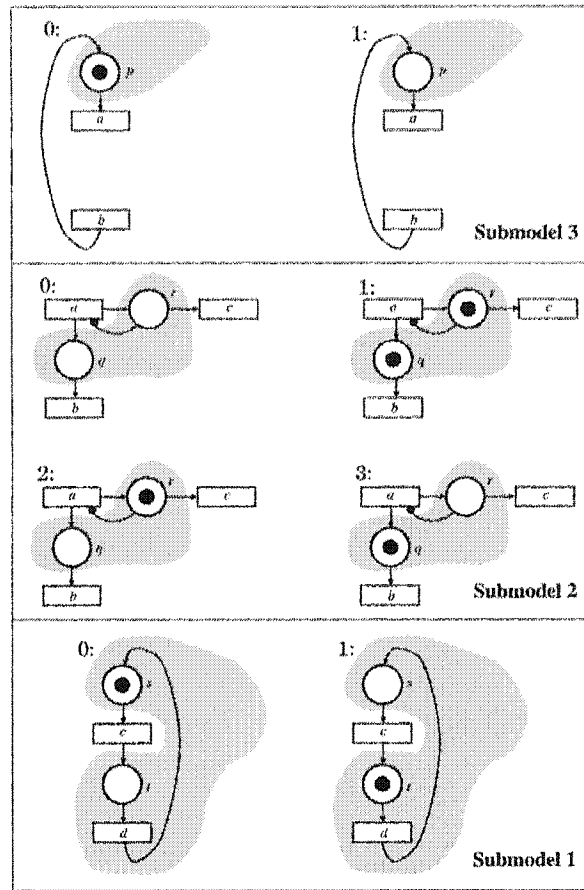
This concludes the summary of traditional techniques. We proceed with the presentation of our contributions, not before introducing a running example that will serve to the better understanding of our techniques.

### Running example: a producer-consumer model



**Figure 3.3:** A producer-consumer model and its submodels.

As a running example, consider Figure 3.3, showing a simplified producer-consumer system represented as a Petri net. This model illustrates the concept of mutual exclusion



**Figure 3.4:** The local states of submodels in the producer-consumer model.

on a shared resource, stored in place  $r$ . The producer creates a resource, by firing event  $a$ , when the place  $p$  holds a token and place  $r$  is empty (this is enforced by the inhibitor arc from place  $r$  to transition  $a$ ). If the consumer has not used an existing resource yet, transition  $a$  cannot fire. After creating a resource, the producer enters the idle state, with a token in place  $q$ . It can later go back to the ready state, by firing event  $b$ , and this can occur independent of the status of the consumer. The consumer is ready to get the resource when there is a token in place  $s$ . If there is an available token in place  $r$ , it is removed by firing transition  $c$ . Thus, the consumer releases the access to  $r$  and enters the idle state,

with a token in place  $t$ . The process may continue indefinitely.

The model has eight reachable (global) states:

$$\begin{aligned} \mathcal{S} = \{ & (p^1 q^0 r^0 s^1 t^0), (p^1 q^0 r^0 s^0 t^1), (p^0 q^1 r^1 s^1 t^0), (p^0 q^1 r^1 s^0 t^1), \\ & (p^1 q^0 r^1 s^1 t^0), (p^1 q^0 r^1 s^0 t^1), (p^0 q^1 r^0 s^1 t^0), (p^0 q^1 r^0 s^0 t^1) \}. \end{aligned}$$

We decompose the model into the three submodels shown in Figure 3.3, each corresponding to exactly one non-terminal level of an MDD. The reachable local states of the example are shown in Figure 3.4. Submodel 1 has two local states, submodel 2 has four, and submodel 3 has two.

### 3.2 Kronecker encoding of the next-state function

The encoding of sets of states using MDDs instead of BDDs has no immediate advantage in itself, since the two data structures are very similar, except for the number of levels. However, the advantages of MDDs become apparent when we consider our encoding of the next-state function and how this encoding is exploited in fixed-point computations, which we discuss next.

We adopt a representation of  $\mathcal{N}$  inspired by work on Markov chains, where the infinitesimal generator matrix of a large continuous-time Markov chain is encoded through a *Kronecker algebra expression* [6] on a set of small matrices [19, 115]. In our setting, this corresponds to a two-dimensional decomposition of  $\mathcal{N}$ : by events and by submodels. This is possible when the decomposition of the model into submodels is *Kronecker consistent*, that is, when we can find  $K \cdot |\mathcal{E}|$  functions  $\mathcal{N}_{k,e} : \mathcal{S}_k \rightarrow 2^{\mathcal{N}_k}$ , describing the (local) effect of

a particular event  $e$  on a particular subsystem  $k$ . Formally, this means that, for each event  $e \in \mathcal{E}$  and (global) state  $(i_K, \dots, i_1) \in \widehat{\mathcal{S}}$ ,

$$\mathcal{N}_e(i_K, \dots, i_1) = \mathcal{N}_{K,e}(i_K) \times \dots \times \mathcal{N}_{1,e}(i_1).$$

Thus, it must be possible to express the effect of firing an event  $e$  in a global state as the cross product of the local effects of  $e$  on each submodel.

The local next-state functions  $\mathcal{N}_{k,e}$  can be encoded as incidence matrices  $\mathbf{N}_{k,e} \in \{0, 1\}^{n_k \times n_k}$ , where

$$\mathbf{N}_{k,e}[i_k, j_k] = 1 \Leftrightarrow j_k \in \mathcal{N}_{k,e}(i_k).$$

Thus, event  $e$  is locally enabled in local state  $i \in \mathcal{S}_k$  of the  $k^{\text{th}}$  submodel,  $\mathcal{N}_{k,e}(i) \neq \emptyset$ , iff not all entries of the  $i^{\text{th}}$  row in  $\mathbf{N}_{k,e}$  are zero,  $\mathbf{N}_{k,e}[i, \cdot] \neq 0$ .

The overall next-state function  $\mathcal{N}$  is then encoded as the incidence matrix given by the (boolean) sum of Kronecker products

$$\mathbf{N} = \sum_{e \in \mathcal{E}} \bigotimes_{K \geq k \geq 1} \mathbf{N}_{k,e}$$

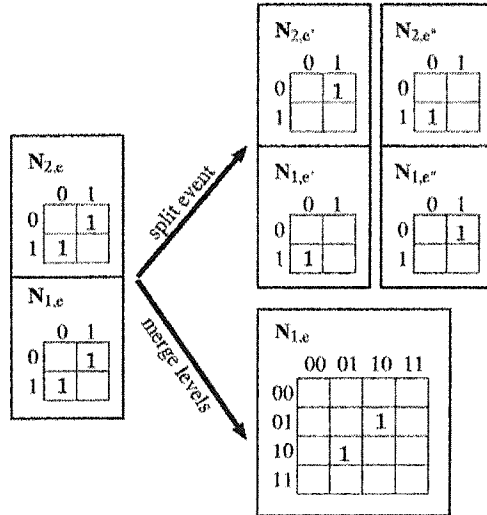
The Kronecker encoding of  $\mathcal{N}$  for the producer-consumer model is shown in Figure 3.5, as full matrices. In general, the  $\mathbf{N}_{k,e}$  matrices are extremely sparse (for Petri nets, each row contains at most one nonzero entry), and are indexed using the same mapping  $\psi_k$  used to index  $\mathcal{S}_k$ .

For some formalisms, such as ordinary Petri nets, the Kronecker consistency requirement is always satisfied, regardless of how the model is partitioned into submodels. In other

$N_{3,a} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$	$N_{3,b} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$	$N_{3,c} = I_2$	$N_{3,d} = I_2$
$N_{2,a} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$	$N_{2,b} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$	$N_{2,c} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$	$N_{2,d} = I_4$
$N_{1,a} = I_2$	$N_{1,b} = I_2$	$N_{1,c} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$	$N_{1,d} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$

**Figure 3.5:** Kronecker encoding of  $\mathcal{N}$  for the producer-consumer model.

formalisms, not all decompositions will be Kronecker consistent; however, we can always find a consistent decomposition by refining the events or coarsening the partition into submodels.

**Figure 3.6:** Satisfying Kronecker-consistency requirements.

Consider for example Figure 3.6, where a model consists of two boolean variables  $x_1$  and  $x_2$  and the partition assigns each variable to a different submodel. If an event  $e$  swaps the values of  $x_1$  and  $x_2$  but is enabled only when  $x_1 \neq x_2$ , the partition is not Kronecker consistent. If it were,  $\mathcal{N}_{2,e}$  and  $\mathcal{N}_{1,e}$  would have to satisfy  $1 \in \mathcal{N}_{2,e}(0)$  and  $1 \in \mathcal{N}_{1,e}(0)$ . However, Kronecker consistency would also imply that  $(1,1) \in \mathcal{N}_{2,e}(0) \times$



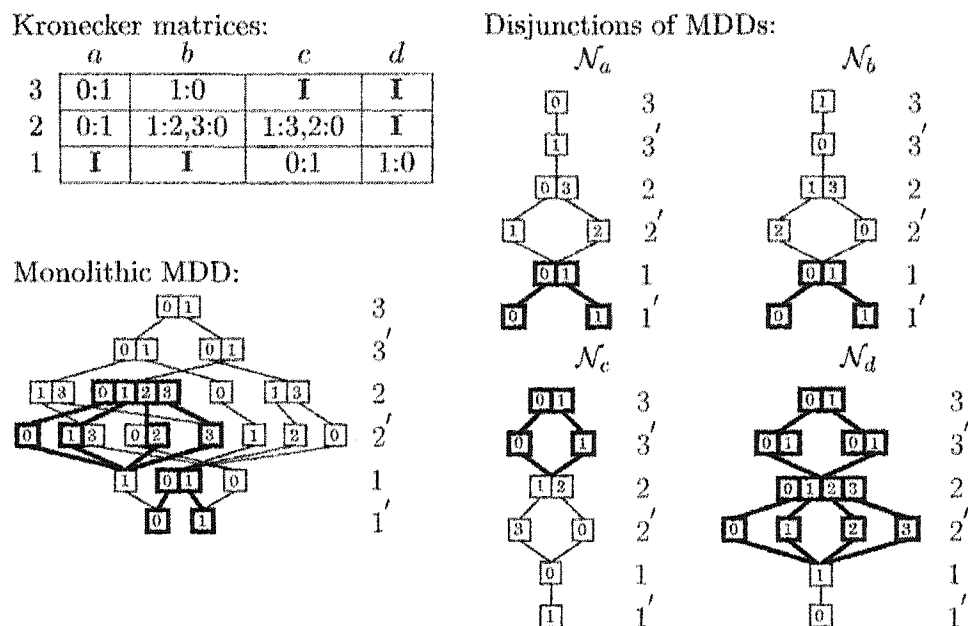
$\mathcal{N}_{1,e}(0) = \mathcal{N}_e(0,0)$ , while a transition from state  $(0,0)$  to state  $(1,1)$  should not be allowed. To achieve Kronecker consistency, one can then either split event  $e$  into  $e'$  and  $e''$ , which allows us to treat the two enabling states,  $(0,1)$  and  $(1,0)$ , separately, or merge the two levels into a single level with four possible local states.

### 3.3 An identity crisis

Event  $e$  is said to be *independent* of level  $k$  if  $\mathbf{N}_{k,e} = \mathbf{I}$ , the identity matrix, that is, if its occurrence is not affected by, nor it affects, the local state of the  $k^{\text{th}}$  submodel. Our approach neither stores nor processes these identity matrices. This results in large savings especially when dealing with asynchronous systems, where most of the  $\mathbf{N}_{k,e}$  matrices are identities.

On the other hand, a decision diagram representation of  $\mathcal{N}$  does not efficiently encode or exploit these identities. The variables (levels) not affected by an event  $e$  are still represented in the decision diagram by “identity patterns”. These are wasteful both in terms of memory, as they require  $n_k + 1$  nodes per variable along each corresponding path in the diagram, and in terms of time, as these additional nodes are processed during the image computation, just like any other node. The compactness of a BDD representation is characterized by the degree of merging of duplicate nodes and the number of redundant nodes that are eliminated and replaced with arcs skipping levels. However, when representing  $\mathcal{N}$  with a BDD, an arc that skips levels  $k$  and  $k'$  (the “from” and “to” variables corresponding to the  $k^{\text{th}}$  submodel, respectively) means that, after an event fires, the  $k^{\text{th}}$  component can be either 0 or 1, regardless of whether it was 0 or 1 before the firing. The more natural

behavior is instead the one where a 0 remains a 0 and a 1 remains a 1, the default in our Kronecker encoding.



**Figure 3.7:** Kronecker versus MDD encoding of the next state function.

Figure 3.7 shows Kronecker, monolithic MDD, and partitioned MDD representations of the next state function for our running example (for brevity, the matrices are in sparse notation and the MDD arcs pointing to 0 are also excluded).

For the Kronecker representation, the total number of nonzero entries is nine and five out of twelve Kronecker matrices are identities. This is not a remarkable ratio because the model is small. For many asynchronous systems, however, the number of events grows linearly in  $K$  but most, if not all, events are described by a constant number (independent of  $K$ ) of non-identity matrices. Thus, while the Kronecker description potentially requires  $O(K^2)$  matrices,  $O(K)$  matrices suffice in practice, which is a huge improvement. On the other hand, the monolithic MDD encoding of  $\mathcal{N}$ , requires 19 nodes, while the partitioned

MDD encoding requires 8 nodes each for  $\mathcal{N}_a$ ,  $\mathcal{N}_b$ , and  $\mathcal{N}_c$ , and 10 nodes for  $\mathcal{N}_d$  (in total, only  $8 + 8 + 8 + 10 - 3 = 31$  nodes are required, since the three nodes at levels 1 and 1' for  $\mathcal{N}_a$  and  $\mathcal{N}_b$  can be shared).

One reason for the large number of nodes in the MDD encoding is the explicit representation of identity transformations, corresponding to patterns where a “from” node  $\langle k|p \rangle$  with  $n_k$  distinct children  $\langle k'|q^{(i)} \rangle$ , such that  $\langle k'|q^{(0)} \rangle[0] = \dots = \langle k'|q^{(n_k-1)} \rangle[n_k - 1]$ , shown with bold lines in the figure. In the partitioned MDD case, these patterns are clearly visible because all other arcs  $\langle k'|q^{(i)} \rangle[j]$ , for  $j \neq i$ , have value 0. In the monolithic MDD case, some of these patterns are harder to recognize because the level- $k'$  nodes may have additional nonzero arcs, as is the case for the identity pattern at levels 2 and 2' in our figure, while other identity patterns are destroyed when performing the union of the next-state functions for multiple events, as is the case for the identity patterns at levels 3 and 3' of  $\mathcal{N}_c$  and  $\mathcal{N}_d$ . However, these transformations are still encoded in the MDD, and they can occupy a non-negligible amount of memory.

### 3.4 Event locality

In large globally-asynchronous locally-synchronous systems, most events are independent of most levels. Then, in addition to efficiently representing  $\mathcal{N}$ , the Kronecker encoding is also able to clearly evidence *event locality* [26, 107], that is, identify which events affect which levels.

Let  $Top(e)$  and  $Bot(e)$  denote the highest and lowest levels on which  $e$  depends:

$$Top(e) = \max\{k : \mathbf{N}_{k,e} \neq \mathbf{I}\}, \quad Bot(e) = \min\{k : \mathbf{N}_{k,e} \neq \mathbf{I}\}.$$

If events that are independent of all levels (hence do not affect the model's behavior) are ignored, these maximum and minimum levels exist, that is:

$$\forall e \in \mathcal{E}, \quad K \geq \text{Top}(e) \geq \text{Bot}(e) \geq 1.$$

We can then partition  $\mathcal{E}$  into  $K$  classes,

$$\mathcal{E}_k = \{e \in \mathcal{E} : \text{Top}(e) = k\}, \quad \text{for } K \geq k \geq 1$$

where some  $\mathcal{E}_k$  might be empty.

We can use this locality information to avoid unnecessary work [26]. When firing an event  $e$ , we do not have to start at the root of the MDD, since we know that all state components from  $K$  down to  $\text{Top}(e) + 1$  are not going to change anyway. Rather, we can “jump-in-the-middle” of the MDD by directly accessing each node at level  $k = \text{Top}(e)$  and fire  $e$  in it and, recursively, on its descendants, up to nodes at level  $l = \text{Bot}(e)$ . Of course, this requires that MDD nodes be organized and stored in such a way that provides easy access by level. Conceptually, for each node  $\langle k|p \rangle$ , we can compute a node  $\langle k|f \rangle$  encoding the effect of firing  $e$  in it:

$$B(\langle k|f \rangle) = \mathcal{N}_{k,e} \times \cdots \times \mathcal{N}_{l,e} \times \underbrace{\mathcal{N}_{l-1,e} \times \cdots \times \mathcal{N}_{1,e}}_{\text{identity functions}}(B(\langle k|p \rangle)).$$

Then, we can *substitute* node  $\langle k|p \rangle$  with a node  $\langle k|u \rangle$ , where  $u$  is the value returned by the call  $\text{Union}(k, p, f)$ , that is,

$$B(\langle k|u \rangle) = B(\langle k|p \rangle) \cup B(\langle k|f \rangle).$$

This is correct because, if  $\mathbf{i} \in \mathcal{S}$  and  $\mathbf{j} \in \mathcal{N}_e(\mathbf{i})$ , we can conclude that  $\mathbf{j} \in \mathcal{S}$  and, in particular, if  $\mathbf{i} \in \mathcal{A}(\langle k|p \rangle) \times \mathcal{B}(\langle k|p \rangle)$ , then  $\mathbf{j} \in \mathcal{A}(\langle k|p \rangle) \times \mathcal{B}(\langle k|f \rangle)$ , and substituting  $\langle k|p \rangle$  with  $\langle k|u \rangle$  will indeed add any such state  $\mathbf{j}$ .

The efficiency of jumps-in-the-middle is due to two factors. First, we avoid work in nodes at levels from  $K$  to  $k+1$ . Second, any algorithm that fires  $e$  starting from the MDD root would reach  $\langle k|p \rangle$  and attempt to fire  $e$  in it as many times as the number of incoming arcs pointing to  $\langle k|p \rangle$  from level  $k+1$ . Of course, all but the first time the effect of this firing is retrieved from the operation cache, but the cost of these cache lookups is not negligible. With jump-in-the-middle, instead, we only need to consider  $\langle k|p \rangle$  once.

One complication with this approach, however, is that substituting  $\langle k|p \rangle$  with  $\langle k|u \rangle$  requires us to *redirect* all incoming arcs so that they point to  $\langle k|u \rangle$  instead of  $\langle k|p \rangle$ . Thus, in [26], we pair jump-in-the-middle with another improvement, *in-place updates* of MDD nodes, which not only avoids the explicit computation of node  $\langle k|f \rangle$ , but also greatly reduces the need to redirect arcs from nodes at level  $k+1$ .

### 3.5 In-place updates

The idea of substituting  $\langle k|p \rangle$  with  $\langle k|u \rangle$  we just discussed can be further improved by realizing that, instead of computing the result of firing  $e$  in  $\langle k|p \rangle$  as a separate node  $\langle k|f \rangle$  and then computing the union node  $\langle k|u \rangle$ , we can incrementally *modify* the arcs of node  $\langle k|p \rangle$ , so that, when we are done, they point to the same nodes that the arcs of node  $\langle k|u \rangle$  would have pointed, had we have built it explicitly.

This is described in Figure 3.8, which provides the pseudocode for functions *Fire* and

<i>Fire</i> ( $e : \text{event}, k : \text{level}, p : \text{index}$ ) : <i>index</i>	
Fire $e$ in $\langle k p \rangle$ , where $k = \text{Top}(e)$ , using in-place updates. *When pipelining is used, return $x$ such that $*B(\langle k x \rangle) = N_{k,e}^*(B(\langle k p \rangle))$ .	
<ol style="list-style-type: none"> <li>1. declare <math>f, u : \text{index}, i, j : \text{local}, \mathcal{L} : \text{set of local};</math></li> <li>2. <math>\mathcal{L} \leftarrow \{i_k \in \mathcal{S}_k : \langle k p \rangle[i_k] \neq 0 \wedge N_{k,e}[i_k, \cdot] \neq 0\};</math></li> <li>3. while <math>\mathcal{L} \neq \emptyset</math> do</li> <li>4. pick and remove <math>i</math> from <math>\mathcal{L};</math></li> <li>5. <math>f \leftarrow \text{RecFire}(e, k-1, \langle k p \rangle[i]);</math></li> <li>6. if <math>f \neq 0</math> then</li> <li>7.   foreach <math>j</math> s.t. <math>N_{k,e}[i, j] = 1</math> do</li> <li>8.     <math>u \leftarrow \text{Union}(k-1, f, \langle k p \rangle[j]);</math></li> <li>9.     if <math>u \neq \langle k p \rangle[j]</math> then</li> <li>10.       <math>\langle k p \rangle[j] \leftarrow u;</math></li> <li>11.       if <math>N_{k,e}[j, \cdot] \neq 0</math> then</li> <li>12.         <math>\mathcal{L} \leftarrow \mathcal{L} \cup \{j\};</math></li> <li>13. <math>p \leftarrow \text{CheckIn}(k, p);</math></li> <li>14. return <math>p;</math></li> </ol>	<ul style="list-style-type: none"> <li>• optional: for pipelining</li> <li>• optional: for pipelining</li> <li>• reinsert <math>\langle k p \rangle</math> in unique table</li> </ul>
<i>RecFire</i> ( $e : \text{event}, l : \text{level}, q : \text{index}$ ) : <i>index</i>	
Recursively fire $e$ in $\langle l q \rangle$ , where $l < \text{Top}(e)$ .	
<ol style="list-style-type: none"> <li>1. declare <math>f, u, s : \text{index}, i, j : \text{local}, \mathcal{L} : \text{set of local};</math></li> <li>2. if <math>l &lt; \text{Bot}(e)</math> then return <math>q;</math></li> <li>3. if <math>\text{Cached}(\text{FIRE}, l, e, q, s)</math> then return <math>s;</math></li> <li>4. <math>s \leftarrow \text{NewNode}(l);</math></li> <li>5. <math>\mathcal{L} \leftarrow \{i_l \in \mathcal{S}_l : N_{l,e}[i_l, \cdot] \neq 0 \wedge \langle l q \rangle[i_l] \neq 0\};</math></li> <li>6. while <math>\mathcal{L} \neq \emptyset</math> do</li> <li>7. pick and remove <math>i</math> from <math>\mathcal{L};</math></li> <li>8. <math>f \leftarrow \text{RecFire}(e, l-1, \langle l q \rangle[i]);</math></li> <li>9. if <math>f \neq 0</math> then</li> <li>10.   foreach <math>j</math> s.t. <math>N_{l,e}[i, j] = 1</math> do</li> <li>11.     <math>u \leftarrow \text{Union}(l-1, f, \langle l q \rangle[j]);</math></li> <li>12.     if <math>u \neq \langle l q \rangle[j]</math> then</li> <li>13.       <math>\langle l q \rangle[j] \leftarrow u;</math></li> <li>14. <math>s \leftarrow \text{CheckIn}(l, s);</math></li> <li>15. <math>\text{PutInCache}(\text{FIRE}, l, e, q, s);</math></li> <li>16. return <math>s;</math></li> </ol>	<ul style="list-style-type: none"> <li>• insert <math>\langle l s \rangle</math> in unique table</li> <li>• remember result</li> </ul>

Figure 3.8: Algorithm for firing events with in-place updates.

*RecFire*. Function *Fire* is called to fire event  $e$  in nodes  $\langle k|p \rangle$ , with  $k = \text{Top}(e)$ , and it updates its arcs  $\langle k|p \rangle[j]$  in place. The recursive function *RecFire* is called to fire  $e$  on nodes  $\langle l|q \rangle$ , with  $\text{Top}(e) > l \geq \text{Bot}(e)$ , but it does not modify these nodes. Instead, it creates a node  $\langle l|s \rangle$  that encodes the local effect of firing  $e$  on node  $\langle l|q \rangle$  and its descendants. The calls *Cached(FIRE, l, e, q, s)* and *PutInCache(FIRE, l, e, q, s)* are exactly analogous to those used in *Union*, except that, when the keyword is *FIRE*, the search key is now an event,  $e$ , and a node index,  $q$ , instead of the two nodes required for a union. It should be noted that only the firing of an event  $e$  on nodes at levels *below*  $\text{Top}(e)$  are cached. This is because, by updating node  $\langle k|p \rangle$  in place, we keep changing (increasing) the set of states it encodes, thus the cached value for the effect of firing  $e$  in  $\langle k|p \rangle$  becomes obsolete after every update. Note that, in a traditional approach without in-place updates, we need to cache this information only to deal with the situation when  $\langle k|p \rangle$  has multiple incoming arcs; each in-place update in our case is equivalent to creating a new node (or a sequence of new nodes, if *Fire* uses pipelining) in the traditional approach, and thus there would be no cached value for such nodes, unless they are duplicates of existing nodes.

By using in-place updates, *Fire* achieves a chaining effect: if two local states  $i$  and  $i'$  “locally enable” event  $e$ , i.e.,  $\mathbf{N}_{k,e}[i, \cdot] \neq 0$  and  $\mathbf{N}_{k,e}[i', \cdot] \neq 0$ , if  $\mathbf{N}_{k,e}[i, i'] = 1$ , and  $i$  is picked before  $i'$  in statement 4, the effect of firing  $e$  in  $\langle k|p \rangle[i]$  has already been incorporated in the updated node  $\langle k|p \rangle$  (that is, in the node pointed by  $\langle k|p \rangle[j]$ ), by the time  $j$  is picked. Thus,  $\langle k|p \rangle[j]$  will include the effect of firing  $e$ , zero, one, or two times. This *pipelining* can be carried even further by adding the last two statements in *Fire*, marked optional in the pseudocode: if we add the new local states  $j$  back to the set  $\mathcal{L}$  of local states to explore (statement 12), the order in which the elements in  $\mathcal{L}$  are picked does not matter, and the

result is a “full” pipelining: event  $e$  is fired in the local states of  $\langle k|p \rangle$  as long as doing so discovers new states. Therefore, the call  $Fire(e, k, p)$  updates  $\langle k|p \rangle$  in-place so that it encodes  $\mathcal{N}_e^*(\mathcal{B}(\langle k|p \rangle))$ , where, for notational convenience, we interpret the application of a global next-state function to a set of substates in the obvious way:  $\mathcal{N}_e(\mathcal{B}(\langle k|p \rangle))$  really means  $\mathcal{N}_{k,e} \times \cdots \times \mathcal{N}_{1,e}(\mathcal{B}(\langle k|p \rangle))$ .

In addition to creating and destroying fewer nodes, in-place updates have the advantage that the node  $\langle k|p \rangle$  being updated is seldom deleted, so that its incoming arcs do not need to be redirected. In other words, if the value of  $\langle k+1|q \rangle[i]$  was  $p$  before a call  $Fire(e, k, p)$ , the value should remain  $p$ . Redirection is needed only when, after having been modified in place, node  $\langle k|p \rangle$  becomes a duplicate of an existing node  $\langle k|d \rangle$ . In this case,  $\langle k|p \rangle$  must be deleted and its incoming arcs must be redirected to  $\langle k|d \rangle$  (in other words, we must set  $\langle k+1|q \rangle[i]$  to  $d$ ). In [26], we do so by using either *upstream arcs* or *forwarding arcs*. In the former case, we actually maintain pointers from each node  $\langle k|p \rangle$  to any node  $\langle k+1|q \rangle$  having pointers to it. In the latter case, we place an arc in the node  $\langle k|p \rangle$  to be deleted, pointing to its active duplicate  $\langle k|d \rangle$ ;  $\langle k|p \rangle$  can then be actually deleted only once all the references to it have been forwarded to  $\langle k|d \rangle$ .

### 3.6 The saturation strategy

An important advantage of using decision diagrams to encode  $\mathcal{N}$  lies in the ability to obtain an entire set of new states reachable from the current set through a single symbolic step. This approach works well in practice for the analysis of mostly synchronous systems. However, in many practical examples, the state space explosion phenomenon that hampers



explicit techniques, reshapes itself in the form of a new obstacle for symbolic methods: “the BDD node explosion”. This usually manifests midway through symbolic exploration, when the number of nodes in the decision diagrams, especially for levels in the middle section, grows extremely fast, before contracting to the final configuration, which is usually much more compact. The *peak size* of the decision diagram (measured in bytes or nodes) is frequently hundreds or thousands of times larger than the *final size*, placing enormous strain on the storage and computational resources.

A fundamental goal of any exploration strategy, then, should be a small peak-to-final size ratio. Attempts have been made in this direction, but most of them are still tied to the idea of breadth-first search through the use of global iterations and have had limited success in reducing peak memory requirements. The algorithms in [111, 126] aim at reducing the number of iterations needed to reach all states, by producing a pipelining effect; using a static causality analysis, events can be ordered so that their firing increases the chance of finding new states at each step. Other approaches attempt a partial symbolic traversal on certain subsets of the newly reached states in the first stages and delay the exploration on the other portion, only to complete the full search in the end. The *high density* approach in [120] targets those subsets that have the most compact symbolic representation to advance the search in the early stages. The technique in [24] computes, in an initial learning phase, the *activity profiles* of each BDD node in the transition relation and uses this information to prune the decision diagrams of the inactive nodes. The *guided* search in [12] uses user-defined hints to simplify the transition relation, but this requires a priori knowledge of the system to predict the evolution of the BDD.

Our approach starts by identifying nodes that contribute to the peak MDD size and are

eventually eliminated, as opposed to nodes that are still (likely to be) present in the final MDD. To this end, we define a restriction of the next-state function to the set of events affecting level  $k$  and below:

$$\mathcal{N}_{\leq k} = \bigcup_{1 \leq l \leq k} \mathcal{N}_{\mathcal{E}_l} = \bigcup_{e: \text{Top}(e) \leq k} \mathcal{N}_e$$

Then, the following condition gives a necessary, but not sufficient, condition for a node to be in the final MDD.

**Definition 3.6.1** *An MDD node  $\langle k|p \rangle$  at level  $k$  is said to be **saturated** if it represents a fixed point with respect to the firing of any event that affects only levels  $k$  and below:  $B(\langle k|p \rangle) = \mathcal{N}_{\leq k}^*(B(\langle k|p \rangle))$ .*

It can be shown by contradiction that any descendant of a saturated node must be saturated as well. Also, since  $\mathcal{N}_{\leq K}$  is simply  $\mathcal{N}$ , we can conclude that

$$B(\langle K|r \rangle) = \mathcal{N}^*(B(\langle K|r \rangle)),$$

when the root  $\langle K|r \rangle$  of the MDD is saturated. Furthermore, if  $\mathbf{s} \in B(\langle K|r \rangle)$ , we have  $B(\langle K|r \rangle) \supseteq \mathcal{N}^*(\mathbf{s})$ . Thus, if we initialize the root node  $\langle K|r \rangle$  so that it encodes exactly the initial state,  $B(\langle K|r \rangle) = \{\mathbf{s}\}$ , and if we saturate  $\langle K|r \rangle$  without ever adding any unreachable state in  $\widehat{\mathcal{S}} \setminus \mathcal{S}$  to  $B(\langle K|r \rangle)$ , the final result will satisfy  $B(\langle K|r \rangle) = \mathcal{S}$ .

This is the idea behind our saturation algorithm [27]. We greedily transform unsaturated nodes into saturated ones in a way that minimizes the number of unsaturated nodes present in the MDD. When working on a node at level  $k$ , only one MDD node per level

is unsaturated, at levels  $K$  through  $k$ . We fire events node-wise and exhaustively, instead of level-wise and just once per iteration, and in a specific *saturation order* that guarantees that, when saturating a node at level  $k$ , all nodes at levels below  $k$  are already saturated. In other words, starting from the MDD encoding the initial state, we first fire all events in  $\mathcal{E}_1$  exhaustively in the node at level 1. Then, we saturate the node at level 2, by repeatedly firing all events in  $\mathcal{E}_2$ . If this creates nodes at level 1, they are immediately saturated, by firing all events in  $\mathcal{E}_1$  on them. Then, we saturate the node at level 3, by exhaustively firing all events in  $\mathcal{E}_3$ . If this creates nodes at levels 2 or 1, we saturate them immediately, using the corresponding set of events. The algorithm halts when we have saturated the root node at level  $K$ . A detailed example of how the saturation algorithm works is presented in the next chapter, where we extend it to a more general case, hence the given example serves both causes.

While we assume a single initial state  $\mathbf{s}$ , thus a single unsaturated node per level in the initial MDD, it is trivial to extend the algorithm to the case where there is a set of initial states encoded by an arbitrary MDD. Also in this case, the number of unsaturated nodes never increases beyond the initial count. Thus, except for those initially unsaturated nodes, only saturated nodes contribute to the peak size of the MDD. These saturated nodes are not guaranteed to be present in the final MDD, as they might become disconnected when arcs pointing to them are redirected to other (saturated) nodes encoding even larger sets. But they have at least a chance to be present in the final MDD, since they satisfy the necessary condition: they are saturated.

The pseudocode for the saturation algorithm is shown in Figures 3.9-3.10. Just as in traditional symbolic state-space generation algorithms, we use a *unique table* to detect

<i>Generate</i> ( $s : \text{state}, \mathcal{N} : \hat{\mathcal{S}} \rightarrow 2^{\hat{\mathcal{S}}} : \text{index}$ )	
Build an MDD rooted at $\langle K r \rangle$ encoding $\mathcal{N}^*(s)$ , return $r$ .	
<ol style="list-style-type: none"> <li>1. declare <math>r, p : \text{index}, \quad k : \text{level};</math></li> <li>2. <math>p \leftarrow 1;</math></li> <li>3. for <math>k = 1</math> to <math>K</math> do</li> <li>4. <math>r \leftarrow \text{NewNode}(k);</math></li> <li>5. <math>\langle k r \rangle[0] \leftarrow p;</math></li> <li>6. <math>\text{Saturate}(k, r);</math></li> <li>7. <math>\text{CheckIn}(k, r);</math></li> <li>8. <math>p \leftarrow r;</math></li> <li>9. return <math>r;</math></li> </ol>	<ul style="list-style-type: none"> <li>• the index <math>\psi_k(s_k)</math> of is 0</li> </ul>
<i>Saturate</i> ( $k : \text{level}, p : \text{index}$ )	
Update $\langle k p \rangle$ in-place, to encode $\mathcal{N}_{\leq k}^*(B(\langle k p \rangle))$ .	
<ol style="list-style-type: none"> <li>1. declare <math>e : \text{event}, \quad \text{chng} : \text{bool};</math></li> <li>2. <math>\text{chng} \leftarrow \text{true};</math></li> <li>3. while <math>\text{chng}</math> do</li> <li>4. <math>\text{chng} \leftarrow \text{false};</math></li> <li>5. foreach <math>e \in \mathcal{E}_k</math> do</li> <li>6. <math>\text{chng} \leftarrow \text{chng} \vee \text{SatFire}(e, k, p);</math></li> </ol>	

**Figure 3.9:** The pseudocode for the *Generate*, *Saturate* algorithms.

duplicate nodes, and *operation caches*, in particular a *union cache* and a *firing cache*, to speed-up computation. In our approach, however, only saturated nodes are checked in the unique table or referenced in the caches. In particular, the *Union* function of Figure 2.8 operates correctly without having to know whether the nodes it operates upon are saturated or not. This is because the result of the union of two saturated nodes is going to be saturated by definition:

$$\begin{aligned}
 B(\langle k|p \rangle) &= \mathcal{N}_{\leq k}^*(B(\langle k|p \rangle)) \wedge B(\langle k|q \rangle) = \mathcal{N}_{\leq k}^*(B(\langle k|q \rangle)) \\
 &\Rightarrow B(\langle k|p \rangle) \cup B(\langle k|q \rangle) = \mathcal{N}_{\leq k}^*(B(\langle k|p \rangle) \cup B(\langle k|q \rangle)).
 \end{aligned}$$

The saturation strategy has several important properties. It is very flexible in al-

<i>SatFire</i> ( <i>e</i> : event, <i>k</i> : level, <i>p</i> : index) : bool
Update $\langle k p \rangle$ in-place, to encode $\mathcal{N}_{\leq k-1}^*(\mathcal{N}_e^*(\mathcal{B}(\langle k p \rangle)))$ .
<ol style="list-style-type: none"> <li>1. declare <i>f, u</i> : index, <i>i, j</i> : local, <i>L</i> : set of local <i>chng</i> : bool;</li> <li>2. <i>chng</i> <math>\leftarrow</math> false;</li> <li>3. <i>L</i> <math>\leftarrow \{i_k \in S_k : \langle k p \rangle[i_k] \neq 0, \mathbf{N}_{k,e}[i_k, \cdot] \neq 0\}</math>;</li> <li>4. while <i>L</i> <math>\neq \emptyset</math> do</li> <li>5.   pick and remove <i>i</i> from <i>L</i>;</li> <li>6.   <i>f</i> <math>\leftarrow</math> <i>SatRecFire</i>(<i>e</i>, <i>k</i>−1, <math>\langle k p \rangle[i]</math>);</li> <li>7.   if <i>f</i> <math>\neq 0</math> then</li> <li>8.     foreach <i>j</i> s.t. <math>\mathbf{N}_{k,e}[i, j] = 1</math> do</li> <li>9.       <i>u</i> <math>\leftarrow</math> <i>Union</i>(<i>k</i>−1, <i>f</i>, <math>\langle k p \rangle[j]</math>);</li> <li>10.      if <i>u</i> <math>\neq \langle k p \rangle[j]</math> then</li> <li>11.       <math>\langle k p \rangle[j] \leftarrow u</math>;</li> <li>12.       <i>chng</i> <math>\leftarrow</math> true;</li> <li>13.       if <math>\mathbf{N}_{k,e}[j, \cdot] \neq 0</math> then</li> <li>14.        <i>L</i> <math>\leftarrow L \cup \{j\}</math>;</li> <li>15. return <i>chng</i>;</li> </ol>
<i>SatRecFire</i> ( <i>e</i> : event, <i>l</i> : level, <i>q</i> : index) : index
Build an MDD rooted at $\langle l s \rangle$ encoding $\mathcal{N}_{\leq l}^*(\mathcal{N}_e(\mathcal{B}(\langle l q \rangle)))$ , return <i>s</i> .
<ol style="list-style-type: none"> <li>1. declare <i>f, u, s</i> : index, <i>i, j</i> : local, <i>L</i> : set of local, <i>chng</i> : bool;</li> <li>2. if <i>l</i> &lt; <i>Bot</i>(<i>e</i>) then return <i>q</i>;</li> <li>3. if <i>Cached</i>(<i>FIRE</i>, <i>l</i>, <i>e</i>, <i>q</i>, <i>s</i>) then return <i>s</i>;</li> <li>4. <i>s</i> <math>\leftarrow</math> <i>NewNode</i>(<i>l</i>); <i>chng</i> <math>\leftarrow</math> false;</li> <li>5. <i>L</i> <math>\leftarrow \{i_l \in S_l : \langle l q \rangle[i_l] \neq 0, \mathbf{N}_{l,e}[i_l, \cdot] \neq 0\}</math>;</li> <li>6. while <i>L</i> <math>\neq \emptyset</math> do</li> <li>7.   pick and remove <i>i</i> from <i>L</i>;</li> <li>8.   <i>f</i> <math>\leftarrow</math> <i>SatRecFire</i>(<i>e</i>, <i>l</i>−1, <math>\langle l q \rangle[i]</math>);</li> <li>9.   if <i>f</i> <math>\neq 0</math> then</li> <li>10.     foreach <i>j</i> s.t. <math>\mathbf{N}_{l,e}[i, j] = 1</math> do</li> <li>11.       <i>u</i> <math>\leftarrow</math> <i>Union</i>(<i>l</i>−1, <i>f</i>, <math>\langle l s \rangle[j]</math>);</li> <li>12.       if <i>u</i> <math>\neq \langle l s \rangle[j]</math> then</li> <li>13.        <math>\langle l s \rangle[j] \leftarrow u</math>;</li> <li>14.       <i>chng</i> <math>\leftarrow</math> true;</li> <li>15. if <i>chng</i> then <i>Saturate</i>(<i>l</i>, <i>s</i>);</li> <li>16. <i>CheckIn</i>(<i>l</i>, <i>s</i>);</li> <li>17. <i>PutInCache</i>(<i>FIRE</i>, <i>l</i>, <i>e</i>, <i>q</i>, <i>s</i>);</li> <li>18. return <i>s</i>;</li> </ol>

Figure 3.10: The pseudocode for *SatFire* and *SatRecFire*.

lowing a “chaotic” order of firing events, as dictated by the dynamics of creating new nodes. Moreover, all firings of events are very lightweight operations, with localized and chained/pipelined effects, as opposed to the monolithic heavyweight image computation of traditional breadth-first strategies. Storing and managing only saturated nodes has additional benefits. Many, if not most, of these nodes will still be present on the final MDD, while the unsaturated nodes are *guaranteed* not to be part of it. These properties lead to enormous time and memory savings, as illustrated in the results section: saturation is up to five orders of magnitude faster and uses up to three orders of magnitude less memory when compared to other state-of-the-art tools, such as NuSMV [33]. The experimental studies for saturation also show that, at times, saturation is *optimal*, in the sense that the peak and final numbers of nodes differ by a small *constant*.

### 3.7 Correctness

**Theorem 3.7.1** *Consider a node  $\langle k|p \rangle$ ,  $K \geq k \geq 1$ , with saturated children. Moreover, (a) let  $\langle l|q \rangle$  be one of its children, satisfying  $q \neq 0$  and  $l = k-1$ ; (b) let  $\mathcal{U}$  stand for  $\mathcal{B}(\langle l|q \rangle)$  before the call  $\text{SatRecFire}(e, l, q)$ , for some event  $e$  with  $l < \text{Top}(e)$ , and let  $\mathcal{V}$  represent  $\mathcal{B}(\langle l|f \rangle)$ , where  $f$  is the value returned by this call; and (c) let  $\mathcal{X}$  and  $\mathcal{Y}$  denote  $\mathcal{B}(\langle k|p \rangle)$  before and after calling  $\text{Saturate}(k, p)$ , respectively. Then, (i)  $\mathcal{V} = \mathcal{N}_{\leq l}^*(\mathcal{N}_e(\mathcal{U}))$  and (ii)  $\mathcal{Y} = \mathcal{N}_{\leq k}^*(\mathcal{X})$ .*

By choosing, for node  $\langle k|p \rangle$ , the root  $\langle K|r \rangle$  of the MDD representing the initial system state  $\mathbf{s}$ , we obtain  $\mathcal{Y} = \mathcal{N}_{\leq K}^*(\mathcal{B}(\langle K|r \rangle)) = \mathcal{N}_{\leq K}^*(\{\mathbf{s}\}) = \mathcal{S}$ , as desired.

**Proof.** To prove both statements we employ a simultaneous induction on  $k$ . For the

induction base,  $k = 1$ , we have: (i) The only possible call  $SatRecFire(e, 0, 1)$  immediately returns 1 because of the test on  $l$  (cf. line 2). Then,  $\mathcal{U} = \mathcal{V} = \{\langle \rangle\}$  and  $\{\langle \rangle\} = \mathcal{N}_{\leq 0}^*(\mathcal{N}_e(\{\langle \rangle\}))$ . (ii) The call  $Saturate(1, p)$  repeatedly explores events in  $\mathcal{E}_1$ , in every local state  $i$  for which  $\mathcal{N}_{1, \mathcal{E}_1}(i) \neq \emptyset$  and for which  $\langle 1|p \rangle[i]$  is either 1 at the beginning of the “while  $\mathcal{L} \neq \emptyset$ ” loop, or has been modified (cf. line 10) from 0 to 1, which is the value of  $f$ , hence  $u$ , since the call  $SatRecFire(e, 0, 1)$  returns 1. The iteration stops when further attempts to fire  $\mathcal{E}_1$  do not add any new state to  $\mathcal{B}(\langle 1|p \rangle)$ . At this point,  $\mathcal{Y} = \mathcal{N}_{\mathcal{E}_1}^*(\mathcal{X}) = \mathcal{N}_{\leq 1}^*(\mathcal{X})$ .

For the induction step we assume that the calls to  $Saturate(k-1, \cdot)$  as well as to  $SatRecFire(e, l-1, \cdot)$  work correctly. Recall that  $l = k-1$ .

- (i) Unlike  $SatFire$  (cf. line 14),  $SatRecFire$  does not add further local states to  $\mathcal{L}$ , since it modifies in-place the new node  $\langle l|s \rangle$ , and not node  $\langle l|q \rangle$  describing the states from where the firing is explored. The call  $SatRecFire(e, l, q)$  can be resolved in three ways. If  $l < Bot(e)$ , then the returned value is  $f = q$  and  $\mathcal{N}_{l,e}(\mathcal{U}) = \mathcal{U}$  for any set  $\mathcal{U}$ ; since  $q$  is saturated,  $\mathcal{B}(\langle l|q \rangle) = \mathcal{N}_{\leq l}^*(\mathcal{B}(\langle l|q \rangle)) = \mathcal{N}_{\leq l}^*(\mathcal{N}_e(\mathcal{B}(\langle l|q \rangle)))$ . If  $l \geq Bot(e)$  but  $SatRecFire$  has been called previously with the same parameters, then the call  $Cached(FIRE, l, e, q, s)$  is successful. Since node  $q$  is saturated and in the unique table, it has not been modified further; note that in-place updates are performed only on nodes not yet in the unique table. Thus, the value  $s$  in the cache is still valid and can be safely used. Finally, we need to consider the case where the call  $SatRecFire(e, l, q)$  performs “real work.” First, a new node  $\langle l|s \rangle$  is created, having all its arcs initialized to 0. We explore the firing of  $e$  in each state  $i$  satisfying  $\langle l|q \rangle[i] \neq 0$  and  $\mathcal{N}_{l,e}(i) \neq \emptyset$ . By induction hypothesis, the recursive call  $SatRecFire(e, l-1, \langle l|q \rangle[i])$  returns  $\mathcal{N}_{\leq l-1}^*(\mathcal{N}_e(\mathcal{B}(\langle l-1|\langle l|q \rangle[i])$ )). Hence, when the “while  $\mathcal{L} \neq \emptyset$ ” loop terminates,

$\mathcal{B}(\langle l|s \rangle) = \bigcup_{i \in \mathcal{S}^l} \mathcal{N}_{l,e}(i) \times \mathcal{N}_{\leq l-1}^*(\mathcal{N}_e(\mathcal{B}(\langle l-1|\langle l|q \rangle[i]))) = \mathcal{N}_{\leq l-1}^*(\mathcal{N}_e(\mathcal{B}(\langle l|q \rangle)))$  holds.

Thus, all children of node  $\langle l|s \rangle$  are saturated. According to the induction hypothesis, the call  $\text{Saturate}(l, s)$  correctly saturates  $\langle l|s \rangle$ . Consequently, we have  $\mathcal{B}(\langle l|s \rangle) = \mathcal{N}_{\leq l}^*(\mathcal{N}_{\leq l-1}^*(\mathcal{N}_e(\mathcal{B}(\langle l|q \rangle)))) = \mathcal{N}_{\leq l}^*(\mathcal{N}_e(\mathcal{B}(\langle l|q \rangle)))$  after the call.

- (ii) As in the base case,  $\text{Saturate}(k, p)$  repeatedly explores the firing of each event  $e$  that is locally enabled in  $i \in \mathcal{S}_k$ ; it calls  $\text{SatRecFire}(e, k-1, \langle k|p \rangle[i])$  that, as shown above and since  $l = k-1$ , returns  $\mathcal{N}_{\leq k-1}^*(\mathcal{N}_e(\mathcal{B}(\langle k-1|\langle k|p \rangle[i])))$ . Further,  $\text{Saturate}(k, p)$  terminates when firing the events in  $\mathcal{E}_k = \{e_1, e_2, \dots, e_m\}$  does not add any new state to  $\mathcal{B}(\langle k|p \rangle)$ . At this point, the set  $\mathcal{Y}$  encoded by  $\langle k|p \rangle$  is the fixed-point of the iteration

$$\mathcal{Y}^{(m+1)} \leftarrow \mathcal{Y}^{(m)} \cup \mathcal{N}_{\leq k-1}^*(\mathcal{N}_{e_1}(\mathcal{N}_{\leq k-1}^*(\mathcal{N}_{e_2}(\dots \mathcal{N}_{\leq k-1}^*(\mathcal{N}_{e_m}(\mathcal{Y}^{(m)})) \dots)))),$$

initialized with  $\mathcal{Y}^{(0)} \leftarrow \mathcal{X}$ . Hence,  $\mathcal{Y} = \mathcal{N}_{\leq k}^*(\mathcal{X})$ , as desired.  $\square$

### 3.8 Implementation issues

**Garbage collection.** MDD nodes can become disconnected, i.e., unreachable from the root, and should be removed. Disconnection is detected by associating an *incoming-arc counter* to each node  $\langle k|p \rangle$ . Recycling disconnected nodes is a major issue in traditional symbolic state-space generation algorithms, where usually many more nodes become disconnected. In our algorithm, this phenomenon is much less frequent, and the best runtime is achieved by removing these nodes only at the end. We refer to this policy as **LAZY** policy.

We also implemented a **STRICT** policy where, if a node  $\langle k|p \rangle$  becomes disconnected, its “delete-flag” is set and its arcs  $\langle k|p \rangle[i]$  are re-directed to  $\langle k-1|0 \rangle$ , with possible recursive



effects on the nodes downstream. When a hit in the cache returns an index  $s$ , this entry is considered stale if the delete-flag of node  $\langle k|s \rangle$  is set. By keeping a per-level count of the nodes with delete-flag set, we can decide in routine *NewNode*( $k$ ) whether to (a) allocate new memory for a node at level  $k$  or (b) recycle the indices and the physical memory of all nodes at level  $k$  with delete-flag set, after having removed all the entries in the cache referring to them. The threshold that triggers recycling can be set in terms of number of nodes or bytes of memory. The policy using a threshold of one node, denoted as STRICT(1), is optimal in terms of memory consumption, but has a higher overhead due to more frequent clean-ups.

**Optimizations.** First, observe that the two loops in *Saturate* ensure that firing some event  $e \in \mathcal{E}_k$  does not add any new state. If we always consider these events in the same order, we can stop iterating as soon as  $|\mathcal{E}_k|$  consecutive events have been explored without revealing any new state. This saves  $|\mathcal{E}_k|/2$  firing attempts on average, which translates to speed-ups of up to 25% in our experimental studies. Also, in *Union*, the call *PutInCache*(*UNION*,  $p, q, s$ ) records that  $\mathcal{B}(\langle k|s \rangle) = \mathcal{B}(\langle k|p \rangle) \cup \mathcal{B}(\langle k|q \rangle)$ . Since this implies  $\mathcal{B}(\langle k|s \rangle) = \mathcal{B}(\langle k|p \rangle) \cup \mathcal{B}(\langle k|s \rangle)$  and  $\mathcal{B}(\langle k|s \rangle) = \mathcal{B}(\langle k|s \rangle) \cup \mathcal{B}(\langle k|q \rangle)$ , we can, optionally, also issue the calls *Cached*(*UNION*,  $p, s, s$ ), if  $s \neq p$ , and *Cached*(*UNION*,  $q, s, s$ ), if  $s \neq q$ . This *speculative union heuristic* improves performance by up to 20% in some models.

### 3.9 Results

To evaluate our saturation algorithm, we have chosen a suite of examples with a wide range of characteristics. Their detailed description is given in Appendix B.

In all cases, the state space sizes depend on a parameter  $N$ .

- The *dining philosophers* and *slotted ring models* [26, 111] are obtained by connecting  $N$  identical safe subnets in a circular fashion. The MDD has  $N/2$  MDD levels (two subnets per level) for the former model and  $N$  levels (one subnet per level) for the latter. Events are either local or synchronize adjacent subnets, thus they span only two levels, except for those synchronizing subnet  $N$  with subnet 1, which span the entire MDD.
- The *round-robin mutex protocol* model [72] also has  $N$  identical safe subnets placed in a circular fashion, which represent  $N$  processes, each mapped to one MDD level. Another subnet models a resource shared by the  $N$  processes, giving rise to one more level, at the bottom of the MDD. There are no local events and, in addition to events synchronizing adjacent subnets, the model contains events synchronizing levels  $n$  and 1, for  $2 \leq n \leq N + 1$ .
- The *flexible manufacturing system* (FMS) model [107] has a fixed shape, but is parameterized by the initial number  $N$  of tokens in some places. We partition this model into 19 subnets, giving rise to a 19-level MDD with a moderate degree of locality, as events span from two to six levels.

We conducted two series of experiments. In the first, we compared the various exploration algorithms presented in Section 3.6, which we implemented in our tool `SMART` using MDDs and Kronecker matrices.

To the left of Tables 3.1-3.2 are the model parameters ( $N$  and  $|S|$ , the state space size), while the runtimes and memory requirements for six different exploration strategies are

listed in the right blocks of columns, respectively. The strategies considered are the two variants of breadth-first search, *BfSsGen* and *AllBfSsGen*, the two variants of BFS with chaining, *ChSsGen* and *AllChSsGen*, the in-place update approach of [26] (*Fwd*), and the saturation algorithm [27] with LAZY garbage collection policy (*Sat*). The last column in Table 3.2 lists the memory needed by the final MDD, encoding the entire state space, this quantity being independent of the algorithm used.

N	Reachable states	Time (sec)					
		Bfs	AllBfs	Chn	AllChn	Fwd	Sat
Dining Philosophers: $K = N$ , $ S_k  = 34$ for all $k$							
50	$2.2 \times 10^{31}$	37.6	36.8	1.3	1.3	0.1	0.0
100	$5.0 \times 10^{62}$	644.1	630.4	5.4	5.3	0.3	0.2
1000	$9.2 \times 10^{626}$	—	—	895.4	915.5	4.6	1.1
10000	$4.3 \times 10^{6269}$	—	—	—	—	704.0	84.4
Slotted Ring Network: $K = N$ , $ S_k  = 15$ for all $k$							
5	$5.3 \times 10^4$	0.2	0.3	0.1	0.1	0.0	0.0
10	$8.3 \times 10^9$	21.5	24.1	2.1	1.2	0.1	0.0
15	$1.5 \times 10^{15}$	745.4	771.5	18.5	8.9	0.5	0.1
50	$1.7 \times 10^{52}$	—	—	—	—	120.3	2.9
100	$2.6 \times 10^{105}$	—	—	—	—	4976.1	21.6
Round Robin: $K = N + 1$ , $ S_k  = 10$ for all $k$ except $ S_1  = N + 1$							
10	$2.3 \times 10^4$	0.2	0.3	0.1	0.1	0.0	0.0
20	$4.7 \times 10^7$	2.7	4.4	0.3	0.3	0.1	0.0
30	$7.2 \times 10^{10}$	16.4	26.7	0.7	0.7	0.3	0.1
50	$1.3 \times 10^{17}$	263.2	427.6	2.9	2.8	1.0	0.2
100	$2.9 \times 10^{32}$	—	—	22.1	19.3	7.7	1.2
200	$7.2 \times 10^{62}$	—	—	—	—	58.1	10.9
FMS: $K = 19$ , $ S_k  = N + 1$ for all $k$ except $ S_{17}  = 4$ , $ S_{12}  = 3$ , $ S_7  = 2$							
5	$2.9 \times 10^6$	0.7	0.7	0.1	0.1	0.0	0.0
10	$2.5 \times 10^9$	7.0	5.8	0.5	0.3	0.1	0.0
15	$2.2 \times 10^{11}$	41.0	30.5	1.9	1.0	0.2	0.0
20	$6.0 \times 10^{12}$	183.6	126.0	5.3	2.3	0.5	0.1
25	$8.5 \times 10^{13}$	677.2	437.9	12.9	5.1	1.0	0.1
150	$4.8 \times 10^{23}$	—	—	—	—	—	8.4

**Table 3.1:** Time for state-space generation algorithms in SMART.

The results show a steady performance gain that “chronologically” follows the series of

N	Reachable states	Memory (MB)						
		Bfs	AllBfs	Chn	AllChn	Fwd	Sat	final
Dining Philosophers: $K=N$ , $ \mathcal{S}_k =34$ for all $k$								
50	$2.2 \times 10^{31}$	146.8	131.6	2.2	2.2	0.0	0.0	0.0
100	$5.0 \times 10^{62}$	> 999.9	> 999.9	8.9	8.9	0.2	0.2	0.0
1000	$9.2 \times 10^{626}$	—	—	895.2	895.0	0.5	0.4	0.3
10000	$4.3 \times 10^{6269}$	—	—	—	—	5.5	3.9	3.1
Slotted Ring Network: $K=N$ , $ \mathcal{S}_k =15$ for all $k$								
5	$5.3 \times 10^4$	0.8	1.1	0.3	0.2	0.0	0.0	0.0
10	$8.3 \times 10^9$	39.0	45.0	5.7	3.3	0.0	0.0	0.0
15	$1.5 \times 10^{15}$	344.3	375.4	35.1	20.2	0.1	0.1	0.0
50	$1.7 \times 10^{52}$	—	—	—	—	4.2	2.2	0.1
100	$2.6 \times 10^{105}$	—	—	—	—	44.5	14.8	0.4
Round Robin: $K=N+1$ , $ \mathcal{S}_k =10$ for all $k$ except $ \mathcal{S}_1 =N+1$								
10	$2.3 \times 10^4$	0.6	1.2	0.1	0.1	0.0	0.0	0.0
20	$4.7 \times 10^7$	5.9	12.8	0.5	0.5	0.0	0.0	0.0
30	$7.2 \times 10^{10}$	22.7	48.2	1.3	1.1	0.1	0.0	0.0
50	$1.3 \times 10^{17}$	126.7	257.7	4.3	3.8	0.2	0.1	0.1
100	$2.9 \times 10^{32}$	—	—	22.1	20.0	0.7	0.4	0.4
200	$7.2 \times 10^{62}$	—	—	—	—	2.7	1.4	1.4
FMS: $K=19$ , $ \mathcal{S}_k =N+1$ for all $k$ except $ \mathcal{S}_{17} =4$ , $ \mathcal{S}_{12} =3$ , $ \mathcal{S}_7 =2$								
5	$2.9 \times 10^6$	2.6	2.2	0.4	0.2	0.0	0.0	0.0
10	$2.5 \times 10^9$	18.2	14.7	2.3	1.3	0.0	0.0	0.0
15	$2.2 \times 10^{11}$	61.9	48.8	8.0	4.3	0.1	0.1	0.0
20	$6.0 \times 10^{12}$	154.0	119.4	20.2	10.3	0.1	0.1	0.1
25	$8.5 \times 10^{13}$	319.7	245.3	42.7	21.2	0.3	0.2	0.1
150	$4.8 \times 10^{23}$	—	—	—	—	—	30.7	15.8

**Table 3.2:** Memory consumption for state-space generation algorithms in SMART.

improvements presented in this chapter. The chaining technique improves on traditional BFS, the in-place updates and locality improve on chaining, while the saturation strategy is vastly superior to all predecessors by several orders of magnitude, both in terms of time and memory consumption.

On the issue of using just the MDD encoding of frontier set (strictly new states) versus the encoding of the entire current set in the BFS iterations, we found a rather surprising fact. Not only does the unanimous preference for the frontier set approach seems to be

unjustified, but, at least for the class of asynchronous systems studied here, the second approach that uses all states performs consistently better (even for the dining philosopher model, where *ChSsGen* and *AllChSsGen* have similar timing performance, *AllChSsGen* uses substantially less peak memory). This is counterintuitive only in the context of explicit methods. Decision diagrams exploit structure in the encoded sets, which seems to be poor in the frontier set.

In the second experiment (Figure 3.3) we compare three variants of our new algorithm using both the LAZY policy and the STRICT policy (with thresholds of 1 or 100 nodes per level). For reference, we also compare them with the RECURSIVE algorithm in [107] and the FORWARDING algorithm in [26]. On the left column, Figure 3.3 reports the size of the state space for each model and value of  $N$ . The graphs in the middle and right columns show the peak and final number of MDD nodes and the CPU time in seconds required for the state-space generations, respectively.

For the above models, saturation is up to two orders of magnitude faster than [107] and up to one order of magnitude faster than [26]. These results are observed for the LAZY variant of the algorithm, which yields the best runtimes; the STRICT policy also outperforms [107] and [26]. Furthermore, the gap keeps increasing as we scale up the models.

Just as important, the saturation algorithm tends to use many fewer MDD nodes, hence less memory. This is most apparent in the FMS model, where the difference between the peak and the final number of nodes is just a constant, 10, for any STRICT policy. Also notable is the reduced memory consumption for the slotted ring model, where the STRICT(1) policy uses 23 times fewer nodes compared to [107], for  $N = 50$ .

## Model &amp; size

## Dining phil.

$N$	$S$
100	$4.97 \cdot 10^{62}$
200	$2.47 \cdot 10^{125}$
400	$6.10 \cdot 10^{250}$
600	$1.51 \cdot 10^{376}$
800	$3.72 \cdot 10^{501}$
1000	$9.18 \cdot 10^{626}$

## Slotted ring

$N$	$S$
10	$8.29 \cdot 10^9$
20	$2.73 \cdot 10^{20}$
30	$1.04 \cdot 10^{31}$
40	$4.16 \cdot 10^{41}$
50	$1.72 \cdot 10^{52}$

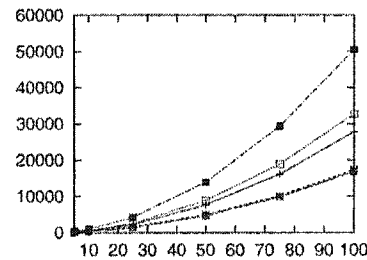
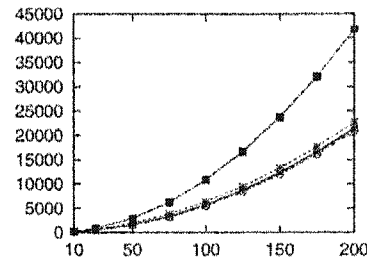
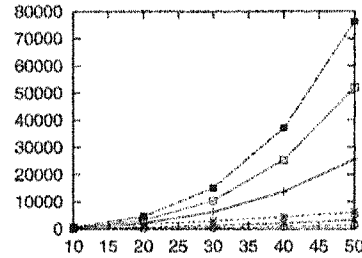
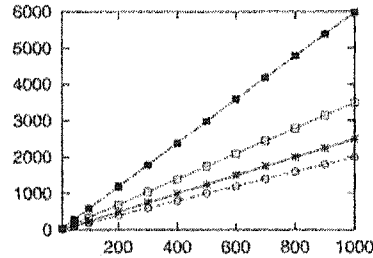
## Round robin

$N$	$S$
10	$2.30 \cdot 10^4$
25	$1.89 \cdot 10^9$
50	$1.27 \cdot 10^{17}$
100	$2.85 \cdot 10^{32}$
150	$4.82 \cdot 10^{47}$
200	$7.23 \cdot 10^{62}$

## FMS

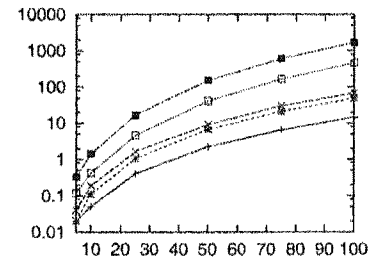
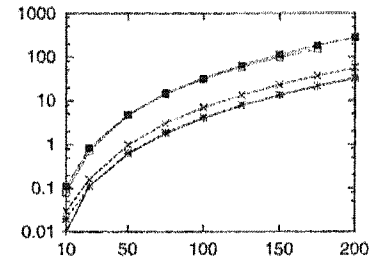
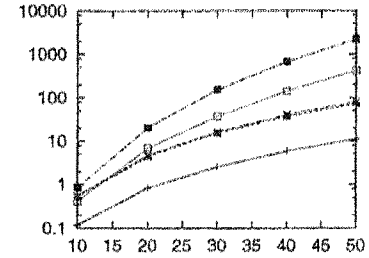
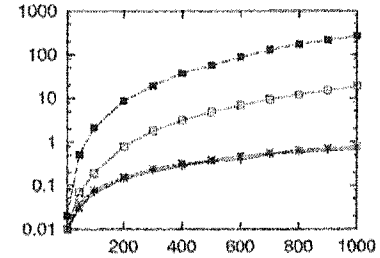
$N$	$S$
5	$2.90 \cdot 10^6$
10	$2.50 \cdot 10^9$
25	$8.54 \cdot 10^{13}$
50	$4.24 \cdot 10^{17}$
75	$6.98 \cdot 10^{19}$
100	$2.70 \cdot 10^{21}$

## Peak &amp; final MDD nodes



Key  
 lazy —○—  
 strict(1) - - - × - - -  
 strict(100) ···· \* ····  
 forwarding —□—  
 recursive —◇—  
 final —○—

## Generation time (sec.)



Key  
 lazy —○—  
 strict(1) - - - × - - -  
 strict(100) ···· \* ····  
 forwarding —□—  
 recursive —◇—  
 final —○—

Table 3.3: Results for the saturation algorithm.

In a nutshell, regarding generation time, the best algorithm is LAZY, followed by STRICT(100), STRICT(1), FORWARDING, and RECURSIVE. With respect to memory consumption, the best algorithm is STRICT(1), followed by STRICT(100), LAZY, FORWARDING, and RECURSIVE. Thus, our new algorithm is consistently faster and uses less memory than previously proposed approaches.

## Chapter 4

# Saturation Unbound

In devising the algorithms presented so far, we have assumed that the model is partitioned so that the state space of each submodel can be generated in isolation. This practice requires careful addition of constraints (such as inhibitor arcs for Petri nets) to ensure correct local behavior without affecting the global behavior. Traditionally, the burden of this task has been placed on the user, and this is often a difficult and error-prone endeavor.

In circuit verification the possible values of each state variable are known: they are simply 0 and 1. For arbitrary systems modeled in high-level formalisms, such as Petri nets or pseudocode, determining the range of the state variables is more challenging. One can argue that this problem is as hard as constructing the global state-space itself.

In this chapter we introduce a new algorithm that merges an explicit, on-the-fly construction of the local state-spaces,  $\mathcal{S}_k$ , with the symbolic global state-space generation. The algorithm produces an MDD representation of the final state-space and an exact representation of the “minimal” local state spaces. This relieves the modeler from worrying about the behavior of submodels in isolation.

Symbolic analysis of unbounded discrete-event systems has been considered before. In most cases, the goal is the study of systems with infinite but *regular* state spaces. For



example, the Queue BDDs of [68] allow one to model systems with a finite number of boolean variables plus one or more unbounded queues, as long as the contents of the queue can be represented by a deterministic finite automaton. The MONA system [76] implements *monadic second-order logic* and can be used to verify *parametric* systems without relying on a proof by induction. These types of approach can be generally classified under the umbrella of *Regular Model Checking* [13].

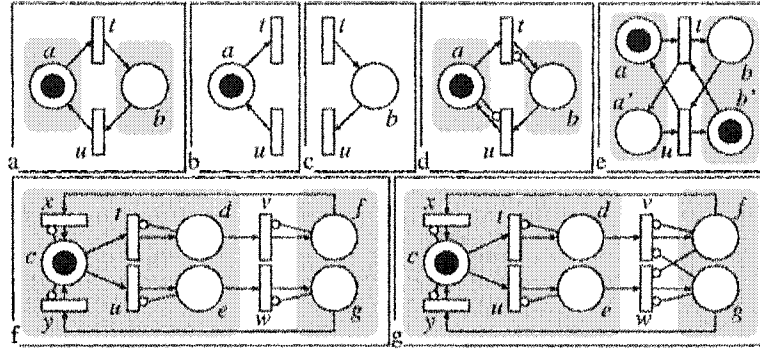
Here, we target a different problem: the analysis of bounded models with unknown bounds of the state variables. Traditional symbolic state-space generation assumes that each local state space is known a priori. One could argue that this is reasonable for BDD-based methods, since each boolean variable takes simply values in  $\{0,1\}$ ; however, this requires a similar assumption, i.e, that we know how many boolean variables are needed to represent some system variable, such as an integer variable in a software module being modeled, or the number of tokens in a Petri net place. In other words, we need to know each  $S_k$ , and in particular its size  $n_k$ , so that we can set up either the correct number  $\lceil \log n_k \rceil$  of boolean variables for it, if we use a BDD, or the correct size  $n_k$  of the nodes at level  $k$ , if we use an MDD.

The algorithm we describe in this section can be used for symbolic state-space generation when all we know (or need to assume) initially about  $S_k$  is that it has a finite but unknown size. The main idea is to interleave symbolic generation of the global state space with explicit, “on-the-fly” generation of the *smallest* local state spaces (plus, possibly, a small *rim* of additional local states that are discarded at the end). Starting only with the knowledge of the initial state, the algorithm discovers new local states at each level. As  $n_k$  increases, the size of the MDD nodes at level  $k$  increases as well (if we used BDDs, the number of

levels would have to increase instead).

Typical applications of this algorithm are found in modeling distributed software, one of the most challenging cases for symbolic methods. For this type of systems, even though the state spaces are finite, the highly “irregular” nature of the local spaces makes it difficult to apply Regular Model Checking methods.

For a given high-level formalism, we can attempt to *pregenerate* the  $k^{\text{th}}$  local state space  $S_k$  with an explicit traversal of the local state-to-state transition graph of the  $k^{\text{th}}$  submodel, obtained by considering all the variables associated with that submodel and all the events affecting those variables. Unfortunately, this may create *spurious* local states. For example, if the two places of the Petri net in Figure 4.1(a) are partitioned into two subsets, the corresponding subnets, (b) and (c), have unbounded local state spaces when considered in isolation. In subnet (b), transition  $u$  can keep adding tokens to place  $a$  since, without the input arc from  $b$ ,  $u$  is always locally enabled. Hence, in isolation,  $a$  may contain arbitrarily many tokens. The same can be said for subnet (c). However,  $S = \{a^1b^0, a^0b^1\}$ , so we would ideally like to define  $S_2 = S_1 = \{0, 1\}$ . This can be enforced by adding either inhibitor arcs, (d), or complementary places, (e). Consider now the Petri net of Figure 4.1(f), partitioned into two subnets, one containing  $c$ ,  $d$ , and  $e$ , the other containing  $f$  and  $g$ . The inhibitor arcs shown avoid unbounded local state spaces in isolation, but they do not ensure that the local state spaces are as small as possible. For example, the local state space built in isolation for the subnet containing  $f$  and  $g$  is  $\{f^0g^0, f^1g^0, f^0g^1, f^1g^1\}$ , while only the first three states are actually reachable in the overall net, since  $f$  and  $g$  can never contain a token at the same time. This is corrected in (g) by adding two more inhibitor arcs, from  $g$  to  $v$  and from  $f$  to  $w$ . An analogous problem exists for the other subnet as well, and



**Figure 4.1:** Local state spaces built in isolation.

correcting it with inhibitor arcs is even more cumbersome.

Thus, there are two problems with pregeneration: a local state space in isolation might be unbounded (causing pregeneration to fail) or it might contain spurious states (causing inefficiencies in the symbolic state-space generation, since  $n_k$  is larger than needed). Asking the modeler to cope with these problems by adding constraints to the original model (e.g., the inhibitor arcs in Figure 4.1) is at best *burdensome*, since it requires a priori knowledge of  $\mathcal{S}$ , the output of state-space generation, and at worst *dangerous*, since adding the wrong constraint might hide undesirable behaviors present in the original model.

## 4.1 Local state spaces with unknown bounds

We now describe an *on-the-fly* algorithm that intertwines explicit generation of the local state spaces with symbolic generation of the global state space and, as a result, builds the *smallest* local state spaces  $\mathcal{S}_k$  needed to encode the correct global state space  $\mathcal{S} \subseteq \hat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$ . Ideally, the additional time spent exploring local state spaces on-the-fly should be comparable to that spent in the pregeneration phase of our previous algorithm.

This is indeed the case for our new algorithm, which incrementally discovers a set  $\hat{\mathcal{S}}_k$  of *locally-reachable* local states, of which a subset  $\mathcal{S}_k$  is known to be also globally reachable. Our algorithm *confirms* that a local state  $i_k \in \hat{\mathcal{S}}_k$  is globally reachable when it appears as the  $k^{\text{th}}$  component of some state encoded by the MDD rooted at  $\langle K|r \rangle$ . Since *unconfirmed* local states in  $\hat{\mathcal{S}}_k \setminus \mathcal{S}_k$  are limited to a “rim” around the confirmed ones, and since unconfirmed states do not affect the size of the MDD nodes, there is only a small memory and time overhead in practice.

In this on-the-fly version of the saturation algorithm, the arcs of the MDD nodes are labeled only with confirmed states, while our Kronecker encoding of the next-state function must describe all possible transitions from confirmed local states to both confirmed and unconfirmed local states. Thus, we only explore *global symbolic firings* originating in confirmed states.

The on-the-fly algorithm follows exactly the same steps as the pregeneration one, except for the need to confirm local states. Rather than providing the entire pseudocode for this modified algorithm, we show procedure *Confirm* in Figure 4.2, and list the three places where it should be called from the pregeneration pseudocode of Figure 3.9. Between lines 3 and 4 of function *Generate* we add the statement

$$\text{Confirm}(k, 0);$$

to confirm each initial local state, since we number local states starting at 0, that is,  $\psi(s_k) = 0$  for all submodels  $k$ . Between lines 10 and 11 of function *SatFire* we add the statement

$$\text{if } j \notin \mathcal{S}_k \text{ then } \text{Confirm}(k, j);$$

and between lines 12 and 13 of function *SatRecFire* we add the statement

if  $j \notin \mathcal{S}_l$  then *Confirm*( $l, j$ );

to confirm each local state  $j$  (if not already confirmed) after it has been determined that a global state with  $j$  as the  $k^{\text{th}}$ , or  $l^{\text{th}}$ , component can be reached through a symbolic firing.

<i>Confirm</i> ( $k : \text{level}, i : \text{local}$ )	
Add local state index $i$ to $\mathcal{S}_k$ and build the corresponding rows $\mathbf{N}_{k,e}[i, \cdot]$ for all matrices $\mathbf{N}_{k,e} \neq \mathbf{I}$ .	
1. declare $e : \text{event}, j : \text{local}, n : \text{int}, \mathbf{i}_k, \mathbf{j}_k : \text{state};$	
2. $\mathbf{i}_k \leftarrow \psi_k^{-1}(i);$	
3. foreach $e$ s.t. $\mathbf{N}_{k,e} \neq \mathbf{I}$ do	
4.   foreach $\mathbf{j}_k \in \mathcal{N}_{k,e}(\mathbf{i}_k)$ do	• access high-level model
5. $j \leftarrow \psi_k(\mathbf{j}_k);$	
6.     if $j = \text{null}$ then	• $\mathbf{j}_k \notin \widehat{\mathcal{S}}_k$ , new local state
7. $\psi_k(\mathbf{j}_k) \leftarrow  \widehat{\mathcal{S}}_k ;$	• assign next local index to $\mathbf{j}_k$
8. $j \leftarrow  \widehat{\mathcal{S}}_k ;$	
9. $\widehat{\mathcal{S}}_k \leftarrow \widehat{\mathcal{S}}_k \cup \{\mathbf{j}_k\};$	
10. $\mathbf{N}_{k,e}[i, j] \leftarrow 1;$	
11. $\mathcal{S}_k \leftarrow \mathcal{S}_k \cup \{i_k\};$	

**Figure 4.2:** The *Confirm* procedure for saturation on-the-fly.

The firing of an event  $e$  in a local state  $i$  for node  $\langle k|p \rangle$  may lead to a state  $j \in \mathcal{S}_k$  or  $j \in \widehat{\mathcal{S}}_k \setminus \mathcal{S}_k$ . In the former case,  $j$  is already confirmed and row  $j$  of  $\mathbf{N}_{k,e}$  has been built, thus the local states reachable from  $j$  are already in  $\widehat{\mathcal{S}}_k$ . In the latter case,  $j$  is unconfirmed: it is locally, but not necessarily globally, reachable, thus it appears as a column index but has no corresponding row in  $\mathbf{N}_{k,e}$ . Local state  $j$  will be confirmed if the global symbolic firing that used the entry  $\mathbf{N}_{k,e}[i, j]$  is actually possible, i.e, if  $e$  can fire in an MDD path from  $Top(e)$  to  $Bot(e)$  passing through node  $\langle k|p \rangle$ . Only when  $j$  is confirmed, the corresponding rows  $\mathbf{N}_{k,e}[j, \cdot]$  (for all events  $e$  that depend on  $k$ ) are built, using one forward step of *explicit local reachability analysis*. This step must consult the description of the model itself, and thus

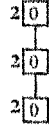
works on actual submodel variables, not state indices. This is the only operation that may discover new unconfirmed local states. Thus, the on-the-fly algorithm uses “rectangular” Kronecker matrices over  $\{0, 1\}^{\mathcal{S}_k \times \widehat{\mathcal{S}}_k}$ . In other words, only confirmed local states “know” their successors, confirmed or not, while unconfirmed states appear only in the columns of the Kronecker matrices.

## 4.2 Example

Figures 4.3-4.5 illustrate the application of the on-the-fly algorithm on our running example. The net is partitioned into three subnets, defined by sets of places  $\{p\}$ ,  $\{q, r\}$ , and  $\{s, t\}$ , respectively. Accordingly, the events are partitioned into  $\mathcal{E}_3 = \{a, b\}$ ,  $\mathcal{E}_2 = \{c\}$ , and  $\mathcal{E}_1 = \{d\}$ . In each snapshot we show the current status of the MDD on the left, and of the Kronecker matrices and local spaces on the right. Local states are described by the number of tokens, listed as superscripts, for each place in the subnet. Their equivalent indices, that is, the mapping  $\psi$ , is also given ( $\equiv$ ), and confirmed local states are underlined. Saturated MDD nodes are shown in dark background, light foreground. For the matrices we adopt a concise description listing only the nonzero entries, in the format *row:col*. This is very close to the actual sparse storage representation we use for each Kronecker matrix.

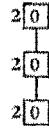
- (a) The algorithm starts with the insertion of the initial state  $(0, 0, 0)$ , encoded with one MDD node per level.
- (b) The initial local states, each indexed 0, are confirmed. Their corresponding rows in the Kronecker matrices are built by consulting the model. This leads to the discovery of new local states. In subspace  $\widehat{\mathcal{S}}_3$ , local state  $p^0$ , indexed 1 is obtained by firing

(a) Initial Configuration:



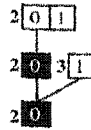
	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
3			I	I
2				I
1	I	I		

$$\begin{aligned} \mathcal{S}_3 &= \{p^1\} \equiv \{0\} \\ \mathcal{S}_2 &= \{q^0 r^0\} \equiv \{0\} \\ \mathcal{S}_1 &= \{s^1 t^0\} \equiv \{0\} \end{aligned}$$

(b) Confirm initial  $p^1 \equiv 0, q^0 r^0 \equiv 0, s^1 t^0 \equiv 0$ :

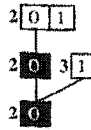
	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
3	0:1	0:2	I	I
2	0:1			I
1	I	I	0:1	

$$\begin{aligned} \mathcal{S}_3 &= \{p^1, p^0, p^2\} \equiv \{0, 1, 2\} \\ \mathcal{S}_2 &= \{q^0 r^0, q^1 r^1\} \equiv \{0, 1\} \\ \mathcal{S}_1 &= \{s^1 t^0, s^0 t^1\} \equiv \{0, 1\} \end{aligned}$$

(c) Saturate  $\langle 3|2 \rangle$  (Fire *a*):

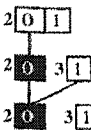
	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
3	0:1	0:2	I	I
2	0:1			I
1	I	I	0:1	

$$\begin{aligned} \mathcal{S}_3 &= \{p^1, p^0, p^2\} \equiv \{0, 1, 2\} \\ \mathcal{S}_2 &= \{q^0 r^0, q^1 r^1\} \equiv \{0, 1\} \\ \mathcal{S}_1 &= \{s^1 t^0, s^0 t^1\} \equiv \{0, 1\} \end{aligned}$$

(d) Confirm  $q^1 r^1 \equiv 1$ :

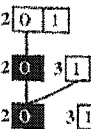
	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
3	0:1	0:2	I	I
2	0:1	1:2	1:3	I
1	I	I	0:1	

$$\begin{aligned} \mathcal{S}_3 &= \{p^1, p^0, p^2\} \equiv \{0, 1, 2\} \\ \mathcal{S}_2 &= \{q^0 r^0, q^1 r^1, q^0 r^1, q^1 r^0\} \equiv \{0, 1, 2, 3\} \\ \mathcal{S}_1 &= \{s^1 t^0, s^0 t^1\} \equiv \{0, 1\} \end{aligned}$$

(e) Saturate  $\langle 2|3 \rangle$  (Fire *c*):

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
3	0:1	0:2	I	I
2	0:1	1:2	1:3	I
1	I	I	0:1	

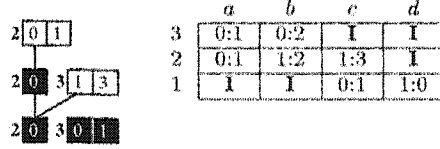
$$\begin{aligned} \mathcal{S}_3 &= \{p^1, p^0, p^2\} \equiv \{0, 1, 2\} \\ \mathcal{S}_2 &= \{q^0 r^0, q^1 r^1, q^0 r^1, q^1 r^0\} \equiv \{0, 1, 2, 3\} \\ \mathcal{S}_1 &= \{s^1 t^0, s^0 t^1\} \equiv \{0, 1\} \end{aligned}$$

(f) Confirm  $s^0 t^1 \equiv 1$ :

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
3	0:1	0:2	I	I
2	0:1	1:2	1:3	I
1	I	I	0:1	1:0

$$\begin{aligned} \mathcal{S}_3 &= \{p^1, p^0, p^2\} \equiv \{0, 1, 2\} \\ \mathcal{S}_2 &= \{q^0 r^0, q^1 r^1, q^0 r^1, q^1 r^0\} \equiv \{0, 1, 2, 3\} \\ \mathcal{S}_1 &= \{s^1 t^0, s^0 t^1\} \equiv \{0, 1\} \end{aligned}$$

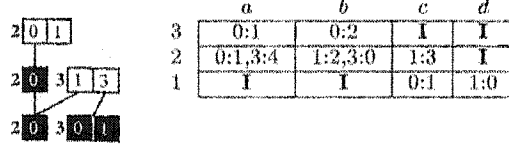
Figure 4.3: Saturation by example (part 1).

(g) Saturate  $\langle 1|3 \rangle$  (Fire  $d$ ):

$$S_3 = \{p^1, p^0, p^2\} \equiv \{0, 1, 2\}$$

$$S_2 = \{q^0 r^0, q^1 r^1, q^0 r^1, q^1 r^0\} \equiv \{0, 1, 2, 3\}$$

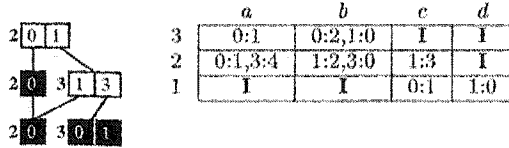
$$S_1 = \{s^1 t^0, s^0 t^1\} \equiv \{0, 1\}$$

(h) Confirm  $q^1 r^0 \equiv 3$ :

$$S_3 = \{p^1, p^0, p^2\} \equiv \{0, 1, 2\}$$

$$S_2 = \{q^0 r^0, q^1 r^1, q^0 r^1, q^1 r^0, q^2 r^1\} \equiv \{0, 1, 2, 3, 4\}$$

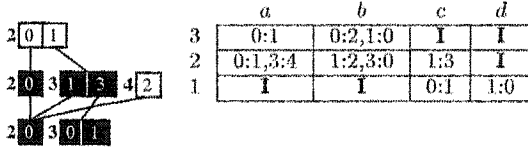
$$S_1 = \{s^1 t^0, s^0 t^1\} \equiv \{0, 1\}$$

(i) Confirm  $p^0 \equiv 1$ :

$$S_3 = \{p^1, p^0, p^2\} \equiv \{0, 1, 2\}$$

$$S_2 = \{q^0 r^0, q^1 r^1, q^0 r^1, q^1 r^0, q^2 r^1\} \equiv \{0, 1, 2, 3, 4\}$$

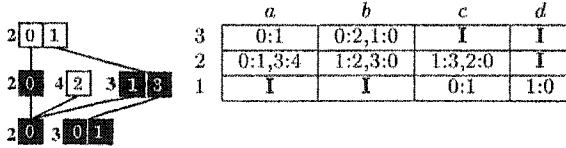
$$S_1 = \{s^1 t^0, s^0 t^1\} \equiv \{0, 1\}$$

(j) Continue Saturating  $\langle 3|2 \rangle$  (Fire  $b$ ):

$$S_3 = \{p^1, p^0, p^2\} \equiv \{0, 1, 2\}$$

$$S_2 = \{q^0 r^0, q^1 r^1, q^0 r^1, q^1 r^0, q^2 r^1\} \equiv \{0, 1, 2, 3, 4\}$$

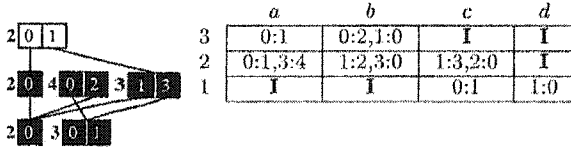
$$S_1 = \{s^1 t^0, s^0 t^1\} \equiv \{0, 1\}$$

(k) Confirm  $q^0 r^1 \equiv 2$ :

$$S_3 = \{p^1, p^0, p^2\} \equiv \{0, 1, 2\}$$

$$S_2 = \{q^0 r^0, q^1 r^1, q^0 r^1, q^1 r^0, q^2 r^1\} \equiv \{0, 1, 2, 3, 4\}$$

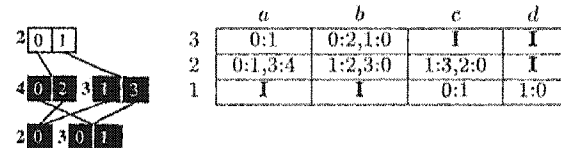
$$S_1 = \{s^1 t^0, s^0 t^1\} \equiv \{0, 1\}$$

(l) Saturate  $\langle 2|4 \rangle$  (Fire  $c$ ):

$$S_3 = \{p^1, p^0, p^2\} \equiv \{0, 1, 2\}$$

$$S_2 = \{q^0 r^0, q^1 r^1, q^0 r^1, q^1 r^0, q^2 r^1\} \equiv \{0, 1, 2, 3, 4\}$$

$$S_1 = \{s^1 t^0, s^0 t^1\} \equiv \{0, 1\}$$

(m) Union  $\langle 2|2 \rangle$  and  $\langle 2|4 \rangle$ :

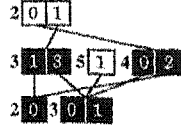
$$S_3 = \{p^1, p^0, p^2\} \equiv \{0, 1, 2\}$$

$$S_2 = \{q^0 r^0, q^1 r^1, q^0 r^1, q^1 r^0, q^2 r^1\} \equiv \{0, 1, 2, 3, 4\}$$

$$S_1 = \{s^1 t^0, s^0 t^1\} \equiv \{0, 1\}$$

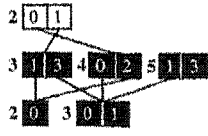
Figure 4.4: Saturation by example (part 2).



(n) Continue Saturating  $\langle 3|2 \rangle$  (Fire  $a$ ):

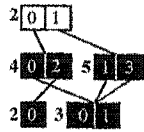
	$a$	$b$	$c$	$d$
3	0:1	0:2,1:0	I	I
2	0:1,3:4	1:2,3:0	1:3,2:0	I
1	I	I	0:1	1:0

$$\begin{aligned} S_3 &= \{p^1, p^0, p^2\} \equiv \{0, 1, 2\} \\ S_2 &= \{q^0 r^0, q^1 r^1, q^0 r^1, q^1 r^0, q^2 r^1\} \equiv \{0, 1, 2, 3, 4\} \\ S_1 &= \{s^1 t^0, s^0 t^1\} \equiv \{0, 1\} \end{aligned}$$

(o) Saturate  $\langle 2|5 \rangle$  (Fire  $c$ ):

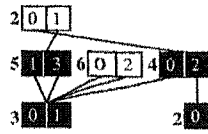
	$a$	$b$	$c$	$d$
3	0:1	0:2,1:0	I	I
2	0:1,3:4	1:2,3:0	1:3,2:0	I
1	I	I	0:1	1:0

$$\begin{aligned} S_3 &= \{p^1, p^0, p^2\} \equiv \{0, 1, 2\} \\ S_2 &= \{q^0 r^0, q^1 r^1, q^0 r^1, q^1 r^0, q^2 r^1\} \equiv \{0, 1, 2, 3, 4\} \\ S_1 &= \{s^1 t^0, s^0 t^1\} \equiv \{0, 1\} \end{aligned}$$

(p) Union  $\langle 2|3 \rangle$  and  $\langle 2|5 \rangle$ :

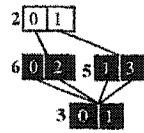
	$a$	$b$	$c$	$d$
3	0:1	0:2,1:0	I	I
2	0:1,3:4	1:2,3:0	1:3,2:0	I
1	I	I	0:1	1:0

$$\begin{aligned} S_3 &= \{p^1, p^0, p^2\} \equiv \{0, 1, 2\} \\ S_2 &= \{q^0 r^0, q^1 r^1, q^0 r^1, q^1 r^0, q^2 r^1\} \equiv \{0, 1, 2, 3, 4\} \\ S_1 &= \{s^1 t^0, s^0 t^1\} \equiv \{0, 1\} \end{aligned}$$

(q) Continue Saturating  $\langle 3|2 \rangle$  (Fire  $b$ ):

	$a$	$b$	$c$	$d$
3	0:1	0:2,1:0	I	I
2	0:1,3:4	1:2,3:0	1:3,2:0	I
1	I	I	0:1	1:0

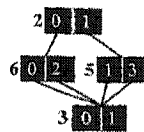
$$\begin{aligned} S_3 &= \{p^1, p^0, p^2\} \equiv \{0, 1, 2\} \\ S_2 &= \{q^0 r^0, q^1 r^1, q^0 r^1, q^1 r^0, q^2 r^1\} \equiv \{0, 1, 2, 3, 4\} \\ S_1 &= \{s^1 t^0, s^0 t^1\} \equiv \{0, 1\} \end{aligned}$$

(r) Union  $\langle 2|4 \rangle$  and  $\langle 2|6 \rangle$ :

	$a$	$b$	$c$	$d$
3	0:1	0:2,1:0	I	I
2	0:1,3:4	1:2,3:0	1:3,2:0	I
1	I	I	0:1	1:0

$$\begin{aligned} S_3 &= \{p^1, p^0, p^2\} \equiv \{0, 1, 2\} \\ S_2 &= \{q^0 r^0, q^1 r^1, q^0 r^1, q^1 r^0, q^2 r^1\} \equiv \{0, 1, 2, 3, 4\} \\ S_1 &= \{s^1 t^0, s^0 t^1\} \equiv \{0, 1\} \end{aligned}$$

(s) Discard unreachable local states:



	$a$	$b$	$c$	$d$
3	0:1	1:0	I	I
2	0:1	1:2,3:0	1:3,2:0	I
1	I	I	0:1	1:0

$$\begin{aligned} S_3 &= \{p^1, p^0\} \equiv \{0, 1\} \\ S_2 &= \{q^0 r^0, q^1 r^1, q^0 r^1, q^1 r^0\} \equiv \{0, 1, 2, 3\} \\ S_1 &= \{s^1 t^0, s^0 t^1\} \equiv \{0, 1\} \end{aligned}$$

Figure 4.5: Saturation by example (part 3).

event  $a$ , and local state  $p^2$ , indexed 2, by firing event  $b$ . Similarly, in subspaces  $\widehat{\mathcal{S}}^2$  and  $\widehat{\mathcal{S}}_1$ , local states  $q^1r^1$  and  $s^0t^1$  are discovered from the initial local state, and are each indexed 1.

- (c) The saturation process starts with the bottom node  $\langle 1|2 \rangle$ , for which no event is enabled, hence it is marked saturated. The same holds for node  $\langle 2|2 \rangle$ . Moving up, we begin saturating the root node,  $\langle 3|2 \rangle$ . Event  $a$  is enabled at level 3 by local state 0.  $SatFire(a, 3, 2)$  calls  $SatRecFire(a, 2, \langle 3|2 \rangle[0])$ , which creates a new node,  $\langle 2|3 \rangle$ . Since local state 0 at level 2 enables  $a$ , the recursive call continues downstream with  $SatRecFire(a, 1, \langle 2|2 \rangle[0])$ . This returns index 2 =  $\langle 2|2 \rangle[0]$ , since  $1 < Bot(a)$ .
- (d) Upon return, an arc is set from  $\langle 2|3 \rangle[1]$  to  $\langle 1|2 \rangle$ , meaning the unconfirmed local state  $q^1r^1 \equiv 1$  has been globally reached and the *Confirm* procedure is called on substate  $q^1r^1$ . As a result, two new local states,  $q^0r^1$  and  $q^1r^0$ , are discovered by locally firing events  $b$  and  $c$ , respectively, and indexed 2 and 3 in  $\widehat{\mathcal{S}}_2$ . The pairs 1:2 and 1:3 are added to the corresponding matrices,  $\mathbf{N}_{2,b}$  and  $\mathbf{N}_{2,c}$ .
- (e) At this point,  $SatRecFire(a, 2, 2)$  has the result of firing  $a$  below level 2, encoded by  $\langle 2|3 \rangle$ , which we need to saturate. The only event in  $\mathcal{E}_2$ ,  $c$ , is enabled by local state 1, and moves subsystem 2 to local state 3, according to  $\mathbf{N}_{2,c}$ .  $SatFire(c, 2, 3)$  makes the recursive call  $SatRecFire(c, 1, \langle 2|3 \rangle[1])$ . This creates the node  $\langle 1|3 \rangle$ , and sets its arc  $\langle 1|3 \rangle[1]$  to  $\langle 0|1 \rangle$ , the base case result of  $SatRecFire(c, 0, 1)$ .
- (f) Local state  $s^0t^1 \equiv 1$  has been globally reached and has to be confirmed in  $\widehat{\mathcal{S}}_1$ . The only possible move discovered from this state is via event  $d$ , and leads submodel 1 to the already known local state  $s^1t^0 \equiv 0$ .

- (g) Since  $\langle 1|3 \rangle$  is a new node, it is immediately saturated, by firing event  $d$ , the only element of  $\mathcal{E}_1$ . This adds the arc  $\langle 1|3 \rangle[0]$  to point to the terminal node  $\langle 0|1 \rangle$ . Local state 0 at level 1 is an old state, hence it does not need to be confirmed again. Node  $\langle 1|3 \rangle$  is declared saturated and returned as the result of  $SatRecFire(c, 1, 2)$ . An arc is set from  $\langle 2|3 \rangle[3]$  to  $\langle 1|3 \rangle$  to represent the local move from local state 1 to local state 3 in  $\widehat{\mathcal{S}}_2$ , as demanded by  $\mathbf{N}_{2,c}$ .
- (h) Local state  $q^1 r^0 \equiv 3$  now has to be confirmed. It leads to the already known state  $q^0 r^0 \equiv 0$  via  $b$ , and a new local state  $q^2 r^1$ , which is indexed 4, via  $a$ . Node  $\langle 2|3 \rangle$  is now saturated and it is returned as the result of  $SatRecFire(a, 2, 2)$ , which started in snapshot (c).
- (i) Arc  $\langle 3|2 \rangle[1]$  is set to point to  $\langle 2|3 \rangle$ , signifying that local state  $p^0 \equiv 1$  has been reached. The confirmation of it in  $\widehat{\mathcal{S}}_3$  adds one entry in the Kronecker matrices: to  $p^1 \equiv 0$  in  $\mathbf{N}_{3,b}$ .
- (j) The saturation of node  $\langle 3|2 \rangle$  continues with the firing of the other event in  $\mathcal{E}_3$ ,  $b$ .  $SatFire(b, 3, 2)$  is enabled by both local states at level 3 represented in the node, 0 and 1. The first subsequent call,  $SatRecFire(b, 2, \langle 3|2 \rangle[0])$  finds no local states at level 2 enabled in node  $\langle 3|2 \rangle[0] = \langle 2|2 \rangle$ . The recursion stops and returns index 0, signaling the failure to fire event  $b$  from this particular combination of local states. The second call,  $SatRecFire(b, 2, \langle 3|2 \rangle[1])$  is successful in firing  $b$ , based on the enabling pattern  $(1, 1, *)$ , and creates node  $\langle 2|4 \rangle$ .
- (k) Since local state  $q^0 r^1 \equiv 2$  is reached in this firing,  $q^0 r^1$  needs to be confirmed in  $\widehat{\mathcal{S}}_2$ . The only move discovered is to  $q^0 r^0 \equiv 0$ , via event  $c$ . The corresponding element,

1:0, is inserted in  $\mathbf{N}_{2,c}$ .

- (l) After setting the arc  $\langle 2|4 \rangle[2]$  to  $\langle 1|3 \rangle$ , node  $\langle 2|4 \rangle$  has to be saturated. Event  $c$  is enabled, and its firing causes node  $\langle 2|4 \rangle$  to be updated by setting its arc  $\langle 2|4 \rangle[0]$  to  $\langle 1|2 \rangle$ . No other firings produce changes to the node, which is returned as the result of *SatRecFire* ( $b, 2, \langle 3|2 \rangle[1]$ ).
- (m) Upon return, node  $\langle 3|2 \rangle$  is updated in-place to incorporate the effect of the recursive calls. This requires a union operation between its existing child in position 0,  $\langle 2|2 \rangle$ , and  $\langle 2|4 \rangle$ . The result of the union is  $\langle 2|4 \rangle$ . The arc  $\langle 3|2 \rangle[0]$  is updated accordingly, causing node  $\langle 2|2 \rangle$  to become disconnected and, therefore, removed from the diagram.
- (n) Even though both events  $a$  and  $b$  have already been fired in  $\langle 3|2 \rangle$ , the saturation of the root node is not complete, since both firings produced new states. The loop continues and event  $a$  is fired again, which induces the creation of node  $\langle 2|5 \rangle$  at level 2.
- (o) Node  $\langle 2|5 \rangle$  is saturated by firing event  $c$ , and adding in-place the arc  $\langle 2|5 \rangle[3]$  to  $\langle 1|3 \rangle$ .
- (p) The union of  $\langle 2|3 \rangle$  and  $\langle 2|5 \rangle$ , required by the in-place update of the root, is node  $\langle 2|5 \rangle$ , which is known to be saturated, as it represents the union of two saturated nodes. Node  $\langle 2|5 \rangle$  is the new successor of the root along arc 1. The old successor, node  $\langle 2|3 \rangle$ , becomes disconnected and is deleted.
- (q) Finally, we fire event  $b$  one more time and set  $\langle 3|2 \rangle[0]$  to the result of the union between  $\langle 2|4 \rangle$  and newly created  $\langle 2|6 \rangle$ .
- (r) The union of the two nodes is encoded by  $\langle 2|6 \rangle$ . Further attempts to fire  $a$  or  $b$  in the root do not discover any other new states, hence the root is finally saturated.

- (s) The algorithm concludes by discarding all local states still unconfirmed. These are  $p^2 \equiv 2$ , at level 3, and  $q^2r^1 \equiv 4$ , at level 2.

### 4.3 Implementation issues

#### 4.3.1 Data structures for the next-state function

We store the next-state function  $\mathcal{N}$  of the model using a sparse matrix  $\mathbf{N}$ , whose entries are themselves sparse matrices. The rows of  $\mathbf{N}$  correspond to the levels of the model. The columns of  $\mathbf{N}$  correspond to the events of the model. Each entry of  $\mathbf{N}$  contains a sparse matrix  $\mathbf{N}_{k,e}$  describing the effect on level  $k$  of firing event  $e$ . Storing  $\mathbf{N}$  as a full two-dimensional array would waste a significant amount of memory, since most  $\mathbf{N}_{k,e}$  are identities, so we use a sparse encoding where a missing entry means “identity”, not “zero”.

We provide sparse access using linked lists by columns (so that we can efficiently access all levels affected by an event  $e$ ), and by rows (so that we can efficiently access all events affecting a level  $k$ ). These two types of access are required by the most common and expensive operations of state space generation, *firing* an event and *confirming* that a local state is globally reachable, respectively. The resulting data structure is shown in Figure 4.6.

Each node in the figure consists of a pointer to the data structure storing  $\mathbf{N}_{k,e}$ , plus pointers to the next elements along its row and along its column. Each  $\mathbf{N}_{k,e}$ , in turn, is stored as a sparse row-wise matrix [114], since the only required access to  $\mathbf{N}_{k,e}$  is the retrieval of all local state indices  $j$  such that  $\mathbf{N}_{k,e}[i, j] = 1$ , for a given  $i$ . The only implementation difficulty is that the matrix size must be dynamic, since as we discover local states on-the-fly, we need to add new rows to it (adding new columns is not an issue, since, in a sparse

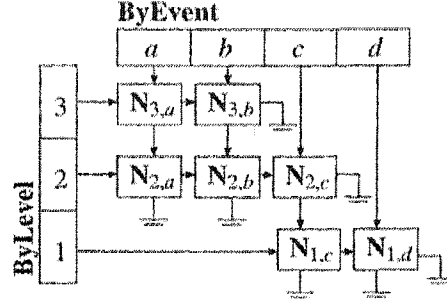


Figure 4.6: Storage for the matrix nodes.

row-wise representation, column indices appear only as values in the entries for each row).

#### 4.3.2 Unconfirmed vs. confirmed states

With the exception of the first local state for each level, which is assigned index 0 and is confirmed by definition, any other local state we find is always unconfirmed at first. Yet, this unconfirmed state  $\mathbf{j}_k$  must be referenced as a column index  $j = \psi_k(\mathbf{j}_k)$  on a confirmed state row  $i$ , that is, we need to be able to set  $\mathbf{N}_{k,e}[i, j] = 1$ . It is natural and efficient to assign increasing sequential state indices to the states in  $\widehat{\mathcal{S}}_k$ , but, unfortunately, not all states in  $\widehat{\mathcal{S}}_k$  will be necessarily confirmed. Worse yet, even those that will be confirmed are not guaranteed to be confirmed in the order they are discovered.

We could ignore this problem and simply index local states using  $\psi_k : \widehat{\mathcal{S}}_k \rightarrow \{0, \dots, \widehat{n}_k - 1\}$  everywhere, where  $\widehat{n}_k = |\widehat{\mathcal{S}}_k|$ , obviously for the currently-known  $\widehat{\mathcal{S}}_k$ . This would be indeed correct, but inefficient in two ways. First, the sparse row-wise storage of each  $\mathbf{N}_{k,e}$  would require  $\widehat{n}_k$  pointers to the rows, while we know that only  $n_k = |\mathcal{S}_k|$  rows corresponding to the confirmed local states need to be built and accessed, thus the remaining  $\widehat{n}_k - n_k$  row pointers would simply be null. Second, each MDD node at level  $k$  would require  $\widehat{n}_k$  arcs,

while, again, we know that only the subset corresponding to states in  $\mathcal{S}_k$  may be pointing to nodes other than  $\langle k-1|0\rangle$ . This second problem can potentially have a large impact, since it may substantially increase the memory requirements, depending on how the MDD nodes are stored.

To avoid these inefficiencies, we then use a second mapping  $\Phi_k : \{0, \dots, \widehat{n}_k - 1\} \rightarrow \{0, \dots, n_k - 1\} \cup \{\text{null}\}$  from “unconfirmed indices” to “confirmed indices”. A local state  $\mathbf{j}_k$ , then, has an unconfirmed index  $j = \psi_k(\mathbf{j}_k)$ , which is used exclusively for column indices in  $\mathbf{N}_{k,e}$ . Once it is confirmed, though, it also has a confirmed index  $j' = \Phi_k(j)$ , which is used as a row index in  $\mathbf{N}_{k,e}$  and as an arc label in the MDD nodes at level  $k$ . Thus, to test whether a local state  $\mathbf{j}_k \in \widehat{\mathcal{S}}$  is confirmed given its unconfirmed index  $j$ , we only need to test whether  $\Phi_k(j) \neq \text{null}$ .

When state-space generation is complete, we know that any unconfirmed state can never be reached, so we:

- Discard the states in  $\widehat{\mathcal{S}}_k \setminus \mathcal{S}_k$ , keep only those in  $\mathcal{S}_k$ .
- Redefine  $\psi_k$ , so that  $\psi_k : \mathcal{S}_k \rightarrow \{0, \dots, n_k - 1\}$ .
- Examine the matrices  $\mathbf{N}_{k,e}$  and, for each of them, delete any entry  $\mathbf{N}_{k,e}[i, j]$  such that  $\Phi_k(j) = \text{null}$ .
- Switch from unconfirmed to confirmed indices in the column indices, i.e, change the entries of each  $\mathbf{N}_{k,e}$  so that  $\mathbf{N}_{k,e}[i, \Phi_k(j)] = 1$  if  $\mathbf{N}_{k,e}[i, j]$  was 1.
- Discard the array used to store the  $\Phi_k$  mapping.

### 4.3.3 Storing MDD nodes

To store the state space, we use a quasi-reduced MDD, which naturally lends itself to a level-oriented node storage. In our implementation, the data for an MDD node  $\langle k|p \rangle$  is divided into two extensible arrays  $nodes_k$  and  $arcs_k$  associated with level  $k$  (Figure 4.7). Entry  $nodes_k[p]$  occupies the same number of bytes for each node (at any level) and stores the following data:

- an integer *offset* into extensible array  $arcs_k$ ,
- an integer *size* describing the length of the portion array used by the node,
- a *count* of incoming arcs,
- and the boolean flags *saturated*, *shared*, *sparse*, and *deleted* (the first two are required only when SMART manages multiple MDDs on the same potential state space  $\mathcal{S}$ , e.g., for Model Checking).

The data for the arcs of node  $\langle k|p \rangle$  occupies instead a variable-size portion of array  $arcs_k$ . If  $nodes_k[p].sparse = true$ , the arcs of node  $\langle k|p \rangle$  are stored in  $2 \cdot nodes_k[p].size$  entries of  $arcs_k$  starting in position  $nodes_k[p].offset$ , as pairs of *local* and *index* values, meaning that  $\langle k|p \rangle[local] = index$ . If instead  $nodes_k[p].sparse = false$ , “truncated full” storage is used, that is,  $arcs_k[x] = q$  if  $\langle k|p \rangle[x - nodes_k[p].offset] = q$ . However, not all  $n_k$  arcs need to be stored: if the last  $m$  arcs are 0, i.e.,  $\langle k|p \rangle[n_k - m] = \dots = \langle k|p \rangle[n_k - 1] = 0$ ,  $nodes_k[p].size$  is set to  $n_k - m$ .



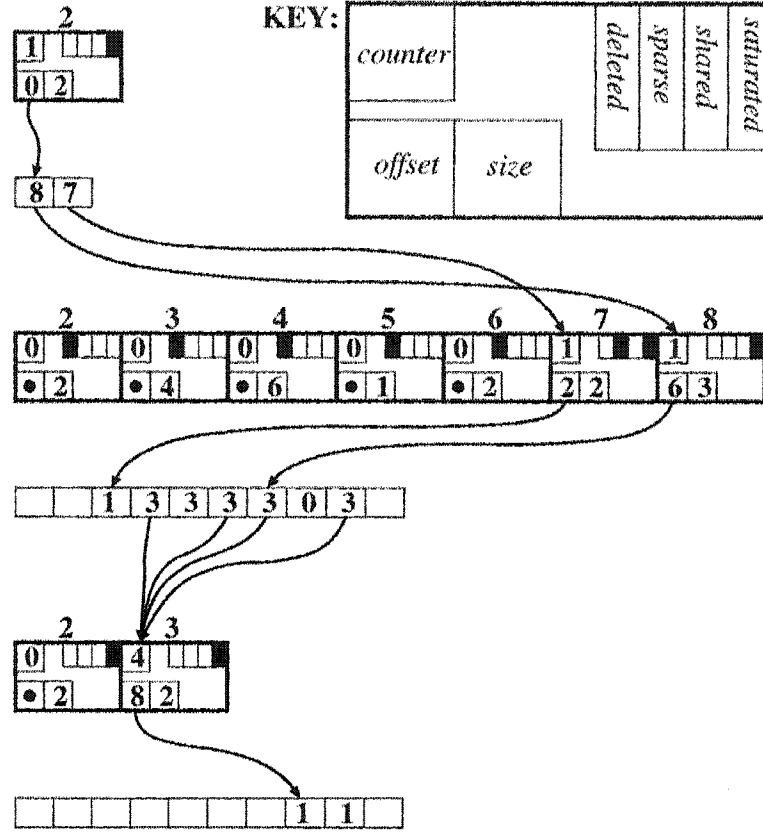


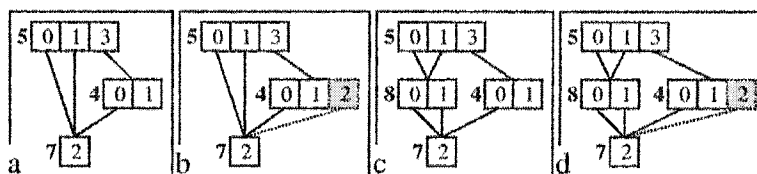
Figure 4.7: Node and arc storage for MDDs.

#### 4.3.4 MDD nodes of variable size

We keep track of the index of the last node in  $nodes_k$  and of the last used position in  $arcs_k$ . Requests for a new node at level  $k$  are satisfied by using the next available position in these arrays. Thus, the node being saturated is always at the end of the array, and so are its arcs. This is particularly important with saturation on-the-fly, where we do not know beforehand how many entries in  $arcs_k$  are needed to store the arcs of a node being saturated.

A fundamental property of our encoding is that a saturated node  $\langle k|p \rangle$  remains saturated and encodes the same set even after a new state  $i$  is added to  $S_k$ . This is because, regardless

of whether the node is using sparse or truncated full storage, it will be automatically and correctly interpreted as having  $\langle k|p\rangle[i] = 0$ . However, the possible growth of  $\mathcal{S}_k$  implies that it becomes hard to exploit the special meaning for node  $\langle k|1\rangle$ , which, we recall is  $\mathcal{B}(\langle k|1\rangle) = \mathcal{S}_k \times \cdots \times \mathcal{S}_1$ . With pregeneration, this optimization speeds up computation whenever node  $\langle k|1\rangle$  is involved in a union, since we immediately conclude that  $\mathcal{B}(\langle k|1\rangle) \cup \mathcal{B}(\langle k|p\rangle) = \mathcal{B}(\langle k|1\rangle)$ . Further, such nodes need not be explicitly stored.



**Figure 4.8:** The implicit index 1 in reduced vs. quasi-reduced MDDs.

To reserve index 1 for the same purpose with the on-the-fly approach is possible, but problematic, since, whenever a new state is added to  $\mathcal{S}_l$ , for  $l \leq k$ , the meaning of  $\mathcal{B}(\langle k|1 \rangle)$  implicitly changes. This is one of the reasons that led us to use *quasi-reduced* instead of *reduced* MDDs. The latter eliminates redundant nodes and is potentially more efficient, but its arcs can span multiple levels. As discussed in [26], such arcs are more difficult to manage and can yield a slower state-space generation when exploiting locality. With the on-the-fly algorithm, they create an even worse problem: they become “incorrect” when a local state space grows. For example, both the reduced and the quasi-reduced 3-level MDDs in Figure 4.8(a) and (c) encode the state space

$$S = \{(0, 0, 2), (0, 1, 2), (1, 0, 2), (1, 1, 2), (3, 0, 2)\},$$

when  $\mathcal{S}_3 = \{0, 1, 2, 3\}$ ,  $\mathcal{S}_2 = \{0, 1\}$ , and  $\mathcal{S}_1 = \{0, 1, 2\}$ . If we want to add global state

$(3, 2, 2)$  to  $\mathcal{S}$ , we need to add local state 2 to  $\mathcal{S}_2$  and set arc  $\langle 2|4 \rangle [2]$  to  $\langle 1|7 \rangle$ . However, while the resulting quasi-reduced MDD in (d) is correct, the reduced one in (b) is not, since now it also encodes global states  $(0, 2, 2)$  and  $(1, 2, 2)$ . To fix the problem, we could reintroduce the formerly-redundant node  $\langle 2|8 \rangle$  so that the new reduced and quasi-reduced MDDs coincide. While it would be possible to modify the MDD to obtain a correct reduced ordered MDD in this manner whenever a local state space grows, the cost of doing so is unjustifiably high. Of course, we can still reserve index 0 for the empty set, and exploit the relation  $\mathcal{B}(\langle k|0 \rangle) \cup \mathcal{B}(\langle k|p \rangle) = \mathcal{B}(\langle k|p \rangle)$ , since the representation of the empty set does not change as the local state spaces expand.

#### 4.3.5 Garbage collection

If a node  $\langle k|p \rangle$  becomes disconnected, we “mark it for deletion” by setting its flag  $nodes_k[p]$ . *deleted* to *true*. However, the node itself remains in memory until the garbage collection manager performs a cleanup. Such a node can then be “resuscitated” by simply resetting this flag. This occurs if a reference to it is retrieved as the result of an operation cache lookup. Of course, if a cleanup is issued after marking  $\langle k|p \rangle$  for deletion and prior to the cache lookup, there will be a cache lookup miss, since all entries referring to nodes marked for deletion are eliminated when these nodes are actually deleted.

One problem with MDD nodes of variable sizes is that the “hole” left in  $arcs_k$  when a node is deleted can be smaller than what is needed for a new node, thus it cannot be easily “recycled”. However, the use of separate  $nodes_k$  and  $arcs_k$  arrays affords us great flexibility in our garbage collection strategy. In particular, we can “compact to the left” the entire  $arcs_k$  array any time the memory used by its holes exceed some threshold (standard value

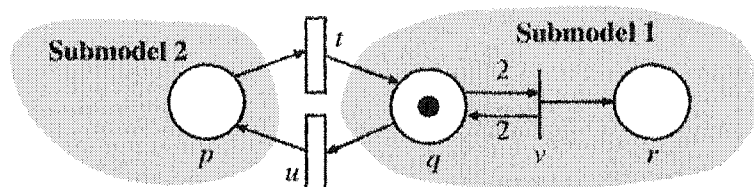
10%) without having to access nodes at other levels. The space freed at the right end of  $arcs_k$  can then be reused for new nodes. This compaction only requires to re-adjust the values of each  $nodes_k[p].offset$ , for each level- $k$  node. There are several ways to do this.

One approach is to store, together with the chunk for  $\langle k|p \rangle$  in  $arcs_k$ , a back pointer to the node itself, i.e, the value  $p$ . This allows compaction of  $arcs_k$  via a linear scan: valid values may be shifted to the left over invalid values until all holes have been removed. Simultaneously, using the back pointer  $p$ , we access and update the value  $nodes_k[p].offset$  after shifting the corresponding arcs. With respect to our previous pregeneration implementation, where array  $arcs_k$  is not used because the arcs are stored directly in  $nodes_k$  as arrays of fixed-size  $n_k$ , this requires 12 additional bytes per node, for  $offset$  and  $size$  in  $nodes_k$  and for the back pointer in  $arcs_k$ . However, it can also save memory, since we can now employ sparse or truncated full storage.

A better alternative is to build a second, temporary, array  $tmp$  when compacting the arcs at level  $k$  by scanning the nodes sequentially and copying the arcs of each non-deleted node from  $arcs_k$  to  $tmp$ . When we are done copying,  $tmp$  becomes our new compacted  $arcs_k$  array, and the old  $arcs_k$  array is deleted. This additional array  $tmp$  temporarily increases the memory requirements, but avoids the need for back pointers, saving four bytes per node.

In addition to compacting the  $arcs_k$  array, we also need to regularly, and independently, compact the  $nodes_k$  array. This can also be done by “compacting to the left” over nodes marked for deletion but, this now means that what was node  $\langle k|p \rangle$  is now node  $\langle k|q \rangle$ , with  $q < p$ . When performing this compaction, we build a temporary indirection array  $old2new$  of size equal to the old number of nodes, so that  $old2new[p] = q$ . Then, once compaction is completed, we scan the arcs in the level above,  $arcs_{k+1}$ , and change each occurrence of

$p$  into  $q$  using a lookup into *old2new*. Finally, *old2new* is destroyed when this scan has completed.



**Figure 4.9:** Example of potential, but not actual, overflow of a local state space.

### 4.3.6 Overflow of potential local state spaces

Our new algorithm eliminates the need to specify additional constraints for any formalism where each state can reach a finite number of states in a single step. A subtle problem remains, however, if an infinite number of states can be reached in one step. For example, in Generalized Stochastic Petri Nets [2], *immediate* transitions, such as  $v$  in Figure 4.9, are processed not by themselves, but as events that can take place instantaneously after the firing of *timed* transitions, such as  $t$  and  $u$  (somewhat analogous to *internal* events in process algebra). In Figure 4.9, we partition the net into submodel 2, containing place  $p$ , and submodel 1, containing places  $q$  and  $r$ .

The initial local states are  $(p^0)$  and  $(q^1r^0)$ , respectively. When the latter state is confirmed into  $S_1$ , an explicit local exploration begins. Transition  $t$  can fire in submodel 1 in isolation, leading to marking  $(q^2r^0)$ . This enables immediate transition  $v$  which, processed right away as part of the firing of  $t$ , leads to markings  $(q^2r^1)$ ,  $(q^2r^2)$ ,  $(q^2r^3)$ , ... and so on. Thus, the explicit local exploration fails with an overflow in place  $r$ , while a traditional explicit global exploration would not, since it would never reach a global marking with two

N	Reachable states	Final memory (KB)			Peak memory (KB)			Time (sec)		
		Otf	Pre	NuSMV	Otf	Pre	NuSMV	Otf	Pre	NuSMV
Dining Philosophers: $K = N$ , $ S_k  = 34$ for all $k$										
20	$3.46 \times 10^{12}$	4	3	4,178	5	4	4,192	0.01	0.01	0.4
50	$2.23 \times 10^{31}$	11	10	8,847	14	12	8,863	0.03	0.02	13.1
100	$4.97 \times 10^{62}$	24	20	8,891	28	25	15,256	0.06	0.05	990.8
200	$2.47 \times 10^{125}$	48	40	21,618	57	50	59,423	0.15	0.11	18,129.3
5,000	$6.53 \times 10^{3134}$	1,210	1,015	—	1,445	1,269	—	65.55	51.29	—
Slotted Ring Network: $K = N$ , $ S_k  = 15$ for all $k$										
5	$5.39 \times 10^4$	1	1	502	5	5	507	0.01	0.01	0.1
10	$8.29 \times 10^9$	5	5	4,332	28	27	8,863	0.06	0.04	6.1
15	$1.46 \times 10^{15}$	10	9	771	80	77	11,054	0.18	0.13	2,853.1
100	$2.60 \times 10^{105}$	434	398	—	15,753	14,486	—	41.72	25.78	—
Round Robin Mutual Exclusion: $K = N + 1$ , $ S_k  = 10$ for all $k$ except $ S_1  = N + 1$										
10	$2.30 \times 10^4$	5	5	917	6	7	932	0.01	0.01	0.2
20	$4.72 \times 10^7$	18	17	5,980	20	21	5,985	0.04	0.03	1.4
30	$7.25 \times 10^{10}$	37	36	2,222	41	41	8,716	0.09	0.07	5.6
100	$2.85 \times 10^{32}$	357	355	13,789	372	372	21,814	2.11	1.55	2,836.5
150	$4.82 \times 10^{47}$	784	781	—	807	807	—	7.04	5.07	—
FMS: $K = 19$ , $ S_k  = N + 1$ for all $k$ except $ S_{17}  = 4$ , $ S_{12}  = 3$ , $ S_7  = 2$										
5	$1.92 \times 10^4$	5	6	2,113	6	9	2,126	0.01	0.01	1.0
10	$2.50 \times 10^9$	16	19	1,152	26	31	8,928	0.02	0.02	41.6
25	$8.54 \times 10^{13}$	86	135	17,045	163	239	152,253	0.16	0.11	17,321.9
150	$4.84 \times 10^{23}$	6,291	15,459	—	16,140	29,998	—	18.50	10.92	—

**Table 4.1:** Generation of the state space: On-the-fly vs. pregeneration vs. NuSMV

tokens in  $q$ . This situation is quite artificial, however. It can occur only if the formalism allows a state to reach an infinite number of states in one “timed step”.

## 4.4 Results

We compared the space and runtime required by our new algorithm with those of its pregeneration predecessor [27] and of NuSMV [33], a symbolic verifier built on top of the CUDD library [127]. The benchmark of examples include: dining philosophers, slotted ring, round robin mutual exclusion, and flexible manufacturing system (FMS).

Table 4.1 lists the peak and final memory and the runtime for these algorithms. For

comparison's sake, we assume that a BDD node in NuSMV uses 16 bytes. To be fair, we point out that our memory consumption refers to the MDD only, while we believe that the number of nodes reported by NuSMV includes also those for the next-state function. However, our Kronecker encoding for  $\mathcal{N}$  is extremely efficient, requiring at most 300KB in any model, except for the model with 5,000 dining philosophers, where it requires 5.2MB. The memory for the operation caches is not included in either our or NuSMV results (for our algorithms, caches never exceeded 20MB on these examples). Both the on-the-fly and the pregeneration MDD algorithms are able to handle significantly larger models than NuSMV. We show the largest value of  $N$  for each of the four models where generation was possible, in the penultimate row for NuSMV and the last row for SMART. When comparisons with NuSMV can be made, our algorithms show speed-up ratios over 100,000 and memory reduction ratios over 1,000.

The results also demonstrate that the overhead of the on-the-fly algorithm versus pregeneration is acceptable. Moreover, the additional 12-byte per node memory overhead required to manage dynamically-sized nodes at a given level  $k$  can be offset by the ability to store nodes with  $m < n_k$  arcs (because they were created when  $\mathcal{S}_k$  contained only  $m$  states, or because the last  $n_k - m$  arcs point to  $\langle k-1|0 \rangle$  and are truncated). In fact, for the FMS model, this results in smaller memory requirements than with pregeneration, suggesting that the use of sparse nodes is advantageous in models with large local state spaces. Even if our on-the-fly implementation is not yet as optimized as that of pregeneration, the runtime of the on-the-fly algorithm is still excellent, at most 70% over that of pregeneration. This is a good tradeoff, given the increase in modeling ease and safety afforded by not having to worry about local state space generation in isolation.

## Chapter 5

# Structural CTL Model Checking

Having defined the abstract concept of a *model* and its *state space* in Chapter 2, next we need a *theory* that makes the model meaningful, by asserting some type of observations about the constructed model. These assertions will be evaluated in the form of truth or falsehood, therefore the theory will be called *logic*. Moreover, when the evolution of the model in time is studied, this is called *temporal logic* [22, 59, 95, 100, 103, 116, 118, 122].

The simplest and most intuitive mathematical structure that captures the above elements is a Kripke structure [91]. This is nothing but a discrete state model endowed with a labeling function of its states with *atomic propositions*. The propositions can be viewed as basic properties, evaluating to *true* or *false*, that can be observed of each state, hence they are given as axioms before the verification process starts. Formally:

**Definition 5.0.1** *Given a finite set of atomic propositions  $\mathcal{P}$ , a Kripke structure is a quadruple  $(\hat{S}, \mathbf{s}, \mathcal{N}, L)$ , consisting of the discrete state system  $(\hat{S}, \mathbf{s}, \mathcal{N})$  and the labeling function  $L : \hat{S} \rightarrow 2^{\mathcal{P}}$ , that assigns to each state the set of propositions satisfied by that state.*

**Definition 5.0.2** *A computation of a Kripke structure  $(\hat{S}, \mathbf{s}, \mathcal{N}, L)$ , is a path (a sequence of states)  $\pi = \mathbf{i}^0 \mathbf{i}^1 \dots \mathbf{i}^n$  such that  $\mathbf{i}^0 = \mathbf{s}$  and  $\mathbf{i}^{j+1} \in \mathcal{N}(\mathbf{i}^j)$ , for all  $0 \leq j < n$ .*



**Definition 5.0.3** *The reachability set of a Kripke structure is the set of states  $S = \{\mathbf{i} \in \widehat{S} \mid \text{exists a path } \pi \text{ from the initial state } \mathbf{s} \text{ to } \mathbf{i}\}$ .*

The traditional definition of a Kripke structure requires that the next state function  $\mathcal{N}$  be total (in the sense that for each state  $\mathbf{i}$  there always exists a successor for it,  $\mathcal{N}(\mathbf{i}) \neq \emptyset$ ). We omitted this requirement from the definition, since this is rather reminiscent from the reactive systems, where the evolution of the system is considered infinite (through a set of finite states, though). We have generalized the classical definition to arbitrary transition relations, since one can always introduce back the infinite paths by adding self loops to absorbing states.

## 5.1 Temporal logic

Given a Kripke structure  $(\widehat{S}, \mathbf{s}, \mathcal{N}, L)$ , over a set of propositions  $\mathcal{P}$ , there is a *temporal relation*  $\prec$  that can be established over the set  $\widehat{S}$ . We say that state  $\mathbf{i}$  precedes states  $\mathbf{j}$ ,  $\mathbf{i} \prec \mathbf{j}$  if there is a sequence of transitions that leads from  $\mathbf{i}$  to  $\mathbf{j}$ . This temporal relation is only transitive, but not necessarily reflexive, symmetric, or antisymmetric, since the reachability graph may contain cycles. If  $\mathbf{i}$  is the current state, the “future” states are all those that can be reached from  $\mathbf{i}$  by a sequence of transitions (transitive closure of  $\mathcal{N}$ ). Conversely, the “past” states of  $\mathbf{i}$  are all those that can reach  $\mathbf{i}$  by some sequence of transitions.

Next we can introduce *tense operators* [95, 116] that are specifically devised to make statements about changes in time. Informally, a tense formula is of the form  $Qp$ , where  $p$  is a boolean predicate and  $Q$  is a tense operator, which is evaluated only in the present state.

In the following, the symbols  $\neg, \vee, \wedge$ , denote the negation, disjunction and conjunction

operators from propositional logic.

- Eventually (in the future):  $\mathbf{F}q$  is true in the present state, if  $q$  is true in some future state, and there is a path that connects the two states;
- Eventually (in the past):  $\mathbf{P}q$  is true in the present state, if  $q$  has held at some predecessor state in the past;
- Generally (globally or always in the future):  $\mathbf{G}q$  is true in the present state, if  $q$  holds in all future states. This is also equivalent to  $\neg\mathbf{F}\neg q$ ;
- Generally in the past (or historically):  $\mathbf{H}q$  is true in the present state, if  $q$  has been true at all moments in the past. Equivalent to  $\neg\mathbf{P}\neg q$ ;
- Until:  $p \mathbf{U} q$  holds in the present state, if  $p$  holds from the current state until a state in the future where  $q$  holds. As a special case,  $p \mathbf{U} q$  holds if  $q$  holds in the present state.

A tense formula acts like an open sentence with one free parameter, representing the present state. In that way, there is a bijection that can be established between states and formulae:

- Each formula is determined by the set of all states in which it holds;
- Each state can be labeled with the set of formulae that are valid in it.

### 5.1.1 Linear time, branching time

There exists a relationship between the characterization of the temporal relation  $\prec$  and the set of axioms that can be used in a similar fashion to the inference rules from the proving

techniques of mathematical logic, to reduce a desired property to another.

For example, the following axioms exactly characterize a partial order (transitive and antisymmetric) relation  $\prec$  [22]:

$$\begin{aligned} \mathbf{G}(p \Rightarrow q) &\Rightarrow (\mathbf{G}p \Rightarrow \mathbf{G}q) \\ \mathbf{H}(p \Rightarrow q) &\Rightarrow (\mathbf{H}p \Rightarrow \mathbf{H}q) \\ p &\Rightarrow \mathbf{G}Pp \\ p &\Rightarrow \mathbf{H}Fp \end{aligned}$$

To make  $\prec$  total (i.e., for any two states either  $\mathbf{i} \prec \mathbf{j}$ ,  $\mathbf{i} = \mathbf{j}$ , or  $\mathbf{j} \prec \mathbf{i}$ , two additional axioms are needed to fully characterize  $\prec$ :

$$\begin{aligned} (\mathbf{F}Pq) &\Rightarrow (\mathbf{P}q \vee q \vee \mathbf{F}q) \\ (\mathbf{P}Fq) &\Rightarrow (\mathbf{P}q \vee q \vee \mathbf{F}q) \end{aligned}$$

This setting corresponds to a view of *linear time*, for which all the states are linearly ordered according to their occurrence in time. In practice, time is modeled as a *discrete* sequence, measured by integers. In this common case, each state has an immediate successor and an immediate predecessor. Formally, a state  $\mathbf{j}$  is an immediate successor of state  $\mathbf{i}$  if  $\mathbf{i} \prec \mathbf{j}$  and there is no other state in between,  $\nexists \mathbf{i}' : \mathbf{i} \prec \mathbf{i}' \prec \mathbf{j}$ . This property can be enforced by adding two more axioms to the linear time frame:

$$\begin{aligned} p \wedge \mathbf{H}p &\Rightarrow \mathbf{F}H p \\ p \wedge \mathbf{G}p &\Rightarrow \mathbf{P}G p \end{aligned}$$

Within the discrete time frame, it is useful to define the next time operator:  $\mathbf{X}p$  is true in the present state  $\mathbf{i}$  when there is an immediate successor of  $\mathbf{i}$  in which  $p$  is true. The dual operator,  $\overline{\mathbf{X}}$  corresponds to the predecessor.

The above frameworks are very intuitive, but they lack the notion of non-determinism. This happens when a state may have multiple successor and/or predecessor states. With

the addition of non-deterministic choice with respect to the future only, the temporal order  $\prec$  defines a tree that branches towards the future. Any state has a unique past, but undetermined future. The branching frame comes natural to many practical models, like reactive systems (which receive unpredictable input from the outside world), or non-deterministic programs.

Formally, *branching time* can be obtained from the linear time by dropping the assumption  $(\mathbf{P}\mathbf{F}q) \Rightarrow (\mathbf{P}q \vee q \vee \mathbf{F}q)$ , which orders the future.

### 5.1.2 The CTL\*, CTL, and LTL temporal logics

Next, we will introduce three of the most popular temporal logics, each having a different expressive power.

The CTL\* (*Computation Tree Logic*) temporal logic [37] is a convenient subset of the branching time family. CTL\* formulae use two types of operators:

- Modal operators (or path quantifiers), which can be either **A** (on all paths) or **E** (for some path).
- Tense operators: **F** (future), **G** (generally), **U** (until), or **X** (next), which were previously defined.

Notice that there are no past operators. They are dropped for reasons of symmetry, as any reasoning frame in terms of past time can be mirrored back in order to get its counterpart in terms of future time.

Formally, for a set of atomic propositions  $\mathcal{P}$  and a Kripke structure  $(\widehat{\mathcal{S}}, \mathbf{s}, \mathcal{N}, L)$ , the set of CTL\* formulae can be introduced by inductive definition. Note that CTL\* formulae

may refer to either a state, hence they are called state formulae, or to a path, thus called path formulae.

**Definition 5.1.1** *State formulae:*

1. if  $p \in \mathcal{P}$ , then  $p$  is a state formula;
2. if  $p, q$  are state formulae, then  $\neg p$ ,  $p \vee q$ ,  $p \wedge q$ ,  $p \Rightarrow q$ ,  $p \Leftrightarrow q$  are state formulae;
3. if  $p$  is a path formula, then  $\mathbf{E}p$ ,  $\mathbf{A}p$  are state formulae;

*Path formulae:*

4. if  $p$  is a state formula, then  $p$  is also a path formula;
5. if  $p, q$  are path formulae, then  $\neg p$ ,  $p \vee q$ ,  $p \wedge q$ ,  $p \Rightarrow q$ ,  $p \Leftrightarrow q$ ,  $\mathbf{X}p$ ,  $\mathbf{F}p$ ,  $\mathbf{G}p$ ,  $p \mathbf{U} q$  are path formulae.

Next, we formally define the  $\models$  (“satisfies”) relation, between states and formulae. This relation can be further extended to paths, as shown later. Let  $\Pi_{\mathbf{i}}$  denote the set of all paths starting in state  $\mathbf{i}$ , and for a path  $\pi = \mathbf{i}^0 \mathbf{i}^1 \dots$ ,  $\pi|_k$  represents the suffix of  $\pi$  starting at position  $k$ :  $\pi|_k = \mathbf{i}^k \mathbf{i}^{k+1} \dots$

**Definition 5.1.2** *For a Kripke structure  $(\widehat{\mathcal{S}}, \mathbf{s}, \mathcal{N}, L)$ , the state  $\mathbf{i}$  is said to satisfy the CTL\* formula  $f$  (written  $\mathbf{i} \models f$ ) if one of the following cases holds (for clarity we use  $p$  for atomic propositions from  $\mathcal{P}$ ,  $f, f_1, f_2$  denote state formulae, while  $g, g_1, g_2$  are path formulae):*

1.  $i \models p \Leftrightarrow p \in L(i)$
2.  $i \models \neg f \Leftrightarrow i \not\models f$
3.  $i \models f_1 \circ f_2 \Leftrightarrow (i \models f_1) \circ (i \models f_2)$ , for any operator  $\circ \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$
4.  $i \models \mathbf{E}g \Leftrightarrow \exists \pi \in \Pi_i$  such that  $\pi \models g$
5.  $i \models \mathbf{A}g \Leftrightarrow \forall \pi \in \Pi_i, \pi \models g$
6.  $\pi \models f \Leftrightarrow \pi \in \Pi_i$  and  $i \models f$
7.  $\pi \models \neg g \Leftrightarrow \pi \not\models g$
8.  $\pi \models g_1 \circ g_2 \Leftrightarrow (\pi \models g_1) \circ (\pi \models g_2)$ , for any operator  $\circ \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$
9.  $\pi \models \mathbf{X}g \Leftrightarrow \pi_{|1} \models g$
10.  $\pi \models \mathbf{F}g \Leftrightarrow \exists k \geq 0: \pi_{|k} \models g$
11.  $\pi \models \mathbf{G}g \Leftrightarrow \forall k \geq 0: \pi_{|k} \models g$
12.  $\pi \models g_1 \mathbf{U} g_2 \Leftrightarrow \exists k \geq 0: \pi_{|k} \models g_2$  and  $\forall 0 \leq j < k, \pi_{|j} \models g_1$

**Lemma 5.1.1** *Adjacent path quantifiers are legal. However, this is a purely syntactical loophole, as semantically:*

$$Q_n \dots Q_2 Q_1 g \Leftrightarrow Q_1 g$$

for any combination of path quantifiers  $Q_i \in \{\mathbf{A}, \mathbf{E}\}$ ,  $1 \leq i \leq n$ .

**Proof.** The key observation is that the quantifier adjacent to the predicate “cancels” the effect of the other quantifiers, based on the conversion of path formulae to state formulae and the definition of the  $\models$  property in this case. To prove the above statement for  $n = 2$ , we have to show that  $Q_2 Q_1 g \Leftrightarrow Q_1 g$ . Since path quantifiers are applied only to path formulae and  $Q_1 g$  is a state formula, this has to be interpreted back as a path formula according

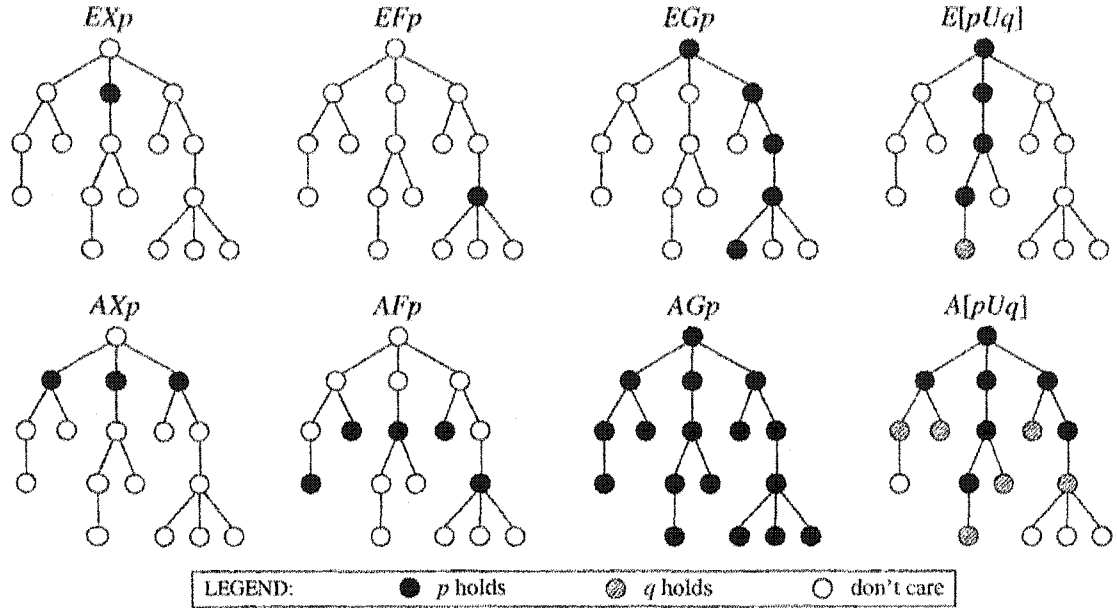


Figure 5.1: The eight basic CTL operators

to item 6 in the above definition. Hence,  $i \models Q_2 Q_1 g \Leftrightarrow \exists/\forall \pi \in \Pi_i \pi \models Q_1 g$ . But, since  $\pi \in \Pi_i$ ,  $\pi \models Q_1 g \Leftrightarrow i \models Q_1 g$  (based on item 6). Together,  $i \models Q_2 Q_1 g \Leftrightarrow i \models Q_1 g$ , q.e.d.

The claim can be easily extended to  $n$  quantifiers, by induction.

The **CTL temporal logic** [37] is a subset of  $\text{CTL}^*$ , in which temporal operators appear only in pairs. The first is a path quantifier, **A** or **E**. The second is the tense part, which can be **F**, **G**, **U**, or **X**. Hence, besides excluding the past tenses, there is one more restriction: path quantifiers cannot be combined directly with the propositional part. This leaves only eight possible combinations for a valid CTL pair.

Their intuitive meaning is described in Figure 5.1 on small examples, where the root node of each branching tree is a state satisfying the listed property.

For convenience, we can determine an even smaller subset of CTL operators that can be used to express any other combination (a complete subset). If the subset is minimal, it

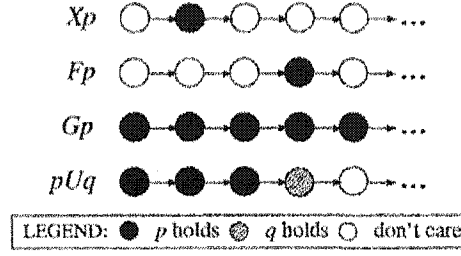


Figure 5.2: The four basic LTL operators

is called a *generator*. One such set is  $\{\mathbf{EX}, \mathbf{AU}, \mathbf{EU}\}$ , as the rest of the operators can be obtained as follows [37]:

**Lemma 5.1.2**

$$\begin{aligned}
 \mathbf{AX}p &= \neg \mathbf{EX} \neg p \\
 \mathbf{AF}p &= \mathbf{A}(\text{true} \mathbf{U} p) \\
 \mathbf{EF}p &= \mathbf{E}(\text{true} \mathbf{U} p) \\
 \mathbf{AG}p &= \neg \mathbf{E}(\text{true} \mathbf{U} \neg p) \\
 \mathbf{EG}p &= \neg \mathbf{A}(\text{true} \mathbf{U} \neg p)
 \end{aligned}$$

Another generator set frequently used for CTL is  $\{\mathbf{EX}, \mathbf{EG}, \mathbf{EU}\}$ .

In the **Linear Temporal Logic** LTL [64], a different view of the behaviors of the system is assumed, in which the branching of the outgoing paths is not important. A property is considered satisfied by a state if all possible interleavings of future paths verify it. The relation between different executions (such as existing common suffixes, prefixes, etc.) is not relevant in LTL.

The interleaving of the branches is consistent with only one path quantifier, **A**, which is always assumed at the beginning of an LTL formula. Therefore the path quantifier can be discarded, leaving four possible choices for LTL path formulae:  $\mathbf{X}p$ ,  $\mathbf{F}p$ ,  $\mathbf{G}p$ , and  $p \mathbf{U} q$ . On the other hand, the connectors may be combined in any order (as opposed to the strict pairing of CTL).



The expressiveness of the three temporal logics that were introduced are not equivalent. The LTL formula  $\mathbf{FG}p$  (meaning that there is some state in the future from which  $p$  will hold forever) has no equivalent in the CTL syntax. On the other hand, the CTL formula  $\mathbf{AG}(\mathbf{EF}p)$  (meaning that on all paths, there exists a branching path along which  $p$  will eventually hold) is not expressible in LTL.

If we combine the two above formulae, say  $\mathbf{A}(\mathbf{FG}p) \vee \mathbf{AG}(\mathbf{EF}p)$  we obtain a formula that is not valid in either LTL or CTL syntax, but it is valid in CTL\*.

More details about the logic hierarchy can be found in [34, 58, 96, 103, 116]. We implement CTL, as its simple syntax is expressive enough to formulate most properties of interest.

### 5.1.3 Computing CTL operators

Given a set of CTL formulae  $\mathcal{F}$ , and a Kripke structure  $(\hat{\mathcal{S}}, s, \mathcal{N}, L)$ , solving the specification  $\mathcal{F}$  means labeling each state  $i$  in  $\mathcal{S}$  with those formulae of  $\mathcal{F}$  that are satisfied by  $i$ . In other words, each formula is identified by the set of states that validates it, therefore solving  $\mathcal{F}$  is reduced to building subsets of reachable states.

Working with sets is very practical, since CTL properties can then be characterized as fixed points of appropriate functionals [35]. The lattice over which the functionals are defined is  $\mathcal{L} = (2^{\hat{\mathcal{S}}}, \subseteq, \cup, \cap)$ , the lattice of subsets of  $\hat{\mathcal{S}}$ , with set inclusion as the partial order relation, and set union and set intersection for infimum and supremum of two elements. It is well known that, for a finite  $\hat{\mathcal{S}}$ , the lattice  $\mathcal{L}$  is complete, hence any monotonic functional  $f : 2^{\hat{\mathcal{S}}} \rightarrow 2^{\hat{\mathcal{S}}}$  (for which  $\mathcal{P} \subseteq \mathcal{Q}$  implies  $f(\mathcal{P}) \subseteq f(\mathcal{Q})$ ), is also continuous with respect to union and intersection: i.e.  $f(\cup_i \mathcal{P}_i) = \cup_i f(\mathcal{P}_i)$ ,  $f(\cap_i \mathcal{P}_i) = \cap_i f(\mathcal{P}_i)$ . In this case, by

Tarski's theorem [130], any monotonic functional  $f$  has a least fixed point  $\mu f = \bigcup_{i \geq 0} f^i(\emptyset)$  and a greatest fixed point  $\nu f = \bigcap_{i \geq 0} f^i(\hat{\mathcal{S}})$ .

We are now ready to give the fixed point characterization of a generator set of CTL operators:

**Theorem 5.1** [37] For a finite Kripke structure  $(\hat{\mathcal{S}}, \mathbf{s}, \mathcal{N}, L)$  and two CTL predicates  $p$  and  $q$  (corresponding to the sets of states  $\mathcal{P}$  and  $\mathcal{Q}$ ):

1.  $\mathbf{EF}p \equiv \mu \mathcal{X}.f_1(\mathcal{X})$ , where  $f_1(\mathcal{X}) = \mathcal{P} \cup h(\mathcal{X})$ ;
2.  $\mathbf{EG}p \equiv \nu \mathcal{X}.f_2(\mathcal{X})$ , where  $f_2(\mathcal{X}) = \mathcal{P} \cap h(\mathcal{X})$ ;
3.  $\mathbf{E}(p \mathbf{U} q) \equiv \mu \mathcal{X}.f_3(\mathcal{X})$ , where  $f_3(\mathcal{X}) = \mathcal{Q} \cup (\mathcal{P} \cap h(\mathcal{X}))$ .

provided  $h : 2^{\hat{\mathcal{S}}} \rightarrow 2^{\hat{\mathcal{S}}}$ ,  $h(\mathcal{X}) = \{\mathbf{i} \in \hat{\mathcal{S}} \mid \exists \mathbf{j} \in \mathcal{X} : \mathbf{j} \in \mathcal{N}(\mathbf{i})\}$ , the functional that computes **EX**.

**Corollary 5.1.1** *For completeness only, the fixed point characterization of the remaining CTL operators is:*

1.  $\mathbf{AF}p \equiv \mu \mathcal{X}.f_4(\mathcal{X})$ , where  $f_4(\mathcal{X}) = \mathcal{P} \cup h'(\mathcal{X})$ ;
2.  $\mathbf{AG}p \equiv \nu \mathcal{X}.f_5(\mathcal{X})$ , where  $f_5(\mathcal{X}) = \mathcal{P} \cap h'(\mathcal{X})$ ;
3.  $\mathbf{A}(p \mathbf{U} q) \equiv \mu \mathcal{X}.f_6(\mathcal{X})$ , where  $f_6(\mathcal{X}) = \mathcal{Q} \cup (\mathcal{P} \cap h'(\mathcal{X}))$ .

provided  $h' : 2^{\hat{\mathcal{S}}} \rightarrow 2^{\hat{\mathcal{S}}}$ ,  $h'(\mathcal{X}) = \{\mathbf{i} \in \hat{\mathcal{S}} \mid \mathbf{j} \in \mathcal{N}(\mathbf{i}) \Rightarrow \mathbf{j} \in \mathcal{X}\}$ , the functional that computes **AX**.

Intuitively, properties that look for a property that *eventually* holds build a least fixed point – by finding in each iteration more states to add to the initial set  $\emptyset$ , and properties that

check that a property holds *globally* build a greatest fixed point – by gradually eliminating states from the initial set  $\hat{S}$ .

## 5.2 Saturation-based CTL Model Checking

Given the excellent performance of the saturation strategy, we next investigate how to extend it to improving the computation of the CTL operators [31]. As said above, only a *generator* subset needs to be implemented in a model checker, as the remaining operators can be expressed in terms of those in the set [41].  $\{\mathbf{EX}, \mathbf{EU}, \mathbf{EG}\}$  is such a set, but the following discusses also  $\mathbf{EF}$  for clarity.

### 5.2.1 The EX operator

**Semantics:**  $i^0 \models \mathbf{EX}p$  iff  $\exists i^1 \in \mathcal{N}(i^0)$  such that  $i^1 \models p$ . (“ $\models$ ” means “satisfies”)

In our notation,  $\mathbf{EX}$  corresponds to the inverse of function  $\mathcal{N}$ , the previous-state function,  $\mathcal{N}^{-1}$ . With our Kronecker matrix encoding, the inverse of  $\mathcal{N}_e$  is simply obtained by transposing the incidence matrices  $\mathbf{N}_{k,e}$  in the Kronecker product, thus  $\mathcal{N}^{-1}$  is encoded as  $\bigvee_{e \in \mathcal{E}} \bigotimes_{K \geq k \geq 1} \mathbf{N}_{k,e}^T$ .

To compute the set of states where  $\mathbf{EX}p$  is satisfied, we can follow the same idea used to fire events in an MDD node during our state-space generation: given the set  $\mathcal{P}$  of states satisfying formula  $p$ , we can accumulate the effect of “firing backward” each event  $\mathcal{N}^{-1}(\mathcal{P}) = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e^{-1}(\mathcal{P})$ , and take advantage of locality. This results in an efficient calculation of  $\mathbf{EX}$ . Computing its reflexive and transitive closure, that is, the backward reachability operator  $\mathbf{EF}$ , is a much more difficult challenge, which we consider next.

### 5.2.2 The EF operator

**Semantics:**  $i^0 \models \mathbf{EF}p$  iff  $\exists n \geq 0, \exists i^1 \in \mathcal{N}(i^0), \dots, \exists i^n \in \mathcal{N}(i^{n-1})$  such that  $i^n \models p$ .

In our approach, the construction of the set of states satisfying  $\mathbf{EF}p$  is analogous to the saturation algorithm for state-space generation, with two differences. Besides using the transposed incidence matrices  $\mathbf{N}_{k,e}^T$ , the execution starts with the set  $\mathcal{P}$ , not a single state. These differences do not affect the applicability of saturation, which retains all its substantial time and memory benefits.

### 5.2.3 The EU operator

**Semantics:**  $i^0 \models \mathbf{E}[p \mathbf{U} q]$  iff  $\exists n \geq 0, \exists i^1 \in \mathcal{N}(i^0), \dots, \exists i^n \in \mathcal{N}(i^{n-1})$  such that  $i^n \models q$  and  $i^m \models p$  for all  $m < n$ .  
(in particular,  $i \models q$  implies  $i \models \mathbf{E}[p \mathbf{U} q]$ )

The traditional computation of the set of states satisfying  $\mathbf{E}[p \mathbf{U} q]$  uses a least fixed point algorithm. Starting with the set  $\mathcal{Q}$  of states satisfying  $q$ , it iteratively adds all the states that reach them on paths where property  $p$  holds (see Algorithm *EUtrad* in Figure 5.3). The number of iterations to reach the fixed point is

$$\max_{i \in \mathcal{S}} \left( \min \{ n \mid \exists i^0 \in \mathcal{Q} \wedge \forall 0 < m \leq n, \exists i^m \in \mathcal{N}^{-1}(i^{m-1}) \cap \mathcal{P} \wedge i = i^n \} \right)$$

#### Applying saturation to EU.

The challenge in applying saturation arises from the need to “filter out” states not in  $\mathcal{P}$  (line 5 of Algorithm *EUtrad*): as soon as a new predecessor of the working set  $\mathcal{X}$  is obtained, it must be intersected with  $\mathcal{P}$ . Failure to do so can result in paths to  $\mathcal{Q}$  that stray, even temporarily, out of  $\mathcal{P}$ . However, saturation works in a highly localized manner,

```

EUtrad(in  $\mathcal{P}, \mathcal{Q} : \text{stateset}$ ) :  $\text{stateset}$ 
1. declare  $\mathcal{X}, \mathcal{Y} : \text{stateset}$ ;
2.  $\mathcal{X} \leftarrow \mathcal{Q}$ ;                                • initialize  $\mathcal{X}$  with all states in  $\mathcal{Q}$ 
3. repeat
4.    $\mathcal{Y} \leftarrow \mathcal{X}$ ;
5.    $\mathcal{X} \leftarrow \mathcal{X} \cup (\mathcal{N}^{-1}(\mathcal{X}) \cap \mathcal{P})$ ;      • add predecessors of states in  $\mathcal{X}$  that are in  $\mathcal{P}$ 
6. until  $\mathcal{Y} = \mathcal{X}$ ;
7. return  $\mathcal{X}$ ;

```

**Figure 5.3:** The traditional algorithm to compute **EU**.

adding states out of breadth-first-search order. Performing an expensive intersection after each firing would add enormous overhead, since our firings are very lightweight operations. To cope with this problem, we propose a “partial” saturation that is applied to a subset of events for which no filtering is needed. These are the events whose firing is *guaranteed* to preserve the validity of formula  $p$ . For the remaining events, BFS with filtration must be used. The resulting global fixed point iteration interleaves these two phases (see Figure 5.4). The following classification of events is analogous to, but different from, the *visible* vs. *invisible* one proposed for partial order reduction [4].

**Definition 5.2.1** In a discrete state model  $(\widehat{\mathcal{S}}, \mathbf{s}, \mathcal{N})$ , an event  $e$  is *dead* with respect to a set of states  $\mathcal{X}$  if there is no state in  $\mathcal{X}$  from which its firing leads to a state in  $\mathcal{X}$ , i.e.,  $\mathcal{N}_e^{-1}(\mathcal{X}) \cap \mathcal{X} = \emptyset$  (this includes the case where  $e$  is always disabled in  $\mathcal{X}$ ); it is *safe* if it is not dead and its firing cannot lead from a state not in  $\mathcal{X}$  to a state in  $\mathcal{X}$ , i.e.,  $\emptyset \subset \mathcal{N}_e^{-1}(\mathcal{X}) \subseteq \mathcal{X}$ ; it is *unsafe* otherwise, i.e.,  $\mathcal{N}_e^{-1}(\mathcal{X}) \setminus \mathcal{X} \neq \emptyset \wedge \mathcal{N}_e^{-1}(\mathcal{X}) \cap \mathcal{X} \neq \emptyset$ .  $\square$

Given a formula  $\mathbf{E}[p \mathbf{U} q]$ , we first classify the safety of events through static analysis. Then, each **EU** fixed point iteration consists of two backward steps: BFS on unsafe events followed by saturation on safe events. Since saturation is in turn a fixed point computation, the resulting algorithm computes a nested fixed point. Note that the operators used in both

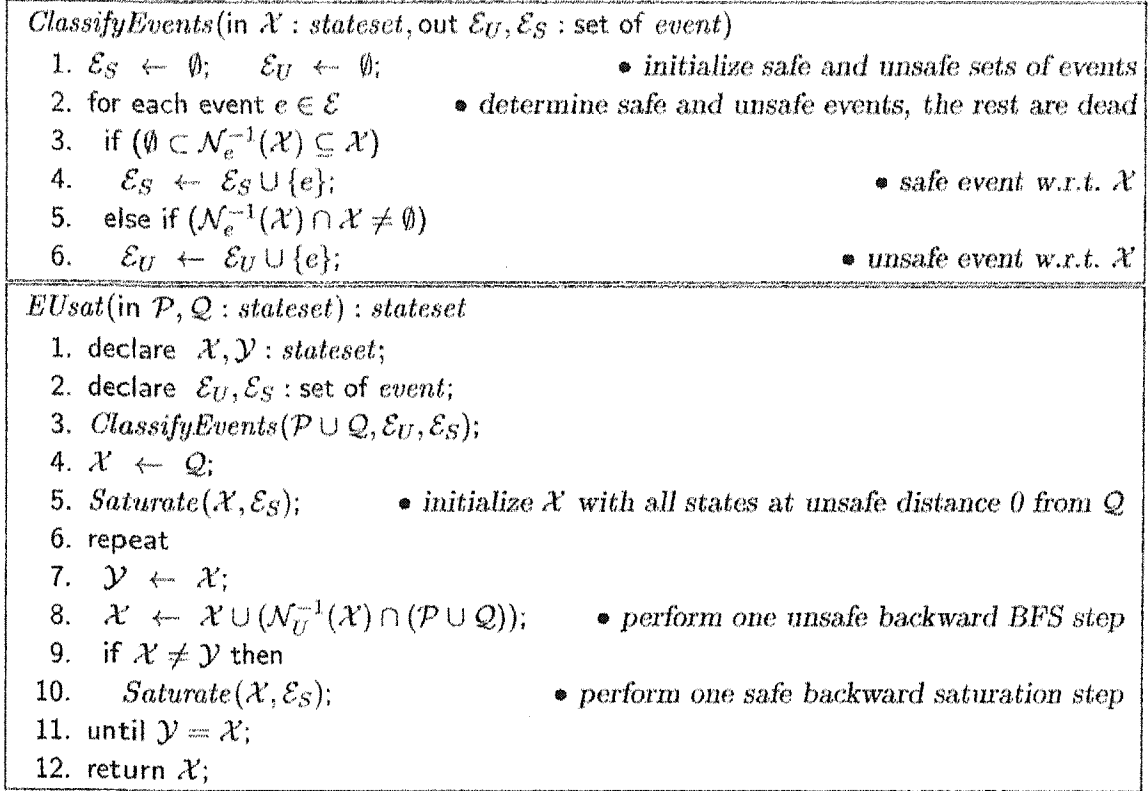


Figure 5.4: The saturation-based algorithm to compute EU.

steps are monotonic (the working set  $\mathcal{X}$  is increasing), a condition for applying saturation and in-place updates.

**Note 5.2.1** Dead events can be ignored altogether by our *EUsat* algorithm, since the working set  $\mathcal{X}$  is always a subset of  $\mathcal{P} \cup \mathcal{Q}$ .

**Note 5.2.2** The *Saturate* procedure in line 10 of *EUsat* is analogous to the one we use in Chapter 3, except that it is restricted to a subset  $\mathcal{E}_S$  of events.

**Note 5.2.3** *ClassifyEvents* has the same time complexity as one **EX** step and is called only once prior to the fixed point iterations.

**Note 5.2.4** To simplify the description of *EUsat*, we call *ClassifyEvents* with the filter  $\mathcal{P} \cup \mathcal{Q}$ , i.e.,  $\mathcal{E}_S = \{e : \emptyset \subset \mathcal{N}_e^{-1}(\mathcal{P} \cup \mathcal{Q}) \subseteq \mathcal{P} \cup \mathcal{Q}\}$ . With a slightly more complex initialization

in *EUsat*, we could use instead the smaller filter  $\mathcal{P}$ , i.e.,  $\mathcal{E}_S = \{e : \emptyset \subset \mathcal{N}_e^{-1}(\mathcal{P}) \subseteq \mathcal{P}\}$ . In practice, both sets of events could be computed. Then, if one is a strict superset of the other, it should be used, since the larger  $\mathcal{E}_S$  is, the more *EUsat* behaves like our efficient **EF** saturation; otherwise, some heuristic must be used to choose between the two.

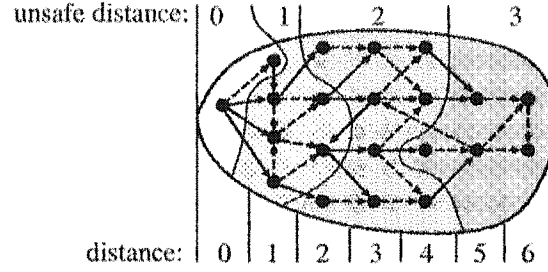
**Note 5.2.5** The number of *EUsat* iterations is 1 plus the “unsafe distance from  $\mathcal{P}$  to  $\mathcal{Q}$ ”,  $\max_{\mathbf{i} \in \mathcal{P}}(\min\{n \mid \exists \mathbf{i}^0 \in \mathcal{R}_S^*(\mathcal{Q}) \wedge \forall 0 < m \leq n, \exists \mathbf{i}^m \in \mathcal{R}_S^*(\mathcal{R}_U(\mathbf{i}^{m-1}) \cap \mathcal{P}) \wedge \mathbf{i} = \mathbf{i}^n\})$ , where  $\mathcal{R}_S(\mathcal{X}) = \bigcup_{e \in \mathcal{E}_S} \mathcal{N}_e^{-1}(\mathcal{X})$  and  $\mathcal{R}_U(\mathcal{X}) = \bigcup_{e \in \mathcal{E}_U} \mathcal{N}_e^{-1}(\mathcal{X})$  are the sets of “safe predecessors” and “unsafe predecessors” of  $\mathcal{X}$ , respectively.

**Lemma 5.2.1** Iteration  $d$  of *EUsat* finds all states  $\mathbf{i}$  at unsafe distance  $d$  from  $\mathcal{Q}$ .

**Proof.** By induction on  $d$ . Base:  $d = 0 \Rightarrow \mathbf{i} \in \mathcal{R}_S^*(\mathcal{Q})$  which is a subset of  $\mathcal{X}$  (lines 4,5). Inductive step: suppose all states at unsafe distance  $m \leq d$  are added to  $\mathcal{X}$  in the  $m^{\text{th}}$  iteration. By definition, a state  $\mathbf{i}$  at unsafe distance  $d+1$  satisfies:  $\exists \mathbf{i}^0 \in \mathcal{R}_S^*(\mathcal{Q}) \wedge \forall 0 < m \leq d+1, \exists \mathbf{j}^m \in \mathcal{R}_U(\mathbf{i}^{m-1}) \cap \mathcal{P}, \exists \mathbf{i}^m \in \mathcal{R}_S^*(\mathbf{j}^m)$ , and  $\mathbf{i} = \mathbf{i}^{d+1}$ . Then,  $\mathbf{i}^m$  and  $\mathbf{j}^m$  are at unsafe distance  $m$ . By the induction hypothesis, they are added to  $\mathcal{X}$  in iteration  $m$ . In particular,  $\mathbf{i}^d$  is a new state found in iteration  $d$ . This implies that the algorithm must execute another iteration, which finds  $\mathbf{j}^{d+1}$  as an unsafe predecessor of  $\mathbf{i}^d$  (line 8). Since  $\mathbf{i}$  is either  $\mathbf{j}^{d+1}$  or can reach it through safe events alone, it is added to  $\mathcal{X}$  (line 10).

**Theorem 5.2.1** Algorithm *EUsat* returns the set  $\mathcal{X}$  of states satisfying  $\mathbf{E}[p \mathbf{U} q]$ .

**Proof.** It is immediate to see that *EUsat* terminates, since its working set is a monotonically increasing subset of  $\widehat{S}$ , which is finite. Let  $\mathcal{Y}$  be the set of states satisfying  $\mathbf{E}[p \mathbf{U} q]$ . We have (i)  $\mathcal{Q} \subseteq \mathcal{X}$  (line 4) (ii) every state in  $\mathcal{X}$  can reach a state in  $\mathcal{Q}$  through a path in  $\mathcal{X}$ , and (iii)  $\mathcal{X} \subseteq \mathcal{P} \cup \mathcal{Q}$  (lines 8,10). This implies  $\mathcal{X} \subseteq \mathcal{Y}$ . Since any state in  $\mathcal{Y}$  is at some



**Figure 5.5:** Comparing BFS and saturation order: distance vs. unsafe distance.

finite unsafe distance  $d$  from  $\mathcal{Q}$ , by Lemma 5.2.1 we conclude that  $\mathcal{Y} \subseteq \mathcal{X}$ . The two set inclusions imply  $\mathcal{X} = \mathcal{Y}$ .

Figure 5.5 illustrates how our exploration differs from BFS. Solid and dashed arcs represent unsafe and safe transitions, respectively. The shaded areas encircle the explored regions after each iteration of *EU<sub>sat</sub>*, four in this case. *EU<sub>trad</sub>* would instead require seven iterations to explore the entire graph (states are aligned vertically according to their BFS depth).

**Note 5.2.6** Our approach exhibits “graceful degradation”. In the best case, all events are safe, and *EU<sub>sat</sub>* performs just one saturation step and stops. This happens for example when  $p \vee q \equiv \text{true}$ , which includes the special case  $p \equiv \text{true}$ . As  $\mathbf{E}[\text{true} \mathbf{U} q] \equiv \mathbf{EF}q$ , we simply perform backward reachability from  $\mathcal{Q}$  using saturation on the entire set of events. In the worst case, all events are unsafe, and *EU<sub>sat</sub>* performs the same steps as *EU<sub>trad</sub>*. But even then, locality and our Kronecker encoding can still substantially improve the efficiency of the algorithm.

#### 5.2.4 The EG operator

**Semantics:**  $i^0 \models \mathbf{EG}p$  iff  $i^0 \models p \wedge \forall n > 0 : \exists i^n \in \mathcal{N}(i^{n-1})$  such that  $i^n \models p$ .



$EG_{trad}(\text{in } \mathcal{P} : \text{stateset}) : \text{stateset}$ 1. declare $\mathcal{X}, \mathcal{Y} : \text{stateset};$ 2. $\mathcal{X} \leftarrow \mathcal{P};$ 3. repeat 4. $\mathcal{Y} \leftarrow \mathcal{X};$ 5. $\mathcal{X} \leftarrow \mathcal{N}^{-1}(\mathcal{X}) \cap \mathcal{P};$ 6. until $\mathcal{Y} = \mathcal{X};$ 7. return $\mathcal{X};$	$EG_{sat}(\text{in } \mathcal{P} : \text{stateset}) : \text{stateset}$ 1. declare $\mathcal{X}, \mathcal{Y}, \mathcal{C}, \mathcal{T} : \text{stateset};$ 2. $\mathcal{C} \leftarrow EU_{sat}(\mathcal{P}, \{\mathbf{i} \in \mathcal{P} : \mathbf{i} \in \mathcal{N}(\mathbf{i})\});$ 3. $\mathcal{T} \leftarrow \emptyset;$ 4. while $\exists \mathbf{i} \in \mathcal{P} \setminus (\mathcal{C} \cup \mathcal{T})$ do 5. $\mathcal{X} \leftarrow EU_{sat}(\mathcal{P} \setminus \mathcal{C}, \{\mathbf{i}\});$ 6. $\mathcal{Y} \leftarrow ES_{sat}(\mathcal{P} \setminus \mathcal{C}, \{\mathbf{i}\});$ 7. if $ \mathcal{X} \cap \mathcal{Y}  > 1$ then 8. $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{X};$ 9. else 10. $\mathcal{T} \leftarrow \mathcal{T} \cup \{\mathbf{i}\};$ 11. return $\mathcal{C};$
--	---

Figure 5.6: Traditional and saturation-based EG algorithms.

In graph terms, consider the reachability subgraph obtained by restricting the transition relation to states in  $\mathcal{P}$ . Then, **EG** $p$  holds in any state belonging to, or reaching, a nontrivial strongly connected component (SCC) of this subgraph.

Algorithm *EGtrad* in Figure 5.6 shows the traditional greatest fixed point iteration. It initializes the working set  $\mathcal{X}$  with all states in  $\mathcal{P}$  and gradually eliminates states that have no successor in  $\mathcal{X}$  until only the SCCs of  $\mathcal{P}$  and their incoming paths along states in  $\mathcal{P}$  are left. The number of iterations equals the maximum length of any path over  $\mathcal{P}$  that does *not* lead to such an SCC.

### Applying saturation to EG.

*EGtrad* is a greatest fixed point algorithm, so to speed it up we must *eliminate* unwanted states faster. The criterion for a state  $\mathbf{i}$  is a *conjunction*:  $\mathbf{i}$  should be eliminated if *all* its successors are not in  $\mathcal{P}$ . Since it considers a single event at a time and makes local decisions that must be globally correct, it would appear that saturation cannot be used to improve *EGtrad*. However, Figure 5.6 shows an algorithm for **EG** which, like [11, 136], enumerates

the SCCs by finding forward and backward reachable sets from a state. However, it uses saturation, instead of breadth-first search. In line 2, Algorithm *EGsat* disposes of selfloop states in  $\mathcal{P}$  and of the states reaching them through paths in  $\mathcal{P}$  (self-loops can be found by performing **EX** using a modified set of matrices  $\mathbf{N}_{k,e}$  where off-diagonal entries are set to zero). Then, it chooses a single state  $i \in \mathcal{P}$  and builds the backward and forward reachable sets from  $i$  restricted to  $\mathcal{P}$ , using *EUsat* and *ESsat*, where **ES** is the dual in the past of **EU**. *ESsat* differs from *EUsat* only in that it does not transpose the matrices  $\mathbf{N}_{k,e}$ . If  $\mathcal{X}$  and  $\mathcal{Y}$  have more than just  $i$  in common,  $i$  belongs to a nontrivial SCC and all of  $\mathcal{X}$  is part of our answer  $\mathcal{C}$ . Otherwise, we add  $i$  to the set  $\mathcal{T}$  of trivial SCCs ( $i$  might nevertheless reach a nontrivial SCC, in  $\mathcal{Y}$ , but we have no easy way to tell). The process ends when  $\mathcal{P}$  has been partitioned into  $\mathcal{C}$ , containing nontrivial SCCs and states reaching them over  $\mathcal{P}$ , and  $\mathcal{T}$ , containing trivial SCCs.

*EGsat* is more efficient than *EGtrad* only in special cases. An example is when the *EUsat* and *ESsat* calls in *EGsat* find each next state on a long path of trivial SCCs through a single lightweight firing, while *EGtrad* always attempts firing each event at each iteration. In the worst case, however, *EGsat* can be much worse than not only *EGtrad*, but even an explicit approach, since it enumerates all SCCs. For this reason, the results section discusses only *EGtrad*, which is guaranteed to benefit from locality and the Kronecker encoding.

### 5.3 Results

We compared our new Model Checking algorithms implemented in *SMART* with those in NuSMV [33]. In all experiments, the time necessary to build the encoding of  $\mathcal{N}$  is not

N	S	State-space generation				EG query					
		NuSMV		SMART		NuSMV			SMART		
						after SS		alone		EGtrad	
		time	mem	time	mem	time	mem	time	mem	time	mem
Leader: $K = 2N$ , $ \mathcal{E}  = N^2 + 13N$						EG( $status_0 \neq leader$ )					
3	$8.49 \times 10^2$	0.1	2	0.04	<.5	0.2	4	0.7	4	0.02	<.5
4	$1.15 \times 10^4$	2.1	10	0.27	<.5	232.8	12	1189.1	235	0.11	2
5	$1.50 \times 10^5$	56.0	29	1.49	1	18023.6	104	—	—	0.44	9
6	$1.89 \times 10^6$	1063.7	295	7.35	3	—	—	—	—	1.64	38
7	$2.39 \times 10^7$	—	—	40.64	7	—	—	—	—	7.15	128
Philosophers: $K = \lceil N/2 \rceil$ , $ \mathcal{E}  = 4N$						EG( $phil_0 \neq eat$ ) (starvation)					
20	$3.46 \times 10^{12}$	0.8	6	0.02	<.5	0.1	8	1.1	6	0.01	<.5
50	$2.23 \times 10^{31}$	36.0	46	0.07	<.5	0.9	46	132.3	50	0.02	1
100	$4.96 \times 10^{62}$	1134.8	316	0.15	<.5	9.0	316	2525.3	358	0.05	3
500	$3.03 \times 10^{313}$	—	—	1.01	1	—	—	—	—	0.28	58
Slotted ring: $K = N$ , $ \mathcal{E}  = 8N$						EG( $slot_0 \neq hg$ )					
5	$5.38 \times 10^5$	0.1	1	0.00	<.5	0.0	1	0.0	<.5	0.00	<.5
10	$8.29 \times 10^9$	3.1	10	0.05	<.5	0.6	10	0.1	1	0.01	<.5
15	$1.46 \times 10^{15}$	1503.9	15	0.17	<.5	4.7	15	0.2	2	0.01	<.5
100	$3.03 \times 10^{105}$	—	—	39.70	16	—	—	—	—	0.62	62
Round robin: $K = N + 1$ , $ \mathcal{E}  = 6N$						EG( $true$ ) (find all cycles)					
5	$3.60 \times 10^2$	0.1	1	<.005	<.5	0.0	1	0.1	1	<.005	<.5
10	$2.30 \times 10^5$	68.2	11	0.01	<.5	0.3	11	78.5	13	<.005	<.5
15	$1.10 \times 10^6$	4201.5	40	0.02	<.5	1.2	40	4739.5	44	0.01	<.5
100	$2.85 \times 10^{32}$	—	—	1.98	1	—	—	—	—	1.29	20
Flexible manuf. sys.: $K = 19$ , $ \mathcal{E}  = 20$						EG( $\neg(P_1s = P_2s = P_3s = N)$ )					
2	$3.44 \times 10^3$	128.3	17	0.01	<.5	0.2	17	128.9	18	<.005	<.5
3	$4.86 \times 10^4$	4107.5	127	0.02	<.5	1.0	127	—	—	0.01	<.5
25	$8.54 \times 10^{13}$	—	—	17.98	<.5	—	—	—	—	50.38	251

**Table 5.1:** Experimental results for computing EF and EU: SMART vs. NuSMV.

included in the tables. While this time is negligible for our Kronecker encoding, it can be quite substantial for NuSMV, at times exceeding the reported runtimes.

Tables 5.1-5.2 show the state-space size, runtime (in seconds), and peak memory consumption (in megabytes) for the five models, counting MDD nodes plus Kronecker matrices in SMART, and BDD nodes in NuSMV. There are three sets of columns: state-space generation (analogous to EF), EU, and EG.

Note that in NuSMV it is possible to evaluate EU expressions without explicitly building the state space first; however, this substantially increases the runtime, so that it almost

N	S	EU query									
		NuSMV				SMART					
		after SS		alone		EUsat			EUsat		
		time	mem	time	mem	iter	time	mem	iter	time	mem
Leader:		$E[(pref_1 = 0) \cup (status_0 = leader)]$									
3	$8.49 \times 10^2$	0.1	3	18.3	12	43	0.02	<.5	22	0.02	<.5
4	$1.15 \times 10^4$	2.3	11	8104.7	371	62	0.36	1	38	0.27	1
5	$1.50 \times 10^5$	52.0	33	—	—	81	3.74	7	52	3.09	7
6	$1.89 \times 10^6$	—	—	—	—	101	46.90	30	66	35.67	28
7	$2.39 \times 10^7$	—	—	—	—	121	690.85	116	85	416.85	101
Philosophers		$E[(phil_1 \neq eat) \cup (phil_0 = eat)]$									
20	$3.46 \times 10^{12}$	0.1	7	0.8	6	40	0.03	<.5	4	0.02	<.5
50	$2.23 \times 10^{31}$	1.2	46	39.7	46	100	0.17	1	4	0.06	1
100	$4.96 \times 10^{62}$	7.9	316	1121.8	316	200	0.67	3	4	0.14	3
500	$3.03 \times 10^{313}$	—	—	—	—	1000	19.09	78	4	0.77	60
Slotted ring		$E[(slot_1 \neq bf) \cup (slot_0 = ag)]$									
5	$5.38 \times 10^5$	0.0	1	0.0	1	33	0.01	<.5	9	0.00	<.5
10	$8.29 \times 10^9$	0.2	10	0.4	3	63	0.01	<.5	9	0.01	<.5
15	$1.46 \times 10^{15}$	1.8	15	2.0	10	93	0.37	1	9	0.02	<.5
100	$3.03 \times 10^{105}$	—	—	—	—	603	—	—	9	1.60	62
Round robin		$E[(p_1 \neq load) \cup (p_0 = send)]$									
5	$3.60 \times 10^2$	0.0	1	0.2	1	19	0.00	<.5	6	0.00	<.5
10	$2.30 \times 10^5$	0.2	11	85.0	11	39	0.01	<.5	11	0.01	<.5
15	$1.10 \times 10^6$	0.6	40	4922.7	40	59	0.03	<.5	16	0.01	<.5
100	$2.85 \times 10^{32}$	—	—	—	—	399	13.32	32	101	4.67	19
FMS		$E[(M_1 > 0) \cup (P_1s = P_2s = P_3s = N)]$									
2	$3.44 \times 10^3$	0.2	17	318.1	43	31	0.04	<.5	6	0.01	<.5
3	$4.86 \times 10^4$	1.0	127	—	—	46	0.16	<.5	8	0.02	<.5
25	$8.54 \times 10^{13}$	—	—	—	—	376	—	—	52	1010.85	293

Table 5.2: Experimental results for computing EG: SMART vs. NuSMV.

equals the sum of the state-space generation and EU entries. The same holds for EG. In SMART, the state-space construction is always executed in advance, hence the memory consumption includes the MDD for the state space. We show the largest parameter for which NuSMV can build the state space in the penultimate row of each model, while the last row shows the largest parameter for which SMART can evaluate the EU and EG.

Overall, SMART outperforms NuSMV time- and memory-wise. Saturation excels at state-space generation, with improvements exceeding 100,000 in time and 1,000 in memory. For EU, we can see the effect of the data structures and of the algorithm separately, since we

report for both *EUtrad* and *EUsat*. When comparing the two, the latter reaches a fixed point in fewer iterations (recall Figure 5.5) and uses less memory. While each *EUsat* iteration is more complex, it also operates on smaller MDDs, one of the benefits of saturation. The performance gain is more evident in large models, where *EUtrad* runs out of memory before completing the task and is up to 20 times slower. The comparison between our *EUtrad*, *EGtrad* and NuSMV highlights instead the differences between data structures. SMART is still faster and uses much less memory, suggesting that the Kronecker representation for the transition relation is much more efficient than the  $2K$ -level BDD representation.

## Chapter 6

# Counterexamples and Witnesses

One of the most attractive features of Model Checking is its ability to provide counterexamples and witnesses for the properties that are checked, as this allows the user to inspect execution traces in the model for debugging purposes. The two notions are dual: whenever a formula prefixed with the universal path quantifier, **A**, is not valid, the model checker provides the user with an execution trace that proves the negation of the formula (a *counterexample*). Similarly, for a valid formula prefixed with the existential path quantifier, **E**, the model checker delivers a *witness* to that.

In many cases, however, this feature is the most time- and space-consuming stage of the entire verification process. For example, [40] shows how to construct traces for queries expressed in the temporal logic CTL [36] under fairness constraints. A different approach is taken in SAT-based Model Checking [1]. The search for counterexamples is conducted incrementally, by looking for execution traces of increasing length, and using a satisfiability checker instead of BDDs to verify the desired properties written as propositional logic formulae. The resulting method, called Bounded Model Checking [9], is able to provide the shortest counterexamples. On the other hand, the method is incomplete, in the sense that it cannot give an answer when no counterexamples exist, and it is limited in practice

to finding errors with short counterexamples. The SAT-based approach becomes complete when appropriately combined with the BDD-based approach, resulting in a hybrid method [135].

Since a trace is usually meant to be examined by a human, it is particularly desirable for a model-checking tool to compute a minimal-length trace. Unfortunately, finding such trace is an NP-complete problem [80], thus a sub-optimal trace is sought in most cases. For some operators, finding minimal-length witnesses is instead easy in principle. An example is the **EF** operator, which is closely related to the (backward) reachability relation: a state satisfies **EF** $p$  if there is an execution path from it to a state where property  $p$  holds. Even using symbolic encodings [21], though, the generation and storage of the sets of states required to generate an **EF** witness can be a major limitation in practice.

In this section we investigate the possibility of using the saturation strategy to build shortest-length execution traces for reachability queries. This is based on the definition of the *distance* function between states/sets of states, which we introduce next.

## 6.1 The distance function

The *distance* of a reachable state  $\mathbf{i} \in \mathcal{S}$  from the initial state  $\mathbf{s}$  is defined as  $\delta(\mathbf{i}) = \min\{d : \mathbf{i} \in \mathcal{N}^d(\mathbf{s})\}$ . We can naturally extend  $\delta : \mathcal{S} \rightarrow \mathbb{N}$  to all states in  $\widehat{\mathcal{S}}$  by letting  $\delta(\mathbf{i}) = \infty$  for any non-reachable state  $\mathbf{i} \in \widehat{\mathcal{S}} \setminus \mathcal{S}$ . Alternatively, given such a function  $\delta : \widehat{\mathcal{S}} \rightarrow \mathbb{N} \cup \{\infty\}$ , we can identify  $\mathcal{S}$  as the subset of the domain where the function is finite:  $\mathcal{S} = \{\mathbf{i} \in \widehat{\mathcal{S}} : \delta(\mathbf{i}) < \infty\}$ .

The formulation of our problem is then: Given a description of a structured discrete-

state system  $(\widehat{S}, s, \mathcal{N})$ , determine the distance to all reachable states, i.e., compute and store  $\delta : \widehat{S} \rightarrow \mathbb{N} \cup \{\infty\}$  (note that the reachable state space  $\mathcal{S}$  is not an input, rather, it is implicitly an output). This can be viewed as a least fixed-point computation for the functional  $\Phi : \mathcal{D} \rightarrow \mathcal{D}$ , where  $\mathcal{D}$  is the set of functions mapping  $\widehat{S}$  onto  $\mathbb{N} \cup \{\infty\}$ . More precisely,  $\Phi$  refines an approximation of the distance function from the initial  $\delta^{[0]} \in \mathcal{D}$ , defined as  $\delta^{[0]}(\mathbf{i}) = 0$ , if  $\mathbf{i} = s$ ,  $\delta^{[0]}(\mathbf{i}) = \infty$  otherwise, via the iteration

$$\delta^{[m+1]}(\mathbf{i}) = \Phi(\delta^{[m]})(\mathbf{i}) = \min \left( \delta^{[m]}(\mathbf{i}), \min \left\{ 1 + \delta^{[m]}(\mathbf{i}') \mid \mathbf{i} \in \mathcal{N}(\mathbf{i}') \right\} \right).$$

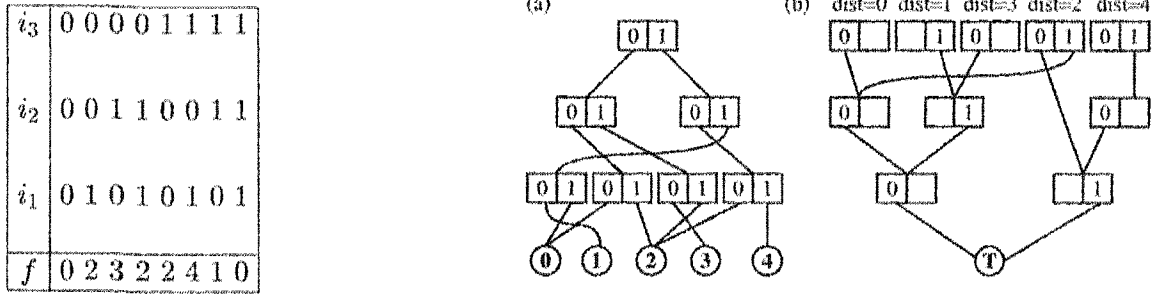
Note that the state-space construction is itself a fixed-point computation, so we seek now to efficiently combine the two fixed-point operations into one. Before showing our algorithm to accomplish this, in Section 6.2, we first describe a few approaches to compute distance information based on existing decision diagrams technology.

### 6.1.1 Explicit decision diagram encoding of state distances

Algebraic decision diagrams (ADDs) [8] are an extension of BDDs where multiple terminals are allowed (thus, they are also called MTBDDs [44]). ADDs can encode arithmetic functions from  $\widehat{S}$  to  $\mathbb{R} \cup \{\infty\}$ . The value of the function on a specific input (representing a state in our case) is the value of the terminal node reached by following the path encoding the input. While ADDs are traditionally associated to boolean argument variables, extending the arguments to finite integer sets is straightforward.

The compactness of the ADD representation is related to the merging of nodes, exploited to a certain degree in all decision diagrams. In this case, there is a unique root, but having many terminal values can greatly reduce the degree of node merging, especially at the lower





**Figure 6.1:** Storing the distance function: ADD vs. forests of MDDs.

levels, with respect to the *support decision diagram*, i.e., the MDD that encodes  $\mathcal{S} \subseteq \widehat{\mathcal{S}}$ . In other words, the number of terminal nodes for the ADD that encodes  $\delta : \widehat{\mathcal{S}} \rightarrow \mathbb{N} \cup \{\infty\}$  equals the number of distinct values for  $\delta$  (hence the “explicit” in the title of this section). If we merged all finite-valued terminals into one, thus encoding just  $\mathcal{S}$  but not the state distances, many ADD nodes may be merged into one MDD node.

An alternative explicit encoding of state distances can be achieved by simply using a forest of MDDs. This approach is derived from the traditional ROBDD method, by extending it to multi-valued variables. Each of the distance sets  $\mathcal{N}^d(\mathbf{s}) = \{\mathbf{i} \in \mathcal{S} \mid \delta(\mathbf{i}) = d\}$  (or  $\{\mathbf{i} \in \mathcal{S} \mid \delta(\mathbf{i}) \leq d\}$ , which may require fewer nodes in some cases) can be encoded using a separate MDD. Informally, this reverses the region where most sharing of nodes occurs compared to ADDs: the roots are distinct, but they may be likely to share nodes downstream.

The range of the function is critical to the compactness of either representation: the wider the range, the less likely it is that nodes are merged. Figure 6.1 (a) and (b) show an example of the same distance function represented as an ADD or as a forest of MDDs, respectively.

### 6.1.2 Symbolic encoding of state distances

In this section we introduce a new data structure that is able to store the distance function more efficiently. This can be achieved by “distributing” the values of the terminal nodes in the ADD as the composition of several numerical values associated to the arcs along paths in the decision diagram. This is based on the intuition that scattering the value that was previously “concentrated” in a single terminal node would increase the degree of merging of nodes, so that they share common arcs.

The idea of associating numerical values to the edges of regular BDDs was proposed in [93, 94], resulting in a new type of decision diagrams, edge-valued BDDs (EVBDDs)<sup>1</sup>.

**Definition 6.1.1** An EVBDD is a directed acyclic graph that encodes a total function  $f : \{0, 1\}^K \rightarrow \mathbb{Z}$  as follows:

1. There is a single terminal node, at level 0, with label 0, denoted by  $\langle 0|0 \rangle$ .
2. A non-terminal node at level  $k$ ,  $K \geq k \geq 1$ , is denoted by  $\langle k|p \rangle$ , where  $p$  is a unique identifier within level  $k$ , and has two children,  $\langle k|p \rangle[0].child$  and  $\langle k|p \rangle[1].child$  (corresponding to the two possible values in  $\mathcal{S}_k$ ) which are nodes at some (not necessarily the same) level  $l$ ,  $k > l \geq 0$ .
3. The 1-edge is labeled with an integer value  $\langle k|p \rangle[1].val \in \mathbb{Z}$ , while the label of  $\langle k|p \rangle[0].val$  is always (implicitly) 0.

---

<sup>1</sup>We observe that also binary moment diagrams (BMDs), independently introduced in [18], associate values to edges. For BMDs however, evaluating the function on a particular argument requires the traversal of multiple paths, as opposed to a unique path for EVBDDs. Thus, while very effective for verifying circuits such as a multiplier, BMDs are not as suited for our approach.

$i_3$	0	0	0	0	1	1	1	1
$i_2$	0	0	1	1	0	0	1	1
$i_1$	0	1	0	1	0	1	0	1
$f$	0	2	3	2	2	4	1	0

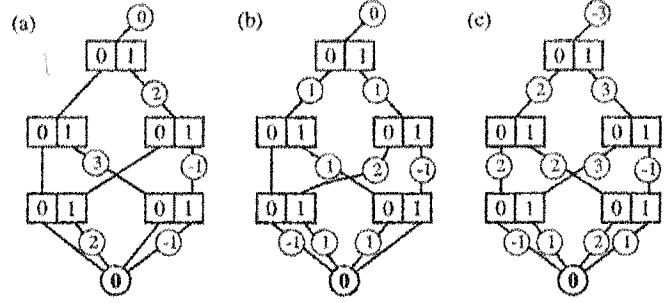


Figure 6.2: Canonical and non-canonical EVBDDs.

4. There is a single *root* node  $\langle k_r | r \rangle$ , for some  $K \geq k_r \geq 0$ , with no incoming edges, except for a “dangling” edge labeled with an integer value  $\rho \in \mathbb{Z}$ .
5. Canonicity restrictions analogous to those of reduced ordered BDDs apply:

*uniqueness*: if  $\langle k|p \rangle[0].child = \langle k|q \rangle[0].child$ ,  $\langle k|p \rangle[1].child = \langle k|q \rangle[1].child$ , and

$\langle k|p \rangle[1].val = \langle k|q \rangle[1].val$ , then  $p = q$ ;

*reducedness*: there is no *redundant* node  $\langle k|p \rangle$  satisfying  $\langle k|p \rangle[0].child = \langle k|p \rangle[1].child$

and  $\langle k|p \rangle[1].val = 0$ .

The function encoded by an EVBDD node  $\langle k|p \rangle$  is recursively defined by

$$f_{\langle k|p \rangle}(i_k, \dots, i_1) = \begin{cases} f_{\langle k|p \rangle[0].child}(i_l, \dots, i_1) & \text{if } i_k = 0 \\ f_{\langle k|p \rangle[1].child}(i_r, \dots, i_1) + \langle k|p \rangle[1].val & \text{if } i_k = 1 \end{cases}$$

where  $l$  and  $r$  are the levels of  $\langle k|p \rangle[0].child$  and  $\langle k|p \rangle[1].child$ , respectively, and  $f_{\langle 0|0 \rangle} = 0$ .

The function encoded by an EVBDD *edge*, that is, a  $(value, node)$  pair is then simply obtained by adding the constant *value* to the function encoded by the *node*. In particular, the function encoded by the EVBDD is  $f = \rho + f_{\langle k_r | r \rangle}$ .

Note that the nodes are *normalized* to enforce canonicity: the value of the 0-edge is always 0. If this requirement were relaxed, there would be an infinite number of EVBDDs

representing the same function, obtained by rearranging the edge values. An example of multiple ways to encode the function of Figure 6.1 with non-canonical EVBDDs is shown in Figure 6.2<sup>2</sup>. Only the EVBDD in Figure 6.2(a) is normalized. This node normalization implies that  $\rho = f(0, \dots, 0)$  and may require the use of both negative and positive edge values even when the encoded function is non-negative, as is the case for Figure 6.2(a). More importantly, if we want to represent functions such as our distance  $\delta : \hat{S} \rightarrow \mathbb{N} \cup \{\infty\}$ , we can allow edge values to be  $\infty$ ; however, if  $\delta(0, \dots, 0) = \infty$ , i.e., state  $(0, \dots, 0)$  is not reachable, we cannot enforce the required normalization, since this implies that  $\rho$  is  $\infty$ , and  $f$  is identically  $\infty$  as well. This prompted us to introduce a more general normalization rule, which we present next.

## 6.2 A new approach

We use quasi-reduced, ordered, non-negative edge-valued, multi-way decision diagrams. To the best of our knowledge, this is the first attempt to use edge-valued decision diagrams of any type in fixed-point computations or in the generation of traces.

### 6.2.1 Edge-valued MDDs

We extend EVBDDs in several ways. The first extension is straightforward: from binary to multi-valued variables. Then, we change the normalization of nodes to a slightly more general one needed for our task. Finally, we allow the value of an edge to be  $\infty$ , since this is required to describe our distance functions. Note that the choice to use quasi-reduced

---

<sup>2</sup>The edge values are written in circled boxes on the corresponding arcs. Also, for better readability, a missing edge-value has to be interpreted as 0 – the neutral element of integer addition, and the arcs labeled with  $\infty$  are completely eliminated.

instead of reduced decision diagrams is not dictated by limitations in the descriptive power of EVBDDs, but by efficiency considerations in the saturation algorithm.

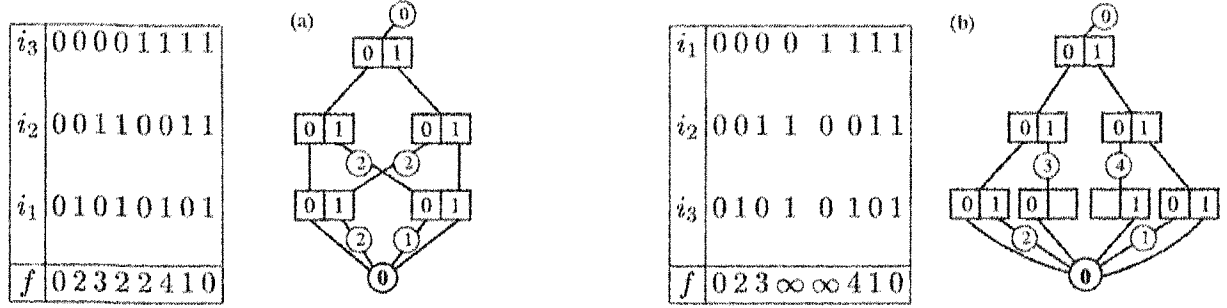
**Definition 6.2.1** Given a function  $f : \widehat{\mathcal{S}} \rightarrow \mathbb{Z} \cup \{\infty\}$ , an  $\text{EV}^+\text{MDD}$  for  $f \neq \infty$  is a directed acyclic graph with labeled edges that satisfies the following properties:

1. There is a single terminal node, at level 0, with label 0, denoted by  $\langle 0|0 \rangle$ .
2. A non-terminal node at level  $k$ ,  $K \geq k \geq 1$ , is denoted by  $\langle k|p \rangle$ , where  $p$  is a unique identifier within the level, and has  $n_k \geq 2$  edges to children,  $\langle k|p \rangle[i_k].child$ , labeled with values  $\langle k|p \rangle[i_k].val \in \mathbb{N} \cup \{\infty\}$ , for  $0 \leq i_k < n_k$ .
3. If  $\langle k|p \rangle[i_k].val = \infty$ , the value of  $\langle k|p \rangle[i_k].child$  is irrelevant, so we require it to be 0 for canonicity; otherwise,  $\langle k|p \rangle[i_k].child$  is the index of a node at level  $k - 1$ .
4. There is a single root node,  $\langle K|r \rangle$ , with no incoming edges, except for a “dangling” incoming edge labeled with an integer value  $\rho \in \mathbb{Z}$ .
5. **Each non-terminal node has at least one outgoing edge labeled with 0.**
6. All nodes are unique, i.e.,  
 if  $\forall i_k, 0 \leq i_k < n_k, \langle k|p \rangle[i_k].child = \langle k|q \rangle[i_k].child, \langle k|p \rangle[i_k].val = \langle k|q \rangle[i_k].val$ , then  
 $p = q$ .

Figure 6.3 shows two  $\text{EV}^+\text{MDD}$ s storing a total and a partial<sup>3</sup> function, respectively (the total function encoded is that of Figures 6.1 and 6.2). Note that, unlike the normalization

---

<sup>3</sup>By “partial”, we mean that some of its values can be  $\infty$ ; whenever this is the case, we omit the corresponding value and edge from the graphical representation.



**Figure 6.3:** Storing total and partial arithmetic functions with EV+MDDs.

for EVBDDs, our normalization requires that the labels on (non-dangling) edges be non-negative, and at least one per node be zero, but not in a pre-determined location; compare the EVBDD of Figure 6.2(a) with the equivalent EV+MDD of Figure 6.3(a). The function encoded by the EV+MDD node  $\langle k|p \rangle$  is

$$f_{\langle k|p \rangle}(i_k, \dots, i_1) = \langle k|p \rangle[i_k].val + f_{\langle k-1|\langle k|p \rangle[i_k].child \rangle}(i_{k-1}, \dots, i_1)$$

and we let  $f_{\langle 0|0 \rangle} = 0$ . As for EVBDDs, the function encoded by the EV+MDD  $(\rho, \langle K|r \rangle)$  is  $f = \rho + f_{\langle K|r \rangle}$ . However, now,  $\rho = \min\{f(i) : i \in \mathcal{S}_k \times \dots \times \mathcal{S}_1\}$ . In our application, we encode distances, which are non-negative, thus  $\rho = 0$ .

### 6.2.2 Canonicity of EV+MDDs

**Lemma 6.2.1** From every non-terminal EV+MDD node, there is an outgoing path with all edges labeled 0 reaching  $\langle 0|0 \rangle$ .

**Corollary 6.2.1** Function  $f_{\langle k|p \rangle}$  encoded by node  $\langle k|p \rangle$  is non-negative and  $\min(f_{\langle k|p \rangle}) = 0$ .

**Definition 6.2.2** The graphs rooted at two EV+MDD nodes  $\langle k|p \rangle$  and  $\langle k|q \rangle$  are isomorphic if there is a bijection  $b$  from the nodes of the first graph to the nodes of the second graph

such that, for each node  $\langle l|s \rangle$  of the first graph and each  $i_l \in \mathcal{S}_l$  (with  $k \geq l \geq 1$ ):

$$b(\langle l|s \rangle)[i_l].child = b(\langle l|s \rangle[i_l].child) \quad \text{and} \quad b(\langle l|s \rangle)[i_l].val = \langle l|s \rangle[i_l].val.$$

**Theorem 6.1** (*Canonicity*) If two  $\text{EV}^+\text{MDDs}$   $(\rho_1, \langle K|r_1 \rangle)$  and  $(\rho_2, \langle K|r_2 \rangle)$  encode the same function  $f : \widehat{\mathcal{S}} \rightarrow \mathbb{N} \cup \{\infty\}$ , then  $\rho_1 = \rho_2$  and the two labeled graphs rooted at  $\langle K|r_1 \rangle$  and  $\langle K|r_2 \rangle$  are isomorphic.

**Proof.** It is easy to see that, since the value on the dangling edges of the two  $\text{EV}^+\text{MDDs}$  equals the minimum value  $\rho$  the encoded function  $f$  can assume, we must have  $\rho_1 = \rho_2 = \rho$ , and the two nodes  $\langle K|r_1 \rangle$  and  $\langle K|r_2 \rangle$  must encode the same function  $f - \rho$ . We then need to prove by induction that, if two generic  $\text{EV}^+\text{MDD}$  nodes  $\langle k|p \rangle$  and  $\langle k|q \rangle$  encode the same function, the labeled graphs rooted at them are isomorphic.

**Induction base** ( $k = 1$ ): if  $\langle 1|p \rangle$  and  $\langle 1|q \rangle$  encode the same function  $f : \mathcal{S}_1 \rightarrow \mathbb{N} \cup \{\infty\}$ ,  $\langle 1|p \rangle[i_1].child = \langle 1|q \rangle[i_1].child = 0$  and  $\langle 1|p \rangle[i_1].val = \langle 1|q \rangle[i_1].val = f(i_1)$  for all  $i_1 \in \mathcal{S}_1$ , thus the two labeled graphs rooted at  $\langle 1|p \rangle$  and  $\langle 1|q \rangle$  are isomorphic.

**Induction step** (assume claim true for  $k - 1$ ): if  $\langle k|p \rangle$  and  $\langle k|q \rangle$  encode the same function  $f : \mathcal{S}_k \times \dots \times \mathcal{S}_1 \rightarrow \mathbb{N} \cup \{\infty\}$ , consider the function obtained when we fix  $i_k$  to a particular value  $t$ , i.e.,  $f_{i_k=t}$ . Let  $g$  and  $h$  be the functions encoded by  $\langle k|p \rangle[t].child$  and  $\langle k|q \rangle[t].child$ , respectively; also, let  $\langle k|p \rangle[t].val = \alpha$  and  $\langle k|q \rangle[t].val = \beta$ , and observe that the functions  $\alpha + g$  and  $\beta + h$  must coincide with  $f_{i_k=t}$ . However, because of Corollary 6.2.1, we know that both the  $g$  and  $h$  evaluate to 0, their minimum possible value, for at least one choice of the arguments  $(i_{k-1}, \dots, i_1)$ . Thus, the minimum of values  $\alpha + g$  and  $\beta + h$  can have are  $\alpha$  and  $\beta$ , respectively. Since  $\alpha + g$  and  $\beta + h$  are the same function, they must have the

<i>UnionMin</i> ( $k : \text{level}, (\alpha, p) : \text{edge}, (\beta, q) : \text{edge}) : \text{edge}$	
Computes the minimum of two partial functions	
1. if $\alpha = \infty$ then return $(\beta, q)$ ;	
2. if $\beta = \infty$ then return $(\alpha, p)$ ;	
3. if $k = 0$ then return $(\min(\alpha, \beta), 0)$ ;	• the only node at level $k = 0$ has index 0
4. if <i>Cached</i> ( <i>EVunion</i> , $k, p, q, \alpha - \beta, (\gamma, u)$ ) then	• match $(k, p, q, \alpha - \beta)$ , return $(\gamma, u)$
5. return $(\gamma + \min(\alpha, \beta), u)$ ;	
6. $u \leftarrow \text{NewNode}(k)$ ;	• create new node at level $k$ with edges set to $(\infty, 0)$
7. $\mu \leftarrow \min(\alpha, \beta)$ ;	
8. for $i_k = 0$ to $n_k - 1$ do	
9. $p' \leftarrow \langle k p \rangle.\text{child}[i_k]$ ; $\alpha' \leftarrow \alpha - \mu + \langle k p \rangle.\text{val}[i_k]$ ;	
10. $q' \leftarrow \langle k q \rangle.\text{child}[i_k]$ ; $\beta' \leftarrow \beta - \mu + \langle k q \rangle.\text{val}[i_k]$ ;	
11. $\langle k u \rangle[i_k] \leftarrow \text{UnionMin}(k-1, (\alpha', p'), (\beta', q'))$ ;	• continue downstream
12. <i>CheckIn</i> ( $k, u$ );	
13. <i>PutInCache</i> ( <i>EVunion</i> , $k, p, q, \alpha - \beta, (\mu, u)$ );	
14. return $(\mu, u)$ ;	

Figure 6.4: The *UnionMin* algorithm for EV<sup>+</sup>MDDs.

same minimum, hence  $\alpha = \beta$ . This implies that  $g = h$  and, by inductive hypothesis, that  $\langle k|p \rangle[t].\text{child}$  and  $\langle k|q \rangle[t].\text{child}$  are isomorphic. Since this argument applies to a generic child  $t$ , the two nodes  $\langle k|p \rangle$  and  $\langle k|q \rangle$  are then themselves isomorphic, completing the proof.

### 6.3 Operations with EV<sup>+</sup>MDDs

We are now ready to discuss manipulation algorithms for EV<sup>+</sup>MDDs. We do so in the context of our state-space and distance generation problem, although, of course, the *UnionMin* function we introduce in Figure 6.4 has general applicability. The new types used in the pseudocode of Figures 6.4 and 6.6 are *value* (for edge values) and *edge*, denoting a pair (*value*, *index*) (i.e., the type of  $\langle k|p \rangle[i]$ ).

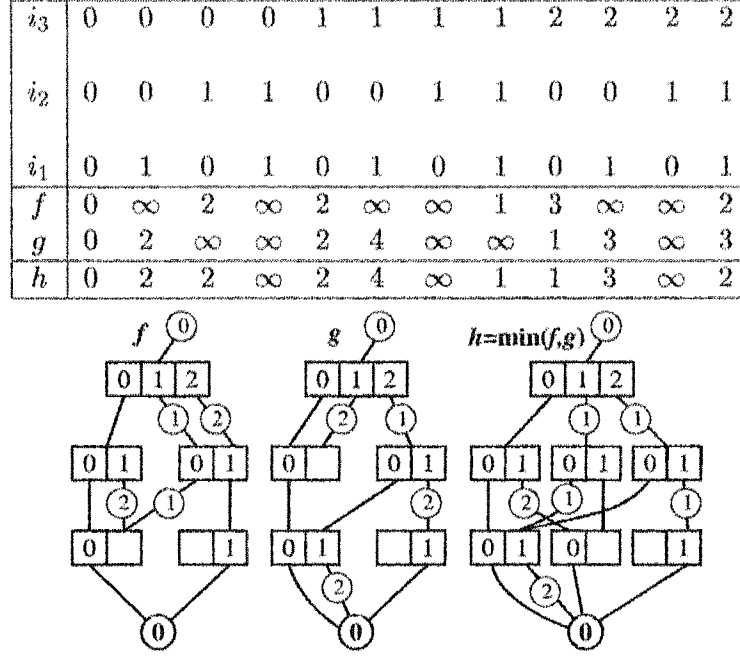


The *UnionMin* algorithm computes the minimum of two partial functions. This acts like a dual operator by performing the union on the support sets of states of the two operands (which must be defined over the same potential state space  $\hat{S}$ ), and by finding the minimum value for the common elements. The algorithm starts at the roots of the two operand EV<sup>+</sup>MDDs, and recursively descends along matching edges. If at some point one of the edges has value  $\infty$ , the recursion stops and returns the other edge (since  $\infty$  is the neutral value with respect to the minimum). If the other edge has value  $\infty$  as well, the returned value is  $(\infty, 0)$ , i.e., no states are added to the union; otherwise, if the other edge has finite value, we have just found states reachable in one set but not in the other. If the recursion reaches instead all the way to the terminal node  $\langle 0|0 \rangle$ , the returned value is the minimum of the two input values  $\alpha$  and  $\beta$ .

If both  $\alpha$  and  $\beta$  are finite and  $p$  and  $q$  are non-terminal, *UnionMin* “keeps” the minimum value on the incoming arcs to the operands,  $\mu$ , and “pushes down” any residual value  $\alpha - \mu$ , if  $\mu = \beta < \alpha$ , or  $\beta - \mu$ , if  $\mu = \alpha < \beta$ , to the children of  $p$  or  $q$ , respectively, in its recursive downstream calls. In this case, the returned edge  $(\mu, u)$  is such that  $\mu + f_{\langle k|u \rangle} = \min(\alpha + f_{\langle k|p \rangle}, \beta + f_{\langle k|q \rangle})$ .

An example of the application of the *UnionMin* algorithm is illustrated in Figure 6.5. The potential state space is  $\mathcal{S}_3 \times \mathcal{S}_2 \times \mathcal{S}_1 = \{0, 1, 2\} \times \{0, 1\} \times \{0, 1\}$ . The functions encoded by the operands,  $f$  and  $g$ , are listed in the table to the left, along with the result function  $h = \min(f, g)$ .

**Lemma 6.3.1** The call *UnionMin*( $k, (\alpha, p), (\beta, q)$ ) returns an edge  $(\mu, u)$  such that  $\mu = \min(\alpha, \beta)$  and  $\langle k|u \rangle$  and its descendants satisfy property 5 of Definition 6.2.1, if  $\langle k|p \rangle$  and



**Figure 6.5:** An example of the *UnionMin* operator for  $EV^+$ MDDs.

$\langle k|q \rangle$  do.

**Proof.** It is immediate to see that  $\mu = \min(\alpha, \beta)$ . To prove that  $\langle k|u \rangle$  satisfies property 5, we use induction: if  $k = 0$ , there is nothing to prove, since property 5 applies to non-terminal nodes only. Assume now that the lemma is true for all calls at level  $k-1$  and consider an arbitrary call  $UnionMin(k, (\alpha, p), (\beta, q))$ , where the input nodes  $\langle k|p \rangle$  and  $\langle k|q \rangle$  satisfy property 5. If  $\alpha$  or  $\beta$  is  $\infty$ , the returned node is one of the input nodes, so it satisfies property 5. Otherwise, since  $\mu = \min(\alpha, \beta)$ , at least one of  $\alpha - \mu$  and  $\beta - \mu$  is 0; say  $\alpha - \mu = 0$ . The values labeling the edges of  $\langle k|u \rangle$  are computed in line 11 of *UnionMin*. Since  $\langle k|p \rangle$  satisfies property 5, there exists  $i_k \in \{0, \dots, n_k - 1\}$  such that  $\langle k|p \rangle.val[i_k] = 0$ . Then, for the corresponding iteration of the for-loop,  $\alpha'$  is 0 and the edge

returned by  $UnionMin(k-1, (\alpha', p'), (\beta', q'))$  is  $(\min(\alpha', \beta'), u') = (0, u')$ , where  $\langle k-1 | u' \rangle$  satisfies property 5 by induction; thus,  $\langle k | u \rangle[i_k].val$  is set to 0.

We conclude the discussion of  $UnionMin$  by observing that the hash-key for the entries in the union/min cache is formed by the two nodes (passed as *level*, *index*, *index*, since the nodes are at the same level) plus the **difference**  $\alpha - \beta$  of the values labeling two edges pointing to these nodes. This is better than using the key  $(k, p, q, \alpha, \beta)$ , which would unnecessarily clutter the cache with entries of the form  $(k, p, q, \alpha + \tau, \beta + \tau, (\mu + \tau, u))$ , for all the values of  $\tau$  arising in a particular execution.

### 6.3.1 State-space and distance generation using EV<sup>+</sup>MDDs

Our fixed-point algorithm to build and store the distance function  $\delta$ , and implicitly the state space  $\mathcal{S}$ , is described by the pseudo-code for  $BuildDistance$ ,  $EVSaturate$ , and  $EVRecFire$ , shown in Figures 6.6-6.7. Given a model  $(\hat{\mathcal{S}}, \mathbf{s}, \mathcal{N})$  we follow these steps:

1. Encode  $\mathbf{s}$  into an initial EV<sup>+</sup>MDD node  $\langle K | r \rangle$ . This can be done by building the MDD for  $\mathbf{s}$ , then setting to 0 all edge values for edges going to *true* (called 1 in the MDD terminology of [27]), setting the remaining edge values to  $\infty$ , eliminating the terminal node *false*, and renaming the terminal node *true* as 0 (in EV<sup>+</sup>MDD terminology). See [27] on how to build an MDD when  $\mathbf{s}$  contains a single state. In general, the MDD encoding of  $\mathbf{s}$  will be derived from some other symbolic computation, e.g., it will be already available as the result of a temporal logic query.
2. Call  $BuildDistance(K, r)$ .

<p><i>BuildDistance</i>(<math>k : \text{level}, p : \text{index}</math>)</p> <ol style="list-style-type: none"> <li>1. if <math>k &gt; 0</math> then</li> <li>2. for <math>i_k = 0</math> to <math>n_k - 1</math> do</li> <li>3. if <math>\langle k p \rangle[i_k].\text{val} &lt; \infty</math> then <i>BuildDistance</i>(<math>k - 1, \langle k p \rangle[i_k].\text{child}</math>);</li> <li>4. <i>EVSaturate</i>(<math>k, p</math>);</li> </ol>
<p><i>EVSaturate</i>(<math>k : \text{level}, p : \text{index}</math>)</p> <ol style="list-style-type: none"> <li>1. repeat</li> <li>2. <math>p\text{Changed} \leftarrow \text{false}</math>;</li> <li>3. foreach <math>e \in \mathcal{E}_k</math> do <math>p\text{Changed} \leftarrow p\text{Changed} \vee \text{EVFire}(e, k, p)</math></li> <li>4. until <math>p\text{Changed} = \text{false}</math>;</li> <li>5. <i>CheckIn</i>(<math>k, p</math>);</li> </ol>
<p><i>EVFire</i>(<math>e : \text{event}, k : \text{level}, p : \text{index}</math>)</p> <ol style="list-style-type: none"> <li>1. <math>p\text{Changed} \leftarrow \text{false}</math></li> <li>2. <math>\mathcal{L} \leftarrow \{i_k : \mathcal{N}_{k,e}(i_k) \neq \emptyset \wedge \langle k p \rangle[i_k].\text{val} \neq \infty\}</math></li> <li>3. while <math>\mathcal{L} \neq \emptyset</math> do</li> <li>4. pick and remove <math>i_k</math> from <math>\mathcal{L}</math>;</li> <li>5. <math>(\alpha, f) \leftarrow \text{EVRecFire}(e, k - 1, \langle k p \rangle[i_k])</math>;</li> <li>6. if <math>\alpha \neq \infty</math> then</li> <li>7. foreach <math>j_k \in \mathcal{N}_{k,e}(i_k)</math> do</li> <li>8. <math>(\beta, u) \leftarrow \text{UnionMin}(k - 1, (\alpha + 1, f), \langle k p \rangle[j_k])</math>;</li> <li>9. if <math>(\beta, u) \neq \langle k p \rangle[j_k]</math> then</li> <li>10. <math>\langle k p \rangle[j_k] \leftarrow (\beta, u)</math>;</li> <li>11. <math>p\text{Changed} \leftarrow \text{true}</math>;</li> <li>12. if <math>\mathcal{N}_{k,e}(j_k) \neq \emptyset</math> then <math>\mathcal{L} \leftarrow \mathcal{L} \cup \{j_k\}</math>; <span style="float: right;">• remember to explore <math>j_k</math> later</span></li> </ol>

Figure 6.6: The pseudocode for *BuildDistance*, *EVSaturate* and *EVFire*

```

EVRecFire( $e : \text{event}, k : \text{level}, (\alpha, q) : \text{edge}$ ) :  $\text{edge}$ 
1. if  $k < \text{Bot}(e)$  then return  $(\alpha, q)$ ;           • level  $k$  is not affected by event  $e$ 
2. if Cached(EVFIRE,  $k, q, e, (\gamma, s)$ ) then      • match  $(k, q, e)$ , return  $(\gamma, s)$ 
3. return  $(\gamma + \alpha, s)$ ;
4.  $s \leftarrow \text{NewNode}(k)$ ;                       • create new node at level  $k$  with edges set to  $(\infty, 0)$ 
5.  $s\text{Changed} \leftarrow \text{false}$ ;
6.  $\mathcal{L} \leftarrow \{i_k : \mathcal{N}_{k,e}(i_k) \neq \emptyset \wedge \langle k|q \rangle[i_k].\text{val} \neq \infty\}$ 
7. while  $\mathcal{L} \neq \emptyset$  do
8.   pick and remove  $i_k$  from  $\mathcal{L}$ ;
9.    $(\phi, f) \leftarrow \text{EVRecFire}(e, k-1, \langle k|q \rangle[i_k])$ ;
10.  if  $\phi \neq \infty$  then
11.    foreach  $j_k \in \mathcal{N}_{k,e}(i_k)$  do
12.       $(\beta, u) \leftarrow \text{UnionMin}(k-1, (\phi, f), \langle k|s \rangle[j_k])$ ;
13.      if  $(\beta, u) \neq \langle k|s \rangle[j_k]$  then
14.         $\langle k|s \rangle[j_k] \leftarrow (\beta, u)$ ;
15.         $s\text{Changed} \leftarrow \text{true}$ ;
16. if  $s\text{Changed}$  then EVSaturate( $k, s$ );
17.  $\gamma \leftarrow \text{Normalize}(k, s)$ ;
18.  $s \leftarrow \text{CheckIn}(k, s)$ ;
19. PutInCache(EVFIRE,  $k, q, e, (\gamma, s)$ );
20. return  $(\gamma + \alpha, s)$ ;

```

Figure 6.7: The pseudo-code for *EVRecFire*.

The new function *Normalize*( $k, s$ ) puts node  $\langle k|s \rangle$  in canonical form by computing  $\mu = \min\{\langle k|s \rangle[i_k].\text{val} : i_k \in \mathcal{S}_k\}$  and subtracting  $\mu$  from each  $\langle k|s \rangle[i_k].\text{val}$  (so that at least one of them becomes 0), then returns  $\mu$ ; in particular, if all edge values in  $\langle k|s \rangle$  are  $\infty$ , it returns  $\infty$  (this is the case in line 17 of *EVRecFire* if the while-loop did not manage to fire  $e$  from any of the local states in  $\mathcal{L}$ ).

The hash-key for the firing cache does not use the value  $\alpha$  on the incoming edge, because the node  $\langle k|s \rangle$  corresponding to the result  $(\gamma, s)$  of *EVRecFire* is independent of this quantity. The edge value returned by *EVRecFire* depends instead on  $\alpha$ : it is simply obtained by adding the result of *Normalize*( $k, s$ ) to  $\alpha$ .

*EVRecFire* may push excess values upwards when normalizing a node in line 17, that is, residual values are moved in the opposite direction as in *UnionMin*. However, the normalization procedure is called only once per node (when the node has been saturated), therefore excess values are not bounced back and forth repeatedly along edges.

### 6.3.2 Trace generation using EV<sup>+</sup>MDDs

Once the EV<sup>+</sup>MDD  $(\rho, \langle K|r \rangle)$  encoding  $\delta$  and  $\mathcal{S}$  is built, a shortest-length trace from any of the states in  $\mathbf{s}$  to one of the states in a set  $\mathcal{X}$  (given at input as an MDD) can be obtained by backtracking. For simplicity, the following algorithm does not output the identity of the events along the trace, but this option could be easily added, if desired:

1. Transform the MDD for  $\mathcal{X}$  into an EV<sup>+</sup>MDD  $(\rho_x, \langle K|x \rangle)$  encoding  $\mathcal{X}$  and  $\delta_x$  using the approach previously described for  $\mathbf{s}$ , where  $\delta_x(\mathbf{i}) = 0$  if  $\mathbf{i} \in \mathcal{X}$  and  $\delta_x(\mathbf{i}) = \infty$  if  $\mathbf{i} \in \widehat{\mathcal{S}} \setminus \mathcal{X}$ .
2. Compute  $\text{IntersectionMax}(K, (\rho, r), (\rho_x, x))$ , which is the dual of *UnionMin*, and whose pseudo-code is exactly analogous; let  $(\mu, \langle K|m \rangle)$  be the resulting EV<sup>+</sup>MDD, which encodes  $\mathcal{X} \cap \mathcal{S}$  and the restriction of  $\delta$  to this set ( $\mu$  is then the length of one of the shortest-paths we are seeking).
3. Extract from  $(\mu, \langle K|m \rangle)$  a state  $\mathbf{j}^{[\mu]} = (\mathbf{j}_K^{[\mu]}, \dots, \mathbf{j}_1^{[\mu]})$  encoded by a path from  $\langle K|m \rangle$  to  $\langle 0|0 \rangle$  labeled with 0 values ( $\mathbf{j}^{[\mu]}$  is a state in  $\mathcal{X}$  at the desired minimum distance  $\mu$  from  $\mathbf{s}$ ). The algorithm proceeds now with an explicit flavor.
4. Initialize  $\nu$  to  $\mu$  and iterate:

- (a) Find all states  $\mathbf{i} \in \widehat{\mathcal{S}}$  such that  $\mathbf{j}^{[\nu]} \in \mathcal{N}(\mathbf{i})$ . With our boolean Kronecker encoding of  $\mathcal{N}$ , this “one step backward” is easily performed: we simply have to use the transpose of the matrices  $\mathbf{N}_{k,e}$ .
  - (b) For each such state  $\mathbf{i}$ , compute  $\delta(\mathbf{i})$  using  $(\rho, \langle K|r \rangle)$  and stop on the first  $\mathbf{i}$  such that  $\delta(\mathbf{i}) = \nu - 1$  (there exists at least one such state  $\mathbf{i}^*$ ).
  - (c) Decrement  $\nu$ .
  - (d) Let  $\mathbf{j}^{[\nu]}$  be  $\mathbf{i}^*$ .
5. Output  $\mathbf{j}^{[0]}, \dots, \mathbf{j}^{[\mu]}$ .

The cost of obtaining  $\mathbf{j}^{[\mu]}$  as the result of the *IntersectionMax* operation is  $O(\# \langle K|r \rangle \cdot \# \langle K|x \rangle)$ , where  $\#$  indicates the number of EV<sup>+</sup>MDD nodes. The complexity of the rest of the algorithm is then simply  $O(\mu \cdot M \cdot K)$ , where  $M$  is the maximum number of incoming arcs to any state in the reachability graph of the model, i.e.,  $M = \max\{|\mathcal{N}^{-1}(\mathbf{j})| : \mathbf{j} \in \mathcal{S}\}$ , and  $K$  comes from traversing one path in the EV<sup>+</sup>MDD. In practice  $M$  is small but, if this were not the case, the set  $\mathcal{N}^{-1}(\mathbf{j}^{[\nu]})$  could be computed symbolically at each iteration instead.

Generating the same trace using traditional symbolic approaches could follow a similar idea. If we used ADDs, we would start with an ADD encoding the same information as the EV<sup>+</sup>MDD  $(\rho_x, \langle K|x \rangle)$ , compute the ADD equivalent to the EV<sup>+</sup>MDD  $(\mu, \langle K|m \rangle)$  using a breadth-first approach, and pick as  $\mathbf{j}^{[\mu]}$  any state leading to a terminal with minimal value  $\mu$ . If we used a forest of MDDs, we would compute  $\mu = \min\{d : \mathcal{N}^d(\mathbf{s}) \cap \mathcal{X} \neq \emptyset\}$ , and pick as  $\mathbf{j}^{[\mu]}$  any state in  $\mathcal{N}^\mu \cap \mathcal{X}$ . Then, the backtracking would proceed in exactly the same way.

In either case, however, we are discovering states symbolically in breadth-first order, thus we could choose to perform an intersection with  $\mathcal{X}$  after finding each additional set of states  $\mathcal{N}^d$ , and stop as soon as  $\mathcal{N}^d(\mathbf{s}) \cap \mathcal{X} \neq \emptyset$ . Overall, we would then have explored only  $\{\mathbf{i} : \delta(\mathbf{i}) \leq \mu\}$ , which might be a strict subset of the entire state space  $\mathcal{S}$ . However, two observations are in order. First, while this “optimization” manages fewer *states*, it may well require many more *nodes* in the symbolic representation: decision diagrams are quite counterintuitive in this respect. Second, in many verification applications, the states in  $\mathcal{X}$  satisfy some property, e.g., “being a deadlock”, and they can only be reached in some obscure and tortuous way, so that the minimum distance  $\mu$  to any state in  $\mathcal{X}$  is in practice close, if not equal, to the maximum distance  $\rho$  to any of the states in  $\mathcal{S}$ .

The advantage of our approach is that, while it must explore the entire  $\mathcal{S}$ , it can do so using the idea of saturation, thus the resulting decision diagrams are built much more efficiently and require much less memory than with breadth-first approaches. The following section confirms this, focusing on the first and expensive phase of trace generation, the computation of the distance information, since the backtracking phase has negligible cost in comparison and is in any case essentially required by any approach.

## 6.4 Results

To stress the importance of using a saturation-based approach, we compare the three types of encodings for the distance function we have discussed, EV<sup>+</sup>MDDs, forests of MDDs, and ADDs, in conjunction with two iteration strategies, based on breadth-first and saturation, respectively (see Table 6.1). Since only breadth-first is applicable in the case of forests of



**Table 6.1:** Experimental results for computing the distance function.

N	S	Time in seconds					Number of nodes									
							final			peak						
		E <sub>s</sub>	E <sub>b</sub>	M <sub>b</sub>	A <sub>s</sub>	A <sub>b</sub>	E <sub>s</sub> E <sub>b</sub>	M <sub>b</sub>	A <sub>s</sub> A <sub>b</sub>	E <sub>s</sub>	E <sub>b</sub>	M <sub>b</sub>	A <sub>s</sub>	A <sub>b</sub>		
Dining philosophers: D=2N, K=⌊N/2⌋,  S <sub>k</sub>  =34 for all k except  S <sub>1</sub>  =8 when N is odd																
5	1.3·10 <sup>3</sup>	0.00	0.01	0.01	0.01	0.03	11	83	38	11	155	172	48	434		
10	1.9·10 <sup>6</sup>	0.01	0.06	0.05	0.12	0.46	21	255	170	21	605	644	238	4022		
20	3.5·10 <sup>12</sup>	0.01	0.34	0.28	1.64	9.00	46	1100	740	46	2990	3079	1163	38942		
25	4.7·10 <sup>15</sup>	0.01	0.59	0.47	4.09	26.08	61	1893	1178	61	5215	5334	1958	79674		
30	6.4·10 <sup>18</sup>	0.02	0.86	0.70	7.39	56.80	71	2545	1710	71	7225	7364	2788	140262		
1000	9.2·10 <sup>626</sup>	0.48	—	—	—	—	2496	—	—	2496	—	—	—	—		
Kanban system: D=14N, K=4,  S <sub>k</sub>  =(N+3)(N+2)(N+1)/6 for all k																
3	5.8·10 <sup>4</sup>	0.01	0.02	0.02	0.04	0.17	7	180	68	29	454	464	284	3133		
5	2.5·10 <sup>6</sup>	0.02	0.14	0.12	0.24	1.55	9	444	133	57	1132	1156	776	13241		
7	4.2·10 <sup>7</sup>	0.04	0.51	0.42	0.94	7.79	11	848	218	93	2112	2166	1600	35741		
10	1.0·10 <sup>9</sup>	0.16	2.10	1.68	4.68	48.86	14	1673	383	162	4041	4160	3616	98843		
12	5.5·10 <sup>9</sup>	0.34	4.34	3.45	11.08	129.46	16	2368	518	218	5633	5805	5585	165938		
50	1.0·10 <sup>16</sup>	179.48	—	—	—	—	58	—	—	2802	—	—	—	—		
Flex. manuf. syst.: D=14N, K=19,  S <sub>k</sub>  =N+1 for all k except  S <sub>17</sub>  =4,  S <sub>12</sub>  =3,  S <sub>2</sub>  =2																
3	4.9·10 <sup>4</sup>	0.00	0.12	0.09	0.26	1.58	88	1925	1191	116	5002	5187	2075	37657		
5	2.9·10 <sup>6</sup>	0.01	0.42	0.34	0.88	11.78	149	5640	2989	211	15205	15693	4903	179577		
7	6.6·10 <sup>7</sup>	0.02	1.05	0.85	2.08	65.32	222	12070	5739	326	32805	33761	9027	523223		
10	2.5·10 <sup>9</sup>	0.04	2.96	2.40	5.79	608.92	354	28225	11894	536	76676	78649	17885	1681625		
140	2.0·10 <sup>23</sup>	20.03	—	—	—	—	32012	—	—	52864	—	—	—	—		
Round-robin mutex protocol: D=8N-6, K=N+1,  S <sub>k</sub>  =10 for all k except  S <sub>1</sub>  =N+1																
10	2.3·10 <sup>4</sup>	0.01	0.06	0.05	0.22	0.50	92	1038	1123	107	1898	1948	1210	9245		
15	1.1·10 <sup>6</sup>	0.01	0.15	0.14	1.00	2.93	177	2578	3136	212	4774	4885	3308	34897		
20	4.7·10 <sup>7</sup>	0.02	0.32	0.31	3.10	12.62	287	4968	6619	322	9270	9467	6901	92140		
25	1.8·10 <sup>9</sup>	0.03	0.59	0.54	7.89	52.29	422	8333	11947	477	15636	15944	12364	198839		
30	7.2·10 <sup>10</sup>	0.05	0.95	0.89	16.04	224.83	582	12798	19495	637	24122	24566	20072	376609		
200	7.2·10 <sup>62</sup>	1.63	—	—	—	—	20897	—	—	21292	—	—	—	—		

MDDs, this leads to five cases:  $EV^+$ MDD with saturation ( $E_s$ ),  $EV^+$ MDD with breadth-first ( $E_b$ ), forest of MDDs with breadth-first ( $M_b$ ), ADD with saturation ( $A_s$ ), and ADD with breadth-first ( $A_b$ ). Note that only  $M_b$  and  $A_b$  have been used in the literature before [8, 39], while  $E_s$  and  $E_b$  use our new data structure and  $A_s$  applies the idea of saturation to ADDs, thus it is also a new approach. The ADD algorithms are conceptually similar to the  $EV^+$ MDD ones with several differences in the *UnionMin* and *RecFire* procedures. The

base cases for the ADD version of  $UnionMin(k, p, q)$  are

```

if  $p = q \vee p = \infty$  return  $q$ ;

else if  $q = \infty$  return  $p$ ;

else if  $k = 0$  return  $\min(p, q)$ ;

```

For the ADD version of the  $RecFire(e, l, k, p)$  procedure, we observe that, for all firings, the recursive calls have to reach the terminal nodes on all occasions, hence they no longer stop at level  $Bot(e)$ . The base case is then

```

if  $l = 0$  then return  $p + 1$ ;

```

We implemented the five algorithms in SMART and used them to generate the distance function for the entire state space. The suite of examples is chosen from the same benchmark we used in previous experiments. For each model, we list the maximum distance  $D$ , the number  $K$  of levels in the decision diagram, and the sizes of the local state spaces. For each experiment we list the maximum distance to a reachable state, which is also the number of iterations in the breadth-first approaches, the runtime, and the number of nodes (both final and peak).

In terms of runtime, there is a clear order:  $E_s < E_b < M_b < A_s < A_b$ , with  $E_s$  easily managing much larger systems;  $E_s, E_b < M_b < A_s, A_b$  clearly attests to the effectiveness of the data structures, while  $E_s < E_b$  and  $A_s < A_b$  attest to the improvements obtainable with saturation-based approaches.

With  $EV^+$ MDDs, in particular with  $E_s$ , we can scale up the models to huge parameters. The other two data structures do not scale up nearly as well and run out of memory. In terms of memory consumption:  $E_s < A_s < E_b \approx M_b < A_b$  for the peak number of nodes,

while  $E_s = E_b < A_s = A_b \approx M_b$  for the final number of nodes. The key observation is that  $E_s$  substantially outperforms all other methods. Compared to  $A_b$ , it is over 1,000 times faster and uses fewer peak nodes, also by a factor of 1,000.

## Chapter 7

# Application: the Runway Safety Monitor

In this chapter, we present the results of applying our new model checker to a real-life industrial-size application: the Runway Safety Monitor. The application was made available to us through a collaboration with NASA Langley Research Center.

### 7.1 The Runway Incursion Prevention System

The Runway Safety Monitor (RSM) is a component of NASA's Runway Incursion Prevention System (RIPS) research. Designed and implemented by engineers at Lockheed Martin, RSM is intended to be incorporated in the Integrated Display System (IDS), under development since 1993. IDS also includes other aircraft and ground-based incursion detection and prevention algorithms, such as TCAS [60]. The IDS design makes it easy for RSM to exploit the existing data communications facilities, displays, Global Positioning System (GPS), available ground surveillance system information, and data links.

Collision avoidance protocols already exist and operate. TCAS has been in use since

1994, and it is nowadays required by FAA on all commercial US aircraft. TCAS has a full formal specification, but it has been verified only partially, not fully, due to its complexity which arises mostly from modeling the trajectories of planes with trigonometric functions and linear equations.

Unlike TCAS, the goal of RSM is not to *prevent* hazards, but rather to *detect* them. The runway incursions are defined by the Federal Aviation Administration [73] as:

“any occurrence at an airport involving an aircraft, vehicle, person, or object on the ground, that creates a collision hazard or results in the loss of separation with an aircraft taking off, intending to take off, landing, or intending to land”.

The prevention part is provided by other components of RIPS in the form of a number of IDS capabilities: heads up display (HUD), electronic moving map (EMM), cockpit display of traffic information (CDTI), taxi routing, and others.

Experimental studies conducted by Lockheed Martin [73] showed that “incursion situations are less likely to occur when IDS technology is employed on aircraft, but in the event that IDS fails to provide complete situational awareness, RSM plays an even more important role.”

Figure 7.1 shows the high level architecture of the RSM algorithm. RSM runs on a device, installed in the cockpit, that is turned on before takeoff and landing procedures on airports. The traffic is monitored in a zone surrounding the runway where the takeoff or landing is to take place. The zone is a three-dimensional volume of space that runs 220 feet laterally from the edge of the runway, up to 400 feet of altitude above the runway, and 1.1 nautical miles from the runway ends. The 400 feet altitude corresponds to a glide slope

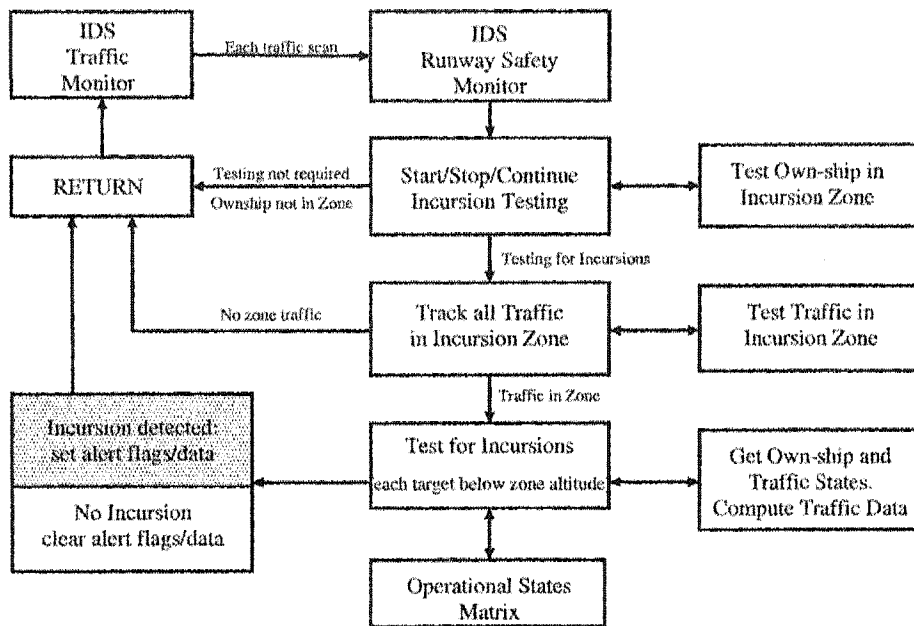


Figure 7.1: The Runway Safety Monitor flow chart.

of 3 degrees for takeoff/landing trajectories.

The protocol refers to the aircraft that is operating on as *ownship*, and to other aircraft, ground vehicles using the same runway, or even physical obstacles, such as equipment, as *targets*.

The protocol, written in the C language, consists of a repeat loop containing three major phases. In the first phase, the device gathers traffic information from radar updates (coming through a data-link). It identifies each target present in the monitored zone and stores its 3-D physical coordinates. Updates may not be received with a constant frequency, but rather irregularly, and can even be faulty. The implications of data link errors or lack of input are not addressed in this study, as it is a challenging task in itself. It has already been the subject of some experimental measurements, and the analysis of faulty behavior carries a stochastic/probabilistic flavor, not captured in our present model, which is concerned only

with *logical* errors.

In the second phase of each iteration, the algorithm assigns to each target a *status*, from a pre-determined set of values that includes taxi, pre-takeoff, takeoff, climb-out, landing, roll-out and fly-through modes. The definition of these states is described in detail when discussing our model of the system.

The third phase is responsible for detecting the incursions, and is performed for each target based on the combination of physical coordinates of ownship and the target (position, heading, etc), and some additional conditions. Table 7.5 shows the operational state matrix of the testing for incursions phase. Our analysis is focused on certifying that this decision procedure is able to detect all possible incursion scenarios.

The model of the RSM protocol in our tool SMART is described next.

## 7.2 The SMART model of RSM

The process of extracting a Petri net model from the specifications of RSM is a typical software engineering task, which starts with identifying the relevant variables to describe the states of the system and the events that transition the system from one state to another. At the end of several iterations of refinement, our model included the variables discussed below.

### 7.2.1 State variables

The number of tracked targets, *num\_targets*, plus ownship as target number 0, gives the number of subsystems in the model. For each subsystem, the relevant attributes are:

- location : the 3D coordinates, as a vector of three variables  $(x, y, z)$ , where the  $X$ -axis is across the width of the runway, the  $Y$ -axis is along the length, and the  $Z$ -axis is on the vertical;
- speed and heading: packed together in a speed vector, for which we need three more variables,  $(vx, vy, vz)$ ,
- acceleration along the runway: one variable  $a_y$ , for which only its sign (positive, negative, or zero) is relevant;
- status : one variable;
- alarm flag : one variable;

All other variables can be removed from the model, which helps in significantly reducing the size of the state-space.

Since SMART operates on discrete-type systems, abstraction (by discretization) is needed to cope with continuous-type variables of RSM. Coming up with “good” representations of the variable domains can start with the “roughest” possible discretization that can be extracted from the protocol specifications, and then refined to an acceptable solution. As modelers, we had to take into consideration a balanced solution between a very rough discretization (which potentially hides too many meaningful behaviors by merging distinct states into a single representative), and a discretization that is too fine to allow an efficient state-space generation (the state-space explosion problem). At the end of this decision making process, the domains were chosen as follows:



- The 3-D coordinates: may be as simple as  $x, y, z \in \{0, 1, 2\}$ : out of the zone, in the vicinity, or on the runway, but a better representation is:

$$\begin{cases} x \in \{0, \dots, max\_x\} \\ y \in \{0, \dots, max\_y\} \\ z \in \{0, \dots, max\_z\} \end{cases}$$

where 0 means outside the zone, and the constants  $max\_x, max\_y, max\_z$  can be adjusted to the user's preference;

- Speed: the only relevant information in the specification is the taxi speed threshold, which is used to classify targets that move slower than 45 knots as being in taxi mode; hence, variables  $vx, vy, vz$  can simply be assigned to the domain

$$vx, vy, vz \in \{0, \pm 1 \sim \text{"slow"}, \pm 2 \sim \text{"fast"}\}$$

Again, a more general representation is  $\{-max\_speed, \dots, 0, \dots, max\_speed\}$ , using another adjustable parameter  $max\_speed$ .

- Status: enumerated type,  $\{out, taxi, pretakeoff, takeoff, climb, land, rollout, flythru\}$ ;
- Alarm: boolean.

The acceleration variable, which helps discern between the status *takeoff* and status *rollout*, can be eliminated altogether at this point, since its value can be computed on the spot, based on the variation of the variable *speed*: the acceleration along the *Y*-axis is positive in the current state if there has been an increase in *vy* compared to the previous update, negative for a decrease in the value of *vy*, and zero, otherwise.

At the end of this first design stage, our Petri net model consists of a number of identical subnets (one for each target and ownship) with one place for each variable. For the

coordinates  $(x, y, z)$ , the number of tokens in a place represents the (integer) value of the variable. For the representation of the speed, however, since a place cannot hold a negative number of tokens, the values of  $vx$ ,  $vy$ , and  $vz$  can be obtained from the marking of the corresponding places with tokens by subtracting  $max\_speed$ : a number of tokens in the interval  $[0..max\_speed - 1]$  represents a negative speed,  $max\_speed$  tokens represents no speed, and the interval  $[max\_speed + 1..2 * max\_speed]$  encodes all positive speed values.

The status variable can be modeled with a single place holding from 0 to 7 tokens, encoding one of its possible enumerated values. For better readability, we adopt an equivalent encoding with eight different safe places (holding 0 or 1 tokens only): *st\_out*, *st\_taxi*, *st\_pretakeoff*, *st\_takeoff*, *st\_climb*, *st\_land*, *st\_rollout*, *st\_flythru*. In each subnet, exactly one of these places can hold a token at any moment

To complete our structural approach, we have to define a Kronecker consistent partitioning of the net. Since the modeling of transitions influences this decision, we discuss this issue in Section 7.2.4.

### 7.2.2 Modeling the state transitions

The events in the model can be grouped in three categories, corresponding to the three phases executed in each iteration of the algorithm:

1. Update aircraft data from the radar scans;
2. Update the status of each target;
3. Set or reset alarm (for each target);

This execution flow suggests that the model displays a globally asynchronous / locally synchronous behavior. The updates on each target can be performed concurrently and independently of each other, while every phase ends with a synchronization step, to ensure that, before assigning the status to every target, all coordinates have been read, and also that before taking the decision to raise an alarm or not, the status has been updated and is consistent with the physical coordinates.

To establish this execution flow, we introduce one more variable per target,  $phase \in \{ph\_radar, ph\_status, ph\_alarm\}$ , that governs the three phases.

### 7.2.2.1 The 3-D motion of targets

By introducing a discretization function, we have practically divided the monitored volume of space into a number of smaller volumes arranged in a three-dimensional grid. As a result, the positions of the aircraft are identified only by a finite number of positions in the grid, from the discrete domain  $\{0, \dots, max\_x\} \times \{0, \dots, max\_y\} \times \{0, \dots, max\_z\}$ . Similarly, the continuous trajectories are identified by an abstract, discretized trajectory traversing the small volumes of the 3D grid. This abstraction technique is frequently used in practice to deal with continuous type variables.

We use some additional modeling rules:

- Location  $(0, 0, 0)$  represents all positions outside the zone. A target that exits the zone or has not entered is assigned this position. No other location can have a 0 value for  $x$ ,  $y$  or  $z$ .
- As an alternative, we could use an outer layer of states surrounding the extended zone

to further distinguish between locations outside the zone, but this would unnecessarily increase the state space with entries of the type  $(0, y, z)$ ,  $(x, 0, z)$ , and  $(x, y, 0)$ , all representing the same state: not monitored.

- Movement is allowed only between adjacent “cubes”, and consistent with the heading: coordinate  $x$  cannot decrease if  $vx$  is positive (unless it becomes 0 upon exiting the zone), cannot increase if  $vx$  is negative, and the equivalent rules for the other axes.
- When a target enters the zone, its position is non-deterministically chosen on the frontier of the volume (i.e.,  $x \in \{1, max\_x\}$  or  $y \in \{1, max\_y\}$  or  $z = max\_z$ ). However the entry speed parameters have to be consistent with the point of entry. For example, one cannot enter from the left (i.e. crossing the 2-D plane  $x = 1$  from an outside position) with a negative  $vx$ . Also, out of the six 2D planes of the frontier of the zone, only five can be actually crossed (no entry is possible from “below ground”).

Note that the model might include unrealistic trajectories. This is acceptable in the verification process as long as all realistic behaviors (all the behaviors we are interested in) are covered by the model. Consequently, if a universal property holds in the abstract model, then it must hold in the realistic model. If a property does not hold, then the provided counterexample has to be checked to determine whether it represents a real counterexample or not. This is a typical procedure when working with abstraction techniques.

#### 7.2.2.2 Status definitions

The value of the status variable for each target is set in the second phase of the execution loop. Using our model variables, we define each value of this variable as follows ( $TS$  denotes

the taxi speed threshold of 45 knots):

1. *out*: not in the monitored zone  $\equiv$   
 $(x = 0) \wedge (y = 0) \wedge (z = 0)$
2. *taxi*: on the ground and (at low speed or not runway heading)  $\equiv$   
 $(z = 1) \wedge ((|vx| \leq TS \wedge |vy| \leq TS) \vee (vx \neq 0))$
3. *takeoff\_roll*: on the ground, runway heading, accelerating  $\equiv$   
 $(z = 1) \wedge (|vy| > TS) \wedge (vx = 0) \wedge (a_y > 0)$
4. *rollout*: on the ground, runway heading, decelerating  $\equiv$   
 $(z = 1) \wedge (|vy| > TS) \wedge (vx = 0) \wedge (a_y < 0)$
5. *climbout*: airborne, runway heading and positive vertical speed  $\equiv$   
 $(z > 1) \wedge (vx = 0) \wedge (vz > 0)$
6. *landing*: airborne, runway heading and negative vertical speed  $\equiv$   
 $(z > 1) \wedge (vx = 0) \wedge (vz \leq 0)$
7. *flythru*: airborne and not climbout or landing  $\equiv$   
 $(z > 1) \wedge (vx \neq 0)$

Note that predicates  $(z = 1)$  and  $(z > 1)$  automatically imply that  $x > 0$  and  $y > 0$  by the way we designed the outside zone to be represented by a single state, not a “rim” of states.

As a derived issue in our analysis, we investigated which status-to-status transitions are never possible in the RSM model. Table 7.1 shows this in the form of a matrix, where, for

each row/column combination, an “X” sign means the transition is possible, and a dash means it is not.

To → From ↓	out	taxi	takeoff	rollout	climbout	landing	flythru
out	X	X	-	-	X	X	X
taxi	X	X	X	X	X	-	X
takeoff	X	X	X	X	X	-	X
rollout	X	X	X	X	-	-	-
climbout	X	-	-	-	X	X	X
landing	X	X	X	X	X	X	X
flythru	X	X	X	X	X	X	X

**Table 7.1:** The state-to-state transition matrix for RSM.

### 7.2.3 State-space measurements

The model that includes the first two phases of the protocol can be used as a building block to further conducting the analysis. The third phase, setting the alarm, can be added into the model as actual transitions, or alternatively, the value of the alarm variable can be implied by the combination of status and positioning after the second phase.

Up to this point, the constructed model exhibits strong event locality. This also allows for a flexible choice of partitioning of the model. A natural choice is to group variables referring to the same target together. However, assigning *all* variables to the same partition leads to extremely large local state spaces:

$$2 \cdot 3 \cdot 8(\max\_x + 1)(\max\_y + 1)(\max\_z + 1)(2\max\_speed + 1)^3$$

possible local states, which is unacceptable. A better choice is to further split the subnets into even smaller ones.

This decision is strongly influenced by the need to satisfy the Kronecker consistency requirements for each event. This proved to be the most challenging part of the modeling phase. As far as the events in the first two phases are concerned, a partitioning of places into five subnets per target yielded very good results. We arranged the variables in  $num\_targets + 1$  clusters, as follows:

$$\left\{ \begin{array}{l} \text{subnet } 5 * i + 1 : \quad ph\_radar[i], ph\_status[i], ph\_status[i] \\ \text{subnet } 5 * i + 2 : \quad st\_out[i], st\_taxi[i], \dots, st\_flythru[i], alarm[i] \\ \text{subnet } 5 * i + 3 : \quad x[i], vx[i] \\ \text{subnet } 5 * i + 4 : \quad y[i], vy[i] \\ \text{subnet } 5 * i + 5 : \quad z[i], vz[i] \end{array} \right.$$

We took measurements on the generated state spaces for different input parameters of the model: number of targets, decomposition grid size, thresholds of speed. The state space size, runtime, and memory consumption are listed in Tables 7.2–7.4. The results show that the reachability set can be constructed for multiple targets, a fairly large size of the grid, and multiple thresholds of speed in a matter of minutes, using under 100 MB of memory.

grid size	speeds	1 target	2 targets	3 targets	4 targets
$3 \times 5 \times 3$	2	$1.0 \times 10^{13}$	$3.4 \times 10^{19}$	$1.1 \times 10^{26}$	$3.5 \times 10^{32}$
$5 \times 10 \times 5$	2	$4.1 \times 10^{14}$	$8.4 \times 10^{21}$	$1.7 \times 10^{29}$	$3.5 \times 10^{36}$
$10 \times 10 \times 10$	2	$7.6 \times 10^{15}$	$6.6 \times 10^{23}$	$5.8 \times 10^{31}$	$5.0 \times 10^{39}$
$3 \times 5 \times 3$	5	$2.7 \times 10^{14}$	$4.4 \times 10^{21}$	$7.2 \times 10^{28}$	$1.2 \times 10^{36}$
$5 \times 10 \times 5$	5	$8.3 \times 10^{15}$	$7.6 \times 10^{23}$	$6.9 \times 10^{31}$	$6.3 \times 10^{39}$
$10 \times 10 \times 10$	5	$1.4 \times 10^{17}$	$5.0 \times 10^{25}$	$1.8 \times 10^{34}$	$6.7 \times 10^{42}$

**Table 7.2:** State space measurements for RSM: number of states.

#### 7.2.4 Setting the alarm

The third, and most important, phase of the RSM algorithm is setting the alarm flag for every target. In pseudo-code, this corresponds to a single variable assignment statement:

grid size	speeds	1 target	2 targets	3 targets	4 targets
$3 \times 5 \times 3$	2	2.01	2.93	3.93	4.91
$5 \times 10 \times 5$	2	5.52	8.27	11.19	13.91
$10 \times 10 \times 10$	2	13.62	20.58	27.50	34.42
$3 \times 5 \times 3$	5	4.41	6.51	8.77	10.98
$5 \times 10 \times 5$	5	12.91	19.07	25.42	32.05
$10 \times 10 \times 10$	5	28.45	42.84	57.25	71.75

**Table 7.3:** State-space construction time for RSM (in seconds).

grid size	speeds	1 target	2 targets	3 targets	4 targets
$3 \times 5 \times 3$	2	2.00	3.00	4.00	4.99
$5 \times 10 \times 5$	2	7.21	10.81	14.41	18.01
$10 \times 10 \times 10$	2	20.86	31.29	41.72	52.15
$3 \times 5 \times 3$	5	4.22	6.33	8.44	10.55
$5 \times 10 \times 5$	5	15.48	23.21	30.95	38.69
$10 \times 10 \times 10$	5	39.73	59.59	79.45	99.31

**Table 7.4:** State-space generation memory consumption for RSM (in Megabytes).

set the (boolean) value of *alarm* based on different combinations of the current values of the other variables, as listed in operational state matrix in Table 7.5 [73].

Modeling this rather complex assign statement in a Petri net is difficult because of two factors. First, predicates such as “distance is closing” or “in the takeoff path” that involve geometry and linear equations are difficult to express in a discretized model. Second, the Kronecker consistency requirements will force splitting of events into many finer-grain events. For example, the predicate “target *i* is in takeoff/landing path of ownship” can be expressed as:

$$(x_i = x_0) \bigwedge (((vy_0 > 0) \wedge (y_i > y_0)) \vee ((vy_0 < 0) \wedge (y_i < y_0)))$$

Since variables  $y_i$  and  $y_0$  are not in the same partition, each term involving the two must



Target → Ownship ↓	taxi	takeoff	climbout	land	rollout	fly-thru
taxi	-	1 ∧ 6	1 ∧ 6	1 ∧ 6	1 ∧ 3 ∧ 6	-
pretakeoff	2 ∧ 6	yes	1 ∨ 4	yes	yes	2 ∧ 3
takeoff	1 ∧ 6	4 ∨ 5	4 ∨ 5	4 ∨ 5	1 ∨ 4	2 ∧ 3
climbout	1 ∧ 6	4 ∨ 5	4 ∨ 5	4 ∨ 5	4 ∨ 5	2 ∧ 3
landing	1 ∧ 6	4 ∨ 5	4 ∨ 5	4 ∨ 5	1 ∨ 4	2 ∧ 3
rollout	1 ∧ 3 ∧ 6	1 ∨ 4	1 ∨ 4	1 ∨ 4	4 ∨ 5	2 ∧ 3
fly-thru	-	2 ∧ 3	2 ∧ 3	2 ∧ 3	2 ∧ 3	-
(1) Distance closing (2) In takeoff/landing path (3) Distance less than minimum separation (4) Takeoff/landing in same direction and distance less than minimum separation (5) Takeoff/landing in opposite direction and closing (6) Taxi/stationary on/near runway						

Table 7.5: RSM protocol: alarm setting criteria.

be split to satisfy Kronecker consistency:

$\forall C_y \in [1..max\_y]$ :

$$(x_i = x_0) \wedge ((vy_0 > 0 \wedge y_i = C_y \wedge C_y > y_0) \vee (vy_0 < 0 \wedge y_i = C_y \wedge C_y < y_0))$$

The same procedure must be applied to  $x_0$  and  $x_i$ , by further splitting terms:

$\forall C_x \in [1..max\_x]$ :

$\forall C_y \in [1..max\_y]$ :

$$(x_i = C_x) \wedge (x_0 = C_x) \wedge (vy_0 > 0) \wedge (y_i = C_y) \wedge (C_y > y_0) \quad \vee$$

$$(x_i = C_x) \wedge (x_0 = C_x) \wedge (vy_0 < 0) \wedge (y_i = C_y) \wedge (C_y < y_0)$$

This generates  $2 \cdot max\_x \cdot max\_y$  events from a single original event.

In general, the number of events needed to model the third phase necessitated over 2000 lines of additional SMART code, compared to just 500 lines needed to model the first two phases together.

An excerpt from the actual model that illustrates this procedure is listed below:

```

/* +++++ Alarm for ownship rollout & target taxi +++++ */
for (int jx in {2..max_x-1}) {
  for (int jy in {2..max_y-1}) {
    trans alarm_roll_taxi1[i][jx][jy],
      alarm_roll_taxi2[i][jx][jy];
    firing(alarm_roll_taxi1[i][jx][jy]:expo(1),
      alarm_roll_taxi2[i][jx][jy]:expo(1));
    arcs(
      ph_alarm[i]:alarm_roll_taxi1[i][jx][jy],
      ph_alarm[i]:alarm_roll_taxi2[i][jx][jy],
      alarm_roll_taxi1[i][jx][jy]:ph_radar[i],
      alarm_roll_taxi2[i][jx][jy]:ph_radar[i],
      alarm_roll_taxi1[i][jx][jy]:fl_alarm[i],
      alarm_roll_taxi2[i][jx][jy]:fl_alarm[i]);
    guard(
      alarm_roll_taxi1[i][jx][jy]:tk(st_taxi[i])>0 & tk(st_rollout[0])>0 &
        tk(x[0])==jx & tk(y[0])==jy & tk(vy[0])>max_speed &
        tk(x[i])>=jx-1 & tk(x[i])<=jx+1 &
        tk(y[i])>=jy & tk(y[i])<=jy+1,
      alarm_roll_taxi2[i][jx][jy]:tk(st_taxi[i])>0 & tk(st_rollout[0])>0 &
        tk(x[0])==jx & tk(y[0])==jy & tk(vy[0])<max_speed &
        tk(x[i])>=jx-1 & tk(x[i])<=jx+1 &
        tk(y[i])>=jy-1 & tk(y[i])<=jy);
  }
}

```

```

/* +++++ No alarm for ownship rollout & target taxi +++++ */
for (int jx in {2..max_x-1}) {
    trans noalarm_roll_taxix[i][jx];

    firing(noalarm_roll_taxix[i][jx]:expo(1));

    arcs(ph_alarm[i]:noalarm_roll_taxix[i][jx],
        noalarm_roll_taxix[i][jx]:ph_radar[i]);

    guard(noalarm_roll_taxix[i][jx]:
        tk(st_taxi[i])>0 & tk(st_rollout[0])>0 &
        tk(x[0])==jx & (tk(x[i])<jx-1 | tk(x[i])>jx+1))
    }

for (int jy in {2..max_y-1}) {
    trans noalarm_roll_taxi1[i][jy],
        noalarm_roll_taxi2[i][jy];

    firing(noalarm_roll_taxi1[i][jy]:expo(1),
        noalarm_roll_taxi2[i][jy]:expo(1));

    arcs(ph_alarm[i]:noalarm_roll_taxi1[i][jy],
        ph_alarm[i]:noalarm_roll_taxi2[i][jy],
        noalarm_roll_taxi1[i][jy]:ph_radar[i],
        noalarm_roll_taxi2[i][jy]:ph_radar[i]);

    guard(
        noalarm_roll_taxi1[i][jy]:tk(st_taxi[i])>0 & tk(st_rollout[0])>0 &
        tk(y[0])==jy & tk(vy[0])>max_speed & (tk(y[i])<jy | tk(y[i])>jy+1),
        noalarm_roll_taxi2[i][jy]:tk(st_taxi[i])>0 & tk(st_rollout[0])>0 &
        tk(y[0])==jy & tk(vy[0])<max_speed & (tk(y[i])<jy-1 | tk(y[i])>jy)
    );
}

```

At the end of this process, the most significant change in the model was a dramatic loss of event locality, leading to a slowdown in generation time and, most importantly, a much higher memory consumption. The peak MDD size increased to over 1000 times larger than the final, causing our model checker to run out of memory for large parameters, including multiple targets. However, we were still able to build the state space for one target and a medium size of the grid, within 1 GB of memory and less than 5 minutes. This was enough to expose several potential problems with the decision procedure of the protocol.

### 7.3 Model Checking RSM

The goal of our analysis is to exhaustively check whether the operational matrix in Table 7.5 is able to detect all incursion scenarios. A situation where two aircraft get too close to each other, within the minimum separation distance (MS) of 900 feet, without the alarm variable being set is called a *missed alarm* scenario. A dual analysis could be conducted on identifying *false alarm* scenarios, which describe the situations where an alarm is raised without the presence of an imminent danger.

In analyzing the scenarios involving a single target, we can use the subscripts  $o$  and  $t$  for the state variables referring to ownship and target, respectively. The following predicates are used to describe properties of interest:

$$\begin{aligned}
 done &\equiv_{\text{def}} phase_o = done\_alarm \wedge phase_t = done\_alarm \\
 sep &\equiv_{\text{def}} distance(o, t) > MS \\
 alarm &\equiv_{\text{def}} alarm_t = \text{true} \\
 track &\equiv_{\text{def}} status_o \notin \{taxi, flythru\} \vee status_t \notin \{taxi, flythru\}
 \end{aligned}$$

The semantics of the (non-trivial) predicates above are:

*done*: all updates are complete

*track*: at least one aircraft is taking off or landing

### 7.3.1 A “memory-less” property

We start our investigation with the most simple interpretation of a missed alarm.

- “Is there a tracked state where minimum separation is lost and the alarm is off?”
- CTL:  $\mathbf{EF}(done \wedge track \wedge \neg sep \wedge \neg alarm)$
- Answer: YES.

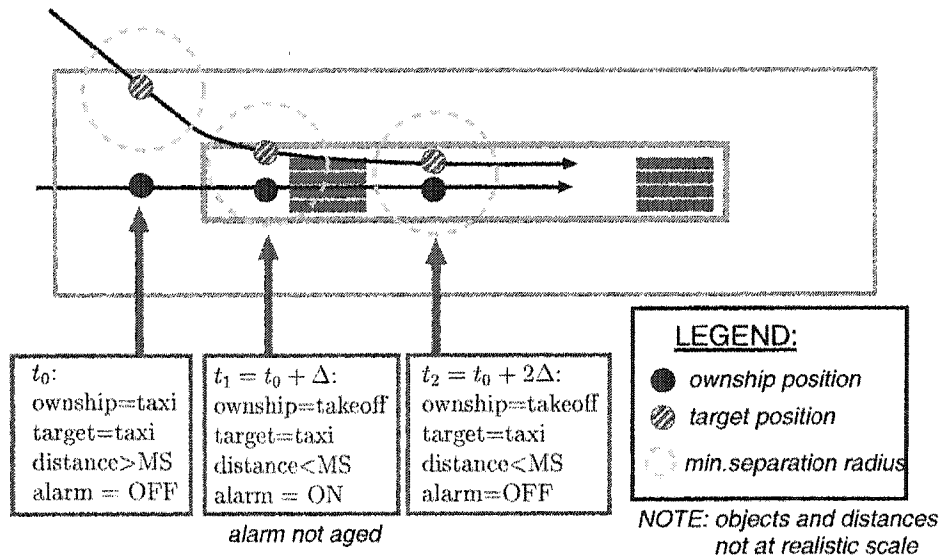


Figure 7.2: RSM: missed alarm scenario 1, at ground level.

A scenario that leads to a state satisfying this query is realized when the criterion “distance is closing” is not satisfied in the current state. This is the case of the third snapshot in Figure 7.2. However, this situation might not be the result of an unwanted

behavior, since the alarm might have been set in a previous state, more precisely in the first state where the minimum separation was lost. The value of the alarm variable being set also depends on whether the alarm is aged for a few more cycles or not.

Nevertheless, the situation is still of potential concern, even with aging of the alarm, since the target can maintain a constant distance (at less than minimum separation) for longer than the duration of the aging, resulting in a “bad state” in the round after the alarm expires.

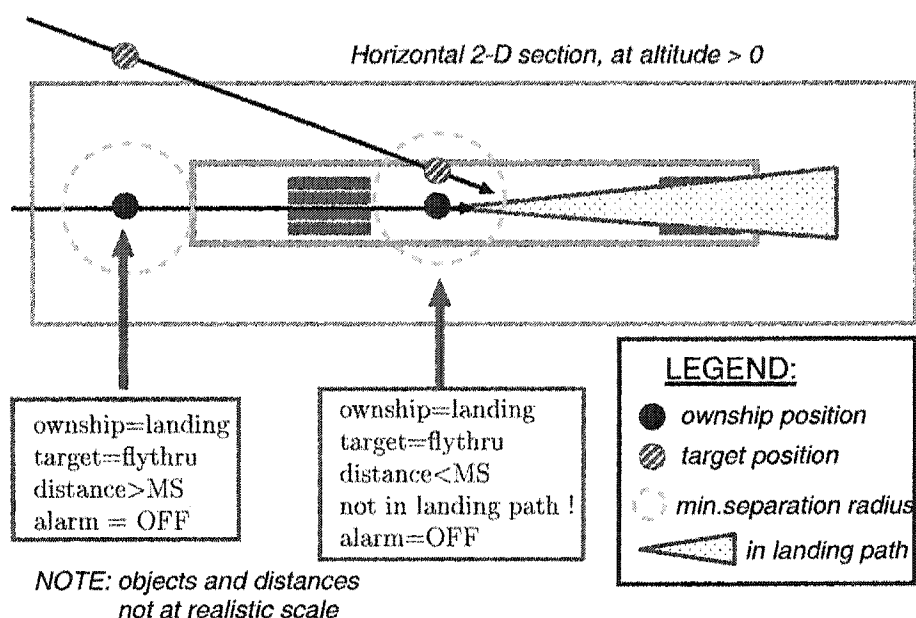
Note that the memory-less nature of the query influences the result. We looked at the property in a particular snapshot in time, without considering the sequence of events leading to the current state. To get a better understanding of the facts, we next investigate the states of the system immediately after the minimum separation distance between two aircraft is lost.

### 7.3.2 Analysis of the transition that causes loss of separation

- *“Is there a state where minimum separation is lost by transitioning to the current state and the alarm is off ?”*
- CTL:  $\mathbf{EF}(done \wedge track \wedge sep \wedge \mathbf{E}[(\neg done) \mathbf{U} (done \wedge track \wedge \neg sep \wedge \neg alarm)])$
- Answer: YES.

**Note 7.3.1** *The nested  $\mathbf{EU}$  operator in the query (instead of an  $\mathbf{EX}$ ) is due to the fact that several transitions are required to finish the update of coordinates (3), status (1) and to set the alarm again (1).*

A witness is shown in Figure 7.3, where ownship is in a landing or climbout state and the target is flying across the runway, faster than ownship, moving within separation distance from the side at an angle. The condition for setting an alarm in this circumstance is “distance less than minimum separation **and** target in takeoff/landing path”. The second term is not satisfied, hence no alarm is raised. Note that aircraft can actually collide (trajectories intersect in Figure 7.3), while none of the participants is ever warned. The above scenario is the only one generating bad states for this query.



**Figure 7.3:** RSM: missed alarm scenario 2, airborne.

This situation can be corrected by changing the criterion for this combination of status values to “distance less than minimum separation and target in takeoff/landing path **or** distance is closing”. As a result of our findings, the designers of the protocol have changed the specification to accommodate this situation.

Note that we included the predicate *track* in both states (before and after the transition), as we are interested in scenarios involving only takeoff and/or landing trajectories. However, this additional constraint may mask some other undesired behaviors. Therefore, we next ask a more general question.

### 7.3.3 A stronger safety property

- “Is there a tracked state where minimum separation is lost which can be reached without *EVER* previously setting the alarm?”
- CTL:  $\mathbf{E}[(\neg \text{alarm}) \mathbf{U} (\text{done} \wedge \text{track} \wedge \neg \text{sep} \wedge \neg \text{alarm})]$ ;
- Answer: YES.

#### 7.3.3.1 Counterexamples for strong safety

**Example 1.** (see Figure 7.4) Actors enter the stage taxiing (not aligned to the runway) at close distance to each other. Once on the runway, one of them (say ownship) changes direction and aligns itself to the runway. Thereafter, it is categorized as takeoff (or climbout, if it becomes airborne). The other aircraft stays within minimum separation, but it does not close in: it can be either behind ownship or, more dangerously, in front of the aircraft that is taking off. There is no alarm raised in this state because the criterion “distance is closing” is, again, not satisfied. If the distance between aircraft at entry is very small, even if the alarm is set on by closing in later, there might not be enough time for an escape manoeuvre.

Figure 7.4 shows an abstract trace that contradicts the safety property. The trajectories are shown for a horizontal section in the monitored zone at ground level. The third snapshot



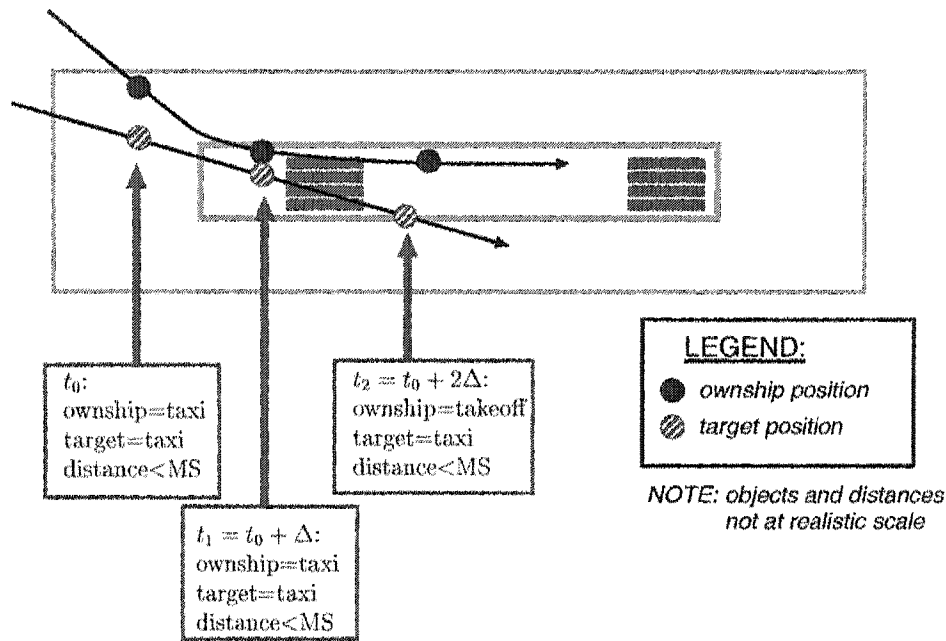


Figure 7.4: RSM: missed alarm scenario 3, at ground level.

illustrates the bad state: the two aircraft are within minimum separation distance, but no alarm has been issued either for the current state or any of the previous states in the scenario.

**Example 2.** A similar scenario exists for airborne states that are not tracked (status *flythru*): both airplanes enter as flying through (not aligned with the runway), either at close distance to each other, or they can lose minimum separation later, while both still being in flythru state. The target aircraft turns and aligns itself to the runway, switching to a landing/climbout status. Ownship stays within minimum separation, but not in front of the target (either flanking on the side, or zig-zagging behind, so that its status remains flythru). The condition for raising the flag in this state asks for distance less than minimum separation **and** ownship in takeoff/landing path. The condition is not met, hence the alarm

stays off.

**Example 3.** There are additional scenarios that do not satisfy the safety property, where events are developing immediately after both planes enter a monitored zone. The bad behavior in these cases is caused by the fact that the previous position is unknown — coordinates  $(0,0,0)$  in our model — for both planes, hence distance cannot be closing in the next state. If the airplanes enter the zone simultaneously, at positions very close to each other (e.g., both are trying to land), the alarm flag will not be raised. However, this particular behavior can be attributed to the fact that the planes enter the monitored zone already in a bad state, and the protocol is not responsible for monitoring the targets outside the zone.

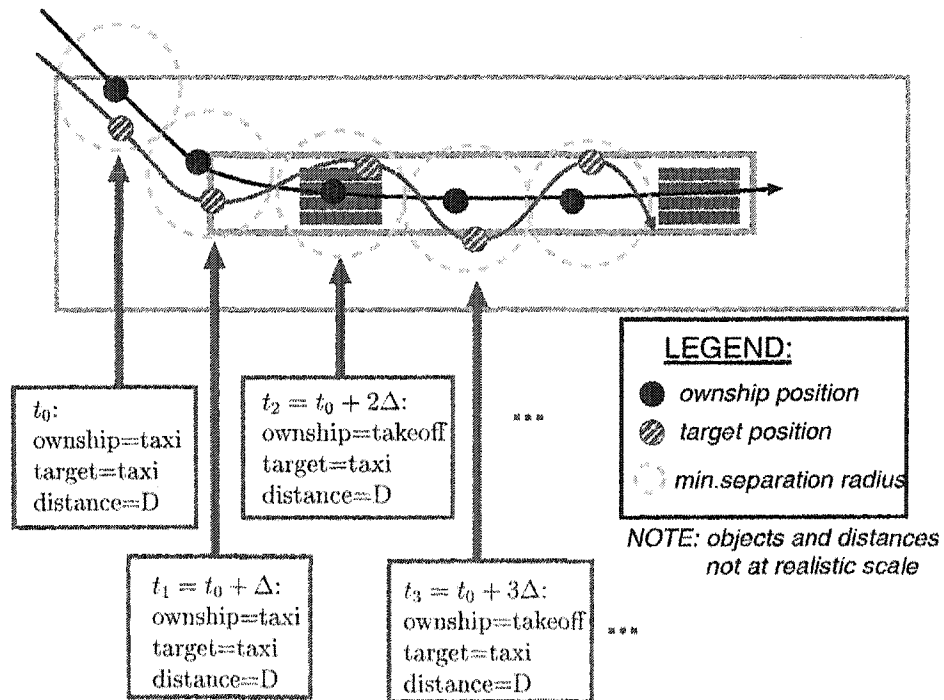


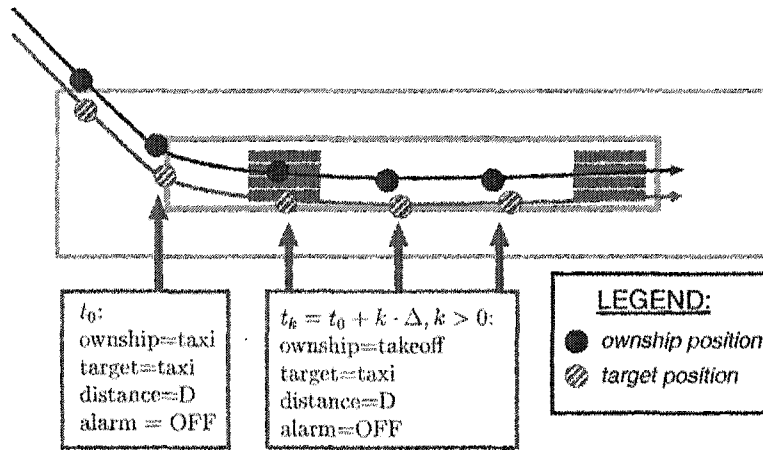
Figure 7.5: RSM: missed alarm scenario 4, at ground level.

To summarize the common characteristics of all the above the scenarios, we observe that the key factor is that both aircraft are in taxi/flythru mode at the moment when minimum separation is lost, hence the situation is not tracked (a potentially bad occurrence is masked by a protocol specification). Subsequently, the predicate “distance closing” is not validated, hence no alarm is issued, whereas the distance remains less than minimum separation (maintained constant or even slightly increasing).

To further extend our discussion, we look at the possible continuations of the scenario, after the bad state is reached. If, after a subsequent update, the distance is closing, then a warning will be issued and the “missed alarm” situation will cease to exist.

The only way to perpetuate the problem is shown in Figure 7.5 which is an extension of scenario 3. The target can stay within minimum separation radius for a longer period of time if it “zig-zags” in front of ownship, and at each radar update, it has the exact same distance to ownship. The target has to zig-zag to maintain the distance, since, if it follows a parallel path to ownship, the algorithm will consider it as taking off. The alarm criterion for the new combination of operational states is “taking off in the same direction and distance less than minimum separation”. Therefore, an alarm will be issued as soon as the target stops zig-zagging.

The case when the target is not an aircraft (vehicle, service truck, etc.) adds an extra degree of freedom for “malicious behavior” of the target (see scenario 5, in Figure 7.6). Ground vehicles are always in taxi mode according to the protocol, regardless of their speed, heading, and physical coordinates. Therefore, like in scenario 4, the target may follow ownship at close distance, but now it can also continue chasing, even after ownship is lined up for takeoff and accelerating. No flag will be raised for the same reasons as in



**Figure 7.6:** RSM: missed alarm scenario 5, aircraft vs vehicle.

scenario 4.

In an ulterior implementation of RSM, the designers have taken into account these facts and eliminated the special treatment given to ground vehicles. This addresses the situation in scenario 5.

The situation in scenario 4 is of lesser concern, since it certainly falls into the category of “freak occurrences”. At the same time, there is some benefit in exposing it: the designers are aware of this low-probability event, and also, by the fact that it is the only remaining unwanted behavior in the model, it serves as a validation of the decision procedure of phase 3 of the algorithm.

### 7.3.4 Conclusions

There were several lessons learned from our analysis. On the positive side, we concluded that our formal verification approach has an undeniable value. At the end of six months of work, most of which has been concerned with the modeling decisions, we have presented

the designers with a list of important findings which were not exposed during the testing activities (involving real aircraft) already underway at different airport locations. The merit of our technique is that, besides being considerably less expensive, it is **exhaustive** (on the constructed model).

We were able to analyze all possible scenarios in our model, with two outcomes. We have found scenarios of potential concern that happen with extremely low probability that are almost impossible to expose during either testing procedures, which usually afford no more than a dozen test flights a day, or simulation sessions, which have the same nature as testing, just that the analysis is conducted on a computer model of the protocol, but still manages to analyze only a limited number of scenarios per session. When compared to the actual state space sizes of the order of  $10^{13} - 10^{42}$  states, this shows the need for our exhaustive analysis. The second outcome of our experiments was that after identifying the problems and suggesting modifications to the protocol to eliminate them, we have presented the designers with our “certificate” of formal verification concerning missed alarms.

Secondly, we need to mention that, by engaging in this industrial-size project, we have identified several areas of possible improvements and extensions to our technique, which will be discussed in more detail in the next chapter.

With respect to the dual analysis of false alarms, this is still on the list of future research plans. From a practical point of view, pilots are equally concerned with both types of situations. Individual reports indicate that frequent false alarms can become a distraction or, in the best case, a nuisance factor in operating an airplane. It is also the case that a system with too many false alarms will tend to be switched off or ignored, thus rendering it useless. Therefore the occurrence rate of false alarms has to be reduced, even though these

are not as critical as missed alarms, which have to be *completely* eliminated. The designers of the protocol had to come up with a balanced solution that trades the simplicity of very “loose” requirements that raise too many alarms for the complexity of “stricter” conditions that decrease the number of false alarms, but make the analysis more difficult.

Another aspect that was not discussed here was fault tolerance. We assumed that all scenarios happen in the absence of faults in the communication devices, meaning that the radars and data-links provide accurate and timely updates to all participants. A natural extension of our analysis is to include faulty behaviors, either of benign nature (in the form of missed or late updates) or malicious/byzantine (such as inconsistent data between participants). However, this type of analysis requires the inclusion of notions of probabilistic nature [92], which are beyond the scope of the logical analysis that Model Checking is concerned with.

## Chapter 8

# Future Research

*“Basic research is what I’m doing  
when I don’t know what I’m doing.”*

Wernher Von Braun (1912-1977)

The work presented in this thesis can be extended and complemented in many ways. One direction is related to the idea of adding more *automation* to all the phases of our verification process where applicable.

First and foremost in this category comes the need for automated *partitioning* heuristics for the input models. As mentioned in Section 2.6.1.1, the variable ordering is critical to the overall performance of a BDD-based model checker. Our approach is subject to a more complex discussion, since we are dealing with multi-valued variables, which have to be ordered *and grouped* according to a partitioning procedure. Our current implementation supports only a predefined partitioning of the model, which has to be part of the input. This requires prior knowledge of the structure of the system, experience, and good intuition on the part of the user. For some models of theoretical nature, such as those considered in our experiments (except RSM), an optimal or nearly optimal partitioning is not difficult to find, but in general the problem is far from trivial. Nevertheless, we believe that the saturation algorithm is suited for *customized* partitioning heuristics, that work well with

our approach. For example, an off-line static analysis of the events in the system can offer hints for minimizing the computation effort of firing events symbolically and translate this analysis in terms of minimizing a goal function. Taking into account the event locality feature, a good candidate is the sum of ranges of (in number of levels that are spanned by) each event. Formally, if  $\mathcal{V}$  is the set of state variables in the system,  $\pi : \mathcal{V} \rightarrow \{1, \dots, K\}$  is the partitioning function that assigns each variable to a submodel (level), then the problem can be formulated as finding:

$$\min_{\pi} \sum_{e \in \mathcal{E}} |Top_{\pi}(e) - Bot_{\pi}(e)|$$

If we take into account the fact that operations on “taller” MDDs are more expensive, the optimization function can include a weight factor, depending on the location of the affected region in the MDD. Preliminary experiments in this direction showed very promising results.

Another direction considers an automated *model extraction* technique that eliminates from the description of the model those aspects that are irrelevant to the verification process. Closely related to this is the automated *abstraction/refinement* technique: a model can be further simplified with respect to validating or invalidating a specific property by using data abstraction. If later the property is verified in the abstract model, it is also valid in the original model, and the process stops. If on the other hand the property is false, the abstraction function has to be refined in order to discern between a real counterexample and an abstract but not original one. The cycle continues until a decision can be made. This property-driven technique is well suited for our structural approach, but has not been investigated yet.

Of significant matter is the issue of automated generation of *all* counterexamples or witnesses in Model Checking (when applicable). As seen in the RSM example, certain



properties may be invalidated by multiple groups of behaviors in the system. A model checker usually provides a single counterexample for a given property under study and this can expose only part of the undesired behaviors. The modeler has to mask the behaviors represented in the provided counterexample before further proceeding with the analysis.

From a practical perspective, automated syntax translators between the SMART input language and other well known tools (such as SPIN [81], SMV [104], LOTOS [79], etc.) will increase the exposure of our tool and also provide access to standard benchmarks of tests that are widely used in practice (such as ISCAS and CADP).

In a second class of improvements we include *extensions* and *generalizations* of our approach.

Of utmost importance is lifting the restrictions of *Kronecker consistency*. When this requirement is not satisfied, the model has to be modified so that it does, by refining the events or by merging submodels. Both actions can lead to certain inefficiencies: the number of events might become too large, which tends to make the exploration algorithm behave more like an explicit one, or the submodels might become too complex, leading to ample local state space sizes and, consequently, extremely large MDD nodes. The Kronecker restriction can be lifted in different ways. One solution is to use a combination of incidence matrices and MDDs to represent the next state functions, which is a middle ground solution between the traditional techniques and the structural approach. The independent encoding scheme for each event can be decided either during the modeling phase, by including options referring to each event explicitly, or it can be done automatically, at run-time, as Kronecker consistency is easy to determine. The resulting approach is gracefully degrading, as it is able to trade the excellent performance of Kronecker compliant techniques with the ease of

use of the general approach.

One of the observations in Chapter 5 was that the CTL operator **EG** is not a good candidate for a saturation-based approach. Despite this fact, we are still interested in ways to improve the computation of this operator together with the construction of corresponding witnesses. The latter issue is closely related to the problem of finding the strongly connected components in a directed graph. Efficient symbolic algorithms for this problem are useful in other areas, including Markov chain analysis. We think that a similar approach to the *partial* saturation introduced for the **EU** operator can be applied with good results to constructing strongly connected components, and also to computing CTL operators under fairness constraints [64].

There are some typical examples for which BDD techniques do not perform well, especially digital circuits such as multipliers, barrel shifters, and bit alternators. For these cases, a substitute for the BDD technology are the SAT solvers [1, 9, 125]. SAT solving consists in adapting the standard algorithms for reachability analysis to work with the classical SAT resolution algorithms, by using satisfiability checkers. The resulting approach is called Bounded Model Checking [9]. Even though SAT solvers are not fully automatic and are not known to outperform BDDs, with the exception of the few typical cases listed above, a direct comparison with our approach is of due course.

On a different note is the intent to extend the area of applicability of our approach. Since our data structures and algorithms could be of a more general purpose, we consider that an MDD/saturation *library* for public distribution would be valuable for the entire Formal Verification community. Along the same lines of enhancing exposure, a new modeling *formalism* in SMART (e.g., similar to the Promela language of SPIN [81]), that is more

adequate for the study of software, is of immediate interest.

Last, but not least, as the major bottleneck for symbolic Model Checking remains the peak memory consumption, *parallel and distributed* versions of our saturation algorithms could offer a valid alternative to the excessive memory consumption of sequential algorithms. On an anecdotal note, we have to mention that this was in fact the original motivation behind our entire work. The original design of the parallel implementation of the structural approach to state space construction has sparked the ideas in the work presented here. We still believe that parallel and distributed implementations are valuable, despite their earned notoriety as “hard-to-parallelize” problems [138].

## Appendix A

# Overview of SMART

SMART (Stochastic Model checking Analyzer for Reliability and Timing) [25] is a software package that integrates multiple logic and stochastic modeling formalisms into a single environment. At the heart of SMART lies the ability to define parametric models for which a variety of *measures* can be computed. For the analysis of logical behavior, both explicit and symbolic state-space generation techniques, as well as CTL Model Checking algorithms, are available. For the study of stochastic and timing behavior, both sparse-storage and Kronecker numerical solution approaches are available.

The SMART project is the result of a collective effort of several developers over the years, starting in 1994. Currently, the C++ source has over 100,000 lines of code. The author of this thesis is responsible for writing the various symbolic state space construction algorithms presented here and the entire Model Checking module, totaling over 12,000 lines of code.

### The SMART Language

SMART uses a strongly-typed, computation-on-demand, declarative language with the following predefined *types*: `bool` (true or false), `int` (integer values, machine dependent range), `bigint` (arbitrary size integer values), `real` (floating-point values, machine-

dependent range and precision), and `string` (character-array values). Sample declarations are listed below:

```
bool c := 3 - 2 > 0;

int i := 12;

bigint i := 1000*2000*3000*4000*5000;

real x := sqrt(2.3);

string s := "Monday";
```

Composite types can be defined using the concepts of *sets* (collection of homogeneous objects:  $\{1, 2, 4, 8, 16\}$ ), *arrays* (multidimensional collections of homogeneous objects indexed by set elements: `a[3][0.2]`), and *aggregates* (analogous to the Pascal “record”: `p:3`).

A type can be further modified by the following *natures*, which describe stochastic characteristics: `const`: (the default) non-stochastic quantities, `ph`: for random variables with discrete or continuous phase-type distribution, `rand`: for random variable with arbitrary distribution. Finally, predefined formalism types can be used to define stochastic processes indexed by time: continuous and discrete time Markov chains (`ctmc`, `dtmc`) and stochastic Petri nets (`spn`).

The basic statements in SMART can be grouped in: declarations, definitions, compound and option statements.

### Function declarations

Syntactically, objects defined in SMART are functions, possibly recursive, and can be overloaded:

```

real pi := 3.14;    /* a parameter-less function */

bool close(real a, real b) := abs(a-b) < 0.00001;

/* a two-parameter function */

int pow(int base, int exp) := cond(exp==1,base,base*pow(base,exp-1));

real pow(real base, int exp) := cond(exp==1,base,base*pow(base,exp-1));

pow(5,3);           /* returns 125, integer */

pow(5.0,3);         /* returns 125.0, real */

```

### Arrays

Arrays are declared using a `for` statement. Their dimensionality is determined by the enclosing iterators. Since the indices along each dimension belong to a finite set, it is legal to define arrays with `real` indices:

```

for (int i in {1..5}, real r in {1..i..0.1}) {

    real res[i][r] := MyModel(i,r).out1;

}

```

fills array `res` with the value of measure `out1` for `MyModel`, when the first input parameter ranges from one to five and the second one ranges from one to the value of the first parameter, with a step of  $1/10$ .

### Fixed-point iterations

The approximate solution of a model is often based on a heuristic decomposition, where (sub)models are solved in a fixed-point iteration. This can be specified with the `converge` statement:

```

converge {
    real x guess 1.0;

    real y := fy(x, y);

    real x := fx(x, y);

}

```

The iterations stop when two subsequent  $x$  and  $y$  values differ by less than  $\epsilon$  in either relative or absolute terms. The values for  $x$  and  $y$  are updated either immediately or at the end of each iteration. Both  $\epsilon$  and the updating criterion are fine-tuned using *option* statements.

The `converge` and `for` statements can be arbitrarily nested within each other.

### Random variables

SMART can manipulate discrete and continuous phase-type distributions. Combining ph types produces another ph type if phase-type distributions are closed under that operation:

```

ph int  X := geom(0.7);

ph int  Y := equilikely(1,5);

ph int  A := min(3*X, Y);

ph int  B := 3*X+Y;

ph int  C := choose(0.4:3*X, 0.6:Y);

ph real x := expo(3.2);

ph real y := erlang(4,5);

ph real a := min(3*x, y);

```

### Model formalisms

A model is declared just like a function except that, instead of a return value, it specifies a block containing *declarations*, *specifications*, and *measures*. Components of the model are

declared using formalism-specific types (e.g., the places of a Petri net). The model structure is specified by using formalism-specific functions (e.g., the arcs of a Petri net). Measures are user-defined functions that specify some quantity of interest (e.g., the expected number of tokens in a given place in steady-state), and are the only model components that can be accessed outside of the model definition block.

For example, the model shown on the right is defined as:

```
spn mynet(int n) := {

  place p5, p4, p3, p2, p1;

  trans a, b, c, d, e;

  arcs(p5:a,a:p4,a:p2,p4:c,c:p3,p3:b,b:p4,
       p2:d,d:p1,p1:e,p3:e,e:p5);

  firing(a:expo(1.1),b:expo(1.2),c:expo(1.3),
        d:expo(1.4),e:expo(1.5));

  init(p5:n);

  bigint count := num_states(false);

  real speed := avg_ss(rate(a));

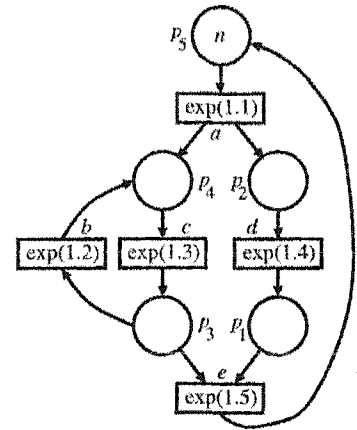
};

for (int n in {1..4}) {

  print("When n=",n," the number of states is ",mynet(n).count:2);

  print(" and the throughput is ",mynet(n).speed,"\n");

}
```



where `place` and `trans` are formalism-specific types; `arcs`, `firing`, and `init` are formalism-specific functions; `count` and `speed` are two measures.



The `for` loop outside the model produces the output

When `n=1` the number of states is 5 and the throughput is 0.284926

When `n=2` the number of states is 14 and the throughput is 0.456624

When `n=3` the number of states is 30 and the throughput is 0.553202

When `n=4` the number of states is 55 and the throughput is 0.612783

where the measure calls cause SMART to perform the appropriate analysis.

Arrays and `for` loops are allowed within a model in the usual manner.

### State space generation and storage

SMART implements a wide variety of algorithms for constructing state spaces, governed by the choice of the `# StateStorage` option. Explicit techniques include AVL, splay trees, and hash tables (option values `AVL`, `SPLAY`, `HASHING`). Symbolic algorithms require that a partition of the model be specified, and are classified as either of pregeneration type (option values `MDD_LOCAL_PREGEN`, `MDD_FORWARDING_PREGEN`, `MDD_SATURATION_PREGEN`), or on-the-fly (`MDD_SATURATION`, the default setting when a partitioning is defined). The size of a state space can be obtained in number of states, by using the function `num_states`, and number of arcs, by calling the `num_arcs` function on a model.

### Model Checking

CTL model-checking queries are available in SMART via a set of model-dependent measures. The answers to CTL queries are stored as sets of states (the states that satisfy the formula) using MDDs with shared nodes. A predefined type `stateset` is used for this. All the algorithms employ symbolic techniques, therefore a user-defined partitioning is required.

Table A.1 lists the functions available for Model Checking in SMART.  $\mathcal{P}$  and  $p$ , or  $\mathcal{Q}$  and

SMART function	Set meaning	Logic meaning
potential(bool b)	$\{i \in \mathcal{S} : b(i)\}$	<i>atom</i>
initialstate	$\{s\}$	<i>atom</i>
nostates	$\emptyset$	<i>false</i>
complement(ss p)	$\mathcal{S} \setminus \mathcal{P}$	$\neg p$
intersection(ss p, ss q)	$\mathcal{P} \cap \mathcal{Q}$	$p \wedge q$
union(ss p, ss q)	$\mathcal{P} \cup \mathcal{Q}$	$p \vee q$
difference(ss p, ss q)	$\mathcal{P} \setminus \mathcal{Q}$	$p \wedge \neg q$
includes(ss p, ss q)	$\mathcal{P} \supseteq \mathcal{Q}$	$q \Rightarrow p$
equal(ss p, ss q)	$\mathcal{P} = \mathcal{Q}$	$p \Leftrightarrow q$
neq(ss p, ss q)	$\mathcal{P} \neq \mathcal{Q}$	$\neg(p \Leftrightarrow q)$
EX(ss p)	$\{i : \exists j \in \mathcal{N}(i), j \in \mathcal{P}\}$	$EX p$
AX(ss p)	$\{i : \forall j \in \mathcal{N}(i), j \in \mathcal{P}\}$	$AX p$
EF(ss p)	$\{i : \exists j \in \mathcal{N}^*(i), j \in \mathcal{P}\}$	$EF p$
AF(ss p)	$\{i : \forall \pi \in \Pi_i, \exists n \geq 0, \pi_n \in \mathcal{P}\}$	$AF p$
EG(ss p)	$\{i : \exists \pi \in \Pi_i, \forall n \geq 0, \pi_n \in \mathcal{P}\}$	$EG p$
AG(ss p)	$\{i : \forall j \in \mathcal{N}^*(i), j \in \mathcal{P}\}$	$AG p$
EU(ss p, ss q)	$\{i : \exists \pi \in \Pi_i, \exists m \geq 0, \pi_m \in \mathcal{Q} \wedge \forall n < m, \pi_n \in \mathcal{P}\}$	$E[p \cup q]$
AU(ss p, ss q)	$\{i : \forall \pi \in \Pi_i, \exists m \geq 0, \pi_m \in \mathcal{Q} \wedge \forall n < m, \pi_n \in \mathcal{P}\}$	$A[p \cup q]$
EXbar(ss p)	$\{j : \exists i \in \mathcal{P}, j \in \mathcal{N}(i)\}$	$EX p$
AXbar(ss p)	$\{j : \forall i \in \hat{\mathcal{S}}, j \in \mathcal{N}(i) \Rightarrow i \in \mathcal{P}\}$	$AX p$
EFbar(ss p)	$\{j : \exists i \in \mathcal{P}, j \in \mathcal{N}^*(i)\}$	$EF p$
AFbar(ss p)	$\{j : \forall \pi \in \bar{\Pi}_j, \exists n \geq 0, \pi_n \in \mathcal{P}\}$	$AF p$
EGbar(ss p)	$\{j : \exists \pi \in \bar{\Pi}_j, \forall n \geq 0, \pi_n \in \mathcal{P}\}$	$EG p$
AGbar(ss p)	$\{j : \forall i \in \hat{\mathcal{S}}, j \in \mathcal{N}^*(i) \Rightarrow i \in \mathcal{P}\}$	$AG p$
EUbar(ss p, ss q)	$\{j : \exists \pi \in \bar{\Pi}_j, \exists m \geq 0, \pi_m \in \mathcal{Q} \wedge \forall n < m, \pi_n \in \mathcal{P}\}$	$E[p \bar{\cup} q]$
AUbar(ss p, ss q)	$\{j : \forall \pi \in \bar{\Pi}_j, \exists m \geq 0, \pi_m \in \mathcal{Q} \wedge \forall n < m, \pi_n \in \mathcal{P}\}$	$A[p \bar{\cup} q]$
prev(ss p)	alias for EX(p), one step backward	
next(ss p)	alias for EXbar(p), one step forward	
backward(ss p)	alias for EF(p), backward reachability	
forward(ss p)	alias for EFbar(p), forward reachability	
reachable	alias for forward(initialstate), the state space $\mathcal{S}$	
addprev(ss p)	alias for union(EX(p), p), zero or one step backward	
addnext(ss p)	alias for union(EXbar(p), p), zero or one step forward	
card(ss p)	return $ \mathcal{P} $ , the number of elements in set $\mathcal{P}$	
printset(ss p)	print the elements in $\mathcal{P}$ , up to MaxPrintedItems	

Table A.1: Model checking functions in SMART.

$q$ , are the set and the formula corresponding to  $p$ , or  $q$ , respectively.  $\Pi_i$  and  $\bar{\Pi}_i$  are the sets of forward and reverse time paths starting at state  $i$ , respectively, and  $\pi_n$  is the  $n^{\text{th}}$  state on path  $\pi$ . For conciseness, the abbreviation **ss** is used for **stateset**. Note that we extended CTL with all the dual operators in the past.

The following example shows one possible way to check for absorbing (deadlocked) states in a model, then verify some stability and safety properties:

```

stateset U          := reachable;           /* reachable states */
stateset NotAbsorb := prev(potential(true)); /* states that have a successor */
stateset Absorb     := difference(U,NotAbsorb); /* reachable absorbing states */
bool    deadlock   := neq(Absorb,nostates);  /* deadlock detection */
bool    prnabs     := printset(Absorb);      /* list deadlocked states */
stateset Good      := potential(expr1);
stateset Stable    := EG(Good);              /* there is an infinite good run */
stateset Safe      := AU(Stable,Good);

```

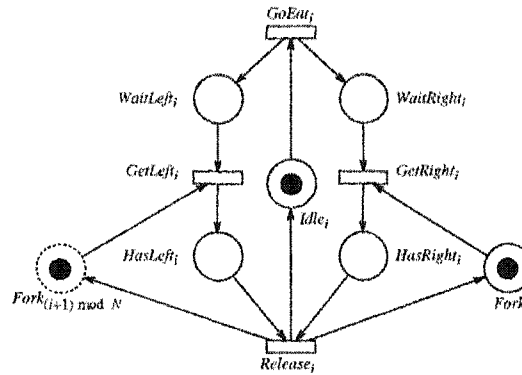
For further reference, a complete SMART User Manual [29] is available on-line, at:

[www.cs.wm.edu/~ciardo/SMART](http://www.cs.wm.edu/~ciardo/SMART).

## Appendix B

# Benchmark of Examples

### B.1 Dining philosophers



**Figure B.1:** The dining philosophers,  $i^{\text{th}}$  subnet.

This model implements the classical dining philosophers protocol, a paradigm for mutual exclusion solutions. The philosophers (representing processes) are seated around a table, with forks (representing the concept of shared resources) between adjacent philosophers. Each philosopher is initially idle, but eventually, and independently become hungry. To be able to eat, they have to obtain their corresponding right-hand and left-hand fork, which are shared with their neighbours. When a fork is secured, it is not released until the philosopher

has finished eating.

The SMART model [111] is composed of  $N$  subnets equal to the one shown in Figure B.1. The net represents a philosopher and the philosopher's right fork. The philosopher's left fork, represented by the dotted place  $Fork_{(i+1) \bmod N}$ , is part of the subnet for the next philosopher and is depicted only to illustrate how the subnets interact. Note that the model implemented here has two deadlocked states, corresponding to the situation when all philosophers hold their right forks, or all hold their left forks, respectively.

The corresponding SMART code is:

```
spn phils(int N) := {
  for (int i in {1..N}) {
    place Idle[i], WaitL[i], WaitR[i], HasL[i], HasR[i], Fork[i];
    trans GoEat[i], GetL[i], GetR[i], Rel[i];
    firing(GoEat[i]:expo(1), GetL[i]:expo(1), GetR[i]:expo(1), Rel[i]:expo(1));
    partition(1+div(i,2):Idle[i]:WaitL[i]:WaitR[i]:HasL[i]:HasR[i]:Fork[i]);
    init(Idle[i]:1, Fork[i]:1);
  }
  for (int i in {1..N}) {
    arcs(Idle[i]:GoEat[i], GoEat[i]:WaitL[i], GoEat[i]:WaitR[i], WaitL[i]:GetL[i],
        Fork[1+mod(i,N)]:GetL[i], GetL[i]:HasL[i], WaitR[i]:GetR[i],
        Fork[i]:GetR[i], GetR[i]:HasR[i], HasL[i]:Rel[i], HasR[i]:Rel[i],
        Rel[i]:Idle[i], Rel[i]:Fork[i], Rel[i]:Fork[1+mod(i,N)]);
  }
};
```

## B.2 A flexible manufacturing system

The Petri net [32] in Fig. B.2 models a flexible manufacturing system (FMS). The system consists of three types of machines,  $M_1$ ,  $M_2$ , and  $M_3$ . There are three machines of type  $M_1$ , that process parts of type  $P_1$ , one machine of type  $M_2$ , processing parts of type  $P_2$ , and two machines of type  $M_3$ , that can assemble finished parts of type  $P_1$  and  $P_2$  together into a single part. When the machine  $M_2$  is idle, it can also process parts of type  $P_3$ . Completed parts of all types are shipped and raw parts of the same type enter the system to maintain a constant inventory (initially, there are  $N$  parts of each type).

The model is parameterized by the initial number  $N$  of tokens in  $P_1$ ,  $P_2$ , and  $P_3$ .

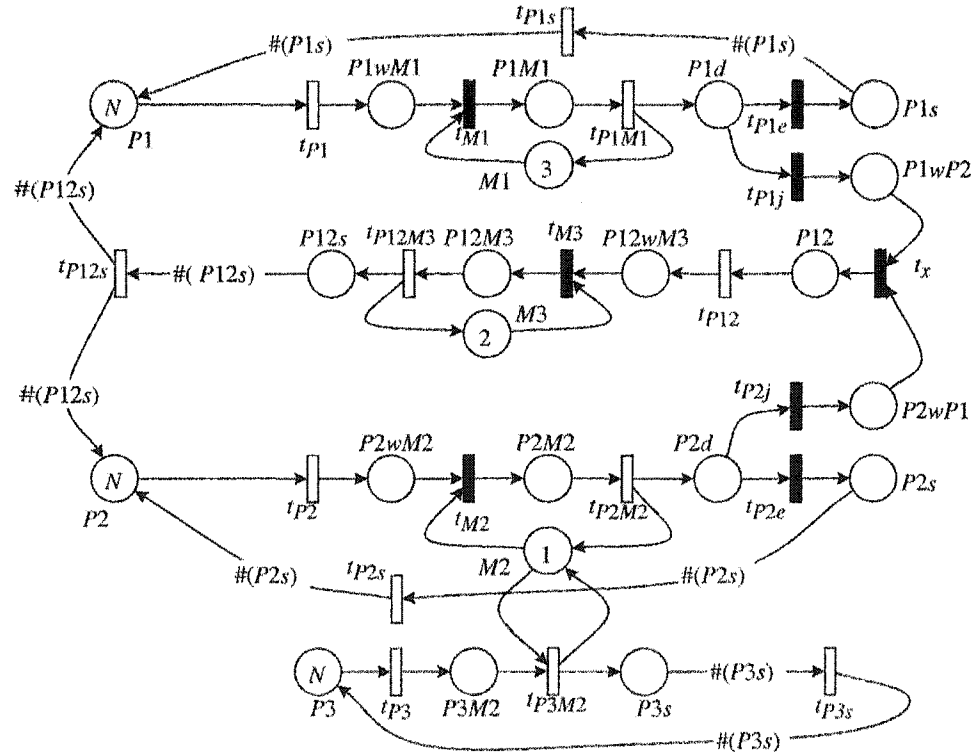


Figure B.2: A flexible manufacturing system.

```

spn fms(int n) := {

  place P1, P2, P3, P12, P2s, P3s, P1s, P12s,

    M1, M2, M3, P1wM1, P2wM2, P3M2, P12wM3,

    P1M1, P2M2, P12M3, P1d, P2d, P1wP2, P2wP1;

  trans tP1, tP2, tP3, tP12, tP1M1, tP2M2, tP3M2,

    tP12M3, tP1s, tP2s, tP3s, tP12s, tM1,

    tM2, tM3, tx, tP1e, tP1j, tP2e, tP2j;

  firing(tP1:expo(tk(P1)*max(1.0, div(3*n,2)/(tk(P1)+tk(P2)+tk(P3)+tk(P12))))),

    tP2:expo(tk(P2)*max(1.0, div(3*n,2)/(tk(P1)+tk(P2)+tk(P3)+tk(P12))))),

    tP3:expo(tk(P3)*max(1.0, div(3*n,2)/(tk(P1)+tk(P2)+tk(P3)+tk(P12))))),

    tP12:expo(tk(P12)*max(1.0, div(3*n,2)/(tk(P1)+tk(P2)+tk(P3)+tk(P12))))),

    tP1M1:expo(tk(P1M1)/4), tP2M2:expo(1/6), tP3M2:expo(1/2),

    tP12M3:expo(tk(P12M3)), tP1s:expo(1/60), tP2s:expo(1/60), tP3s:expo(1/60),

    tP12s:expo(1/60), tM1:0, tM2:0, tM3:0, tx:0, tP1e:0, tP1j:0, tP2e:0, tP2j:0);

  weight(tM1:1, tM2:1, tM3:1, tx:1, tP1e:0.8, tP1j:0.2, tP2e:0.6, tP2j:0.4);

  arcs(P1:tP1, tP1:P1wM1, P1wM1:tM1, M1:tM1, tM1:P1M1, P1M1:tP1M1, tP1M1:M1,

    tP1M1:P1d, P1d:tP1e, P1d:tP1j, tP1e:P1s, tP1j:P1wP2, P1s:tP1s:tk(P1s),

    tP1s:P1:tk(P1s), P1wP2:tx, tx:P12, P12:tP12, tP12:P12wM3, P12wM3:tM3,

    M3:tM3, tM3:P12M3, P12M3:tP12M3, tP12M3:M3, tP12M3:P12s, P12s:tP12s:tk(P12s),

    tP12s:P1:tk(P12s), tP12s:P2:tk(P12s), P2:tP2, tP2:P2wM2, P2wM2:tM2, M2:tM2,

    tM2:P2M2, P2M2:tP2M2, tP2M2:M2, tP2M2:P2d, P2d:tP2j, tP2j:P2wP1, P2wP1:tx,

    P2d:tP2e, tP2e:P2s, P2s:tP2s:tk(P2s), tP2s:P2:tk(P2s), P3:tP3, tP3:P3M2,

    P3M2:tP3M2, M2:tP3M2, tP3M2:P3s, tP3M2:M2, P3s:tP3s:tk(P3s), tP3s:P3:tk(P3s));

  init(P1:n, P2:n, P3:n, M1:3, M3:2, M2:1);

  partition(P1:P1wM1:P1M1:M1:P1d:P1s, P12s:P12M3:M3:P12wM3:P12:P1wP2:P2wP1,

    P2:P2wM2:P2M2:M2:P2d:P2s, P3:P3M2:P3s);

};

```

### B.3 Slotted ring

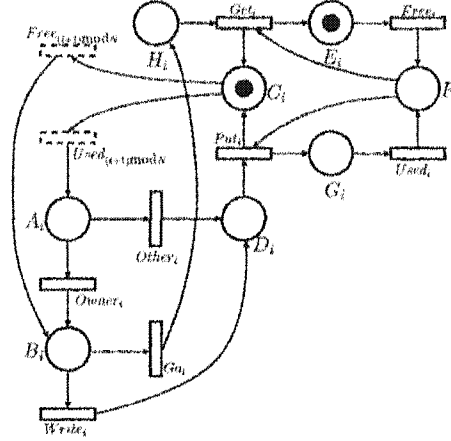


Figure B.3: Slotted ring model,  $i^{\text{th}}$  subnet.

Figure B.3 shows the Petri net for a single node of a slotted ring network protocol [111]. The overall model is composed of  $N$  such subnets connected by merging transitions (i.e.,  $Free_{(i+1) \bmod N}$  and  $Used_{(i+1) \bmod N}$  really belong to the “next” subnet).

The following SMART code shows a decomposition where each node of the ring is in a different level.

```

spn slot(int N) := {
  for (int i in {0..N-1}) {
    place pA[i], pB[i], pC[i], pD[i], pE[i], pF[i], pG[i], pH[i];
    trans other[i], owner[i], write[i], go[i], give[i], put[i], used[i], free[i];
    firing(other[i]:expo(1), owner[i]:expo(1), write[i]:expo(1), go[i]:expo(1),
           give[i]:expo(1), put[i]:expo(1), free[i]:expo(1), used[i]:expo(1));
    partition(pA[i]:pB[i]:pC[i]:pD[i]:pE[i]:pF[i]:pG[i]:pH[i]);
    init(pC[i]:1, pE[i]:1);
  }
}

```



```

for (int i in {0..N-1}) {
    arcs(used[mod(i+1,N)]:pA[i], pA[i]:other[i], pA[i]:owner[i],
        free[mod(i+1,N)]:pB[i], owner[i]:pB[i], pB[i]:go[i], pB[i]:write[i],
        write[i]:pD[i], other[i]:pD[i], pD[i]:put[i], go[i]:pH[i],
        pH[i]:give[i], give[i]:pC[i], put[i]:pC[i], pC[i]:free[mod(i+1,N)],
        pC[i]:used[mod(i+1,N)], free[i]:pF[i], used[i]:pF[i], pF[i]:give[i],
        pF[i]:put[i], give[i]:pE[i], pE[i]:free[i], put[i]:pG[i], pG[i]:used[i]);
    }
}

```

#### B.4 A Kanban system

This model [131] consists of four stations that process parts. A part enters a station only if a ticket is available. The part is then processed, and if the work is performed correctly, the part can move to the next station, otherwise it is sent back to be fixed. A part that passes station 1 is split into two parts, one for stations 2 and 3 each, and when the two parts pass their stations, they are joined and move to station 4.

The model shown in Figure B.4 is parameterized by the number  $N$  of tokens initially in  $p_1$ ,  $p_2$ ,  $p_3$ , and  $p_4$ . The following code shows a partition into four levels, one per kanban station.

```

spn kanban(int N) := {
    place pm1, pback1, pkan1, pout1, pm2, pback2, pkan2, pout2,
        pm3, pback3, pkan3, pout3, pm4, pback4, pkan4, pout4;
    trans tin1, tredo1, tok1, tback1, tin23, tredo2, tok2, tback2, tout2,
        tredo3, tok3, tback3, tredo4, tok4, tback4, tout4;
}

```

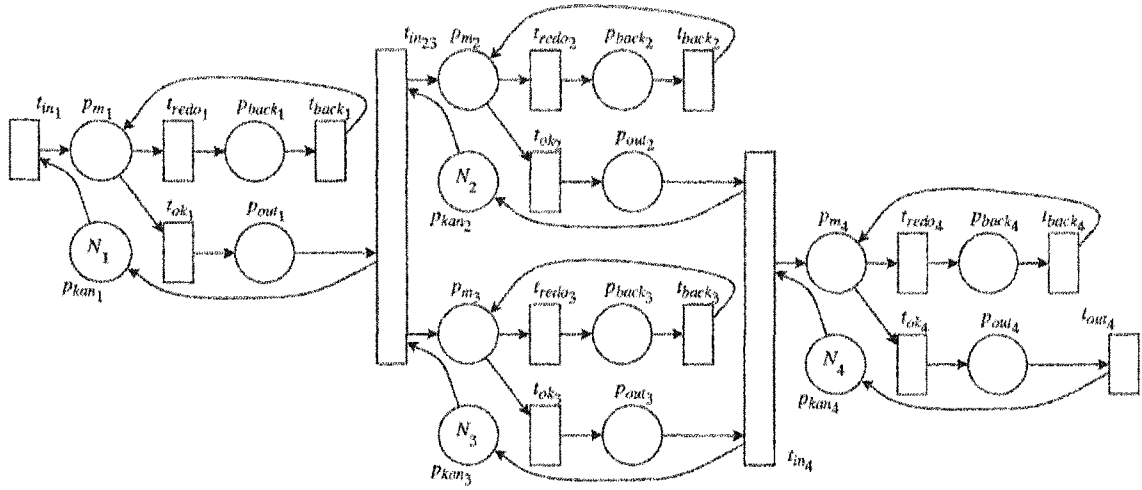


Figure B.4: A kanban system.

```

partition(pm1:pback1:pkani:pout1, pm2:pback2:pkani:pout2,
          pm3:pback3:pkani:pout3, pm4:pback4:pkani:pout4);

firing(tin1:expo(1.0), tredo1:expo(0.36), tok1:expo(0.84), tback1:expo(0.3),
       tin23:expo(0.4), tredo2:expo(0.42), tok2:expo(0.98), tback2:expo(0.3),
       tout2: expo(0.5), tredo3:expo(0.39), tok3:expo(0.91), tback3:expo(0.3),
       tredo4:expo(0.33), tok4:expo(0.77), tback4:expo(0.3), tout4:expo(0.9));

arcs(pkani:tin1, tin1:pm1, pm1:tredo1, pm1:tok1, tredo1:pback1,
      tok1:pout1, pback1:tback1, tback1:pm1, pout1:tin23, tin23:pkani,
      pkan2:tin23, tin23:pm2, pm2:tredo2, pm2:tok2, tredo2:pback2,
      tok2:pout2, pback2:tback2, tback2:pm2, pout2:tout2, tout2:pkani,
      pkan3:tin23, tin23:pm3, pm3:tredo3, pm3:tok3, tredo3:pback3,
      tok3:pout3, pback3:tback3, tback3:pm3, pout3:tout2, tout2:pkani,
      pkan4:tout2, tout2:pm4, pm4:tredo4, pm4:tok4, tredo4:pback4,
      tok4:pout4, pback4:tback4, tback4:pm4, pout4:tout4, tout4:pkani);

init(pkani:N, pkan2:N, pkan3:N, pkan4:N);

};

```

## B.5 Randomized leader election protocol

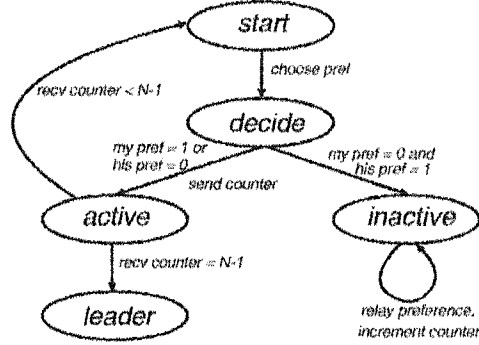
The randomized asynchronous leader election protocol in [84] solves the following problem: given a ring of  $N$  processors, the participants are required to designate a unique processor as leader by sending messages around the ring. The ring is unidirectional, meaning that the processes send messages to their unique successor (e.g. the one to the right), and receive messages from their unique predecessor. It is known that if the processors are indistinguishable (no unique identifiers are assigned), then there is no deterministic algorithm to solve the problem.

The randomized algorithm works in phases. At the beginning of every round, each process flips a coin to decide whether it will continue running for election or not. Initially all processes are valid candidates. After choosing a value (0 = don't run this round, 1 = run), this is communicated to the neighbour to the right. A process is eliminated from the race only if it chose not to run and its predecessor chose to run. After being eliminated from the race, a process never becomes eligible again (it enters the inactive state), and it is used only to relay messages between active nodes around the ring. They do not initiate any communication. Termination is detected by the active processes (at least one active node exists at all times) by sending a token around the ring to count the inactive nodes. The process that receives its own token with count  $N - 1$  is the elected leader.

In our model, each process has 5 state variables:

```
status[i]      : {start, wait, active, inactive, leader}; // init start
preference[i]  : {0, 1};                                     // init 0
counter[i]     : 0..N-1;                                     // init 0
sent[i]        : {none, pref, counter};                     // init none
```

```
recv[i]      : {none, pref, counter};           // init none
```



**Figure B.5:** Statechart for the leader election protocol.

The state-transition diagram for the *status* of a process is shown in Figure B.5.

The SMART code for this model is:

```

spn leader(int N) := {
  for (int i in {0..N-1}) {
    place st_choose[i], st_wait[i], st_active[i], st_inactive[i], st_leader[i],
      pref[i], c[i], rec_none[i], rec_choice[i], rec_counter[i],
      sent_none[i], sent_choice[i], sent_counter[i];
    trans pick_value0[i], pick_value1[i], send_pref[i],
      recv_stay_active0[i], recv_stay_active1[i], recv_become_inactive[i],
      send_new_counter[i], recv_counter[i], become_leader[i],
      i_recv_pref0[i], i_recv_pref1[i], i_send_pref[i], i_send_counter[i];
    partition(st_choose[i]:st_wait[i]:st_active[i]:st_inactive[i]:
      st_leader[i]:pref[i]:c[i],
      rec_none[i]:rec_choice[i]:rec_counter[i]:
      sent_none[i]:sent_choice[i]:sent_counter[i]);
    init(st_choose[i]:1, sent_none[i]:1, rec_none[i]:1);
  }
}

```

```

    for (int j in {0..N-1}) { trans inc_counter[i][j]; }
}

for (int i in {0..N-1}) {
  arcs(
    st_choose[i]:pick_value0[i], pick_value0[i]:st_wait[i],
    sent_none[i]:pick_value0[i], pick_value0[i]:sent_none[i],
    rec_none[i]:pick_value0[i], pick_value0[i]:rec_none[i],
    pref[i]:pick_value0[i]:tk(pref[i]),

    st_choose[i]:pick_value1[i], pick_value1[i]:st_wait[i],
    sent_none[i]:pick_value1[i], pick_value1[i]:sent_none[i],
    rec_none[i]:pick_value1[i], pick_value1[i]:rec_none[i],
    pref[i]:pick_value1[i]:tk(pref[i]), pick_value1[i]:pref[i],

    st_wait[i]:send_pref[i], send_pref[i]:st_wait[i],
    sent_none[i]:send_pref[i], send_pref[i]:sent_choice[i],

    st_wait[i]:recv_stay_active0[i], recv_stay_active0[i]:st_active[i],
    rec_none[i]:recv_stay_active0[i], recv_stay_active0[i]:rec_choice[i],
    sent_choice[i]:recv_stay_active0[i], recv_stay_active0[i]:sent_choice[i],

    st_wait[i]:recv_stay_active1[i], rec_none[i]:recv_stay_active1[i],
    recv_stay_active1[i]:st_active[i], recv_stay_active1[i]:rec_choice[i],
    pref[i]:recv_stay_active1[i], recv_stay_active1[i]:pref[i],
    sent_choice[i]:recv_stay_active1[i], recv_stay_active1[i]:sent_choice[i],

```

```

st_wait[i]:recv_become_inactive[i], recv_none[i]:recv_become_inactive[i],

recv_become_inactive[i]:st_inactive[i], recv_become_inactive[i]:rec_choice[i],

sent_choice[i]:recv_become_inactive[i], recv_become_inactive[i]:sent_choice[i],

pref[mod(i+1,N)]:recv_become_inactive[i], recv_become_inactive[i]:pref[mod(i+1,N)],

st_active[i]:send_new_counter[i], send_new_counter[i]:st_active[i],

sent_choice[i]:send_new_counter[i], send_new_counter[i]:sent_counter[i],

st_active[i]:recv_counter[i], recv_counter[i]:st_choose[i],

rec_choice[i]:recv_counter[i], recv_counter[i]:rec_none[i],

sent_counter[mod(i+1,N)]:recv_counter[i],

recv_counter[i]:sent_none[mod(i+1,N)],

pref[i]:recv_counter[i]:tk(pref[i]), c[i]:recv_counter[i]:tk(c[i]),

c[mod(i+1,N)]:recv_counter[i]:tk(c[mod(i+1,N)]),

st_active[i]:become_leader[i], become_leader[i]:st_leader[i],

rec_choice[i]:become_leader[i], become_leader[i]:rec_none[i],

sent_counter[mod(i+1,N)]:become_leader[i],

become_leader[i]:sent_none[mod(i+1,N)],

c[mod(i+1,N)]:become_leader[i]:N-1,

st_inactive[i]:i_recv_pref0[i], i_recv_pref0[i]:st_inactive[i],

rec_none[i]:i_recv_pref0[i], i_recv_pref0[i]:rec_choice[i],

pref[i]:i_recv_pref0[i]:tk(pref[i]),

c[i]:i_recv_pref0[i]:tk(c[i]),

```

```

    st_inactive[i]:i_recv_pref1[i], i_recv_pref1[i]:st_inactive[i],
    rec_none[i]:i_recv_pref1[i], i_recv_pref1[i]:rec_choice[i],
    pref[mod(i+1,N)]:i_recv_pref1[i], i_recv_pref1[i]:pref[mod(i+1,N)],
    pref[i]:i_recv_pref1[i]:tk(pref[i]), i_recv_pref1[i]:pref[i],
    c[i]:i_recv_pref1[i]:tk(c[i]),
    st_inactive[i]:i_send_pref[i], i_send_pref[i]:st_inactive[i],
    sent_none[i]:i_send_pref[i], i_send_pref[i]:sent_choice[i],

    st_inactive[i]:i_send_counter[i], i_send_counter[i]:st_inactive[i],
    rec_counter[i]:i_send_counter[i], i_send_counter[i]:rec_none[i],
    sent_choice[i]:i_send_counter[i], i_send_counter[i]:sent_counter[i],
    pref[i]:i_send_counter[i]:tk(pref[i])
);
for (int j in {0..N-1}) {
    arcs(
        st_inactive[i]:inc_counter[i][j], inc_counter[i][j]:st_inactive[i],
        rec_choice[i]:inc_counter[i][j], inc_counter[i][j]:rec_counter[i],
        sent_counter[mod(i+1,N)]:inc_counter[i][j],
        inc_counter[i][j]:sent_none[mod(i+1,N)],
        c[mod(i+1,N)]:inc_counter[i][j]:j, inc_counter[i][j]:c[i]:j+1);
}
inhibit(
    pref[mod(i+1,N)]:recv_stay_active0[i],
    sent_none[mod(i+1,N)]:recv_stay_active0[i],
    sent_none[mod(i+1,N)]:recv_stay_active1[i],
    sent_none[mod(i+1,N)]:recv_become_inactive[i],

```

```

    sent_none[mod(i+1,N)]:i_recv_pref0[i],
    sent_none[mod(i+1,N)]:i_recv_pref1[i],
    pref[i]:recv_become_inactive[i],
    sent_choice[i]:recv_counter[i],
    c[mod(i+1,N)]:recv_counter[i]:N-1,
    pref[mod(i+1,N)]:i_recv_pref0[i],
    rec_none[i]:i_send_pref[i]
);
for (int j in {0..N-1}) {
    inhibit(sent_none[mod(i+1,N)]:inc_counter[i][j],
           c[mod(i+1,N)]:inc_counter[i][j]:j+1);
}
}
}

```

## B.6 A round-robin mutual exclusion protocol

The protocol regulates the access to a shared resource (e.g. a communication channel) for a ring of  $N$  processors [72]. The resource manager gives permission to use the channel to each process in order, by moving a token around the ring. When a process has the token, it reads in a message from the channel and stores it in its own buffer. It can then release the token to the next processor before reading the contents of the buffer, or read the buffer first and then send the token to the neighbour.

Figure B.6 shows the subnet for process  $i$ . Each subnet is initially marked with one token in place  $wait_i$ , except for process 0 which starts with one token in  $req_i$ , meaning that it is the first process to have access when the protocol starts.



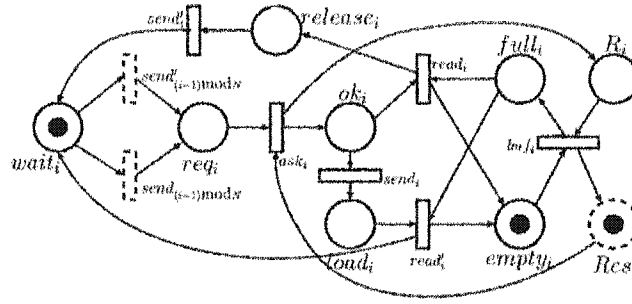


Figure B.6: The round-robin mutual exclusion protocol.

```

spn robin(int N) := {
  place Res;
  partition(1:Res);
  for (int i in {0..N-1}) {
    place
      R[i], bufidle[i], buffull[i],
      pwait[i], pask[i], pok[i], pload[i], psend[i];
    trans
      task[i], tbuf[i], t1load[i], t2load[i], t1send[i], t2send[i];
    partition(
      i+2:bufidle[i]:buffull[i]:pwait[i]:pask[i]:pok[i]:pload[i]:psend[i],
      1:R[i]);
    firing(task[i]:expo(1.0), tbuf[i]:expo(1.0), t1load[i]:expo(1.0),
      t1send[i]:expo(1.0), t2load[i]:expo(1.0), t2send[i]:expo(1.0));
  }
}

```

```

for (int i in {0..N-1}) {
  arcs(Res:task[i], pask[i]:task[i],
       task[i]:R[i], task[i]:pok[i],
       R[i]:tbuf[i], bufidle[i]:tbuf[i],
       tbuf[i]:buffull[i], tbuf[i]:Res,
       buffull[i]:t1load[i], pok[i]:t1load[i],
       t1load[i]:bufidle[i], t1load[i]:psend[i],
       buffull[i]:t2load[i], pload[i]:t2load[i],
       t2load[i]:bufidle[i], t2load[i]:pwait[i],
       pok[i]:t1send[i], pwait[mod(i+1,N)]:t1send[i],
       t1send[i]:pload[i], t1send[i]:pask[mod(i+1,N)],
       psend[i]:t2send[i], pwait[mod(i+1,N)]:t2send[i],
       t2send[i]:pwait[i], t2send[i]:pask[mod(i+1,N)]);
}

init(Res:1, pask[0]:1);

for (int i in {1..N-1}) {
  init (pwait[i]:1);
}

for (int i in {0..N-1}) {
  init(bufidle[i]:1);
}
}

```

# Bibliography

- [1] P. AZIZ ABDULLA, P. BJESSE, AND N. EÉN. Symbolic reachability analysis based on SAT-solvers. In *TACAS 2000*, volume 1785 of *LNCS*, pages 411–425. Springer-Verlag, 2000.
- [2] MARCO AJMONE MARSAN, GIANFRANCO BALBO, GIOVANNI CONTE, SUSANNA DONATELLI, AND GIULIANA FRANCESCHINIS. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, New York, 1995.
- [3] S. B. AKERS. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27(6), June 1978.
- [4] R. ALUR, R.K. BRAYTON, T.A. HENZINGER, S. QADEER, AND S.K. RAJAMANI. Partial-order reduction in symbolic state-space exploration. In *CAV '97*, volume 1254 of *LNCS*, pages 340–351. Springer-Verlag, 1997.
- [5] R. ALUR, T. A. HENZINGER, AND P.-H. HO. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, 1996.
- [6] V. AMOIA, G. DE MICHELI, AND M. SANTOMAURO. Computer-oriented formulation of transition-rate matrices via Kronecker algebra. *IEEE Trans. Rel.*, 30:123–132, June 1981.
- [7] H. R. ANDERSEN, J. STAUNSTRUP, AND N. MARETTI. Partial model checking with ROBDDs. *Lecture Notes in Computer Science*, 1217:35–49, 1997.
- [8] R. I. BAHAR, E. A. FROHM, C. M. GAONA, G. D. HACHTEL, E. MACIII, A. PARDO, AND F. SOMENZI. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206, April 1997.
- [9] A. BIERE, A. CIMATTI, E. CLARKE, AND Y. ZHU. Symbolic model checking without BDDs. *LNCS*, 1579:193–207, 1999.
- [10] N. BJORNER, A. BROWNE, E. CHANG, M. COLON, A. KAPUR, Z. MANNA, H. B. SIPMA, AND T. E. URIBE. STeP: The stanford temporal prover (educational release) user’s manual. Technical Report CS-TR-95-1562, Stanford University, 1995.
- [11] RODERICK BLOEM, HAROLD N. GABOW, AND FABIO SOMENZI. An algorithm for strongly connected component analysis in  $n \log n$  symbolic steps. In *Proc. of FMCAD 2000*, pages 37–54. Springer-Verlag, 2000.

- [12] RODERICK BLOEM, KAVITA RAVI, AND FABIO SOMENZI. Symbolic guided search for CTL model checking. In *Proc. DAC*, pages 29–34, Los Angeles, CA, USA, 2000. ACM Press.
- [13] AHMED BOUAJJANI, BENGT JONSSON, MARCUS NILSSON, AND TAYSSIR TOUIL. Regular model checking. In *Computer Aided Verification*, pages 403–418, 2000.
- [14] R. S. BOYER AND J. S. MOORE. *A Computational Logic*. Academic Press, New York, 1979.
- [15] R. K. BRAYTON, G. D. HACHTEL, A. SANGIOVANNI-VINCENTELLI, AND F. SOMENZI. VIS: a system for verification and synthesis. *Lecture Notes in Computer Science*, 1102:428–437, 1996.
- [16] R. E. BRYANT. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comp.*, 35(8):677–691, August 1986.
- [17] R. E. BRYANT. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comp. Surv.*, 24(3):393–318, 1992.
- [18] R. E. BRYANT AND Y.A. CHEN. Verification of arithmetic circuits with binary moment diagrams. In *Proc. of Design Automation Conf. (DAC)*, pages 535–541, 1995.
- [19] PETER BUCHHOLZ, GIANFRANCO CIARDO, SUSANNA DONATELLI, AND PETER KEMPER. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS J. Comp.*, 12(3):203–222, 2000.
- [20] J. R. BURCH, E. M. CLARKE, AND D. E. LONG. Symbolic model checking with partitioned transition relations. In *Int. Conference on Very Large Scale Integration*, A. Halaas and P.B. Denyer, editors, pages 49–58, Edinburgh, Scotland, August 1991. IFIP Transactions, North-Holland.
- [21] J. R. BURCH, E. M. CLARKE, K. L. McMILLAN, D. L. DILL, AND L. J. HWANG. Symbolic model checking:  $10^{20}$  states and beyond. In *Proc. 5th Annual IEEE Symp. on Logic in Computer Science*, pages 428–439, Philadelphia, PA, 4–7 June 1990. IEEE Comp. Soc. Press.
- [22] JOHN P. BURGESS. Basic tense logic. In *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic*, D. Gabbay and F. Guenther, editors, volume 165 of *Synthese Library*, chapter II.2, pages 89–133. D. Reidel Publishing Co., Dordrecht, 1984.
- [23] RICKY W. BUTLER. Formal methods for life-critical software. In *AIAA Computing in Aerospace 9 Conference*, pages 319–29, San Diego, CA, 19–21 October 1993.
- [24] G. CABODI, P. CAMURATI, AND S. QUER. Improving symbolic traversals by means of activity profiles. In *Design Automation Conference*, pages 306–311, 1999.

- [25] GIANFRANCO CIARDO, ROBERT L. JONES, ANDREW S. MINER, AND RADU SIMINICEANU. Logical and stochastic modeling with SMART. In *Proc. Modelling Techniques and Tools for Computer Performance Evaluation*, Peter Kemper and William H. Sanders, editors, LNCS 2794, pages 78–97, Urbana, IL, USA, September 2003. Springer-Verlag.
- [26] GIANFRANCO CIARDO, GERALD LUETTGEN, AND RADU SIMINICEANU. Efficient symbolic state-space construction for asynchronous systems. In *Proc. 21th Int. Conf. on Applications and Theory of Petri Nets*, Mogens Nielsen and Dan Simpson, editors, LNCS 1825, pages 103–122, Aarhus, Denmark, June 2000. Springer-Verlag.
- [27] GIANFRANCO CIARDO, GERALD LUETTGEN, AND RADU SIMINICEANU. Saturation: An efficient iteration strategy for symbolic state space generation. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Tiziana Margaria and Wang Yi, editors, LNCS 2031, pages 328–342, Genova, Italy, April 2001. Springer-Verlag.
- [28] GIANFRANCO CIARDO, ROBERT MARMORSTEIN, AND RADU SIMINICEANU. Saturation unbound. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Hubert Garavel and John Hatchiff, editors, LNCS 2619, pages 379–393, Warsaw, Poland, April 2003. Springer-Verlag.
- [29] GIANFRANCO CIARDO, ANDREW S. MINER, ROBERT L. JONES, ROBERT MARMORSTEIN, AND RADU SIMINICEANU. SMART: Stochastic Model checking Analyzer for Reliability and Timing, User Manual. Available at <http://www.cs.wm.edu/~ciardo/SMART/>.
- [30] GIANFRANCO CIARDO AND RADU SIMINICEANU. Using edge-valued decision diagrams for symbolic generation of shortest paths. In *Proc. Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Mark D. Aagaard and John W. O’Leary, editors, LNCS 2517, pages 256–273, Portland, OR, USA, November 2002. Springer-Verlag.
- [31] GIANFRANCO CIARDO AND RADU SIMINICEANU. Structural symbolic CTL model checking of asynchronous systems. In *Computer Aided Verification (CAV’03)*, Warren Hunt, Jr. and Fabio Somenzi, editors, volume 2725 of LNCS, pages 40–53, Boulder, CO, USA, July 2003. Springer-Verlag.
- [32] GIANFRANCO CIARDO AND KISHOR S. TRIVEDI. A decomposition approach for stochastic Petri net models. In *Proc. 4th Int. Workshop on Petri Nets and Performance Models (PNPM’91)*, pages 74–83, Melbourne, Australia, December 1991. IEEE Comp. Soc. Press.
- [33] A. CIMATTI, E. M. CLARKE, F. GIUNCHIGLIA, AND M. ROVERI. NUSMV: a new Symbolic Model Verifier. In *Proceedings Eleventh Conference on Computer-Aided Verification (CAV’99)*, N. Halbwachs and D. Peled, editors, LNCS 1633, pages 495–499, Trento, Italy, July 1999. Springer.

- [34] E. M. CLARKE AND I.A. DRAGHICESCU. Expressibility results for linear time and branching time logics. *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, 354:428–437, 1988.
- [35] E. M. CLARKE AND E. A. EMERSON. Characterizing properties of parallel programs as fixpoints. In *Seventh International Colloquium on Automata, Languages, and Programming*, volume 85 of *LNCS*, 1981.
- [36] 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, April 1986.
- [37] E. M. CLARKE AND E.A. EMERSON. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights NY*, volume 131 of *LNCS*, May 1981.
- [38] E. M. CLARKE, T. FILKORN, AND S. JHA. Exploiting symmetry in model checking. In *CAV '93*, volume 697 of *LNCS*, pages 450–462. Springer-Verlag, 1993.
- [39] E. M. CLARKE, M. FUJITA, P. C. MCGEER, K. MCMILLAN, J. C.-Y. YANG, AND X. ZHAO. Multi-terminal binary decision diagrams: an efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2/3):149+, April 1997.
- [40] E. M. CLARKE, O. GRUMBERG, KEN 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, San Francisco, CA, USA, 1995.
- [41] E. M. CLARKE, O. GRUMBERG, AND D. A. PELED. *Model Checking*. MIT Press, 1999.
- [42] E. M. CLARKE, D. E. LONG, AND K. L. MCMILLAN. Compositional model checking. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 353–362, Asilomar Conference Center, Pacific Grove, California, 5–8 June 1989. IEEE Computer Society Press.
- [43] E. M. CLARKE AND X. ZHAO. Analytica — A theorem prover in Mathematica. *Lecture Notes in Computer Science*, 607:761–765, 1992.
- [44] E. M. CLARKE AND X. ZHAO. Word level symbolic model checking: A new approach for verifying arithmetic circuits. Technical Report CS-95-161, Carnegie Mellon University, School of Computer Science, May 1995.
- [45] R. CLEAVELAND, J. PARROW, AND B. STEFFEN. The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems. *ACM TOPLAS*, 15(1):36–72, 1993.
- [46] R. L. CONSTABLE, S. ALLEN, H. BROMELY, W. CLEVELAND, ET AL. *Implementing Mathematics with the NuPRL Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

- [47] JAMES CORBETT, MATTHEW DWYER, JOHN HATCLIFF, CORINA PASAREANU, ROBBY, SHAWN LAUBACH, AND HONGJUN ZHENG. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. IEEE Computer Society.
- [48] C. CORNES, J. COURANT, J-C. FILLIATRE, G. HUET, P. MANOURY, C. MUNOZ, C. MURTHY, C. PARENT, A. SAIBI, AND B. WERNER. The Coq proof assistant, reference manual, version 5.10. Technical Report RT-0177, Inria, 1995.
- [49] D. CRAIGEN, S. KROMODIMOELJO, I. MEISELS, A. NEILSON, B. PASE, AND M. SAALTINK. m-EVES: A tool for verifying software. In *Proceedings of the 10th International Conference on Software Engineering*, pages 324–333. IEEE Press, 1988.
- [50] W. DAMM, B. JOSKO, AND R. SCHLÖR. Specification and verification of VHDL-based system-level hardware designs. In *Specification and Validation Methods*, pages 331–410. Oxford University Press, 1995.
- [51] D. DAMS, R. GERTH, G. DOHMEN, R. HERRMANN, P. KELB, AND H. PARGMANN. Model checking using adaptive state and data abstraction. In *Proc. 6th International Computer Aided Verification Conference*, pages 455–467, 1994.
- [52] M. DAVIO, J-P. DESCHAMPS, AND A. THAYSE. *Discrete and Switching Functions*. McGraw-Hill, New York, 1st edition, 1978.
- [53] D. DEHARBE AND D. BORRIONE. Semantics of a verification-oriented subset of VHDL. *Lecture Notes in Computer Science*, 987:293–310, 1995.
- [54] D. L. DILL, J. DREXLER, A. J. HU, AND C. H. YANG. Protocol verification as a hardware design aid. In *International Conference on Computer Design, VLSI in Computers and Processors*, pages 522–525, Los Alamitos, Ca., USA, October 1992.
- [55] MARK DOWSON. The ARIANE 5 software failure. *Software Engineering Notes*, 22(2):84, March 1997.
- [56] R. DRECHSLER AND D. SIELING. Binary decision diagrams in theory and practice. *Software Tools for Technology Transfer*, 3:112–136, 2001.
- [57] ALAN EDELMAN. The mathematics of the Pentium division bug. *SIAM Rev.*, 39(1):54–67, 1997.
- [58] E. A. EMERSON AND J. Y. HALPERN. Sometimes and Not Never revisited: On branching time versus linear time. *Journal of the ACM*, 33:151–178, 1986.
- [59] E. ALLEN EMERSON AND JOSEPH Y. HALPERN. Decision procedures and expressiveness in the temporal logic of branching time. In *Proceedings of the fourteenth annual ACM Symposium on Theory of Computing, San Francisco, California, May 5–7, 1982*, ACM, editor, pages 169–180, New York, NY, USA, 1982. ACM Press.
- [60] FEDERAL AVIATION ADMINISTRATION, US DEPARTMENT OF TRANSPORTATION. Introduction to TCAS II, March 1990.

- [61] T. FILKORN, H. A. SCHNEIDER, A. SCHOLZ, A. STRASSER, AND P. WARKENTIN. *SVE User's Guide*. Siemens AG, Munich, 1994.
- [62] M. FUJITA, H. FUJISAWA, AND N. KAWATO. Evaluation and improvements of boolean comparison method based on binary decision diagrams. In *Int. Conference on Computer Aided Design*. IEEE, 1988.
- [63] M. FUJITA, H. FUJISAWA, AND Y. MATSUNAGA. Variable ordering algorithms for ordered binary decision diagrams and their evaluation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 12(1):6–12, 1993.
- [64] D. GABBAY, A. PNUELI, S. SHELAH, AND J. STAVI. The temporal analysis of fairness. In *Conference Record of the Seventh annual ACM Symposium on Principles of Programming Languages*, pages 163–173. ACM, ACM, January 1980.
- [65] S. J. GARLAND AND J. V. GUTTAG. LP: The Larch prover. In *Proceedings on the 9th International Conference on Automated Deduction*, volume 310 of *LNCS*, pages 748–749, Berlin, May 1988. Springer.
- [66] A. GESER, J. KNOOP, G. LÜTTGEN, B. STEFFEN, AND O. RÜTHING. Chaotic fixed point iterations. Technical Report MIP-9403, Univ. of Passau, 1994.
- [67] PATRICE GODEFROID. *Partial-order Methods for the Verification of Concurrent Systems – An Approach to the State-explosion Problem*, volume 1032 of *LNCS*. Springer-Verlag, 1996.
- [68] PATRICE GODEFROID AND DAVID E. LONG. Symbolic protocol verification with queue BDDs. *Formal Methods in System Design*, 14(3):257–271, May 1999.
- [69] M. GORDON. A proof generating system for higher-order logic. Technical Report 103, University of Cambridge, January 1987.
- [70] M. GORDON, R. MILNER, AND C. WADSWORTH. Edinburgh LCF. *Lecture Notes in Computer Science*, 78, 1979.
- [71] J. ARTHUR GOWAN, CHRIS JESSE, AND RICHARD G. MATTHIEU. Y2K compliance and the distributed enterprise. *Communications of the ACM*, 42(2):68–73, February 1999.
- [72] S. GRAF, B. STEFFEN, AND G. LÜTTGEN. Compositional minimisation of finite state systems using interface specifications. *Formal Asp. of Comp.*, 8(5):607–616, 1996.
- [73] DAVID F. GREEN JR. Runway safety monitor algorithm for runway incursion detection and alerting. Technical Report CR-2002-211416, NASA Langley Research Center, Hampton, VA, 2002.
- [74] D. HAREL. Statecharts: A visual approach to complex systems. Technical Report CS84-05, Weizmann Institute, Rehovot, Israel, July 1984.



- [75] SCOTT HAZELHURST. Compositional model checking of partially ordered state spaces. Technical Report TR-96-02, Department of Computer Science, University of British Columbia, January 1996.
- [76] JESPER G. HENRIKSEN, JAKOB L. JENSEN, MICHAEL E. JØRGENSEN, NILS KLARLUND, ROBERT PAIGE, THEIS RAUHE, AND ANDERS SANDHOLM. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems*, Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, volume 1019, pages 89–110. Springer, 1995.
- [77] T. A. HENZINGER, X. NICOLLIN, J. SIFAKIS, AND S. YOVINE. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, June 1994.
- [78] C. A. R. HOARE. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.
- [79] DIETER HOGREFE. *Estelle, Lotos und SDL*. Springer Verlag, Berlin, 1989.
- [80] R. HOJATI, R. K. BRAYTON, AND R. P. KURSHAN. BDD-based debugging of designs using language containment and fair CTL. In *Proc. 5th International Computer Aided Verification Conference*, pages 41–58, 1993.
- [81] G. HOLZMANN. The model checker SPIN. *IEEE Trans. on Softw. Eng.*, 23(5):279–295, 1997.
- [82] G. HOLZMANN AND DORON PELED. An improvement in formal verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211, Berne, Switzerland, October 1994. Chapman & Hall.
- [83] S. I. MINATO. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, ACM-SIGDA; IEEE, editor, pages 272–277, Dallas, TX, June 1993. ACM Press.
- [84] A. ITAI AND M. RODEH. Symmetry breaking in distributive networks. In *22th Annual Symp. on Foundations of Computer Science*, pages 150–158, Los Alamitos, CA, USA, October 1981. IEEE Comp. Soc. Press.
- [85] C. B. JONES. *Systematic Development using VDM*. International Series in Computer Science. Prentice-Hall, Englewood Cliffs, 1986.
- [86] JPL SPECIAL REVIEW BOARD. Report on the Loss of the Mars Polar Lander Deep Space 2 Missions, 2000.
- [87] 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.
- [88] D. KAPUR AND D. R. MUSSER. Proof by consistency. *Artificial Intelligence*, 31(2):125–157, 1987.

- [89] M. KAUFMANN AND J. S. MOORE. ACL2: An industrial strength version of Nqthm. *COMPASS — Proceedings of the Annual Conference on Computer Assurance*, pages 23–34, 1996.
- [90] SHINJI KIMURA AND EDMUND M. CLARKE. A parallel algorithm for constructing binary decision diagrams. In *Proc. Int. Conf. on Computer Design (ICCD)*, pages 220–223, Cambridge, MA, September 1990. IEEE Comp. Soc. Press.
- [91] SAUL A. KRIPKE. Semantic considerations on modal logic. *Acta Philosophica Fennica*, 24:83–94, 1963.
- [92] MARTA Z. KWIATKOWSKA, GETHIN NORMAN, AND DAVID PARKER. PRISM: Probabilistic Symbolic Model Checker. In *Proc. Computer Performance Evaluation / TOOLS*, pages 200–204, London, UK, April 2003.
- [93] Y.-T. LAI, M. PEDRAM, AND B. K. VRUDHULA. Formal verification using edge-valued binary decision diagrams. *IEEE Trans. Comp.*, 45:247–255, 1996.
- [94] 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, Los Alamitos, CA, USA, June 1992. IEEE Computer Society Press.
- [95] L. LAMPORT. Time, clocks and the ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–564, 1978.
- [96] L. LAMPORT. Sometimes is sometimes Not Never. *Annual ACM Symposium on Principles of Programming Languages*, pages 174–185, 1980.
- [97] K. LARSEN, P. PETTERSSON, AND W. YI. Compositional and symbolic model-checking of real-time systems. In *RTSS '95*, pages 76–89. Computer Society Press, 1995.
- [98] C. Y. LEE. Representation of switching circuits by binary-decision programs. *Bell Syst. Techn. J.*, 38(4):985–999, July 1959.
- [99] P. LESCANNE. Computer experiments with the REVE term rewriting systems generator. In *Proceedings of 10th ACM Symposium on Principles of Programming Languages*, pages 99–108, 1983.
- [100] LESLIE LAMPORT. What good is temporal logic? In *Proceedings of the IFIP Congress on Information Processing*, R. E. A. Mason, editor, pages 657–667, Amsterdam, 1983. North-Holland.
- [101] Z. LUO AND R. POLLACK. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.
- [102] N. A. LYNCH AND M. TUTTLE. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR-387, Massachusetts Institute of Technology, April 1987.

- [103] ZOHAR MANNA AND AMIR PNUELI. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992.
- [104] K. L. MCMILLAN. *Symbolic Model Checking: An Approach to the State-explosion Problem*. PhD thesis, Carnegie-Mellon Univ., 1992.
- [105] R. MILNER. *A Calculus for Communicating Processes*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.
- [106] KIM MILVANG-JENSEN AND ALAN J. HU. BDDNOW: A parallel BDD package. In *Formal Methods in Computer-Aided Design*, H. Beilner and F. Bause, editors, LNCS 1522, pages 501–507. Springer-Verlag, 1998.
- [107] ANDREW S. MINER AND GIANFRANCO CIARDO. Efficient reachability set generation and storage using decision diagrams. In *Proc. 20th Int. Conf. on Applications and Theory of Petri Nets*, H.C.M. Kleijn and Susanna Donatelli, editors, LNCS 1639, pages 6–25, Williamsburg, VA, USA, June 1999. Springer-Verlag.
- [108] ANDREW STEPHEN MINER. *Data structures for the analysis of large structured Markov models*. PhD thesis, College of William and Mary, Williamsburg, VA, 2000.
- [109] TADAO MURATA AND R. CHURCH. Analysis of marked graphs and Petri nets by matrix equations. Research report MDC 1.1.8, Department of information engineering, Univeristy of Illinois, Chicago,IL, November 1975.
- [110] S. OWRE, J. M. RUSHBY, AND N. SHANKAR. PVS: A prototype verification system. *Lecture Notes in Computer Science*, 607:748–752, 1992.
- [111] E. PASTOR, O. ROIG, J. CORTADELLA, AND R. BADIA. Petri net analysis using boolean manipulation. In *Proc. 15th Int. Conf. on Applications and Theory of Petri Nets*, Robert Valette, editor, LNCS 815, pages 416–435, Zaragoza, Spain, June 1994. Springer-Verlag.
- [112] JAMES L. PETERSON. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.
- [113] C. A. PETRI. *Kommunikation mit Automaten*. PhD thesis, Inst. fur Instrumentelle Mathematik, Un. Bonn, FR Germany, 1962.
- [114] SERGIO PISSANETZKY. *Sparse Matrix Technology*. Academic Press, 1984.
- [115] BRIGITTE PLATEAU. On the stochastic structure of parallelism and synchronisation models for distributed algorithms. In *Proc. ACM SIGMETRICS*, pages 147–153, Austin, TX, USA, May 1985.
- [116] AMIR PNUELI. The temporal logic of programs. *IEEE FOCS*, pages 46–57, October 1977.
- [117] HARRY PREUSS AND ANAND SRIVASTAV. Blockwise variable orderings for shared BDDs. In *MFCS: Symp. on Mathematical Foundations of Computer Science*, 1998.

- [118] ARTHUR N. PRIOR. *Time and modality*. Oxford University Press, Oxford, UK, 1957.
- [119] J.P. QUIELLE AND J. SIFAKIS. Specification and verification of concurrent systems in CESAR. In *5th International Symposium on Programming*, pages 337–350, 1981.
- [120] KAVITA RAVI AND FABIO SOMENZI. High-density reachability analysis. In *ICCAD '95*, pages 154–158. IEEE Computer Society Press, 1995.
- [121] KAVITA RAVI AND FABIO SOMENZI. Efficient fixpoint computation for invariant checking. In *Proc. Int. Conference on Computer Design (ICCD)*, pages 467–474, Austin, TX, October 1999. IEEE Comp. Soc. Press.
- [122] NICHOLAS RESCHER AND ALASDAIR URQUHART. *Temporal Logic*. Springer-Verlag, Berlin, 1971.
- [123] O. ROIG, J. CORTADELLA, AND E. PASTOR. Verification of asynchronous circuits by BDD-based model checking of Petri nets. In *Proc. 16th Int. Conf. on Applications and Theory of Petri Nets, Turin, Italy*, Giorgio De Michelis and Michel Diaz, editors, LNCS 935, pages 374–391. Springer-Verlag, June 1995.
- [124] R. RUDELL. Dynamic variable reordering for ordered binary decision diagrams. In *International Conference on Computer Aided Design, Santa Clara CA*, Nov. 1993.
- [125] M. SHEERAN AND G. STÅLMARCK. A tutorial on Stålmärck’s proof procedure for propositional logic. *Formal Methods in System Design*, 16(1):23–58, 2000.
- [126] MARC SOLÉ AND ENRIC PASTOR. Traversal techniques for concurrent systems. *Lecture Notes in Computer Science*, 2517:220–237, 2002.
- [127] FABIO SOMENZI. CUDD: CU Decision Diagram Package, Release 2.3.1. <http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html>.
- [128] J. M. SPIVEY. *Introducing Z: A Specification Language and its Formal Semantics*. Cambridge University Press, 1988.
- [129] B. STEFFEN, T. MARGARIA, A. CLASSEN, AND V. BRAUN. The METAFrame’95 environment. In *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 450–453. Springer Verlag, 1996.
- [130] ALFRED TARSKI. A lattice-theoretical fixpoint theorem and its application. *Pacific Journal of Mathematics*, pages 285–309, 1955.
- [131] MARCO TILGNER, YUKIO TAKAHASHI, AND GIANFRANCO CIARDO. SNS 1.0: Synchronized Network Solver. In *1st Int. Workshop on Manufacturing and Petri Nets*, pages 215–234, Osaka, Japan, June 1996.
- [132] ANTTI VALMARI. A stubborn attack on the state explosion problem. In *CAV '90*, pages 25–42. AMS, 1990.

- [133] M.Y. VARDI AND P. WOLPER. An automata-theoretic approach to automatic program verification. *LICS*, pages 332–344, 1986.
- [134] F. VERNADAT, P. AZEMA, AND F. MICHEL. Covering step graph. In *ICATPN '96*, volume 1091 of *LNCS*, pages 516–535. Springer-Verlag, 1996.
- [135] POUL F. WILLIAMS, ARMIN BIERE, EDMUND M. CLARKE, AND ANUBHAV GUPTA. Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking. In *Proceedings of CAV'00*, pages 124–138, 2000.
- [136] AIGUO XIE AND PETER A. BEEREL. Implicit enumeration of strongly connected components. In *ICCAD'99*, pages 37–40. ACM Press, 1999.
- [137] BWOLEN YANG AND DAVID R. O'HALLARON. Parallel breadth-first BDD construction. In *Proc. of the Sixth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'97)*, pages 145–156, Las Vegas, NV, June 1997.
- [138] SHIPEI ZHANG, OLEG SOKOLSKY, AND SCOTT A. SMOLKA. On the parallel complexity of model checking in the modal  $\mu$ -calculus. In *Proceedings of the 9th Annual IEEE Symposium on Logic in Computer Science*, pages 154–163, Paris, France, 4–7 July 1994. IEEE Computer Society Press.

## VITA

Radu Siminiceanu

Radu Siminiceanu was Born in Iași, Romania, on September 21, 1973. He graduated from “C. Negruzzi” High School in 1992, as Valedictorian. He earned B.S. and M.S. degrees in Computer Science from “Al.I.Cuza” University of Iași, in 1997 and 1998, respectively. He also graduated college as Valedictorian, and was later appointed as Teaching Fellow by the same department, during the Spring semester of 1998.

Mr. Siminiceanu entered the Ph.D. program of the Computer Science Department at the College of William and Mary in August 1998. During the next five years, he worked as Research Assistant under the supervision of Dr. Gianfranco Ciardo, focusing on symbolic Model Checking and Formal Verification. He was granted a Teaching Fellowship by the Computer Science Department, in the Fall semester of 2003, and was the recipient of the “Stephen K. Park” award for graduate student research in 2003. He completed the Ph.D. program in December 2003.

In 2004, Dr. Siminiceanu will start working as a staff scientist at the National Institute of Aerospace, in Hampton, Virginia.