UNIVERSITY OF CALIFORNIA
RIVERSIDE

Decision Diagram Algorithms for Logic and Timed Verification

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

Doctor of Philosophy

in

Computer Science

by

Min Wan

December 2008

Dissertation Committee:

Professor Gianfranco Ciardo, Chairperson
Professor Rajiv Gupta
Professor Neal E. Young

The Dissertation of Min Wan is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

Foremost, I am grateful and thankful for my advisor Gianfranco Ciardo, who introduced me to the research topic and taught me not only the knowledge but also the way to tackle difficult problem. This thesis would have been impossible without him. I am also grateful for his financial support throughout my Ph.D. study and generous support for my travel to conferences.

I am deeply grateful to my parents, Xiaochu Wan and Xiaofeng Li, who shared the joy and trouble with me in my research adventure. Their support and trust have always given me the strength to overcome difficulties.

I owe special thanks to Andrew S. Miner and Jinqing (Andy) Yu. Their research established the foundation for my research, and the collaboration with them has been a great experience. Dr. Miner is a co-designer of the SMART tool, which gave me a strong platform to carry out my own research.

Finally, I would like to thank all my lovely friends, who accompanied me during these five years and made the Ph.D. study an wonderful and unforgettable journey.

To my parents.

ABSTRACT OF THE DISSERTATION

Decision Diagram Algorithms for Logic and Timed Verification

by

Min Wan

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2008
Professor Gianfranco Ciardo, Chairperson

Symbolic verification has received much attention from both academia and industry in the past two decades. In particular, techniques based on decision diagrams have been successfully applied to various asynchronous and synchronous models.

Decision diagrams can compactly encode sets and relations, or vectors and matrices. For canonicity, variables associated to the nodes must be found in a predefined order on any path from the root, and duplicate nodes cannot be present. In addition, a *reduction rule* is enforced, the simplest being the *quasi-reduced* rule, where no variable is ever skipped. However, more efficient rules exist, where edges skip *redundant* nodes. With the *fully-reduced* rule, a node is redundant if all its outgoing edges point to the same node. With the *zero-suppressed* rule, a node is redundant if only its 0-edge is not pointing to a pre-defined default *terminal* node. None of these rules, however, is particularly effective when encoding transition relations of asynchronous systems or rate matrices of Markov models. We then introduce an *identity-reduced* rule, which generalizes *Kronecker* encodings to take advantage of state variables that remain unchanged after an event occurrence, and a *c-reduced* rule, which generalizes the zero-suppressed rule, and propose a new *generally-reduced* form of decision diagrams where each variable uses a specific reduction rule. We then illustrate the effectiveness of this new canonical form of decision diagrams with a wide set of applications.

State-space generation is usually the first and fundamental step for symbolic verification. Generally-reduced decision diagrams allow a more efficient symbolic state-space generation for general asynchronous systems by allowing *on-the-fly extension* of the state variable domains. After implementing both breadth-first and *saturation-based* state-space generation with this new data structure, we are able to exhibit substantial efficiency improvements with respect to traditional decision diagrams. Since previous works demonstrated that saturation outperforms breadth-first approaches, saturation with this new structure is now arguably the state-of-the-art algorithm for symbolic state-space generation of asynchronous systems. When state-space generation completes, we also obtain the complete transition relation which can be used for further analysis.

For synchronous systems, we study a type of *timed Petri nets*, which extends the traditional Petri nets to explicitly include real time in the model. We consider two fundamental reachability problems for timed Petri nets with positive integer firing times: *timed reachability* (find all markings where the model can be at a given finite time) and *earliest reachability* (find the minimum time when each reachable marking is entered). For these two problems, we define efficient symbolic algorithms that make use of both generally-reduced decision diagrams without edge value and *edge-valued decision diagrams*, which associate integer value to the edges of decision diagrams. Runtime results on an extensive suite of models are provided to show the effectiveness and capability of our algorithm to cope with large state spaces.

Then, we study the use of decision diagrams in stochastic models. We present a new type of edge-valued decision diagrams which can be used to encode non-negative real-valued functions. We then utilize it in a new *approximate numerical solution algorithm for general structured ergodic models*. The approximation uses a state-space encoding provided by decision diagrams and a transition rate matrix encoding provided by these new edge-valued decision diagrams. The new method retains the favorable properties of a previously proposed Kronecker-based approximation, while eliminating the need for a Kronecker-consistent model decomposition. Removing this restriction allows for a greater utilization of event *locality*, which facilitates both state-space generation and transition rate matrix generation, thus extends the applicability of this algorithm to larger and more complex models.

All these algorithms are implemented based on our newly-designed *Decision Diagram Library* (DDL), which provides a user-friendly interface to create and manipulate generally-reduced decision diagrams with or without edge value It is the first library written for this purpose. This library is written in C++ and we adopt *smart pointers* for decision diagram node interface, similar to those in the Boost libraries [1], which automatically handle reference counts and garbage collection; this technique prevents memory leak and simplifies the interface, which greatly facilitate library users.

All above algorithms are integrated into our verification tool *Symbolic Model-checking Analyzer for Reliability and Timing* (SMART) version 2.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we explain the motivation of our work and describe the scope of this thesis.

1.1 Why verification

Our daily life dependency on computer-based applications (both hardware and software) has motivated Computer Science researchers to develop new techniques that increase our confidence on their correctness. Such applications range from simple coffee machines to nuclear plants and flight control towers. Many of those applications are undoubtedly critical and a failure may cause high damages, both economically and physically.

How can one be sure of the correctness of critical systems, with the increasing design complexities? *Testing*, often performed on an actual product, which “feeds” the system with sensitive data to check that it really behaves as it is supposed to, is a widely used and extremely useful approach in practice. However, it is clearly not possible to use it in highly critical systems were the testing data could cause damages in case of errors before real deployment.

Another solution is to *simulate* the behavior of the system via a *model* on a computer. A model is an abstract representation of the real system. One advantage of simulation is that one does not need to build the real system in order to be applied, and it is thus usually much cheaper than testing.

Both testing and simulation are widespread in industrial applications and become an essential part in product life cycle. One drawback, however, is that it is not possible, in general, to simulate or test all the possible scenarios or behaviors of a given system. That is, those techniques are in general not exhaustive due to the high number of possible cases to be taken into account, and the fact that the failure cases may not be among those tested or simulated.

Formal verification, in particular, *model checking* [30, 41] is the technique to enhance and complement existing validation techniques as testing and simulation.

Model checking consists of a systematically exhaustive exploration of the mathematical model (this is possible for finite models, but also for some infinite models where infinite sets of states can be effectively represented), to verify a certain property, often an *invariant property*, will always held. Usually this consists of exploring all *states* and *events* in the model, by using smart and domain-specific abstraction techniques to consider entire groups of states in a single operation and reduce computing time. Here, a state corresponds to one possible status of the system, while an event describes a atomic status change of the system.

Probability is an important component in the design and analysis of software and hardware systems. Traditionally, probability has been used as a tool to analyze system performance, e.g., steady-state probabilities for computing estimates of measures such as throughput and mean waiting time for queuing networks. *Stochastic model checking* verifies properties related to a probability or likelihood during the execution of a system. There are a number of probabilistic models. In this thesis, we consider continuous-time Markov chains (CTMC) [61] in detail, where each event is associated with an exponentially distributed delay.

1.2 Why decision diagrams

Given an arbitrary model and some property, however, the problem of asking whether the model satisfies the property is *undecidable*. To restrict the problem to a decidable one, model checking is (initially) designed for a system with a *finite state space*.

A state space is the collection of all states reachable from an initial configuration of the system. For exhaustive approaches such as model checking, state-space generation or *reachability analysis* is usually the first and fundamental step.

In practice, most models are compactly described using some *high-level formalism*, e.g., Petri nets (Sec. 2.2), but their underlying state space may be so large that computing and storing the state space may overwhelm even the largest computers.

To deal with this *state-space explosion problem*, implicit data structure such as *decision diagrams* (Sec. 2.3) emerged and widely applied in the past two decades, for both *asynchronous* models and *synchronous* models, including asynchronous circuits, distributed software systems, globally-asynchronous-locally-synchronous systems (GALS), and real-timed systems.

Decision diagrams are trie-like directed acyclic edge-labeled multi-graphs. The first decision diagrams are used to encode binary functions [11], while, later on, variations have been applied to encode arbitrary functions, e.g., sets, relations and matrices. For example, decision diagrams are able to encode billions of states just in a few nodes and the manipulation of those states can be done through operations upon those nodes. The compactness and efficiency of decision diagrams lie in the node sharing and intensive usage of operation caches.

1.3 Our contribution

This thesis focuses on decision diagrams and their usage for verification.

First, we studied variations of decision diagrams and defined a generally-reduced canonical form of decision diagrams, which generalize and also extend all previous decision diagrams without edge value (Ch. 3).

We also extend our previous edge-valued decision diagram to allow it to efficiently encode any non-negative real-valued functions (Ch. 6).

We illustrate the usefulness of the new generally-reduced decision diagrams on two applications, one for the on-the-fly state-space generation of asynchronous models (Ch. 4), the other for a timed extension of tradition Petri nets (Ch. 5).

We also show the effectiveness of the new edge-valued decision diagram on an approximate solution for large CTMCs (Ch. 7).

In order to efficiently implement those algorithms, we designed and coded *Decision Diagram Library* (DDL), which provides a user-friendly interface to create and manipulate generally-reduced decision diagrams without edge value and edge-valued decision diagrams. It is the first library for this purpose (Ch. 8).

This thesis includes work from the following publications.

- M. Wan and G. Ciardo. Extensible decision diagrams for symbolic state-space generation of asynchronous systems. In *Proc. 35th Int. Conf. Current Trends in Theory and Practice of Computer Science (SOFSEM)*, Špindlerův Mlýn, Czech Republic, Jan. 2009. Springer-Verlag. To appear.

- M. Wan and G. Ciardo. Symbolic reachability analysis of integer timed Petri nets. In *Proc. 35th Int. Conf. Current Trends in Theory and Practice of Computer Science (SOFSEM)*, Špindlerův Mlýn, Czech Republic, Jan. 2009. Springer-Verlag. To appear.

- G. Ciardo, A. S. Miner, M. Wan, and A. J. Yu. Approximating stationary measures of structured continuous-time Markov models using matrix diagrams. *ACM SIGMETRICS Perf. Eval. Rev.*, 35(3):16–18, Dec. 2007.

- G. Ciardo and M. Wan. Generally-reduced decision diagrams: definition and applications. In preparation.

- M. Wan, G. Ciardo and A. S. Miner Decision-diagram-based approximate steady-state analysis of large structured Markov models. In preparation.

1.4 Organization of the thesis

The remainder of the thesis is organized as follows. Ch. 2 reviews required background information. Ch. 3 present the new generally-reduced canonical form of decision diagrams. Ch. 4 applies the new decision diagram to state-space generation. Ch. 5 extends method in Ch. 4 to a new synchronous system. Ch. 6 introduces the new canonical type of decision diagrams to encode arbitrary real-valued functions. Ch. 7 utilizes these new decision diagrams on an approximate solution of CTMCs. Ch. 8 describes the new Decision Diagram Library, an elegant library that allows us to implement all algorithms in this thesis, and a complete platform for decision diagram users and developers. Ch. 9 summarizes this thesis.

Chapter 2

Background

In this chapter, we present notation and necessary background for the rest of the thesis. This chapter is organized as follows. In Sec. 2.1, we define the discrete-state model where our algorithms are deployed. Sec. 2.2 introduce a high-level formalism to describe a discrete-state model and also a running example. Sec. 2.3 reviews background on decision diagrams. Finally, Sec. 2.4 summarizes well-known symbolic state space generation techniques.

For reference, Fig. 2.1 and Fig. 2.2 summarizes the symbols we consistently use throughout the thesis.

2.1 Discrete-state systems

Logical specification We consider a structured discrete-state model given by $(\mathcal{S}_0, \mathcal{E}, \{\mathcal{T}_e | e \in \mathcal{E}\})$, whose state is a tuple $\mathbf{x} = (x_L, \cdots, x_1)$ of L variables with a predefined order $x_L \succ \cdots \succ x_1$ imposed on them, with each x_k taking value in a finite set $\mathcal{X}_k = \{0, 1, ..., n_k\} \subset \mathbb{N}$,

Symbol	Definition or Meaning
L	number of submodels/variables
$\mathbf{x}, \mathbf{x}'$	a tuple of state variables with a predefined order according to $\succ$
x_k, x'_k	k^{th} state variable
$\mathbf{i}, \mathbf{i}', \mathbf{j}, \mathbf{j}'$	a state or a tuple of non-negative integers
i_k, j_k, i'_k, j'_k	k^{th} state component
$\mathcal{X}_k$	domain of x_k/x'_k
$\mathcal{X}$	$\mathcal{X}_L \times \cdots \times \mathcal{X}_1$, potential state space
$\mathcal{Y}$	set of states/tuples
$\mathcal{S}$	reachable state space
$\mathcal{S}_0/\mathcal{M}_0$	set of initial states/markings
$\mathcal{T}, \mathcal{Z}$	a binary relation/transition relation
$\mathcal{E}$	set of events/Petri net transitions
$\mathcal{P}/\mathcal{A}/\mathcal{I}$	set of Petri net places/arcs/inhibitor arcs
$\mathbf{R}$	transition rate matrix
$spt(f)$	a set of variables in function f's support, f can be a set or relation
$top(f)$	the maximum variable in $spt(f)$ according to $\succ$

Figure 2.1: Symbols.

Symbol	**Definition or Meaning**
$\mathbf{0},\mathbf{1}/\Omega$	MDD/EVMDD terminal node
$p.var$	the variable to which p is associated
$p[i]$	i^{th} edge of node p
$\langle\omega,r\rangle$	edge with value ω and destination r
$p[\alpha]$	when α is a tuple, extends edge notation to paths
$p[\alpha].v/.d$	edge or path's value/destination
$f_p/f_{\langle\omega,r\rangle}$	the function node encoded by node p/edge $\langle\omega,r\rangle$
$\mathcal{B}(p)$	the set of tuples node encoded by p
$enc(f)$	the node/edge encoding f, f can be a set or relation

Figure 2.2: Symbols (continued).

$\mathcal{X} = \mathcal{X}_L \times \cdots \times \mathcal{X}_1$ is the *potential* state space. $\mathcal{S}_0 \subset \mathcal{X}$ is a finite set of *initial* states, $\mathcal{E}$ is a set of *asynchronous* events, and, for each event $e \in \mathcal{E}$, $\mathcal{T}_e$ is a transition relation, i.e., a binary relation defined on $(\mathbf{x}, \mathbf{x}')$, where unprimed $\mathbf{x}$ corresponds to the "from" state and primed $\mathbf{x}' = (x'_L, \cdots, x'_1) \in \mathbb{N}^L$ corresponds to the "to" state, such that $(\mathbf{i}, \mathbf{i}') \in \mathcal{T}_e$ if the model can move from $\mathbf{i}$ to $\mathbf{i}'$ in one step when e occurs. The overall transition relation is defined as $\mathcal{T} = \bigcup_{e \in \mathcal{E}} \mathcal{T}_e$. We refer to the choice of these variables and their order as a *decomposition* of the model.

When $\mathcal{Z}$ is a transition relation for an event, we assume it is given as a conjunction of $C + 1$ *sub-relations*:

$$\mathcal{Z} \equiv \left(\bowtie_{c=1}^{C} \mathcal{Z}_c \right) \bowtie \mathcal{Z}^{eq}, \tag{2.1}$$

where $\bowtie$ is the natural join operator. Each $\mathcal{Z}_c$ depends on or updates a set of state variables, namely variables in its *support*, denoted as $spt(\mathcal{Z}_c)$, defined so that, if $x'_k \in spt(\mathcal{Z}_c)$, then $x_k \in spt(\mathcal{Z}_c)$ as well. Let $spt(\mathcal{Z}) = \bigcup_{c=1}^{C} spt(\mathcal{Z}_c)$ and $top(\mathcal{Z})$ be the maximum variable in $spt(\mathcal{Z})$. The unchanged variables are instead captured by the sub-relation $\mathcal{Z}^{eq} \equiv \bowtie_{x'_k \notin spt(\mathcal{Z})} [x'_k = x_k]$. Thus, we can have $x'_k \notin spt(\mathcal{Z})$ and $x_k \in spt(\mathcal{Z})$, but not $x'_k \in spt(\mathcal{Z})$ and $x_k \notin spt(\mathcal{Z})$. Then, $top(\mathcal{Z})$ is always an unprimed variable.

For example, if $\mathbf{x} = (x_3, x_2, x_1)$ and $\mathcal{Z}$ is defined by the boolean expression $\mathcal{Z} \equiv [x'_3 = x_3 + 1 \wedge x'_2 = x_1 - 2]$, this transition relation can be written as the natural join of the three sub-relations $\mathcal{Z}_1 \equiv [x'_3 = x_3 + 1]$, $\mathcal{Z}_2 \equiv [x'_2 = x_1 - 2]$, and $\mathcal{Z}^{eq} \equiv [x'_1 = x_1]$. Then, $spt(\mathcal{Z}_1) = \{x_3, x'_3\}$, $spt(\mathcal{Z}_2) = \{x_2, x'_2, x_1\}$, and $spt(\mathcal{Z}) = \{x_3, x'_3, x_2, x'_2, x_1\}$, thus $top(\mathcal{Z}) = x_3$.

Stochastic specification We also study the case where the discrete-state system is a

CTMC where each event can fire after a exponentially distributed delay with a *constant* or *deterministic* state-dependent rate, i.e., the rate function f_e of event e only depends on the *current* state, and is further broken into a product of C *sub-functions*:

$$f_e \equiv \prod_{c=1}^{C} f_e^c, \tag{2.2}$$

where each f_e^c depends on a set of state variables, namely variables in its *support*, denoted as $spt(f_e^c)$, and $spt(f_e) = \bigcup_{c=1}^{C} spt(f_e^c)$. Event e may bring the model to a new state chosen from a set of states according to a probability function φ_e, which also depends on a set of state variables and can be broken in a similar style. The difference between f and φ is that f only depends on the current state (or "from" state, denoted with unprimed symbols) while φ also depends on the next state (or "to" state, denoted with primed symbols).

Then, the transition rate *due to* event e is, for any two states $\mathbf{i}, \mathbf{i}'$,

$$\mathbf{R}_e[\mathbf{i}, \mathbf{i}'] \equiv \begin{cases} f_e(\mathbf{i}) \cdot \varphi_e(\mathbf{i}, \mathbf{i}') & \text{if } (\mathbf{i}, \mathbf{i}') \in \mathcal{T}_e, \\ 0 & \text{otherwise,} \end{cases} \tag{2.3}$$

where, although we wrote $f_e(\mathbf{i})$, only the values of variables in $spt(f_e)$ for state $\mathbf{i}$ are needed, similarly for φ_e. We stress that, for the encoding we consider, it might be inefficient to require $f_e(\mathbf{i}) = 0$ if e is disabled in $\mathbf{i}$, but this is not a problem since we can use $\mathcal{T}_e$ to "filter" the value of f_e. Finally, the overall transition rate $\mathbf{R} = \sum_{e \in \mathcal{E}} \mathbf{R}_e$.

2.2 Petri nets

Petri nets [53] are a popular formalism for specifying concurrent and distributed systems. In this section, we cover the class of Petri nets used in this thesis.

2.2.1 Generalized stochastic Petri nets

We use generalized stochastic Petri nets (GSPNs) [4], extended to allow variable-arc cardinality arcs, inhibitor arcs, and transition guards, as the high-level formalism to illustrate our work.

Definition 2.2.1. GSPN [4] is a finite directed, bipartite graphs described by a tuple of the form $(\mathcal{P}, \mathcal{E}, \mathcal{A}, \mathcal{I}, \mathcal{M}_0, w, g, \lambda)$, where

- $\mathcal{P}$ and $\mathcal{E}$ are sets of *places* and *transitions* satisfying $\mathcal{P} \cap \mathcal{E} = \emptyset$ and $\mathcal{P} \cup \mathcal{E} \neq \emptyset$. A *marking* $\mu \in \mathbb{N}^{\mathcal{P}}$ assigns a number of *tokens* μ_p to each place $p \in \mathcal{P}$.
- $\mathcal{A} \subseteq \mathcal{P} \times \mathcal{E} \cup \mathcal{E} \times \mathcal{P}$ is a set of directed arcs which connect places to transitions (*input arcs*) and transitions to places (*output arcs*).
- $\mathcal{I} \subseteq \mathcal{P} \times \mathcal{E}$ is a set of *inhibitor arcs.*
- $\mathcal{M}_0 \subset \mathbb{N}^{\mathcal{P}}$ specifies a finite set of *initial markings.*
- $w : \mathcal{A} \times \mathbb{N}^{\mathcal{P}} \to \mathbb{N}^{+} \cup \{0\}$ *marking-dependent cardinality* [14] for each arc.
- $g : \mathcal{E} \times \mathbb{N}^{\mathcal{P}} \to \mathbb{B}$ to describe the transition guards.
- $\lambda : \mathcal{E} \times \mathbb{N}^{\mathcal{P}} \to \mathbb{R}^{\geq 0}$ or $\lambda \equiv \infty$ to describe the transition rates. $\lambda \equiv \infty$ corresponds to an *immediate* transition. $\lambda : \mathcal{E} \times \mathbb{N}^{\mathcal{P}} \to \mathbb{R}^{\geq 0}$ corresponds to a *timed* transition with a *marking-dependent* rate; $\lambda = 0$ *disables* the transition.

Note that, transitions can be immediate, but *events* must not, in order to make those computations depending on **R** valid, which means somehow we need to “merge” those immediate transitions to timed transitions. When we only deal with logic aspects of the net, we usually neglect λ.

2.2.2 Evolution of a GSPN

Given a marking $\mu \in \mathbb{N}^{\mathcal{P}}$, it is convenient to define the *input flow matrix* of the net as $D^- : \mathbb{N}^{\mathcal{P}\times\mathcal{E}}$, where, for $p \in \mathcal{P}$ and $t \in \mathcal{E}$, $D^-[p,t] = w(p,t)$, if $(p,t) \in \mathcal{A}$, and 0 otherwise. The *inhibitor matrix* as $D^I : \mathbb{N}^{\mathcal{P}\times\mathcal{E}}$, where for $D^I[p,t] = w(p,t)$, if $(p,t) \in \mathcal{I}$, and ∞ otherwise. If we treat the marking μ of the net as a (column) vector, we can then say that a transition $t \in \mathcal{E}$ is *enabled* in μ if $\mu \geq D^-[\cdot,t]$, $\mu < D^I[\cdot,t]$, $g(t) = 1$ and $\lambda(t) \neq 0$. Let $\mathcal{E}(\mu) \subseteq \mathcal{E}$ be the set of marking-enabled transitions in μ.

Analogously, we can define the *output flow matrix* as $D^+ : \mathbb{N}^{\mathcal{P}\times\mathcal{E}}$, where, for $p \in \mathcal{P}$ and $t \in \mathcal{E}$, $D^+[p,t] = w(t,p)$, if $(t,p) \in \mathcal{A}$, and 0 otherwise. Then, the *incidence matrix* of the net is given by $D = D^+ - D^-$, and the vector $D[\cdot,t]$ represents the net effect on the marking when transition t fires. More precisely, the *firing* of transition $t \in \mathcal{E}(\mu)$ in marking μ changes the marking to $\mu' = \mu + D[\cdot,t]$, we write this as $\mu\ [t\rangle\!\!\Rightarrow \mu'$. Keep in mind that the cardinality of any arc, as well as D^-, D^I, D^+, g and λ is evaluated in the current marking, i.e., prior to the firing of any transition.

2.2.3 Running example

The GSPN of Fig. 2.3 models a simple fork-and-join queuing network with two customers (the two tokens initially in place a), and will be our running example. The black transition is immediate, and fires as soon as it is enabled. The white transitions have an exponentially distributed firing time, with the rates listed in the figure, where "$\#(z)$" means the number of tokens in place z. We can see that t_1 is enabled, and after it fires, the GSPN moves from marking $a^2b^0c^0d^0e^0$ to marking $a^1b^1c^1d^0e^0$, where a^2 means place a has 2 tokens.

We consider the following decompositions for this model, $L = 4$, $\{x_4 \equiv [\#(d), \#(e)], x_3 \equiv [\#(c)], x_2 \equiv [\#(b)], x_1 \equiv [\#(a)]\}$. Fig. 2.4 shows, on the left, the underlying CTMC state graph for our running model. The order of the values for each x_k is as given on the right side. For example, d^0e^0, meaning both place e and place f are empty, has index 0 in $\mathcal{X}_4$, thus state 0111 then corresponds to marking $a^1b^1c^1d^0e^0$.

This running example will be used for this chapter and some later chapters.

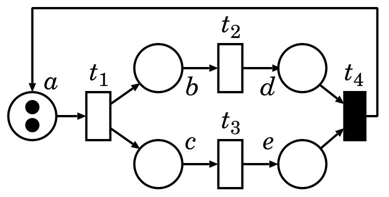
2.3 Decision diagrams

Since the introduction of reduced ordered binary decision diagrams (BDDs) [11], further variations of decision diagrams have been proposed in the literature to compactly encode and efficiently compute functions on structured sets. We now outline the two types of decision diagrams used in our work.

2.3.1 Multiway decision diagrams

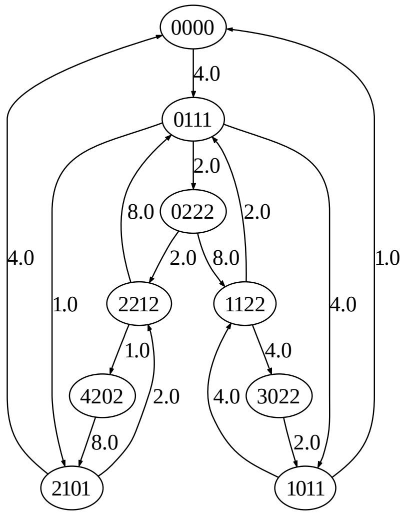
Given L variables $\mathbf{x} = (x_L, ..., x_1)$ with an order $x_L \succ \cdots \succ x_1$, each x_k taking value in a finite set $\mathcal{X}_k = \{0, 1, ..., n_k\} \subset \mathbb{N}$, a (quasi-reduced) *multi-way decision diagram (MDD)* [42] defined on $\mathbf{x}$ is a directed acyclic edge-labeled multi-graph where:

- Each nonterminal node s is associated to a variable $s.var = x_k \in \{x_L, ..., x_1\}$.



trans.	**rate**
t_1	$2.0 \cdot \#(a)$
t_2	$1.0 \cdot \#(b)$
t_3	$4.0 \cdot \#(c)$
t_4	immediate

Figure 2.3: Running example: the GSPN model of our fork-and-join system.



$\mathcal{X}_4 = \{d^0e^0, d^0e^1, d^1e^0, d^0e^2, d^2e^0\} = \{0, ..., 4\}$

$\mathcal{X}_3 = \{c^0, c^1, c^2\} = \{0, 1, 2\}$

$\mathcal{X}_2 = \{b^0, b^1, b^2\} = \{0, 1, 2\}$

$\mathcal{X}_1 = \{a^2, a^1, a^0\} = \{0, 1, 2\}$

Figure 2.4: Running example: CTMC state graph.

- **0** and **1** are the *terminal* nodes. Let $\mathbf{0}.var = \mathbf{1}.var = x_0$ and $x_k \succ x_0$, for $L \geq k \geq 1$.

- Each nonterminal node p associated with x_k has $|\mathcal{X}_k|$ edges, labeled with a different index $i \in \mathcal{X}_k$ and pointing to a node q with $q.var = v_{k-1}$ or $q = \mathbf{0}$, we write $p[i] = q$. At least one edge does not point to **0**.

- *Duplicate* nodes are not allowed, i.e., given two distinct nonterminal nodes p and q with $p.var = q.var = x_k$, there must be an index $i \in \mathcal{X}_k$ such that $p[i] \neq q[i]$.

- There is a single *root* node associated to x_L, with no incoming edges, so that we can identify the MDD with its unique root.

Each node p, with $p.var = x_k$, encodes a set of tuples: $\mathcal{B}(\mathbf{0}) = \emptyset$, $\mathcal{B}(\mathbf{1}) = \{()\}$, the set containing only the empty tuple, and, for $L \geq k \geq 1$, $\mathcal{B}(s) = \bigcup_{i=0}^{n_k} \{i\} \times \mathcal{B}(p[i])$. $\mathcal{B}$ is mnemonic for "below". These MDDs are canonical [22, 42]: given a set $\mathcal{Y} \subseteq \mathcal{X}_L \times \cdots \times \mathcal{X}_1$, there is a unique MDD $p = enc(\mathcal{Y})$ satisfying $\mathcal{Y} = \mathcal{B}(p)$ and $p.var = x_L$ (except for $\mathcal{Y} = \emptyset$, where $p = \mathbf{0}$, thus $p.var = x_0$).

We can extend the edge notation to paths, so that the node reached from node p associated to x_k through a tuple $\alpha = (i_k, i_{k-1}, ..., i_h) \in \mathcal{X}_k \times ... \times \mathcal{X}_h$, for $L \geq k \geq h \geq 1$, is defined recursively as

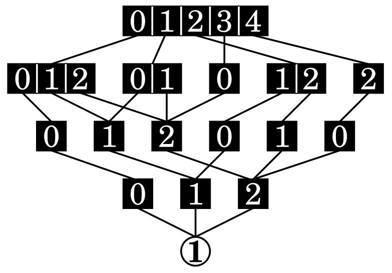
$$p[\alpha] = \begin{cases} p[i_k][i_{k-1}, ..., i_h] & \text{if } p[i_k] \neq \mathbf{0}, \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

Each node p associated to x_k encodes a set of tuples $\mathcal{B}(p) = \{\alpha \in \mathcal{X}_k \times ... \times \mathcal{X}_1 : p[\alpha] = \mathbf{1}\}$, Fig. 2.5 shows the 4-variable MDDs encoding the state space for our fork-and-join model. Only paths leading to node **1** are shown. The order of the values for each x_k is as given on the right side.

2.3.2 Edge-valued decision diagrams

A (quasi-reduced) *additive edge-valued multi-way decision diagram* (EV$^+$MDD) [25] defined on $\mathbf{x}$ is a directed acyclic edge-labeled multi-graph where:

- Each nonterminal node p is associated to a variable $p.var = x_k \in \{x_L, ..., x_1\}$.
- Ω is the only *terminal* node. Let $\Omega.var = x_0$ and $x_k \succ x_0$, for $L \geq k \geq 1$.
- Each nonterminal node s associated with x_k has $|\mathcal{X}_k|$ edges, labeled with a different index $i \in \mathcal{X}_k$ and associated to a value $\mathbb{N} \cup \{\infty\}$. We write $p[i] = \langle \omega, q \rangle = \langle p[i].v, p[i].d \rangle$ if the edge labeled by i points to node q and is associated to ω, and require that $q = \Omega$ if $\omega = \infty$, and $q.var = x_{k-1}$ otherwise. Also, at least one edge leaving each node must have an associated value equal to 0.



$$
\begin{aligned}
\mathcal{X}_4 &= \{d^0e^0, d^0e^1, d^1e^0, d^0e^2, d^2e^0\} &&= \{0, ..., 4\} \\
\mathcal{X}_3 &= \{c^0, c^1, c^2\} &&= \{0, 1, 2\} \\
\mathcal{X}_2 &= \{b^0, b^1, b^2\} &&= \{0, 1, 2\} \\
\mathcal{X}_1 &= \{a^2, a^1, a^0\} &&= \{0, 1, 2\}
\end{aligned}
$$

Figure 2.5: Running example: the MDD encoding $\mathcal{S}$.

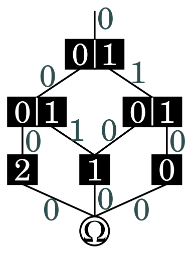
- There is a single *root* node associated to x_L with an incoming *dangling* edge. If the root node is r and the dangling edge associates with value ω, then the EV$^+$MDD can be denoted by $\langle\omega,r\rangle$.

- *Duplicate* nodes are not allowed: given two distinct nonterminal nodes p and q with $p.var = q.var = x_k$, there must be an index $i \in \mathcal{X}_k$ such that $p[i] \neq q[i]$.

The function encoded by edge $\langle\omega,p\rangle$, with $p.var = x_k$, is recursively defined by

$$\forall(i_k,...,i_1)\in\mathcal{X}_k\times\cdots\times\mathcal{X}_1,\ f_{\langle\omega,p\rangle}(i_k,...,i_1)=\begin{cases}\omega+f_{p[i_k]}(i_{k-1},...,i_1) & \text{if } \omega\in\mathbb{N}\\ \infty & \text{if } \omega=\infty.\end{cases}$$

EV$^+$MDDs are canonical [25]. Given a function $g : \mathcal{X}_L \times \cdots \times \mathcal{X}_1 \rightarrow \mathbb{N}\cup\{\infty\}$, there is a unique value $\rho\in\mathbb{N}$ and a unique node r with $r.var=x_L$ such that $f_{\langle\omega,r\rangle}=g$, or $\langle\omega,r\rangle = enc(g)$, with the exception of the constant function $g \equiv \infty$, for which we have the special case $\omega = \infty$ and $r = \Omega$. Fig. 2.6 show the EV$^+$MDD encoding the function f on the right side, omitting edges value with ∞.



x_3	0	0	1	1
x_2	0	1	0	1
x_1	2	1	1	0
f	0	1	1	0

Figure 2.6: The EV*MDD (left) encoding f (right).

```
$mdd\ Or(mdd\ s, mdd\ u)$                                   • assume $s.var = x_k$
    1 if $s = \mathbf{0}$ or $s = u$ then return $u$;
    2 if $u = \mathbf{0}$ then return $s$;                            • trivial cases
    3 if $CacheHit(OR, s, u, r)$ then return $r$;                     • check cache
    4 $r \leftarrow NewNode(x_k)$;                  • create a new node associated to $x_k$
    5 foreach $i \in \mathcal{X}_k$ do $r[i] \leftarrow Or(p[i], u[i])$;      • recursively invoke on children
    6 $CacheAdd(OR, s, u, UTInsert(r))$;                             • add to cache
    7 return $r$;
```

```
$mdd\ And(mdd\ s, mdd\ u)$                                  • assume $s.var = x_k$
    1 if $s = u$ or $u = \mathbf{0}$ then return $u$;
    2 if $s = \mathbf{0}$ then return $s$;
    3 if $CacheHit(AND, s, u, r)$ then return $r$;
    4 $r \leftarrow NewNode(x_k)$;
    5 foreach $i \in \mathcal{X}_k$ do $r[i] \leftarrow And(s[i], u[i])$;
    6 $CacheAdd(AND, s, u, UTInsert(r))$;
    7 return $r$;
```

```
$mdd\ Difference(mdd\ s, mdd\ u)$                           • assume $s.var = x_k$
    1 if $s = \mathbf{0}$ or $s = u$ then return $\mathbf{0}$;
    2 if $u = \mathbf{0}$ then return $s$;
    3 if $CacheHit(DIF, s, u, r)$ then return $r$;
    4 $r \leftarrow NewNode(x_k)$;
    5 foreach $i \in \mathcal{X}_k$ do $r[i] \leftarrow Difference(s[i], u[i])$;
    6 $CacheAdd(DIF, s, u, UTInsert(r))$;
    7 return $r$;
```

Figure 2.7: Standard MDD operations.

```
idd Minimum(idd ⟨$\alpha$,$s$⟩, ⟨$\beta$,$u$⟩)                         • assume $s.var = x_k$
 1 if $\alpha = \infty$ then return ⟨$\beta$,$u$⟩;
 2 if $\beta = \infty$ then return ⟨$\alpha$,$s$⟩;
 3 $\gamma \leftarrow \min\{\alpha, \beta\}$;         • since $\alpha = \min f_{\langle\alpha,s\rangle}, \beta = \min f_{\langle\beta,u\rangle}$
 4 if $s = u$ then return ⟨$\gamma$,$s$⟩;             • $\min(f_{\langle\alpha,s\rangle}, f_{\langle\beta,s\rangle}) = f_{\langle\min(\alpha,\beta),s\rangle}$
 5 if $\alpha < \beta$ then $Swap(\langle\alpha,s\rangle, \langle\beta,u\rangle)$                 • commutativity
 6 if $CacheHit(MIN, \alpha - \beta, s, u, r)$ then return ⟨$\gamma$,$r$⟩;
 7 $r \leftarrow NewNode(x_k)$;
 8 foreach $i \in \mathcal{X}_k$ do
 9   $\alpha' \leftarrow \alpha - \gamma + s[i].val$;
10   $\beta' \leftarrow u[i].val$;
11   $r[i] \leftarrow Minimum(\langle\alpha', s[i]\rangle, \langle\beta', u[i]\rangle)$;
12 $CacheAdd(MIN, \alpha - \beta, s, u, UTInsert(r))$;
13 return ⟨$\gamma$,$r$⟩;
```

Figure 2.8: EV$^+$MDD *Minimal* operation.

2.3.3 Standard operations for decision diagrams

We now list standard operations including Union/*Or*, Intersection/*And* and *Difference* for MDDs in Fig. 2.7 and *Minimum* for EV$^+$MDDs in Fig. 2.8 In the pseudocode, we use type *mdd* for MDD node and type *idd* for integer-valued EV$^+$MDD edge. We omit discussing implementation details here; they can be found in next chapter, where we discuss generally-reduce decision diagrams and Ch. 8 where we discuss the Decision Diagram Library.

2.4 Symbolic reachability analysis

We now briefly review state-of-the-art symbolic state-space generation techniques to produce the MDD encoding the state space $\mathcal{S}$. We assume the model is described using a high-level formalism such as Petri nets.

Let $Img(\mathcal{Y}, \mathcal{Z}) = \{\mathbf{i}' : \exists \mathbf{i} \in \mathcal{Y}, (\mathbf{i}, \mathbf{i}') \in \mathcal{Z}\}$ denote the *image* of the set of states $\mathcal{Y}$ under the transition relation $\mathcal{Z}$. Then, $\mathcal{S}$ is the minimal set satisfying $\mathcal{S} \supseteq \mathcal{S}_0$ and $\mathcal{S} \supseteq Img(\mathcal{S}, \mathcal{T})$. If $\mathcal{T}$ can be computed a priori, we can simply start from $\mathcal{S}_0$ and repeatedly perform image computations under $\mathcal{T}$ until reaching a fixpoint; otherwise, we need to generate $\mathcal{T}$ *on-the-fly* alongside $\mathcal{S}$ [21,28,62]. As $\mathcal{T} = \bigcup_{e \in \mathcal{E}} \mathcal{T}_e$, we can use the transition relations $\mathcal{T}_e$ instead of $\mathcal{T}$ for state-space generation, as long as each $\mathcal{T}_e$ is applied often enough to avoid "missing" any possible transition.

We know that any set of states $\mathcal{Y}$ can be encoded into an MDD $p = enc(\mathcal{Y})$ defined on $(x_L, ..., x_1)$ with the order $x_L \succ \cdots \succ x_1$. Similarly, any transition relation $\mathcal{Z}$ can be encoded into an MDD $r = enc(\mathcal{Z})$ defined on $(x_L, x'_L, ..., x_1, x'_1)$ with the order $x_L \succ x'_L \succ \cdots \succ x_1 \succ x'_1$, where the unprimed variables refer to the "from" states and the primed variables refer to the "to" states, such that, if we let $\mathbf{i} = (i_L, ..., i_1)$ and $\mathbf{i}' = (i'_L, ..., i'_1)$, then $(\mathbf{i}, \mathbf{i}') \in \mathcal{Z} \Leftrightarrow (i_L, i'_L, ..., i_1, i'_1) \in \mathcal{B}(r)$. We choose an *interleaved* order for the variables in the MDD for $\mathcal{Z}$ because it usually results in a compact encoding and allows an efficient computation [28], but we nevertheless let $(\mathbf{i}, \mathbf{i}')$ denote the path tuple $(i_L, i'_L, ..., i_1, i'_1)$ and any (unprimed,primed) tuple-variable pair, e.g., (α, α'), with $\alpha = (i_l, ..., i_h)$ and $\alpha' = (i'_l, ..., i'_h)$, denote the interleaving $(i_l, i'_l, ..., i_h, i'_h)$ of these two tuples.

The image computation $Img(\mathcal{Y}, \mathcal{Z})$ corresponds to a *RelationalProduct* operation or one of its variants [28] on MDD p and MDD r. The simplest symbolic state-space generation uses a breadth-first strategy [54] and performs an image computation at each iteration, until reaching a global fixpoint. This method can be improved through chaining [58], to compound the effect of asynchronous events within a given breadth-first iteration. The saturation algorithm [19] uses instead a completely different iteration strategy to compute a fixpoint for each node in ascending order with respect to its associated variable, and has been shown to excel when applied to asynchronous systems.

The saturation algorithm was first defined for transition relations that admit a Kronecker encoding. While this encoding always exists, it might be unwieldy, thus the approach was not fully general in practice. [51] avoids the Kronecker encoding by partitioning the transition relation of each event into groups, encoded by matrix diagrams [49], while [28] allows an even finer partition for the transition relations, where conjuncts can share state variables, which results in a more efficient computation.

Among the variants of saturation mentioned above, [19] has restrictions on the decomposition of the model and is adopted by the approximate numerical solution in [52], while [28, 51] allow an arbitrary decomposition of the model. Empirically, a finer decomposition using more state variables can be much more efficient both time- and memory-wise than [19], which may be forced to use a coarser Kronecker decomposition.

Detailed algorithms for breadth-first and saturation-based state-space generation are listed in Sec. 4.1.3.

Chapter 3

Generally-reduced Decision Diagrams

Implicit encodings have also been used in real-valued contexts. For example, the tool PRISM [44] uses *multi-terminal BDDs (MTBDDs)* [37], also known as *algebraic decision diagrams* [6], to store the transition rate matrix **R** of a CTMC. However, most stochastic modeling tools [16,31] for CTMC analysis employ techniques based on the *Kronecker product* operator [32], or, more recently, on *matrix diagrams* [23, 49], which combine the idea of decision diagrams with that of Kronecker encoding. One property essential to the efficiency of Kronecker encodings but lacking in BDDs and MTBDDs is the ability to exploit *identity transformations* for a subset of the L state variables when a transition occurs, i.e., the fact that an event may affect only a small fraction of the state variables and leave the other variables unchanged. This phenomenon is extremely common in asynchronous models, such as those arising from modeling distributed software systems or from formalisms such as Petri nets [53]. To our knowledge, this issue has been addressed only recently outside the Kronecker or matrix-diagram framework, by considering a disjunctive partition of $\mathcal{T}$ [8], or $\mathbf{R}$ [45], and encoding an event e with a BDD, or MTBDD, that ignores both variables x_k and $x_{k'}$, if state variable x_k neither is modified by event e, nor it affects its occurrence or effect.

[19,21] exploited identity transformations in the *Saturation* algorithm, which can be used for state-space generation and CTL model checking [26] and improves both peak memory and time requirements by several orders of magnitude with respect to traditional symbolic breadth-first iterations. The tool SMART [17] implements this algorithm, and also offers symbolic encodings for the analysis of Markov models (although, in this case, the enormous reductions in memory are not accompanied by reductions in execution time, unless the user is willing to accept an approximate numerical solution [24,52]).

This chapter presents a new canonical form of multi-terminal decision diagrams where each variable of the decision diagram can be reduced, independently of the others, according to a variable-specific *reduction rule*. In addition to the traditional *fully-reduced* rule [11], which skips a node if all its children are identical, and to the less-well-known *quasi-reduced* rule [43], which never skips a node, possible reduction rules include the new *c-reduced* rule, which skips a node if only its c-child is nonzero, for some fixed but variable-specific $c \in \mathbb{N}$ (this generalizes the *zero-suppressed* rule [48], which assumes $c = 0$) and the new *identity-reduced* rule, which skips a node along an i-edge if only its i-child is nonzero (this generalizes the reduction employed in matrix diagrams [23,49]). An obvious and important application of this last reduction rule is the storage of (Boolean or real) transition matrices, where, for the k^{th} state component, the new state i'_k coincides with the old state i_k, even just some of the times, and these two variables correspond to contiguous variables in the decision diagram.

For our presentation, we use multi-way decision diagrams, as these are more natural when modeling non-Boolean systems. Furthermore, decision diagrams are normally used to encode functions of the form $f : \mathcal{X} \to \mathcal{X}_0$, where the *domain* $\mathcal{X}$ has an L-dimensional structure $\mathcal{X} = \mathcal{X}_L \times \cdots \times \mathcal{X}_1$ and, for $L \geq k \geq 1$, the *local domain* $\mathcal{X}_k$ is a set of size $n_k \geq 1$ of the form $\{0, 1, \ldots, n_k - 1\}$. For example, in BDDs, $\mathcal{X}_k = \mathcal{X}_0 = \mathbb{B}$, the Boolean set $\{0, 1\}$. However, in applications such as state-space generation, some local domains $\mathcal{X}_k$ might not be known a priori, rather, they might be discovered during the fixpoint computations that manipulate the decision diagrams [21, 28, 62]. Here, then, we allow variables to take values over the natural numbers $\mathbb{N}$, and put constraints on the decision diagrams that, while ensuring that the decision diagram is finite, allow us to encode some (obviously not all) functions over structured countably infinite domains. In our approach, the *range* $\mathcal{X}_0$ is an arbitrary set simply required to contain a distinguished, or *default*, element δ; this is needed to define the c-reduced and identity-reduced cases, where δ plays the role of the value 0 for traditional sparse arrays or identity matrices, respectively. δ usually also acts as an “absorbing” element in TDD operations which is very important for efficiency; more details can be found in Sec. 3.2.3.

This chapter is organized as follows. Sec. 3.1 introduces our new class of decision diagrams and discusses their canonicity. Sec. 3.2 presents algorithms to canonize and manipulate them. Sec. 3.3 discusses their application to the storage of sets and relations, or, in greater generality, vectors and matrices, when $\mathcal{X}_0$ is not Boolean. Finally, Sec. 3.4 concludes the chapter outlining some directions for further research.

3.1 Terminal-valued decision diagrams

3.1.1 Definition of TDDs

Definition 3.1.1. Given L *domain* variables $x_L, ..., x_1$, where each x_k takes values over $\mathcal{X}_k = \{0, ..., n_k - 1\}$, $n_k \geq 1$, or $\mathcal{X}_k = \mathbb{N}$, given a *range* variable x_0 taking values over an arbitrary range set $\mathcal{X}_0$ with a *default* element $\delta \in \mathcal{X}_0$, and given the *variable order* $x_L \succ \cdots \succ x_1 \succ x_0$, a *finite ordered multi-way terminal-valued decision diagram*, TDD for short, is a finite acyclic edge-labeled directed multi-graph where:

- Each node p is associated to a variable x_k, $L \geq k \geq 0$, we write $p.var = x_k$.
- If $p.var = x_0$, the node is *terminal*, i.e., has no outgoing edges, otherwise it is *nonterminal*, i.e., it has outgoing edges.
- The terminal nodes are identified with elements of $\mathcal{X}_0$, i.e., if p is a terminal node, then $p \in \mathcal{X}_0$.
- A nonterminal node p associated with variable x_k has an outgoing edge labeled with each different index $i_k \in \mathcal{X}_k$, pointing to node q with $p.var \succ q.var$, we write $p[i_k] = q$.
- If nonterminal node p is associated with variable x_k and $\mathcal{X}_k = \mathbb{N}$, there must be a node q such that $\{i_k \in \mathbb{N} : p[i_k] \neq q\}$ is finite.

Symbol	**Definition or Meaning**
$\mathcal{X}_k$	domain of x_k, either $\{0, ..., n_k - 1\}$ or $\mathbb{N}$
$\mathcal{X}_0$	an arbitrary range set that x_0 can take
δ	the default value of $\mathcal{X}_0$, or the terminal node labeled δ
$p[*]$	the common destination for infinite number of edges of p, when $\mathcal{X}_k = \mathbb{N}$
$\boldsymbol{\rho}$	reduction rule vector, $\boldsymbol{\rho} : \{L, ..., 1\} \to \{Q, F, I\} \cup \mathbb{N}$
$\rho(x_k)$ or ρ_k	reduction rule associated to variable x_k
$\boldsymbol{\rho}^Q$	the reduction rule vector where all entries are Q
$p[\mathcal{X}_k] = q$	redundant node p, $p.var = x_k$ and all edges pointing to q
singular-c	a singular node where only the edge labeled c does not point to δ
$\mathcal{A}_{r,p}$	the set of tuples leading from node r to node p

Figure 3.1: New symbols and notations in this chapter.

We call a (terminal or nonterminal) node with no incoming edges a *root*. Since we assume that the number of roots is finite, the finiteness of the overall TDD is then guaranteed by the last item above. This is because, whenever nonterminal node p is associated with variable x_k and $\mathcal{X}_k = \mathbb{N}$, we can represent its infinite set of outgoing edges with a nonempty finite set of edges: one edge labeled by "$*$", so that $p[*] = q$ describes the common destination node q for an infinite number of edges, and zero or more additional edges of the form $p[i_k] = r$, with $i_k \in \mathbb{N}$ and $r \neq q$; if $j_k \in \mathbb{N}$ is not explicitly listed in these additional edges, it means that $p[j_k] = p[*] = q$.

Given node p associated with variable x_k and an $i_k \in \mathcal{X}_k$, we say that an edge

$p[i_k] = q$ *skips* variable x_h if $x_k \succ x_h \succ q.var$. In a TDD where no edge skips variables, a node p associated to x_k encodes a function f_p recursively defined as, $\forall i_k, ..., i_1 \in \mathcal{X}_k \times \cdots \times \mathcal{X}_1$,

$$f_p(i_k, ..., i_1) = \begin{cases} p & \text{if } k = 0, \\ f_{p[i_k]}(i_{k-1}, ..., i_1) & \text{otherwise.} \end{cases}$$

However, if some edges skip variables, the function encoded by a node p associated to x_k can be defined only after we decide an interpretation for these edges. We consider the following three possibilities.

Fully-reduced: An edge $p[i_k] = q$ with $x_k \succ x_h \succ q.var$ is equivalent to an edge $p[i_k] = r$, where r is a node with $r.var = x_h$ and $r[i_h] = q$ for each $i_h \in \mathcal{X}_h$.

Identity-reduced: An edge $p[i_k] = q$ with $x_k \succ x_h \succ q.var$ is equivalent to an edge $p[i_k] = r$, where r is a node with $r[i_k] = q$ and $r[i_h] = \delta$ for each $i_h \in \mathcal{X}_h \setminus \{i_k\}$ (of course, we must have $i_k \in \mathcal{X}_h$).

c-**reduced**: An edge $p[i_k] = q$ with $x_k \succ x_h \succ q.var$ is equivalent to an edge $p[i_k] = r$, where r is a node with $r[c] = q$ and $r[i_h] = \delta$ for each $i_h \in \mathcal{X}_h \setminus \{c\}$ (of course, we must have $c \in \mathcal{X}_h$).

Then, unlike previous definitions of canonical decision diagrams where the same reduction rule is applied to all nodes, we define a reduction rule vector $\boldsymbol{\rho} : \{L, ..., 1\} \to \{Q, F, I\} \cup \mathbb{N}$, so that, if $\rho_k = Q$, no edge is allowed to skip variable x_k unless it is pointing to δ, if $\rho_k = F$, an edge skipping variable x_k follows the fully-reduced interpretation, if $\rho_k = I$, an edge skipping variable x_k follows the identity-reduced interpretation, and if $\rho_k = c \in \mathcal{X}_k$, an edge skipping variable x_k follows the c-reduced interpretation. We use

$\rho_0 \equiv Q$ to denote the reduction rule associated to x_0, since x_0 can never be skipped. Given such a reduction rule vector $\boldsymbol{\rho}$, the function encoded by a node p associated with variable x_l with respect to a variable $x_n \succeq x_l$ is then recursively defined as, $\forall i_n, ..., i_1 \in \mathcal{X}_n \times \cdots \times \mathcal{X}_1$,

$$f_p(i_n, ..., i_1) = \begin{cases} p & \text{if } p = \delta \text{ or } n = l = 0 \text{ (cases below assume } p \neq \delta), \\ \text{undefined} & \text{if } n > l, \text{ and } \rho_n = I \text{ or } \exists m, n \geq m > l \text{ s.t. } \rho_m = Q, \\ f_{p,i_n}(i_{n-1}, ..., i_1) & \text{if } n > l, \text{ and } \rho_n = F \text{ or } \rho_n = i_n \ , \\ f_{q,i_n}(i_{n-1}, ..., i_1) & \text{if } n = l > 0 \text{ and } p[i_l] = q, \\ \delta & \text{otherwise,} \end{cases} \tag{3.1}$$

where

$$f_{p,j}(i_n, ..., i_1) = \begin{cases} \delta & \text{if } p = \delta \text{ (cases below assume } p \neq \delta), \\ f_p(i_n, ..., i_1) & \text{if } \rho_n \neq I, \text{ or } n = l \\ \text{undefined} & \text{if } \rho_n = I, \ n > l \text{ and } j \notin \mathcal{X}_n, \\ f_{p,j}(i_{n-1}, ..., i_1) & \text{if } \rho_n = I, \ n > l \text{ and } i_n = j, \\ \delta & \text{otherwise.} \end{cases} \tag{3.2}$$

The undefined cases should never happen in the actual TDDs and needs to be forbidden. This definition makes a node p associate to x_l meaningful with respect to x_n when $l < n$, as long as $\rho_n \neq I$, by treating variables from x_n to x_{l+1} as skipped, even if there is no edge leading to p. Finally, we can define a *canonical* form of TDDs.

Definition 3.1.2. A TDD is *generally-reduced* according to the reduction rule vector $\boldsymbol{\rho} : \{L, ..., 1\} \to \{Q, F, I\} \cup \mathbb{N}$, with $\rho_L \neq I$, if:

1. There are no *duplicate* nonterminal nodes: $p.var = q.var = x_k$, with $k > 0$, and $\forall i_k \in \mathcal{X}_k, p[i_k] = q[i_k]$ imply $p = q$.

2. There are no δ-*valued* nonterminal nodes: $p.var = x_k$, with $k > 0$, implies $\exists i_k \in \mathcal{X}_k, p[i_k] \neq \delta$.

3. If $\rho_k = F$, there is no node p associated with x_k such that $\forall i_k \in \mathcal{X}_k, p[i_k] = q$, for some node q; such node p is called a *redundant* node and also denoted as $p[\mathcal{X}_k] = q$.

4. If $\rho_k = Q$, only edges to node δ can skip variable x_k.

5. If $\rho_k = I$, we define node q to be a *singular* node, or, more specifically, a singular-i_q node if it has exactly one edge $q[i_q] \neq \delta$; no edge $p[i]$ can point to q if $i = i_q$ or if the edge skips a variable x_l with $\rho(x_l) = F$ and $i_q \in \mathcal{X}_l$.

6. If $\rho_k = c \in \mathcal{X}_k$, there is no singular-$c$ node p with $p.var = x_k$.

Without causing confusion, we sometimes use *reduced* for short, instead of generally-reduced in this thesis.

Fig. 3.2 shows different TDD with different $\boldsymbol{\rho}$, listed on the left side, all encode the function $f(000) = f(110) = f(220) = 1$ and $f = \delta$ for other inputs, assuming $\mathcal{X}_3 = \mathcal{X}_2 = \mathcal{X}_1 = \{0, 1, 2\}$. Edges pointing to δ and δ itself are omitted.

3.1.2 Canonicity of reduced TDDs

Theorem 3.1.1. *Given a function $g : \mathcal{X}_L \times \cdots \times \mathcal{X}_1 \to \mathcal{X}_0$, a default element $\delta \in \mathcal{X}_0$, and a reduction vector $\boldsymbol{\rho} : \{L, ..., 1\} \to \{Q, F, I\} \cup \mathbb{N}$ with $\rho_L \neq I$, there is a unique reduced TDD node p satisfying $f_p = g$.*

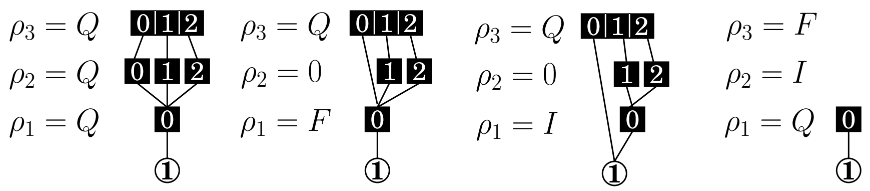


Figure 3.2: TDDs with different $\boldsymbol{\rho}$, encode the same function.

Proof. The proof for the existence of a TDD p such that $f_p = g$ will be given in Sec. 3.2. Now we prove the uniqueness of the encoding, i.e., $f_p = f_q \Rightarrow p = q$.

The proof is based on Eq. 3.1, Eq. 3.2 and Def. 3.1.2. Let $p.var = x_{l_p}$ and $q.var = x_{l_q}$, $L \geq l_p, l_q \geq 0$ and without loss of generality, assume $l_p \geq l_q$.

For the basis of the induction, when $L = 1$, if $l_p > l_q$,

- If $\rho_1 = F$, then $g = f_q \equiv q$, and $\forall i \in \mathcal{X}_1$, $g(i) = f_p(i) = p[i] = q$, so p contradicts Def. 3.1.2 : item3.
- If $\rho_1 = c \in \mathcal{X}_1$, then $f_p(c) = f_q(c) = q$, and $\forall i \in \mathcal{X}_1 \setminus \{c\}$, $f_p(i) = f_q(i) = \delta$, so p contradicts Def. 3.1.2 : item6.

So $l_p = l_q$, and there are two cases:

- If $l_p = l_q = 0$ then:
  - If $\rho_1 = F$, then $p \equiv g \equiv q$;
  - If $\rho_1 = c \in \mathcal{X}_1$, then $g(c) = f_p(c) = p = f_q(c) = q$.
- If $l_p = l_q = 1$, then $\forall i \in \mathcal{X}_1$, $f_p(i) = p[i] = f_q(i) = q[i]$, which implies $p = q$ according to Def. 3.1.2 : item1.

For the inductive step, assume the theorem holds when $L \leq n$ for some $n \geq 1$ and, then show it holds when $L = n + 1$. Consider two cases:

Case 1. If $l_p < n{+}1$, according to Eq. 3.1, $\forall i_{n+1}, i_n, ..., i_1 \in \mathcal{X}_{n+1} \times \cdots \times \mathcal{X}_1$, $f_p(i_{n+1}, i_n, ..., i_1) = f_{p,i_{n+1}}(i_n, ..., i_1)$ and $f_q(i_{n+1}, i_n, ..., i_1) = f_{q,i_{n+1}}(i_n, ..., i_1)$ when $\rho_{n+1} = F$ or $\rho_{n+1} = i_{n+1}$, or δ otherwise.

- If $\rho_n \neq I$, then $f_p = f_q \Rightarrow f_p(i_n, ..., i_1) = f_q(i_n, ..., i_1)$; since the theorem holds when $L = n$, the above statement implies $p = q$.

- If $\rho_n = I$, consider two cases:

  – If $\rho_{n+1} = c \in \mathcal{X}_{n+1}$, we introduce a new reduction rule vector $\boldsymbol{\rho}'$ of size n, where $\rho'_n = Q$ and $\forall l, n > l \geq 1$, $\rho'_l = \rho_l$.

    * If $l_p = l_q = n$, let f'_r be the function node r encodes with $\boldsymbol{\rho}'$, then $\forall i_n, ..., i_1 \in \mathcal{X}_n \times \cdots \times \mathcal{X}_1, f'_p(i_n, ..., i_1) = f_p(i_n, ..., i_1) = g(c, i_n, ..., i_1) = f_q(i_n, ..., i_1) = f'_q(i_n, ..., i_1)$. Thus $p = q$ since theorem holds when $L = n$.

    * If $n > l_p$, then we construct new singular-c nodes p' and q', with $p'.var = q'.var = x_n$, and $p'[c] = p, q'[c] = q$ then when $i_n \neq c$, $f'_{p'}(i_n, ..., i_1) = \delta = f'_{q'}(i_n, ..., i_1)$; when $i_n = c$, $f'_{p'}(i_n, ..., i_1) = f'_{p,c}(i_{n-1}, ..., i_1) = f_{p,c}(i_{n-1}, ..., i_1) = f_{q,c}(i_{n-1}, ..., i - 1) = f'_{q,c}(i_{n-1}, ..., i_1) = f'_{q'}(i_n, ..., i_1)$. So $p' = q'$, which implies $p = q$.

    * if $n = l_p$ and $l_p > l_q$, then we construct q' as above, and similarly $f'_p = f'_{q'}$, so $p = q'$, which contradicts Def. 3.1.2 : item5 with $\boldsymbol{\rho}$.

  So $p = q$ in this case.

– If $\rho_{n+1} = F$, let $h = \max\{l : l \leq n, \rho_l \neq I \text{ or } l = l_p\}$, then $f_{p,i_{n+1}}(i_n, ..., i_1) = f_p(i_h, ..., i_1)$, when $\forall k, n \geq k \geq h, i_k = i_{n+1}$, or δ otherwise.

  * If $h > l_p$, or $h = l_p$ and $\rho_h \neq I$, then $f_{q,i_{n+1}}(i_n, ..., i_1) = f_q(i_h, ..., i_1)$, when $\forall k, n \geq k \geq h, i_k = i_{n+1}$, so $f_p = f_q \Rightarrow \forall i_h, ..., i_1 \in \mathcal{X}_h \times \cdots \times \mathcal{X}_1, f_p(i_h, ..., i_1) = f_q(i_h, ..., i_1)$. Thus $p = q$ since the theorem holds when $L = h$.

  * If $h = l_p = l_q$ and $\rho_h = I$, consider a new reduction rule vector $\boldsymbol{\rho'}$ of size h, where $\rho'_h = Q$ and $\forall l, h > l \geq 1$, $\rho'_l = \rho_l$. Let f'_r be the function node r encodes with $\boldsymbol{\rho'}$, then $\forall i_h, ..., i_1 \in \mathcal{X}_h \times \cdots \times \mathcal{X}_1, f'_p(i_h, ..., i_1) = f_p(i_h, ..., i_1) = f_q(i_h, ..., i_1) = f'_q(i_h, ..., i_1)$, so $p = q$.

  * If $h = l_p > l_q$ and $\rho_h = I$, then p must not be a singular node according to Def. 3.1.2 : item5, so p has at least two outgoing edges not pointing to δ. Then let $i_{n+1} = i_n = ... = i_{h+1} \neq i_h$ and $p[i_h] \neq \delta$, $\exists i_{h-1}, ..., i_1 \in \mathcal{X}_{h-1} \times \cdots \times \mathcal{X}_1$, s.t. $f_p(i_{n+1}, ..., i_1) = f_p(i_h, ..., i_1) \neq \delta$, but $f_q(i_{n+1}, ..., i_1) \equiv \delta$ since $i_h \neq i_{h+1}$, we get a contradiction. So $l_p = l_q$, and we return to the above bullet item.

Case 2. If $l_p = n + 1$, first we prove $p = q$ when $l_p = l_q$.

For any $c \in \mathcal{X}_{n+1}$, consider a reduction rule vector $\boldsymbol{\rho'}$ of size $n + 1$, where $\rho'_{n+1} = c$ and $\forall l, n + 1 \geq l \geq 1$, $\rho'_l = \rho_l$, and let f'_r be the function node r encodes with $\boldsymbol{\rho'}$. Let $p' = p[c], q' = q[c]$, then $\forall i_n, ..., i_1 \in \mathcal{X}_n \times \cdots \times \mathcal{X}_1$, $f_p(c, i_n, ..., i_1) = f_{p',c}(i_n, ..., i_1)$ and $f_q(c, ..., i_1) = f_{q',c}(i_n, ..., i_1)$. Then $\forall i_{n+1} \in \mathcal{X}_{n+1}$, $f'_{p'}(i_{n+1}, ..., i_1) = f'_{p',c}(i_n, ..., i_1) = f_p(c, i_n, ..., i_1)$ when $i_{n+1} = c$ or δ otherwise; similarly $f'_{q'}(i_{n+1}, ..., i_1) = f_q(c, i_n, ..., i_1)$ when $i_{n+1} = c$ or δ otherwise; so $f'_{p'} = f'_{q'}$ which implies $p' = q'$. Since c is arbitrarily chosen, we have $\forall c \in \mathcal{X}_{n+1}$, $p[c] = q[c]$, which implies $p = q$.

Now prove $l_p = l_q$ is always held, otherwise, $\rho_{n+1} = F$ or $\rho_{n+1} = c \in \mathcal{X}_{n+1}$. Consider a new reduction rule vector $\boldsymbol{\rho}'$ of size $n+1$ where $\rho'_{n+1} = Q$ and $\rho'_l = \rho_l$ for $n \geq l \geq 1$, and let f'_r be the function node r encodes with $\boldsymbol{\rho}'$; obvious $f'_p = f_p$,

- If $\rho_{n+1} = F$, we construct a new redundant node q' with $q'.var = x_{n+1}$ and $q'[\mathcal{X}_{n+1}] = q$, then $f'_{q'} = f_q = f_p = f'_p$. Since $p.var = q'.var$, we have $p = q'$, then node p contradicts Def. 3.1.2 : item3 subject to $\boldsymbol{\rho}$.

- If $\rho_{n+1} = c \in \mathcal{X}_{n+1}$, we construct a new singular-c node q' with $q'.var = x_{n+1}$ and $q'[c] = q$, then similarly $f'_{q'} = f'_p \Rightarrow p = q'$. So node p contradicts Def. 3.1.2 : item6 with $\boldsymbol{\rho}$.

So we can conclude when $L = n+1$, $f_p = f_q \Rightarrow p = q$ still holds and thus the theorem is proved. $\square$

3.2 Operations with TDDs

TDDs offer a very compact encoding for many functions, but, to be of practical use, they must also possess efficient manipulation algorithms. We now discuss the most important such algorithms.

One type of TDD manipulations relates to canonization: given a function f, how to create the canonical TDD encoding it with a single root node. Sec. 3.2.1 discusses how to canonize a TDD according to a given reduction rule vector, while Sec. 3.2.2 discusses how to change the reduction rule of a given canonical TDD.

Other TDD operations take instead one or more TDD nodes in input and return a TDD node in output. Given the canonization routines just mentioned, these operations can safely assume that the TDD is correctly reduced according to a given reduction rule vector $\boldsymbol{\rho}$, and must simply guarantee that it remains so. Rather than consider specific operations, Sec. 3.2.3 presents a generic TDD operator, *Apply*, which can be used as the base of most TDD operations.

Our pseudocode uses the type *tdd* for a TDD node. All algorithms in this chapter are implemented and tested in the Decision Diagram Library (DDL), introduced in Ch. 8. We omitted some implementation details for succinctness, see Ch. 8 for additional information.

3.2.1 The *CheckIn* operation

Usually, given a reduction rule vector $\boldsymbol{\rho}$, after a TDD node is created, it might not be reduced according to $\boldsymbol{\rho}$, e.g., it could be a duplicate node, or a redundant node and the reduction rule associate to its variable is F. So a *unique table* and a check-in step is necessary here to ensure canonicity. Before check-in, a node is called a *temporal* node, and does not reside in the unique table; the check-in procedure reads in a temporal node and outputs a *canonical* node which resides in the unique table.

The pseudocode is shown in Fig. 3.3. The unique table insertion is done by *UTInsert* which is part of *CheckIn*. *CheckIn* is also responsible for discovering redundant nodes or singular nodes that should be eliminated according to Def. 3.1.2. The elimination of a node is handled through a reference counting scheme in DDL, which is automatically taken care of and not shown in the pseudocode. A subtle issue is how to eliminate the singular nodes, which contradict Def. 3.1.2, or, in short, *identity singular nodes.* since for this case we require the information of its *incoming* edge.

All nodes except for the terminal nodes need go through the *CheckIn* step to become canonical nodes (we assume terminal nodes are canonical); a canonical node should never have an edge pointing to a temporal node; before checking in a temporal node, all its children must be canonical nodes; and only a temporal node can modify its edges. These are important requirement for all decision-diagram-based algorithms to ensure correctness.

3.2.2 The *Translate* operation

Given a TDD node p reduced with a reduction rule vector $\boldsymbol{\rho}$, we might want to change the reduction rule vector to be $\boldsymbol{\rho}'$ with the same size, and output a TDD node p' such that $f_p = f'_{p'}$, where $f'_{p'}$ is the function p' encodes with $\boldsymbol{\rho}'$.

This "translation" operation is important in the sense that it allows a possible choice of a better reduction rule vector for a TDD, which requires fewer nodes to encode the same function; it also contributes to the constructive proof for the existence part of Thm. 3.1.1 given in Sec. 3.2.3.

The idea of this operation is first to transfer p into a temporal node q under $\boldsymbol{\rho}^Q$, the size-L vector that all entries are Q, so that $f_p = f_q^Q$. Then *CheckIn* all derivative nodes of q in a bottom-up fashion. Look at a simple example, let $p.var = k+1, r.var = k-1$ and $p[i] = r$ skips x_k for some $k, L \geq k \geq, i \in \mathcal{X}_{k+1}$, there are three possible cases:

- If $\rho_k = F$, we can create a redundant temporal node r', such that, $q[i] = r'$ and

```
tdd CheckIn(tdd p)                                          • assume $p.var = x_k$
1 if $\rho_{k-1} = I$ then
2   foreach $i \in \mathcal{X}_k$ s.t. $p[i] \neq \delta$ do          • check identity singular nodes
3     $ch \leftarrow p[i]$
4     if $ch[i] \neq \delta$ and $\forall j \in \mathcal{X}_k \setminus \{i\}$, $ch[j] = \delta$ then
5       $p[i] \leftarrow ch[i]$;
6 else if $\rho_k = F$ and $p[0] = p[1] = \cdots = p[n_k - 1]$ then
7   return $p[0]$;                                                  • redundant node
8 else if $\rho_k = c \in \mathcal{X}_k$, $p[c] \neq \delta$ and $\forall i \in \mathcal{X}_k \setminus \{c\}$, $p[i] = \delta$ then
9   return $p[c]$;                                                  • singular-$c$ node
10 else return $UTInsert(p)$;
```

Figure 3.3: The *CheckIn* algorithm.

$r'[\mathcal{X}_k] = r$.

- if $\rho_k = c \in \mathcal{X}_k$, we create a singular-$c$ temporal node r', such that $q[i] = r'$ and $r'[c] = r$.

- if $\rho_k = I$, we create a singular-i temporal node r', such that $q[i] = r'$ and $r'[i] = r$ ($i \in \mathcal{X}_k$ is required).

For more complicated TDD structures, basically we do the above recursively from top down, shown in Fig. 3.4.

Procedure *Translate*(p) takes node p reduced with $\boldsymbol{\rho}$ and produce p' reduced with $\boldsymbol{\rho}'$, such that $f_p = f'_{p'}$. It calls the recursive procedure *RecTranslate* to finish the task. *RestoreIdentity* handles the case an variable associated to I is skipped. *NewNode*(x_k) create a new temporal node associate to x_k, with all edges initialized pointing to δ. *Copy*(p) makes a temporal copy of a canonical node p.

```
$tdd\ Translate(tdd\ p)$                    • assume original reduction rule vector $\boldsymbol{\rho}$, new vector $\boldsymbol{\rho}'$
1 if $\boldsymbol{\rho} = \boldsymbol{\rho}'$ then return $p$;                                    • do nothing
2 else return $RecTranslate(L, p)$;
```

```
$tdd\ RecTranslate(integer\ n, tdd\ p)$                                    • $p.var = x_k$
1 if $n = 0$ return $p$;                                                     • terminal case
2 if $CacheHit(TRANSLATE, n, p, q)$ return $q$;
3 if $n = k$ then $q \leftarrow Copy(p)$;                                   • no skipping edge
4 else if $\rho_n = c \in \mathcal{X}_k$ then
5   $q \leftarrow NewNode(x_k)$;
6   $q[c] \leftarrow p$;                                             • restore singular-$c$ node
7 else                                                                    • $\rho_n = F$
8   $q \leftarrow NewNode(x_k)$;
9   foreach $i \in \mathcal{X}_k$
10    do $q[i] \leftarrow p$;                                        • restore redundant node
11 foreach $i \in \mathcal{X}_n$ s.t. $q[i] \neq \delta$ do
12   $RestoreIdentity(q, i, q[i])$;                         • restore identity singular nodes
13 $q \leftarrow CheckIn(q)$;                   • $CheckIn$ depends on the reduction rule vector $\boldsymbol{\rho}'$
14 $CacheAdd(TRANSLATE, n, p, q)$;
15 return $q$;
```

```
$RestoreIdentity(tdd\ p, integer\ i, tdd\ ch)$                         • $p.var = x_k$, $ch.var = x_l$
1 if $\rho_{k-1} = I$ and $k - 1 > l$ then                • $i_{k-1}$ is skipped with $I$ interpretation
2   $q \leftarrow NewNode(x_{k-1})$;
3   $q[i] \leftarrow ch$;                                            • restore singular-$i$ node
4   $RestoreIdentity(q, i, ch)$;
5   $p[i] \leftarrow q$;
6 else
7   $p[i] \leftarrow RecTranslate(k - 1, ch)$;          • no skipping or $\rho_{k-1} = F$ or $\rho_{k-1} = c$
```

Figure 3.4: Algorithm for changing from $\boldsymbol{\rho}$ to $\boldsymbol{\rho}'$.

An operation cache is adopted here and used throughout the thesis, which is essential for efficiency of most decision-diagram based algorithms. *CacheHit* search the tuple of the operator and the operand(s) as the search key in the cache to see if there is a hit and if so, return the result that was computed before. *CacheAdd* adds the tuple of the operator, the operand(s) and corresponding result to the cache. To ensure the validity of cache entries, we require that nodes as operands cannot be modified, i.e. canonical nodes, and given the operands, the result is unique. More details can be found in Ch. 8.

3.2.3 The *Apply* operation

Given a binary operation $\odot : \mathcal{X}_0 \times \mathcal{X}_0 \rightarrow \mathcal{X}_0$, we can define the result of this binary operation on the functions encoded by two reduced TDD nodes p and q, and generate a reduced TDD node r encoding the result, or $f_r = f_p \odot f_q$.

r can be computed recursively, e.g., assume $p.var = q.var = x_n$, and $\rho = \rho^Q$, then, for $i_n, ..., i_1 \in \mathcal{X}_n \times \cdots \times \mathcal{X}_1$,

$$\begin{aligned} f_{r[i_n]}(i_{n-1}, ..., i_1) &= f_r(i_n, ..., i_1) = f_p(x_n, ..., x_1) \odot f_q(x_n, ..., x_1) \\ &= f_{p[i_n]}(i_{n-1}, ..., i_1) \odot f_{q[i_n]}(i_{n-1}, ..., i_1) \end{aligned}$$

When there are skipping edges, we might need to restore nodes depending on the reduction rules.

Algorithm *Apply* in Fig. 3.5 and Fig. 3.6 shows how to compute node r. It assumes that the TDD is already in reduced form, and it leaves it in reduced form. The main difference from the traditional *Apply* of fully-reduced decision diagram is in the management of the c-reduced and identity-reduced variables. *RecApply*(p, q) assumes that either node

```
$tdd\ Apply(tdd\ p, tdd\ q)$
    return $Restore(0, p, q, L)$;
```

```
$tdd\ RecApply(tdd\ p, tdd\ q)$                                $\bullet\, p.var = x_k, q.var = x_l$
    if $k = 0$ and $l = 0$ then return $p \odot q$;                    $\bullet$ *terminal case*
    if $CacheHit(APPLY, p, q, r)$ then return $r$;
    $r \leftarrow NewNode(x_{\max(k,l)})$;
    if $k = l$ then foreach $i \in \mathcal{X}_k$ do
      $r[i] \leftarrow Restore(i, p[i], q[i], k)$;
    else if $k > l$ then
      if $\rho_k = F$ then foreach $i \in \mathcal{X}_k$ do
        $r[i] \leftarrow Restore(i, p[i], q, k)$;
      else if $\rho_k = c \in \mathcal{X}_k$ then
        $r[c] \leftarrow Restore(c, p[c], q, k)$;
        foreach $i \in \mathcal{X}_k \setminus \{c\}$ do
          $r[i] \leftarrow Restore(c, p[c], \delta, k)$;
    else                                                       $\bullet\, k < l$
      if $\rho_l = F$ then foreach $i \in \mathcal{X}_l$ do
        $r[i] \leftarrow Restore(i, p, q[i], k)$;
      else if $\rho_l = c \in \mathcal{X}_k$ then
        $r[c] \leftarrow Restore(c, p, q[c], k)$;
        foreach $i \in \mathcal{X}_k \setminus \{c\}$ do
          $r[i] \leftarrow Restore(c, \delta, q[c], k)$;
    $r \leftarrow CheckIn(r)$;
    $CacheAdd(APPLY, p, q, r)$;
    return $r$;
```

Figure 3.5: The *Apply* procedure for a generic operator $\odot$.

```
$tdd\ Restore(integer\ i, tdd\ p, tdd\ q, integer\ n)$                    • $p.var = x_k, q.var = x_l$
if $k = l$ or $\rho_{\max(k,l)} \neq I$ then return $RecApply(p, q)$;          • no need to restore
if $k > l$ then
  $m \leftarrow \min\{m : k < m < n, \rho_m \neq I$ or $m = n\}$
  if $m = n$ then                                                      • add a singular-$i$ node
    $r \leftarrow NewNode(x_k)$;
    $r[i] \leftarrow q$;
  else if $\rho_m = c \in \mathcal{X}_m$ then                                    • add a singular-$c$ node
    $r \leftarrow NewNode(x_k)$;
    $r[c] \leftarrow q$;
  else if $\rho_m = F$ then                            • add a redundant node associated to $x_m$
    $r \leftarrow NewNode(x_m)$;
    foreach $j \in \mathcal{X}_m$ do
      $r[j] \leftarrow q$;
  $r \leftarrow UTInsert(r)$;                                       • so $r$ can be used for cache
  return $RecApply(p, r)$
else                                                                              • $k < l$
  $m \leftarrow \min\{m : l < m < n, \rho_m \neq I$ or $m = n\}$
  if $m = n$ then
    $r \leftarrow NewNode(x_l)$;
    $r[i] \leftarrow p$;
  else if $\rho_m = c \in \mathcal{X}_m$ then
    $r \leftarrow NewNode(x_l)$;
    $r[c] \leftarrow p$;
  else if $\rho_m = F$ then
    $r \leftarrow NewNode(x_m)$;
    foreach $j \in \mathcal{X}_m$ do
      $r[j] \leftarrow p$;
  $r \leftarrow UTInsert(r)$;
  return $RecApply(r, q)$
```

Figure 3.6: The *Apply* procedure for a generic operator $\odot$ (continued).

p and q are associated to the same variable or that the higher one is not identity-reduced; this ensures $p \odot q$ will produce a unique result, not depending on its incoming edge, which is desired for the cache. When p and q do not satisfy this assumption, either during the recursion or at the beginning, it is then handled by procedure *Restore*.

Note that, to be fully general, Algorithm *Apply* does not assume anything about the operator $\odot$. Its efficiency can be improved, for example, if we know that $\odot$ is commutative, since the entry $(APPLY, q, p, r)$ in the cache is also a hit, or that $\delta \odot x = \delta$ (as is the case for multiplication, when $\delta = 0$), since the recursion can be stopped as soon as one of the operators is δ, or that $p \odot p = p$ (as is the case, $\mathcal{X}_0 = \mathbb{B}$ and $\odot$ is *Or* or *And*), etc. We will see some specific *Apply* operations in the following chapters for different applications.

We now have a brief discussion of *Apply*'s complexity. Compared to the traditional algorithm for fully-reduced decision diagrams, *Apply* has the possible additional cost of inserting redundant or singular nodes when some variables are identity-reduced, but these insertions are not truly overhead with respect to fully-reduced decision diagrams, since those nodes would be present in the fully-reduced case anyway. On the other hand, when two edges $p[i]$ and $q[i]$ both skipped a series of variables in a reduced TDD, the *Apply* operation could be much more efficient than the fully-reduced version since these variables are probably present in the fully-reduced version; We will see several examples in the following chapters.

Now we can give the proof for the first part of Thm. 3.1.1 by construction.

Given function $g : \mathcal{X}_L \times \cdots \times \mathcal{X}_1 \Rightarrow \mathcal{X}_0$, a default element δ, and a reduction rule vector $\boldsymbol{\rho}$, in order to prove there is a unique TDD node encoding g with $\boldsymbol{\rho}$, we only need to build a TDD node p such that $f_p = g$ with $\boldsymbol{\rho}^Q$, since we can always change the reduction rule vector to be $\boldsymbol{\rho}$ using *Translate*. For each $i_n, ..., i_1 \in \mathcal{X}_L \times \cdots \times \mathcal{X}_1$ s.t. $g(i_n, ..., i_1) \neq \delta$, build a TDD node q, such that there is only one path from q to a non-δ terminal : $q[i_n][i_{n-1}]...[i_1] = g(i_n, ..., i_1)$. We define the operator $\odot$ such that

$$a \odot b = \begin{cases} a & \text{if } a \neq \delta, \\ b & \text{otherwise ,} \end{cases}$$

which is like a generalized logical *or* ($\vee$) operation. Then let $p = \delta$ initially and apply $p = p \odot q$ for each of those nodes q, we get the desired p finally.

3.3 Applications

We present two sets of experimental results to illustrate the usefulness of our identity-reduced rule. The first set considers the encoding of binary relations and matrices, which naturally possess a concept of identity. The second set considers a perhaps more surprising application, the encoding of sets and vectors.

3.3.1 Encoding relations and matrices

We study the number of nodes needed to encode the transition relation $\mathcal{T}$ or rate matrix $\mathbf{R}$ for several largely asynchronous models, using TDDs on the domain $\mathcal{X}_L^2 \times \cdots \times \mathcal{X}_1^2$. For $\mathcal{T}$, the range $\mathcal{X}_0$ contains only 0 and 1 while, for $\mathbf{R}$, it contains the unique non-negative values in the matrix $\mathbf{R}$. In both cases, $\delta = 0$. We present results for both a *disjunctive* encoding, where $\mathcal{T}$ (resp. $\mathbf{R}$) is encoded as a Boolean (resp. real) sum over a set of events $\mathcal{E}$, each encoded as a TDD, and a *monolithic* encoding, where $\mathcal{T}$ is encoded using a single root TDD. All models have an integer parameter N; we obtained consistent results across a wide range but, for lack of space, we show results for just two values of N. A brief description of each model follows.

Philosophers is the classic dining philosopher problem. A deadlock occurs if all N philosophers pick their right, or their left, fork.

Leader is the randomized leader election protocol of [33].

Mutex is a mutual exclusion protocol for a set of N processes arranged in a circular fashion, where access is granted in a round-robin fashion [38].

FMS models a flexible-manufacturing system where parts move around in a factory, to be processed by various machines, assembled, and shipped [27].

Queuing models a bounded queuing network [35].

Tab. 3.1 reports the size of the state space $\mathcal{S}$, whether $\mathcal{T}$ or $\mathbf{R}$ has a Kronecker representation for the given partition of each model, and the number of nodes needed to encode $\mathcal{T}$ or $\mathbf{R}$ with a disjunctive or monolithic representation. The label X-Y in the eight rightmost columns describes the reduction rules used for each pair of variables x_l and $x_{l'}$.

Several observations can be made. First of all, a comparison of the Q-Q and F-F columns reveals that, in all cases, *there are no redundant nodes.* Comparing the Q-I and F-I columns, instead, shows that, with the exception of the FMS model, *applying the identity-reduced rule to primed variables makes some, possibly many, unprimed nodes redundant.* Most importantly, though, we can see by comparing the Q-Q and Q-I, or F-F and F-I, columns that *the identity-reduced rule results in substantial savings by eliminating* *a large number of singular nodes.* In particular, the combination F-I is quite efficient and consistently the best choice, as it takes advantage of the numerous "identity patterns" (a redundant node p associated to variable x_k with $|\mathcal{X}_k|$ children associated to variable x'_k, where $p[i_k] = q_{i_k}$ and $q_{i_k}[i_k]$ is the only edge not pointing to 0, as shown in Fig. 3.2 on the right), which are skipped altogether, thus do not increase the memory requirements. This is most visible in the Philosophers and Mutex models, where each event affects at most two submodels (each variable corresponds to a submodel) out of L, and L is large.

3.3.2 Encoding sets and vectors

Effectively employing the identity-reduced rule when encoding a set or vector, as opposed to a relation or matrix, requires intuition about how many singular nodes we can save by adopting the identity-reduced for a variable x_l, i.e., which i_l is likely to be equal to i_{l+1} in most, or all, cases.

An interesting possibility requiring further investigation is the use of the identity-reduced rule as a way to *discover* such dependencies. We motivate this idea with an artificial experiment where we randomly generate five sets of 1,000 16-bit vectors with uniform probability for all bits, except $Pr\{x_7 = x_8\} = Pr\{x_6 = x_7\} = Pr\{x_5 = x_6\} = 0.9$, and encode them in a 16-variable TDD. The results reported in Tab. 3.2 show the number of nodes required to encode these random sets; for column $X \cdot Y$, the reduction rule X is used for all variables except (x_7, x_6, x_5) where we use rule Y. While decision diagrams are most effective with highly structured sets, this experiment shows that the appropriate reduction rule saves a substantial fraction of nodes even in relatively unstructured data. The challenge is then

N	$\|\mathcal{S}\|$	**Kr**	**Disjunctive encoding**				**Monolithic encoding**			
			Q-Q	*F-F*	*Q-I*	*F-I*	*Q-Q*	*F-F*	*Q-I*	*F-I*
Number of TDD nodes required to encode $\mathcal{T}$										
Philosophers $L=\lceil N/2\rceil$ $\|\mathcal{S}_l\|=34$ for all l $\|\mathcal{E}\|=3L$										
200	$2.5 \cdot 10^{125}$	yes	527,948	527,948	22,134	7,187	14,008	14,008	5,578	5,283
300	$1.2 \cdot 10^{188}$	yes	1,185,723	1,185,723	44,459	10,787	21,108	21,108	8,378	7,933
Leader $L=2N+1$ $\|\mathcal{S}_l\|\leq 16$ for all l $\|\mathcal{E}\|=5N+2$										
30	$1.6\cdot 10^{18}$	no	34,074	34,074	3,979	938	4,846	4,846	1,290	929
50	$1.9\cdot 10^{31}$	no	89,914	89,914	9,639	1,558	8,086	8,086	2,150	1,549
Mutex $L=N+1$ $\|\mathcal{S}_l\|=7$ for all l except $\|\mathcal{S}_L\|=N+1$ $\|\mathcal{E}\|=5N$										
50	$1.3 \cdot 10^{17}$	yes	57,447	57,447	7,224	950	20,974	20,974	3,198	651
150	$4.8 \cdot 10^{47}$	yes	517,347	517,347	59,174	2,850	182,974	182,974	24,598	1,951
Number of TDD nodes required to encode R										
FMS $L = 16$ $\|\mathcal{X}_l\| = N+1$ $\|\mathcal{E}\| = 17$ $\|\mathcal{X}_0\| = 15$										
2	$1.8 \cdot 10^{3}$	no	2,189	2,189	1,006	1,006	1,547	1,547	860	860
3	$2.1 \cdot 10^{5}$	no	4,639	4,639	1,980	1,980	3,612	3,612	1,959	1,959
Queuing $L=8$ $\|\mathcal{X}_l\|=N+1$ for all $l\geq 1$ $\|\mathcal{E}\| = 8$ $\|\mathcal{X}_0\|=2N$										
30	$2.4 \cdot 10^{8}$	yes	2,920	2,920	693	627	2,256	2,256	552	499
50	$4.6 \cdot 10^{9}$	yes	5,798	5,798	1,113	1,027	4,714	4,714	892	819

Table 3.1: TDD nodes to encode a transition relation or a rate matrix.

	$\rho_7 = \rho_6 = \rho_5$			
	$Q{\cdot}Q$	$F{\cdot}F$	$Q{\cdot}I$	$F{\cdot}I$
set 1	1636	1617	1003	988
set 2	1639	1622	1011	996
set 3	1622	1608	995	980
set 4	1634	1618	995	980
set 5	1634	1618	1010	997

Table 3.2: Encoding five random sets of 1,000 tuples in $\{0, 1\}^{16}$.

to recognize partial dependencies of the type present in our artificial data set.

3.3.3 Discussion

It is easy to compare the advantages and disadvantages of the quasi- and fully-reduced rules. Changing from $\rho_k = F$ to $\rho_k = Q$ can only increase the number of nodes associated to variable x_k, while leaving the other nodes unchanged. However, the quasi-reduced rule might still be a good choice. First of all, there are cases where one might want to access all nodes associated to a given variable x_k, even redundant ones [18]. With a fully-reduced decision diagram, this requires finding all edges that skip variable x_k, thus accessing all nodes associated to higher variables, not just those associated to x_k as in the quasi-reduced case. Second, the recursive manipulation algorithms for quasi-reduced decision diagrams are much easier, since they always operate on nodes associated to the same variable; this also allows us to structure the operation cache by and avoid storing variable information in them. Third, node storage requires less memory because the variable that the node stores does not have to be explicitly stored in the node itself, since any recursive algorithm is aware of the variable at which it is operating, and because, given that an edge from a node p with $p.var = x_l$ can only point to a node q with $q.var = x_{l-1}$, we need in principle fewer bits to store the address of q (and in practice, if we use integer indices instead of actual pointers). Finally, perhaps the main reason is that, often, few redundant nodes are present in the TDDs needed to study a model, as our experimental results show.

More interesting is the comparison of the identity- and fully-reduced rules. Consider a node r with $r.var = x_L$ reaching a redundant node p with $p.var = x_l$, with $\rho_l = Q$ (p would then be eliminated if ρ_l were changed to F). Let the set of tuples leading from r to p be $\mathcal{A}_{r,p} \subseteq \mathcal{X}_L \times \cdots \times \mathcal{X}_{l+1}$. Then, we can see that a redundant node is the manifestation of a form of *independence*:

$$\forall \alpha \in \mathcal{A}_{r,p}, \forall i_l, j_l \in \mathcal{X}_l, \forall \beta \in \mathcal{X}_{l-1} \times \cdots \times \mathcal{X}_1, f_r(\alpha, i_l, \beta) = f_r(\alpha, j_l, \beta).$$

In other words, given any tuple α leading to p, the value of the function is independent of the value of the l^{th} argument.

In the scenario where p is a singular-c node and every tuple in $\mathcal{A}_{r,p}$ has the last component equal to c (p would then be eliminated if ρ_l were changed to I), p is instead the manifestation of a form of *dependence*:

$$\forall \alpha \in \mathcal{A}_{r,p}, \forall i_l \in \mathcal{X}_l \setminus \{c\}, \exists \beta \in \mathcal{X}_{l-1} \times \cdots \times \mathcal{X}_1, f_r(\alpha, c, \beta) \neq f_r(\alpha, i_l, \beta) = \delta.$$

Both independence and dependence can be present to some extent in a system, even within a single variable x_l. In this case, the choice between using the fully- or the identity-reduced rule could be based on which one saves the most nodes. In the extreme case, all the nodes associated to x_l are redundant (any function encoded by the TDD is then completely independent of the l^{th} argument, which can be eliminated from the TDD altogether), or all the nodes can be identity-reduced (any function g encoded by the TDD evaluates to δ whenever $i_l \neq i_{l+1}$, thus, again, variable x_l can be eliminated from the TDD).

Independence is a simpler concept, since it is a property that a TDD node, or even an entire variable, can possess in an absolute sense, while dependence is always relative to the higher variable, thus even testing for it is harder. This is probably the reason why the fully-reduced rule was introduced early on, while the identity-reduced rule has not been proposed before. Nevertheless, it is clear that this new rule can be extremely effective, especially when describing transition matrices, as we have illustrated.

3.4 Conclusion

We presented a new canonical form of decision diagrams where different variables can be reduced according to different reduction rules. Our novel *identity-reduced* rule is particularly effective in the presence of identity transformations, when encoding relations or matrices, and of dependencies, when encoding sets or vectors. We also presented algorithms for the manipulation of this new data structure and provided several numerical examples of its effectiveness.

Future directions of research along these lines include

- Explore more uses for the c-reduced rule. c-reduction is a generalized 0-reduction as used in zero-suppressed decision diagrams [48]. In general, c-reduction would be a good candidate if system behavior is quite different when some certain variables take one special value c, from when they take other values, which probably results in many singular-c nodes.

- Explore other uses of the identity-reduced rule, especially when storing sets or vectors, such as automatically identifying and quantifying dependencies.

- Explore the automatic discovery of good reduction rule vectors, periodically monitor the TDD and update our choice of reduction vector automatically and efficiently.

Chapter 4

Symbolic State-space Generation of Asynchronous Systems

The simplest symbolic state-space generation uses a breadth-first strategy [54] and performs an image computation at each iteration, until reaching a global fixpoint. This method can be improved through chaining [58], to compound the effect of asynchronous events within a given breadth-first iteration. The saturation algorithm [22] uses instead a completely different iteration strategy to compute a fixpoint for each decision diagram node in ascending order and has been shown to excel when applied to asynchronous systems. The algorithm was first defined for transition relations that admit a Kronecker encoding. While this encoding always exists, it might be unwieldy, thus the approach was not always practical. [51] avoids the Kronecker encoding by conjunctively partitioning the transition relation of each event into groups, encoded by matrix diagrams with an identity reduction. [28] allows an even finer partition, where conjuncts can share state variables, resulting in a more efficient computation for the symbolic encoding of the transition relation. Fully-reduced decision diagrams encode each conjunct and fully-identity-reduced decision diagrams (similar to identity-reduced matrix diagrams) encode the transition relation of each event.

When computing the fixpoint for a node associated to variable x_k, saturation applies the union of all events whose root nodes are associated to x_k. If the state variables have known bounds, the symbolic encoding for the transition relation of each event, as well as arbitrary unions among them, can be computed a priori, and the decision diagram structures chosen by [28, 51] work well. If the bounds on the state variables are unknown, however, we need to update the transition relation of any event affected by or affecting x_k when the domain of x_k is extended during state-space generation, resulting in an on-the-fly discovery process. Then, the domain of skipped variables can affect the result of a union operation, thus operation caches might be invalidated. This inefficiency comes from the reduction rules chosen for the decision diagrams, which produce a compact encoding but cause semantic errors when the variable domain is growing if the old cached values were used.

To tackle this problem, we utilize the new generally-reduced decision diagrams from Ch. 3, which allow an infinite variable domain. Using appropriate reduction rules and symbolic operation algorithms, this new canonical representation never requires to invalidate the operation cache, allows more node reuse, and still produces a compact symbolic encoding. As it allows efficient bookkeeping of the transition relation for events and their union, the data structure works well for both breadth-first and saturation.

We provide revised breadth-first and saturation algorithms, as well as a framework to update variable domains on-the-fly and symbolic operations for these new decision diagrams. Experimental results support our contribution.

This chapter is organized as follows. In Sec. 4.1, we define the symbolic data structure for encoding set of states and transition relations. Sec. 4.2 discusses the framework for symbolic on-the-fly state-space generation. Sec. 4.3 introduces how we apply the new generally-reduced decision diagram to on-the-fly state-space generation. Sec. 4.4 presents experimental results and Sec. 4.5 concludes this chapter.

4.1 Symbolic encoding

In this chapter, we use generally-reduced multi-way decision diagrams (MDDs) to encode set of states and transition relations. MDD here is defined as TDD with $\mathcal{X}_0 = \{\mathbf{0}, \mathbf{1}\}$ and $\delta = \mathbf{0}$. It is an enhanced version of the previous MDD [42], but we use the same abbreviation since they share a lot of properties and they can be used interchangeably when the domain of variable is finite. When both types of MDDs are present in the context, we call the latter traditional MDDs. We discuss three reduction rules in this chapter, F, Q and I. For this chapter, when we talk about skipping variables, edges pointing to $\mathbf{0}$ do not count, and without causing confusion, an MDD p is short for an MDD node p.

4.1.1 Symbolic encoding of sets of states

We use an MDD p for a given $\boldsymbol{\rho}$ with $p.var = x_k$ to encode a set of tuples $\mathcal{B}(p) \in \mathcal{X}_k \times \cdots \times \mathcal{X}_1$, such that, let $\alpha = i_k, ..., i_1$, then $\alpha \in \mathcal{B}_p \Leftrightarrow f_p(\alpha) = \mathbf{1}$, or $\mathcal{B}(p) = \mathcal{A}_{p,\mathbf{1}}$.

More specifically, according to Eq. 3.1 and Eq. 3.2, let $\mathcal{B}(\mathbf{0}) = \emptyset$, $\mathcal{B}(\mathbf{1}) = \{()\}$, the

Symbol	Definition or Meaning
$\boldsymbol{\rho}^{F}$	the reduction rule vector that all entries are F
$\boldsymbol{\rho}^{FI}$	$\rho(x_k) = F$ and $\rho(x'_k) = I$, for $L \geq k \geq 1$
$\boldsymbol{\rho}^{QF}$	(w.r.t. $\mathcal{Z}$, same below) $\rho(x) = Q$ if $x \in spt(\mathcal{Z})$, and $\rho(x) = F$ otherwise
$\boldsymbol{\rho}^{QFI}$	$\rho(x'_k) = I$, $\rho(x_k) = Q$ if $x_k \in spt(\mathcal{Z})$, and $\rho(x_k) = F$ otherwise, for $L \geq k \geq 1$
p_f	$\max\{i : p[i] \neq p[*]\}$

Figure 4.1: New symbols and notations in this chapter.

set containing only the empty tuple, then $\mathcal{B}(p)$ is defined by

$$\mathcal{B}(p) = \textstyle\bigcup_{i=0}^{n_{x_k}} \ \{i\} \times \mathcal{B}(x_{k-1}, i, p[i]),$$

where, $\forall x_l, \forall q$, with $q.var = x_h$ and $x_l \succeq x_h$,

$$\mathcal{B}(x_l, i, q) = \begin{cases} \mathcal{B}(q) & \text{if } x_l = x_h \\ \bigcup_{j=0}^{n_{x_l}} \{j\} \times \mathcal{B}(x_{l-1}, j, q) & \text{if } x_l \succ x_h \wedge \rho_l = F \\ \{i\} \times \mathcal{B}(x_{l-1}, i, q) & \text{if } x_l \succ x_h \wedge \rho_l = I. \end{cases}$$

Fig. 4.2 shows three MDDs on (x_3, x_2, x_1), with $\mathcal{X}_3 = \mathcal{X}_2 = \mathcal{X}_1 = \{0, 1, 2\}$, encoding the same set $\{0, 1, 2\} \times \{0\} \times \{0\}$ using different choices for $\boldsymbol{\rho}$. For clarity, we omit edges pointing to $\mathbf{0}$, as well as the terminal $\mathbf{0}$.

Our application needs to store sets of states, thus L-tuples. We do so using MDDs over $\mathbf{x} = (x_L, ..., x_1)$ and $\boldsymbol{\rho} = \boldsymbol{\rho}^{Q}$.

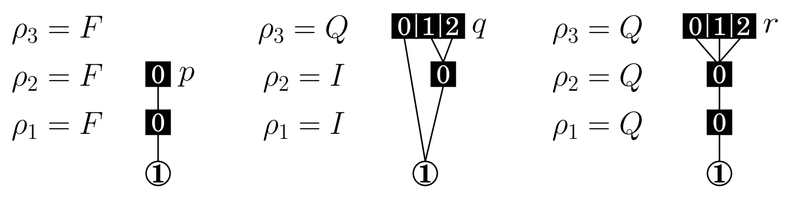


Figure 4.2: MDDs with different reductions encoding the same set

4.1.2 Symbolic encoding of transition relations

Given a transition relation $\mathcal{Z}$, we can use a $2L$-variable MDD on $(x_L, x'_L, ..., x_1, x'_1)$ with the order $x_L \succ x'_L \succ \cdots \succ x_1 \succ x'_1$, to store it, where the unprimed variables refer to the "from" states and the primed variables refer to the "to" states.

If we use a reduction rule vector $\boldsymbol{\rho}^{FI}$, where $\rho(x_k) = F$ and $\rho(x'_k) = I$, for $L \geq k \geq 1$, then $\mathcal{Z}$ is encoded into an MDD where every node is associated with a variable in $spt(\mathcal{Z})$. For sub-relations $\mathcal{Z}_c$, instead, we use a reduction rule vector $\boldsymbol{\rho}^F$, where all entries are F, then each $\mathcal{Z}_c$ is encoded into an MDD where, again, its nodes can only be associated with a variable in $spt(\mathcal{Z}_c)$. Assuming all variables take values in $\{0, 1, 2\}$, Fig. 4.3 shows $p = enc(\mathcal{Z}_1)$ and $q = enc(\mathcal{Z}_2)$ subject to ρ^F and $r = enc(\mathcal{Z})$ subject to $\boldsymbol{\rho}^{FI}$. The rightmost MDD, also subject to $\boldsymbol{\rho}^{FI}$, is incorrect, demonstrating why the identity-reduced rule does not allow edges from nodes associated to x'_3 to skip over x_2, which is fully-reduced. Without this requirement, the two rightmost MDDs would both encode $\mathcal{Z}$, thus canonicity would be lost. Using MDD subject to $\boldsymbol{\rho}^{FI}$ affords an enormous advantage: neither storing $\mathcal{Z}^{eq}$ nor performing a natural join with it is needed. The MDD for $\bowtie_{c=1}^{C} \mathcal{Z}_c$, originally subject to ρ^F, if we interpret it as subject to $\boldsymbol{\rho}^{FI}$, encodes $\left(\bowtie_{c=1}^{C} \mathcal{Z}_c\right) \bowtie \mathcal{Z}^{eq}$ "for free"; minor changes

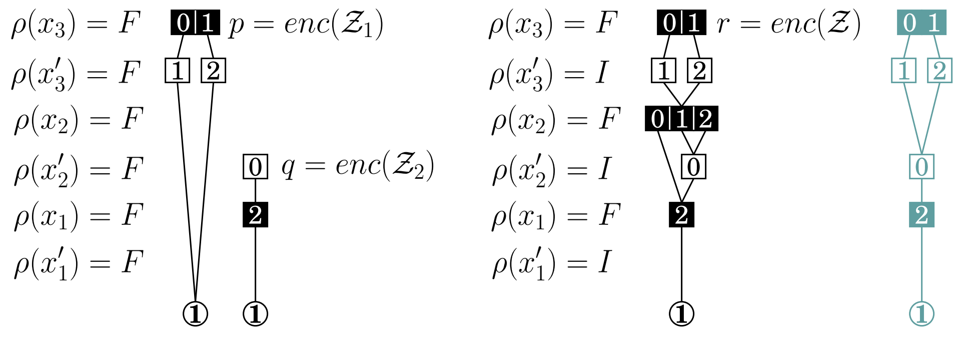


Figure 4.3: MDDs encoding $\mathcal{Z}$ and its sub-relations.

may apply to nodes associated to the primed variables in $spt(\mathcal{Z})$ when they are skipped.

In general discrete-state systems, both asynchronous and synchronous behavior can be present. Then, $\{\mathcal{T}_e | e \in \mathcal{E}\}$ gives a disjunctive partition of $\mathcal{T}$ and describes the asynchronous behavior of the system. Each $\mathcal{T}_e$, when expressed as a natural join of sub-relations as in Eq. 2.1, corresponds to the synchronous behavior of the system. Each $\mathcal{T}_e$ and its sub-relations can be encoded into MDDs as discussed above, and a *union* operation over each $enc(\mathcal{T}_e)$ results in $enc(\mathcal{T})$.

4.1.3 Symbolic generation of the reachable states

Let $Img(\mathcal{X}, \mathcal{Z}) = \{\mathbf{i}' : \exists \mathbf{i} \in \mathcal{X}, (\mathbf{i}, \mathbf{i}') \in \mathcal{Z}\}$ denote the *image* of the set of states $\mathcal{X}$ subject to the transition relation $\mathcal{Z}$. The set of reachable states $\mathcal{S} \subset \mathbb{N}^L$ is the minimal set satisfying $\mathcal{S} \supseteq \mathcal{S}^{init}$ and $\mathcal{S} \supseteq Img(\mathcal{S}, \mathcal{T})$. If $\mathcal{T}$ can be computed a priori, i.e., the bounds on the state variables are known, we can start from $\mathcal{S}^{init}$ and repeatedly perform image computations subject to $\mathcal{T}$ until reaching a fixpoint. As $\mathcal{T} = \bigcup_{e \in \mathcal{E}} \mathcal{T}_e$, we can use the transition relations $\mathcal{T}_e$ instead of $\mathcal{T}$ for state-space generation, as long as each $\mathcal{T}_e$ is applied often enough. Alternatively, we can define transition relations indexed by state variables, $\mathcal{T}_{x_k} = \bigcup_{top(\mathcal{T}_e)=x_k} \mathcal{T}_e$, for $L \geq k \geq 1$, which leave variable x_l unchanged if $x_l \succ x_k$. Since $\mathcal{X}$ and $\mathcal{Z}$ are encoded as MDDs, the MDD encoding $Img(\mathcal{X}, \mathcal{Z})$ is given by the *relational product* of these MDDs.

In the pseudocode, we use the type *mdd* for an MDD node. Two classes of symbolic state-space generation algorithms exist: *breadth-first* [54,58] and *saturation* [22,28]. Fig. 4.4 shows the breadth-first algorithm and two variations based on *chaining* [56]: breadth-first with chaining by events or with chaining by variables. For simplicity, the pseudocode assume $\boldsymbol{\rho}^Q$, i.e., no edge skips variables. The advantage of chaining is that more states can be found by each global iteration. When chaining by events, the number of iterations required to reach the fixpoint is affected by the order in which events are applied; experimentally, applying them in increasing order of *top* tends to work well [22].

Fig. 4.5 shows two saturation algorithms, by variables and by events. These differs from breadth-first generation in that they recursively compute a fixpoint at each node, in a low-to-high order of the associated variables. In the pseudocode, Breadth-first uses the "8b" variant of *RelProd*, saturation the "8s" one. The two variants differ in that, with "8b", *RelProd*(s, r) computes only a *one-step* image of the set encoded by s and the relation encoded by r, while, with "8s", it returns the *fixpoint* of the computed image w.r.t. $\mathcal{T}_{x_h}$ for $s.var \succeq x_h$.

Experimentally, saturation performs far better than breadth-first methods and saturation by variables is preferable to saturation by events. (i.e., in the best cases it is much better and in the worst case it is only slightly worse).

```
$mdd\ GenerateByBFS()$
  1  $s \leftarrow enc(\mathcal{S}^{init})$;
  2  repeat $s \leftarrow AddStates(s)$;
  3  until $s$ does not change;
  4  return $s$;
```

```
$mdd\ AddStates(mdd\ s)$
  1   $r \leftarrow enc(\mathcal{T})$;                                   • standard breadth-first
  2   return $Or(s, RelProd(s, r))$;
----------------------------------------------------------------------
  1'  for $k = 1$ to $L$ do                                     • chain by events
  2'    foreach $e \in \mathcal{E}$ s.t. $top(\mathcal{T}_e) = x_k$ do
  3'      $r \leftarrow enc(\mathcal{T}_e)$;
  4'      $s \leftarrow Or(s, RelProd(s, r))$;
  5'  return $s$;
----------------------------------------------------------------------
  1"  for $k = 1$ to $L$ do                                     • chain by variables
  2"    $r \leftarrow enc(\mathcal{T}_{x_k})$;
  3"    $s \leftarrow Or(s, RelProd(s, r))$;
  4"  return $s$;
```

```
$mdd\ RelProd(mdd\ s, mdd\ r)$                                  • assume $s.var = x_k$
  1   if $s = \mathbf{1}$ and $r = \mathbf{1}$ then return $\mathbf{1}$;
  2   if $CacheHit(RPR, s, r, t)$ then return $t$;
  3   $t \leftarrow NewNode(s.var)$;
  4   foreach $i \in \mathcal{X}_k$ s.t. $s[i], r[i] \neq \mathbf{0}$ do                • assume $r[i].var = x_l$
  5     foreach $i' \in \mathcal{X}_l$ s.t. $r[i][i'] \neq \mathbf{0}$ do
  6       $u \leftarrow RelProd(s[i], r[i][i'])$;
  7       $t[i'] \leftarrow Or(t[i'], u)$;
----------------------------------------------------------------------
  8b  $t \leftarrow UTInsert(t)$;
----------------------------------------------------------------------
  8s  $t \leftarrow Saturate(UTInsert(t))$;
----------------------------------------------------------------------
  9   $CacheAdd(RPR, s, r, t)$;
  10  return $t$;
```

Figure 4.4: State-space generation: breadth-first.

```
mdd GenerateBySaturation()
1 $mdd\ s \leftarrow enc(\mathcal{S}^{init})$;
2 return $Saturate(s)$;

mdd Saturate(mdd s)                                          • assume $s.var = x_k$
1 if $CacheHit(SAT, s, t)$ then return $t$;
2 $t \leftarrow NewNode(s.var)$;
3 foreach $i \in \mathcal{X}_k$ s.t. $s[i] \neq \mathbf{0}$ do
4   $t[i] \leftarrow Saturate(s[i])$;                              • saturate below
5 $t \leftarrow UTInsert(DoFixPoint(t))$;
6 $CacheAdd(SAT, s, t)$;
7 return $t$;

mdd DoFixPoint(mdd t)                                        • assume $t.var = x_k$
1 $r \leftarrow enc(\mathcal{T}_{t.var})$;                                    • by variables
2 repeat
3   foreach $i \in \mathcal{X}_k$ s.t. $t[i] \neq \mathbf{0}$ and $r[i] \neq \mathbf{0}$ do          • assume $r[i].var = x_l$
4     foreach $i' \in \mathcal{X}_l$ s.t. $r[i][i'] \neq \mathbf{0}$ do
5       $u \leftarrow RelProd(t[i], r[i][i'])$;                            • use 8s
6       $t[i'] \leftarrow Or(t[i'], u)$;
7 until $t$ does not change;
--------------------------------------------------------------------------------
1' repeat                                                         • by events
2'   foreach $e \in \mathcal{E}$ s.t. $top(\mathcal{T}_e) = t.var$
3'     $mdd\ \ r \leftarrow enc(\mathcal{T}_e)$;
4'     foreach $i \in \mathcal{X}_k$ do s.t. $t[i] \neq \mathbf{0}$ and $r[i] \neq \mathbf{0}$       • assume $r[i].var = x_l$
5'       foreach $i' \in \mathcal{X}_l$ s.t. $r[i][i'] \neq \mathbf{0}$
6'         $u \leftarrow RelProd(t[i], r[i][i'])$;                          • use 8s
7'         $t[i'] \leftarrow Or(t[i'], u)$;
8' until $t$ does not change;
```

Figure 4.5: State-space generation: saturation.

4.2 On-the-fly discovery of the MDD domain

We now review symbolic on-the-fly state-space generation (where the sets $\mathcal{X}_k$ are not a priori known) and present an optimized way to update $enc(\mathcal{T}_e)$.

4.2.1 Extensible state variable domains

When $\mathcal{T}$ or each $\mathcal{T}_e$ is known a priori, we can compute $\mathcal{S}$ with a simple breadth-first or saturation algorithm. However, the bounds on the state variables are often unknown. Then, a state variable can be considered to have an *extensible* domain, which grows during the generation of $\mathcal{S}$. In other words, if we do not know the actual bounds on the state variables, we can generate $\mathcal{T}_e$ on-the-fly alongside $\mathcal{S}$. When using decision diagrams, this means adding boolean variables (for BDDs) or increasing the set $\mathcal{X}_k$ (for MDDs), as new values for a state variable x_k are discovered. The key difference from the algorithms of Fig. 4.4 and Fig. 4.5 is that the transition relations must be updated before calling *RelProd*, (*AddStates* in Lines 2, 5', 4" or *DoFixPoint* in Lines 6, 7').

[28,51] proposed the first on-the-fly saturation algorithms with transition relations encoded by decision diagrams. Fig. 4.6 summarizes their approach to update transition relations: when a *local state index* is *confirmed*, i.e., a new value i for an unprimed state variable x_k is found, each sub-relation $\mathcal{T}_{e,c}$ (*group* in [51], *conjunct* in [28]) with $x_k \in spt(\mathcal{T}_{e,c})$ is updated to include the required new state-to-state transitions. This is done by first generating a sub-relation $\mathcal{T}_{e,c}^{x_k=i}$ encoding these new pairs, then integrating it into $\mathcal{T}_{e,c}$. Finally, $enc(\mathcal{T}_e)$ is obtained by *intersection* of the sub-relations $enc(\mathcal{T}_{e,c})$ of e, while each $enc(\mathcal{T}_{x_l})$ and $enc(\mathcal{T})$ are obtained through *unions*. Each $enc(\mathcal{T}_{e,c})$ will be the least updating

unit; it cannot be further broken. This approach is applicable to both breadth-first and saturation symbolic on-the-fly state-space generation algorithms.

[51] encodes transition relations with identity-reduced *matrix diagrams*, similar to MDDs subject to $\boldsymbol{\rho}^{FI}$ and requires the support sets of sub-relations from the same $\mathcal{T}_e$ to be disjoint. [28], on which our work is based, allows overlapping support sets for the sub-relations from the same $\mathcal{T}_e$, thus a finer partitioning of the transition relation and a more efficient state-space generation, using $\boldsymbol{\rho}^{F}$ when encoding sub-relations and $\boldsymbol{\rho}^{FI}$ when encoding each $\mathcal{T}_e$.

4.2.2 An improved implementation

We have seen in Sec. 4.1.2 that $\boldsymbol{\rho}^{F}$ are well suited for sub-relations and $\boldsymbol{\rho}^{FI}$ for each $\mathcal{T}_e$, as these rules produce compact MDDs, respectively [28]. However, two problems arise with these choices. First, the semantic of a skipped variable in an MDD subject to $\boldsymbol{\rho}^{F}$ changes when its domain grows, so we must be careful when updating the sub-relations $enc(\mathcal{T}_{e,c})$. Second, the change from $\boldsymbol{\rho}^{F}$ to $\boldsymbol{\rho}^{FI}$ when building transition relations from sub-relations requires extra handling, refer to the *Translate* operation in Sec. 3.2.2.

To illustrate the first problem, consider the example of Fig. 4.3, modified so that $\mathcal{Z}_2 \equiv [x'_2 = x_1 - 2]$ applies only if $x_2 \leq 2$. When $\mathcal{X}_2 = \{0, 1, 2\}$, q correctly encodes $\mathcal{Z}_2$ subject to $\boldsymbol{\rho}^{F}$. However, if we later add 3 to $\mathcal{X}_2$, q now (incorrectly) also encodes transitions of the form $(i_3, 3, 2) \to (i_3, 0, i_1)$, for $i_3 \in \mathcal{X}_3$ and $i_1 \in \mathcal{X}_1$. For the second problem, Fig. 4.7 assumes a transition relation $\mathcal{Z}$ with one sub-relation $\mathcal{Z}_1$, encoded by node p_1 subject to $\boldsymbol{\rho}^{F}$, in addition to $\mathcal{Z}^{eq} = [x'_1 = x_1]$. After intersecting all sub-relations (there is only one), we should obtain $p_2 = enc(\mathcal{Z})$ subject to $\boldsymbol{\rho}^{FI}$. No intersection is performed, but a transformation from p_1 to p_2 is needed since x'_2 changes reduction rule, from $\rho(x'_2) = F$ to $\rho(x'_2) = I$.

We solve these problem through two alternate reduction approaches. With $\boldsymbol{\rho}^{QF}$, $\rho(x) = Q$ if $x \in spt(\mathcal{Z}_c)$, and $\rho(x) = F$ otherwise. With $\boldsymbol{\rho}^{QFI}$, for $L \geq k \geq 1$, $\rho(x'_k) = I$, while $\rho(x_k) = Q$ if $x_k \in spt(\mathcal{Z})$, and $\rho(x_k) = F$ otherwise. We now give an optimized implementation using an MDD subject to $\boldsymbol{\rho}^{QF}$ to encode each sub-relation $\mathcal{T}_{e,c}$. The intersection algorithm of Fig. 4.8, applied to all sub-relations $enc(\mathcal{T}_{e,c})$, produces an MDD r, which, subject to $\boldsymbol{\rho}^{QFI}$ is $enc(\mathcal{T}_e)$. It is a specialized and optimized *Apply* algorithm (see Sec. 3.2.3) for a certain $\boldsymbol{\rho}$. Although MDDs subject to $\boldsymbol{\rho}^{QF}$ and $\boldsymbol{\rho}^{QFI}$ might not be as compact as their equivalence subject to $\boldsymbol{\rho}^{F}$ and $\boldsymbol{\rho}^{FI}$, respectively, we gain efficiency. With the new reductions, in Fig. 4.7, $p_2 = enc(\mathcal{Z}_1)$ subject to $\boldsymbol{\rho}^{QF}$, while $p_2 = enc(\mathcal{Z})$ subject to $\boldsymbol{\rho}^{QFI}$, no transformation is needed. Also, node q in Fig. 4.3, when changed to $\boldsymbol{\rho}^{QF}$, is q_2 in Fig. 4.7, still encoding the correct set after $\mathcal{X}_2$ is enlarged.

4.3 MDDs with infinite domain to encode transition relations

Consider a call *UpdateVariableTR*(x_l) (Fig. 4.6) after a variable domain $\mathcal{X}_k$ has grown, with $x_l \succeq x_k$. It seems redundant to perform the union of all $enc(\mathcal{T}_e)$ with $top(\mathcal{T}_e) = x_l$ to recompute $enc(\mathcal{T}_{x_l})$, when only those $enc(\mathcal{T}_e)$ containing x_k in their domain might change. Ideally, we could simply add the new $enc(\mathcal{T}_e)$, which have changed due growing $\mathcal{X}_k$, to the original $enc(\mathcal{T}_{x_l})$. However, the otherwise very effective $\boldsymbol{\rho}^{FI}(\boldsymbol{\rho}^{QFI})$ reduction used for $\mathcal{T}_e$ and $\mathcal{T}_{x_l}$ makes this problematic, as Fig. 4.9 illustrates. Assume that $\mathcal{X}_2 = \{0,1,2\}$, $\mathcal{X}_1 = \{0,1\}$, and that $r_1 = enc(\mathcal{T}_{x_2})$ is obtained as the union of p and q. If a new value 2 is added to $\mathcal{X}_1$ but p and q do not change due to this, one would imagine that r_1 still correctly encodes $\mathcal{T}_{x_2}$, thus nothing needs to be done. In fact, r_2, not r_1, now correctly encodes $\mathcal{T}_{x_2}$, while r_1 lacks some of the required state-to-state transitions. Using r_1 instead of r_2 could then miss reachable states during image computation. Worse yet, the cache for union operations needs to be cleared, further reducing efficiency, since the union of two MDDs becomes incorrect when a domain $\mathcal{X}_k$ changes and x_k is in the support of one but not the other.

We both solve this correctness issue and improve efficiency by letting the domain of variables to be $\mathbb{N}$ instead of a finite set, for MDDs encoding transition relations, which is allowed in the new generally-reduced TDDs but not allowed in the traditional MDDs.

This will not affect the domain of MDDs encoding set of states, so we have an MDD variable, for a different purpose, have different domains. We still use $\mathcal{X}_k$ to refer to the domain of x_k when the MDDs are used to encode set of states, so is still finite. For MDDs encoding transition relations, all variable domain are then $\mathbb{N}$ by default.

Recalling Def. 3.1.1, although the domain is $\mathbb{N}$, a nonterminal node p can only have a finite number of edges not pointing to $p[*]$. So, there is a value $p_f \in \mathbb{N}$ such that, for $i > p_f$, all edges $p[i]$ point to the same node $p[*]$, while $p[p_f] \neq p[*]$. So we can still have a finite encoding, since we only need to store the $p_f + 2$ edges $p[0], p[1], ..., p[p_f], p[*]$.

This solves the problem of Fig. 4.9: the union of p and q is now r_3 and correctly encodes $\mathcal{T}_{x_2}$ even when $\mathcal{X}_1$ changes. Fig. 4.10 shows the version of the union operation for two MDDs subject to $\boldsymbol{\rho}^{QFI}$. Again, it is a specified and optimized *Apply* operation for a certain $\boldsymbol{\rho}$, and details the handling of the $[*]$ part. This is the *only* operation that can create a node r with $r[*] \neq \mathbf{0}$ in our algorithm. In the pseudocode, the recursive union is guaranteed to be invoked on nodes p and q only when $p.var$ and $q.var$ are quasi-reduced unprimed variables:

if $p.var \succ q.var$ then $r.var = p.var$, $r_f = p_f$, $r[*] = Or(p[*], q)$;
if $p.var = q.var$ then $r.var = p.var$, $r_f = \max\{p_f, q_f\}$, $r[*] = Or(p[*], q[*])$.

The intersection of Fig. 4.8, in other word, need not to deal with the $[*]$ part, since $p[*] = \mathbf{0}$ in any node p subject to $\boldsymbol{\rho}^{QF}$. Then, the update of $enc(\mathcal{T}_{x_l})$ and $enc(\mathcal{T})$ can be done more efficiently, as shown in Fig. 4.11.

4.4 Experimental results

We implemented the proposed approach in SMART [17] and report on experiments run on an Intel Xeon 3.0Ghz workstation with 16GB RAM under SuSE Linux 9.1. Each experiment set (run on a variety of models, referenced in Tab. 4.1) is denoted by a combination X-Y, corresponding to state-space generation algorithm X and reduction Y. For the reduction choices, we use:

- FR: sub-relations and transition relations encoded by traditional fully-reduced MDDs
- QFI: sub-relations encoded by MDDs subject to $\boldsymbol{\rho}^{QF}$, transition relations encoded by MDDs subject to $\boldsymbol{\rho}^{QFI}$, both with finite variable domains, (Sec. 4.2.2); on-the-fly updates of Fig. 4.6.

- EQFI: sub-relations encoded by MDDs subject to $\boldsymbol{\rho}^{QF}$, transition relations encoded by MDDs subject to $\boldsymbol{\rho}^{QFI}$, both with variable domain $\mathbb{N}$; on-the-fly updates of Fig. 4.11.

For the state-space algorithm choices, we use:

- B: Standard breadth-first. B-FR is close to NuSMV's [29] approach, which uses CUDD's [59] fully-reduced BDDs.
- CV: breadth-first with chaining by variables.
- CE: breadth-first with chaining by events (no union of transition relations).
- V: saturation by variables. V-QFI is the improvement of [28] in Sec. 4.2.2.
- E: saturation by events (as for CE, no union of transition relations is needed).

Tab. 4.1 shows runtime, peak memory consumption and number of unions invoked in procedure *UpdateVariableTR* and *UpdateOverallTR* for state-space generation, on a set of models. From it, we can make the following observations:

- For the same algorithm, QFI and EQFI is much better than FR, in both time and memory.
- For the same algorithm, EQFI is $20-80\%$ faster than QFI for most models, while is it is at worst 2% slower in the remaining models. QFI and EQFI have similar memory consumption.
- The improvement due to utilizing infinite domain ability of new generally-reduced MDDs is more substantial for saturation than for breadth-first.

			BREADTH-FIRST GENERATION																		SATURATION									
			B-FR		B-QFI			B-EQFI			CV-FR		CV-QFI			CV-EQFI			CE-QFI		V-FR		V-QFI			V-EQFI			E-QFI	
Model	N	$\lvert\mathcal{S}\rvert$	sec	MB	sec	MB	Or	sec	MB	Or	sec	MB	sec	MB	Or	sec	MB	Or	sec	MB	sec	MB	sec	MB	Or	sec	MB	Or	sec	MB
bitshift	6	$2.62 \cdot 10^{5}$	0.02	0.44	0.01	0.26	173	0.01	0.26	40	0.01	0.18	0.01	0.16	21	0.00	0.16	20	0.01	0.16	0.01	0.10	0.01	0.08	22	0.01	0.08	20	0.01	0.08
[28]	256	$4.63 \cdot 10^{77}$	4.95	140	3.23	94.6	2573	2.84	94.6	520	0.75	33.1	0.72	30.6	261	0.59	30.6	260	0.58	30.6	0.47	17.8	0.28	13.6	262	0.28	13.6	260	0.28	13.6
bittog–	100	$1.26 \cdot 10^{30}$	2.40	10.0	0.13	4.49	20199	0.09	4.50	400	2.23	67.9	1.44	67.8	299	1.40	67.8	200	2.51	117	0.05	0.15	0.02	0.10	299	0.02	0.10	200	0.03	0.09
gle [39]	200	$1.60 \cdot 10^{60}$	21.4	39.9	0.60	17.8	80399	0.43	17.8	800	30.0	508	22.5	508	599	22.4	508	400	41.5	717	0.15	0.30	0.06	0.19	599	0.04	0.19	400	0.04	0.18
bqueue	50	$4.57 \cdot 10^{9}$	13.3	68.9	2.39	36.5	2167	2.38	36.5	813	10.0	46.9	1.41	35.1	602	1.42	35.1	551	16.7	316	-	-	3.20	11.2	854	1.43	11.2	702	8.29	50.1
[55]	100	$2.70 \cdot 10^{11}$	118	178	20.3	91.5	4333	20.3	91.0	1620	95.9	187	15.1	145	1202	15.1	145	1101	-	-	-	-	40.1	65.9	1704	18.2	66.0	1402	171	160
bubsort	11	$3.99 \cdot 10^{7}$	16.5	38.2	15.1	35.9	650	15.1	35.9	180	1.48	33.1	1.08	25.6	65	1.08	25.6	65	1.09	25.6	-	-	1.27	7.21	210	0.62	7.21	210	0.63	7.21
[28]	15	$1.30 \cdot 10^{12}$	-	-	-	-	-	-	-	-	108	188	113	259	119	113	259	119	113	259	-	-	61.4	201	210	35.0	201	210	35.1	201
dme	50	$6.30 \cdot 10^{48}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	362	49.4	1.36	24.5	3314	0.94	25.2	3103	1.46	40.6
[29]	80	$8.59 \cdot 10^{76}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	2.38	41.2	5294	1.70	42.3	4963	2.60	67.2
fms	10	$2.50 \cdot 10^{9}$	15.6	165	9.37	127	2087	9.33	127	797	4.24	129	3.44	115	343	3.42	115	292	3.17	112	47.8	12.6	0.36	6.29	442	0.26	6.30	374	0.26	5.63
[50]	20	$6.02 \cdot 10^{12}$	-	-	-	-	-	-	-	-	121	395	94.6	359	678	94.7	359	582	-	-	-	-	5.31	62.1	872	4.19	62.1	744	4.05	55.1
ftoler–	10	$2.95 \cdot 10^{26}$	3.35	63.9	1.25	46.2	6680	1.24	46.5	3780	0.65	28.6	0.58	27.6	950	0.56	27.6	920	3.75	140	3.22	6.22	0.18	3.44	2370	0.14	3.63	2270	0.25	5.50
ant [28]	30	$6.47 \cdot 10^{78}$	-	-	-	-	-	-	-	-	20.5	726	19.8	727	2850	19.8	726	2760	-	-	246	83.1	3.89	65.8	17540	2.84	67.4	17220	6.65	113
kanban	10	$1.00 \cdot 10^{9}$	1.61	31.3	0.49	16.5	1242	0.50	16.5	404	0.48	9.41	0.15	6.12	296	0.15	6.12	291	0.23	10.1	1.77	3.36	0.07	1.82	370	0.06	1.83	332	0.08	2.65
[50]	50	$1.04 \cdot 10^{16}$	-	-	-	-	-	-	-	-	89.3	1087	33.9	704	1496	33.8	704	1471	48.6	1040	-	-	13.8	252	1890	13.3	252	1692	32.5	266
knight	5	$6.76 \cdot 10^{7}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	12.8	28.3	1.14	25.9	450	0.72	25.8	192	0.62	20.3
[34]	6	$1.63 \cdot 10^{11}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	313	64.5	727	259	64.6	320	2.72	75.8
intshift	32	$9.35 \cdot 10^{49}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	14.2	22.0	1.20	19.0	100	1.19	18.9	66	1.26	20.3
[28]	45	$2.23 \cdot 10^{76}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	76.9	40.3	4.76	25.7	139	4.76	25.7	92	4.98	28.3
leader	8	$3.04 \cdot 10^{8}$	97.9	73.3	82.1	58.1	3360	81.9	58.1	1548	80.1	315	68.5	262	2939	68.4	262	1425	-	-	-	-	3.02	38.7	3439	2.40	38.8	1354	13.1	50.6
[28]	10	$5.02 \cdot 10^{10}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	45.2	58.8	5516	37.7	58.4	2173	461	86.5
phils	50	$2.22 \cdot 10^{31}$	3.79	16.8	0.19	5.40	5600	0.19	5.41	648	0.19	2.63	0.11	2.26	402	0.10	2.27	304	0.17	6.69	0.96	0.89	0.09	0.42	552	0.08	0.43	449	0.08	0.32
[50]	100	$4.96 \cdot 10^{62}$	29.6	65.9	0.78	20.9	21200	0.73	28.2	1298	0.66	9.09	0.28	8.34	802	0.27	8.36	604	0.68	26.7	3.80	1.80	0.17	0.83	1102	0.16	0.85	899	0.16	0.65
polling	15	$3.35 \cdot 10^{19}$	38.4	28.6	0.44	7.04	4244	0.45	7.05	975	1.48	7.17	0.26	4.28	525	0.26	4.29	509	0.27	3.80	5.74	1.25	0.21	0.86	384	0.19	0.86	341	0.24	0.83
[50]	30	$3.25 \cdot 10^{46}$	-	-	4.99	71.3	30314	4.91	126	3750	38.6	65.1	1.66	42.9	1950	1.65	43.0	1919	1.83	37.9	196	6.26	0.89	4.65	1224	0.86	4.65	1136	0.88	4.62
queen	12	$8.56 \cdot 10^{5}$	2.39	83.2	2.30	82.5	356	2.31	82.5	145	3.74	128	3.71	127	266	3.72	127	133	15.9	553	85.1	65.6	2.98	65.2	266	1.79	65.2	133	2.16	65.2
[34]	13	$4.67 \cdot 10^{6}$	12.2	381	11.9	378	418	11.9	378	170	19.3	595	19.1	595	314	19.0	595	157	-	-	536	330	16.7	330	314	10.9	330	157	12.9	330
rips	5	$2.97 \cdot 10^{13}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	296	810	2103	290	807	949	-	-
[57]	10	$8.87 \cdot 10^{14}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	414	904	2103	404	902	949	-	-
robin	100	$2.85 \cdot 10^{32}$	233	517	55.7	333	97540	55.4	331	1699	78.0	439	15.2	520	21494	15.1	520	700	39.3	1004	581	217	3.50	51.3	22088	2.00	51.2	799	2.99	77.6
[28]	200	$7.23 \cdot 10^{62}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	25.2	331	84188	13.5	331	1599	25.2	593
slot	50	$1.72 \cdot 10^{52}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1.09	6.00	1256	0.43	6.02	600	1.02	12.5
[28]	70	$3.12 \cdot 10^{73}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	2.64	14.1	1756	1.00	14.1	840	2.54	31.3
swap-	20	$1.31 \cdot 10^{11}$	1.89	23.4	0.54	15.1	15716	0.50	15.1	232	0.26	14.4	0.21	13.7	96	0.21	13.7	96	0.21	13.7	0.21	0.22	0.02	0.17	96	0.01	0.17	96	0.01	0.17
per [28]	100	$8.96 \cdot 10^{58}$	-	-	-	-	-	-	-	-	312	443	300	448	496	297	448	496	300	448	25.9	3.50	0.27	2.94	496	0.11	2.94	496	0.11	2.94

Table 4.1: Results (“sec”: time in sec; “MB”: memory in MB; “Or”: number of unions; “–”: runtime $>$ 600sec).

- CV is preferable to CE, and V is preferable to E. Clearly, it is best to merge events according to their top state variable, even at the cost of additional unions. This further stresses the benefits of the new generally-reduced MDDs with infinite domain.
- Saturation (V, E) with QFI or EQFI performs much better than breadth-first (B, CV, CE) with any reduction.

We compare the number of unions for transition relations with or without adoption of infinite domain, specifically, for algorithms B–QFI vs. B–EQFI, CV-QFI vs. CV–EQFI and V–QFI vs. V–EQFI, which accounts for part of the improvements we achieve. For all experiments, thanks to the updating approach of Fig. 4.11, the number of unions decreases when using infinite domains for transition relations, and this decrease is positively correlated to the time improvement, especially for saturation-based algorithms. When this decrease is large ($\geq 50\%$), e.g., B with *bitshift* and *bittoggle* or V with *knight*, *leader*, *queen*, *robin*, or *slot*, we achieve substantial time improvement ($20 - 80\%$) from fewer unions; when the decrease is relatively small, we achieve a minor or no improvement (V with *bitshift*, *kanban*, *phils*, *polling*). Sometimes, even if the decrease is small or there is no decrease in the number of unions, we still achieve a sharp improvement (e.g., V with *bubsort*, *bqueue*, *dme*, *fms*, *ftolerant*, *swapper*), because new generally-reduced MDDs with infinite domain do not require to invalidate the caches, making unions more efficient. *intshift* and *rips* show a sharp decrease in the number of unions under V, but do not improve runtime much, because unions are a small part of state-space generation for these models.

We can then experimentally surmise that EQFI usually improves the runtime, for some benchmarks greatly so, while never degrades it in appreciable ways, and does not consume more memory, so it is preferable to QFI.

4.5 Conclusion

We presented how we utilize the new class of generally-reduced decision diagrams, especially its ability to allow variable domain to be $\mathbb{N}$, which is a perfect match for symbolic state-space generation on-the-fly, i.e., when the bounds of the state variables are not known a priori. Through a set of experiments, we showed how its use benefits both breadth-first and saturation-based algorithms. In the future, we intend to explore the application of this infinite domain to other classes of symbolic algorithms, beyond state-space generation.

```
$AddValue(variable\ x_k, index\ i)$
1 $\mathcal{X}_k \leftarrow \mathcal{X}_k \cup \{i\}$;
2 foreach $\mathcal{T}_{e,c}$ s.t. $x_k \in spt(\mathcal{T}_{e,c})$ do
3   build $\mathcal{T}_{e,c}^{x_k=i}$ by querying the model;
4   $p \leftarrow enc(\mathcal{T}_{e,c}^{x_k=i})$;
5   $enc(\mathcal{T}_{e,c}) \leftarrow Or(enc(\mathcal{T}_{e,c}), p)$;
```

```
$UpdateEventTR(event\ e)$
1 if $\exists x_k \in spt(\mathcal{T}_e), \mathcal{X}_k$ changed then
2   $mdd\ r \leftarrow \mathbf{1}$;
3   foreach $\mathcal{T}_{e,c}$ do
4     $r \leftarrow And(r, enc(\mathcal{T}_{e,c}))$;
5   $enc(\mathcal{T}_e) \leftarrow r$;
```

```
$UpdateVariableTR(variable\ x_l)$
1 if $\exists x_k \in spt(\mathcal{T}_{x_l}), \mathcal{X}_k$ changed then
2   $mdd\ r \leftarrow \mathbf{0}$;
3   foreach $e \in \mathcal{E}$ s.t. $top(\mathcal{T}_e) = x_l$ do
4     $r \leftarrow Or(r, enc(\mathcal{T}_e))$.
5   $enc(\mathcal{T}_{x_l}) \leftarrow r$;
```

```
$UpdateOverallTR()$
1 $mdd\ r \leftarrow \mathbf{0}$;
2 for $k = 1$ to $L$ do
3   $r \leftarrow Or(r, enc(\mathcal{T}_{x_k}))$;
4 $enc(\mathcal{T}) \leftarrow r$;
```

Figure 4.6: Updating the transition relations.

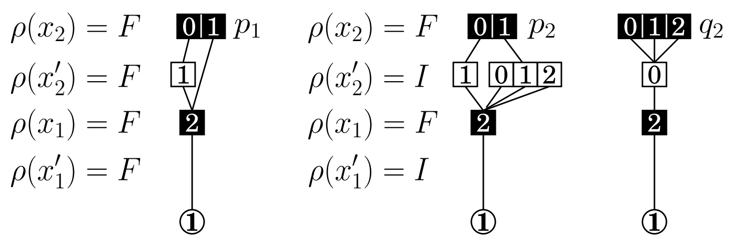


Figure 4.7: MDDs subject to $\boldsymbol{\rho}^{QF}$ and $\boldsymbol{\rho}^{QFI}$.

```
$mdd\ And(mdd\ p, mdd\ q)$                                    • assume $p.var = x_k$
if $p = \mathbf{0}$ or $q = \mathbf{0}$ then return $\mathbf{0}$;                   • trivial cases
if $p = \mathbf{1}$ or $p = q$ then return $q$
if $q = \mathbf{1}$ then return $p$;
if $q.var \succ p.var$ then $Swap(p, q)$;                      • commutativity
if $CacheHit(AND, p, q, r)$ then return $r$;                   • check cache
$r \leftarrow NewNode(p.var)$;
if $p.var \succ q.var$ then
  foreach $i \in \mathcal{X}_k$ do $r[i] \leftarrow And(p[i], q)$;
else                                                           • $p.var = q.var$
  foreach $i \in \mathcal{X}_k$ do $r[i] \leftarrow And(p[i], q[i])$;
$r \leftarrow UTInsert(r)$;                                    • insert into unique-table
$CacheAdd(AND, p, q, r)$;                                      • add to cache
return $r$;
```

Figure 4.8: Intersection of two MDDs subject to $\boldsymbol{\rho}^{QF}$.

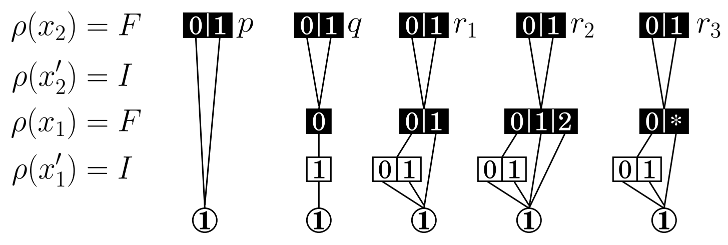


Figure 4.9: Problem with updating $\mathcal{T}_{x_2}$.

```
$mdd\ Or(mdd\ p, mdd\ q)$
if $p = \mathbf{0}$ or $p = q$ then return $q$;
if $q = \mathbf{0}$ then return $p$;                                    • trivial cases
if $q.var \succ p.var$ then $Swap(p, q)$;                               • commutativity
if $CacheHit(OR, p, q, r)$ then return $r$;
$r \leftarrow NewNode(p.var)$;
if $p.var \succ q.var$ then
  $r[*] \leftarrow Or(r[*], q)$;
  foreach $i \in \{0, 1, ..., p_f\}$
    if $\rho(p[i].var) = I$ then
      $r[i] \leftarrow Copy(p[i])$;  $r[i][i] \leftarrow Or(p[i][i], q)$;  $r[i] \leftarrow UTInsert(r[i])$;
    else $r[i] \leftarrow Or(p[i], q[i])$;
else                                                                    • $p.var = q.var$
  $r_f \leftarrow \max\{p_f, q_f\}$;
  foreach $i \in \{0, 1, \cdots, r_f\}$ do
    if $\rho(p[i].var) = \rho(q[i].var) = I$ then
      $r[i] \leftarrow NewNode(p[i].var)$
      foreach $j \in \{0, 1, ..., \max(p[i]_f, q[i]_f)\}$ do
        $r[i][j] \leftarrow Or(p[i][j], q[i][j])$;
      $r[i] \leftarrow UTInsert(r[i])$;
    else if $\rho(p[i].var) = I$ and $\rho(q[i].var) = Q$ then
      $r[i] \leftarrow Copy(p[i])$   $r[i][i] \leftarrow Or(r[i][i], q)$;   $r[i] \leftarrow UTInsert(r[i])$;
    else if $\rho(p[i].var) = Q$ and $\rho(q[i].var) = I$ then
      $r[i] \leftarrow Copy(q[i])$;   $r[i][i] \leftarrow Or(r[i][i], p)$;   $r[i] \leftarrow UTInsert(r[i])$;
    else                                                                • $\rho(p[i].var) = \rho(q[i].var) = Q$
      $r[i] \leftarrow Or(p[i], q[i])$;
  $r[*] \leftarrow Or(p[*], q[*])$;
$r_f \leftarrow \max\{i : r[i] \neq r[*]\}$;                           • reset $r_f$ according to definition
$r \leftarrow UTInsert(r)$;
$CacheAdd(OR, p, q, r)$;
return $r$;
```

Figure 4.10: Union of two MDDs subject to $\boldsymbol{\rho}^{QFI}$.

```
$UpdateVariableTR(variable\ x_l)$
1 if $\exists x_k \in spt(\mathcal{T}_{x_l}), \mathcal{X}_k$ changed then
2   foreach $e \in \mathcal{E}$ s.t. $top(\mathcal{T}_e) = x_l$ do
3     if $enc(\mathcal{T}_e)$ changed then
4       $enc(\mathcal{T}_{x_l}) \leftarrow Or(enc(\mathcal{T}_{x_l}), enc(\mathcal{T}_e));$
```

```
$UpdateOverallTR()$
1 for $k = 1$ to $L$ do
2   if $enc(\mathcal{T}_{x_k})$ changed then
3     $enc(\mathcal{T}) \leftarrow Or(enc(\mathcal{T}), enc(\mathcal{T}_{x_k}));$
```

Figure 4.11: Updating the transition relations using infinite variable domain

Chapter 5

Symbolic Reachability Analysis of Timed Petri Nets

Petri nets are very effective at modeling distributed systems, and have been extensively researched [53]. For systems where timing is an integral part of the dynamics and can affect the logical evolution, various time-related extension of Petri nets have been proposed. Of particular relevance to our present work are the so-called *time Petri nets* [47] and *timed Petri nets* [64], where the durations of events are either known to lie in a given interval, or to be constants.

We consider a class of timed Petri nets where the durations of the transition firing times are integer-valued, but can be chosen from an arbitrary finite set of not necessarily contiguous values. For this class, we explore two fundamental reachability problems: *timed reachability* (find the set of markings where the Petri net can be at a given finite point in time) and *earliest reachability* (find the first instant of time when the Petri net can enter each reachable marking).

These problems can be tackled in principle with explicit methods that explore states one by one, where a "state" describes both the marking of the Petri net and the remaining firing time of each enabled transition. However, even more than for untimed nets, the size of the state space is a formidable obstacle in practice. Following the trend in logical reachability and model checking, where efficient *symbolic* algorithms based on decision diagrams have been widely adopted [22,54], we develop novel symbolic algorithms based on both ordinary and edge-valued decision diagrams, and demonstrate their effectiveness on a suite of models.

This chapter is organized as follows. In Sec. 5.1, we define the class of timed Petri net we adopt in our work. Sec. 5.2 analyzes the timed reachability problem and propose a solution based on generally-reduced decision diagrams. Sec. 5.3 solves the earliest reachability problem. Sec. 5.4 discusses the decidability, complexity and related research. Sec. 5.5 presents experimental results and Sec. 5.6 concludes this chapter.

5.1 Integer Timed Petri Nets

Several extensions of standard Petri nets [53] have been proposed to explicitly represent a notion of time. Usually, a firing time is associated with each transition of the net. In *time Petri nets* [47], the firing times lie in a given interval $[t_{min}, t_{max}]$, where $0 \leq t_{min} < \infty$ and $t_{min} \leq t_{max} \leq \infty$. If $t_{min} = t_{max}$ for all transitions, i.e., all firing times are constants, as in *timed Petri nets* [64], nondeterminism can only arise from the resolution of conflicts among transitions.

Symbol	**Definition or Meaning**
$\mathcal{F}$	firing times for each transition
τ	remaining firing time
θ	elapsed time
$\langle\mu,\tau@\theta\rangle$	the ITPN is in tangible state $\langle\mu,\tau\rangle$ at time θ
$\langle\mu,\tau\not@\theta\rangle$	the ITPN is in vanishing state $\langle\mu,\tau\rangle$ at time θ
$\mathcal{E}(\mu)$	the set of marking-enabled transitions in μ
$\mathcal{E}(\langle\mu,\tau\rangle)$	the set of state-enabled transitions in μ
$\mathcal{E}_f$	maximally serializable subset of state-enabled transitions for a state
τ_b	$\min_{t\in\mathcal{T},\langle\mu,\tau\rangle\in\mathcal{S}_\theta}\{\tau(t)\}$, the breakpoint at time θ

Figure 5.1: New symbols and notations in this chapter.

The class of nets we consider is a restriction of time and timed Petri nets, on the one hand, since we require that all firings occur only at integer times, and an extension, on the other, as we allow the firing time to be nondeterministically chosen among a finite set of non-necessarily contiguous values. Such nets have been shown to give rise to discrete-time Markov chains if choices are resolved probabilistically [15]. Formally, *integer timed Petri nets* (ITPNs) are specified by a tuple $(\mathcal{P}, \mathcal{E}, \mathcal{A}, \mathcal{S}_0, \mathcal{F}, w)$ where:

- $\mathcal{P}$ is a finite, nonempty set of *places*.
- $\mathcal{E}$ is a finite, nonempty set of *transitions*, with $\mathcal{P} \cup \mathcal{E} \neq \emptyset$ and $\mathcal{P} \cap \mathcal{E} = \emptyset$.
- $\mathcal{A} \subseteq \mathcal{P} \times \mathcal{E} \cup \mathcal{E} \times \mathcal{P}$ is a set of directed arcs which connect places to transitions (*input*

arcs) and transitions to places (*output arcs*).

- $w : \mathcal{A} \to \mathbb{N}^+$ specifies a constant positive integer *cardinality* for each arc.
- $\mathcal{F} : \mathcal{E} \to 2^{\mathbb{N}^+}$ specifies a finite set of integer *firing times* for each transition.
- $\mathcal{M}_0 \subset \mathbb{N}^{\mathcal{P}}$ specifies a finite set of *initial markings*.

As an extension of untimed net (Sec. 2.2), we can define input flow matrix, D^-, the output flow matrix D^+ and the incidence matrix D in the same way. If we treat the marking $\mu \in \mathbb{N}^{\mathcal{P}}$ of the net as a (column) vector, we can then say that a transition $t \in \mathcal{E}$ is *marking-enabled* in μ if $\mu \geq D^-[\cdot, t]$. Let $\mathcal{E}(\mu) \subseteq \mathcal{E}$ be the set of marking-enabled transitions in μ. The firing of transition $t \in \mathcal{E}(\mu)$ in marking μ changes the marking to $\mu' = \mu + D[\cdot, t]$, we write this as $\mu \; [t\rangle\!\!\Rightarrow \mu'$.

The *state* of an ITPN is a pair $\langle \mu,\tau \rangle \in \mathbb{N}^{\mathcal{P}} \times (\mathbb{N} \cup \{\infty\})^{\mathcal{E}}$, where μ is the marking and τ is the *remaining firing times*, which, for each transition $t \in \mathcal{E}$, must satisfy $\tau[t] \leq \max \mathcal{F}(t)$ if $t \in \mathcal{E}(\mu)$, and $\tau[t] = \infty$ otherwise. We say that a transition $t \in \mathcal{E}$ is *state-enabled* in $\langle \mu,\tau \rangle$ if it is marking-enabled in μ and its firing time has elapsed, $\tau[t] = 0$. Let $\mathcal{E}(\langle \mu,\tau \rangle) \subseteq \mathcal{E}(\mu)$ be the set of state-enabled transitions in $\langle \mu,\tau \rangle$. If the ITPN is in $\langle \mu,\tau \rangle$ at time θ, it evolves as follows:

- If $\mathcal{E}(\mu) = \emptyset$, thus $\tau = \{\infty\}^{\mathcal{E}}$, the marking μ, as well as the state $\langle \mu,\tau \rangle$, is *dead*. The net will remain forever in that state.
- If $\mathcal{E}(\mu) \neq \emptyset$ but $\mathcal{E}(\langle \mu,\tau \rangle) = \emptyset$, there are marking-enabled transitions but none of their firing times has elapsed, the state is *tangible*. The net remains in marking μ

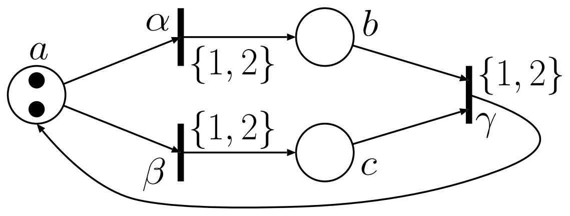


Figure 5.2: Running example: the integer timed Petri net.

for $\tau_* = \min_{t \in \mathcal{E}}\{\tau[t]\} > 0$ time units, while the remaining firing times elapse at unit rate. At time $\theta + \tau_*$, the state is $\langle \mu, \tau - \tau_* \rangle$, where the subtraction of scalar τ_* from vector τ is interpreted elementwise, i.e., $(\tau - \tau_*)[t] = \tau[t] - \tau_*$ for $t \in \mathcal{E}$. We write $\langle \mu, \tau \rangle \; [\tau_1 \rangle\!\!\Rightarrow \langle \mu, \tau - \tau_1 \rangle$, for $0 \le \tau_1 \le \tau_*$.

- If $\mathcal{E}(\langle \mu, \tau \rangle) \neq \emptyset$, there are marking-enabled transitions with elapsed firing times, the state is *vanishing*. The state then immediately changes by firing a nondeterministically chosen *maximally serializable* set $\mathcal{E}_f \subseteq \mathcal{E}(\langle \mu, \tau \rangle)$ as a single operation, we write $\langle \mu, \tau \rangle \; [\mathcal{E}_f \rangle\!\!\Rightarrow \langle \mu', \tau' \rangle$, where the new marking is $\mu' = \mu + \sum_{t \in \mathcal{E}_f} D[\cdot, t]$, while the remaining firing times τ' are chosen so that
  - $\tau'[t] = \infty$ if $t \notin \mathcal{E}(\mu')$, i.e., if t is disabled in the new marking,
  - $\tau'[t] \in \mathcal{F}(t)$ if $t \in (\mathcal{E}_f \cap \mathcal{E}(\mu')) \cup (\mathcal{E}(\mu') \setminus \mathcal{E}(\mu))$, i.e., if t is newly (re)enabled in the new marking, a firing time is nondeterministically chosen for it.
  - $\tau'[t] = \tau[t]$ otherwise, i.e., if the firing time of t continues to elapse undisturbed in the new marking, it remains unchanged.

A set of n transitions $\mathcal{E}_f \subseteq \mathcal{E}(\langle \mu, \tau \rangle)$ is maximally serializable if it is fireable, i.e., its elements can be ordered as $(t_1, ..., t_n)$ s.t. $\mu\,[t_1\rangle\!\!\Rightarrow \mu_1\,[t_2\rangle\!\!\Rightarrow \cdots\,[t_n\rangle\!\!\Rightarrow \mu_n$, and maximal, i.e., any other transition that was state-enabled in $\langle\mu,\tau\rangle$ becomes marking disabled in μ_n, i.e, $(\mathcal{E}(\langle\mu,\tau\rangle)\backslash\mathcal{E}_f) \cap \mathcal{E}(\mu_n) = \emptyset$.

To make explicit the time evolution of the state, we write $\langle\mu,\tau\cancel{@}\theta\rangle$ to signify that the ITPN is in vanishing state $\langle\mu,\tau\rangle$ at time θ, just before the firing of a maximally serializable set $\mathcal{E}_f$ of transitions, and $\langle\mu,\tau@\theta\rangle$ to signify that the ITPN is in tangible state $\langle\mu,\tau\rangle$ at time θ. Thus, a possible sequence of "snapshots" for the evolution of the ITPN from tangible (non-dead) state $\langle\mu,\tau\rangle$ at time θ is

$$\langle\mu,\tau@\theta\rangle\ [\tau_1\rangle\!\!\Rightarrow \langle\mu,\tau-\tau_1@\theta+\tau_1\rangle\ [\tau_2\rangle\!\!\Rightarrow \langle\mu,\tau-\tau_*\cancel{@}\theta+\tau_*\rangle\ [\mathcal{E}_f\rangle\!\!\Rightarrow \langle\mu',\tau'@\theta+\tau_*\rangle,$$

where $\langle\mu',\tau'\rangle$ is tangible and $\tau_1 + \tau_2 = \tau_* = \min_{t\in\mathcal{E}}\{\tau[t]\}$. Then, we write $\langle\mu_0,\tau_0@\theta\rangle$ $[*\rangle\!\!\Rightarrow \langle\mu_n,\tau_n@\theta'\rangle$ iff, from tangible state $\langle\mu_0,\tau_0\rangle$ at time θ, the ITPN can reach tangible state $\langle\mu_n,\tau_n\rangle$ at time $\theta' = \theta + \sum_{i=1}^{n}\tau_{i*}$ visiting vanishing states $\langle\mu_i,\tau_i-\tau_{i*}\rangle$ and tangible states $\langle\mu_i,\tau_i\rangle$, where $\tau_{i*} = \min_{t\in\mathcal{E}}\{\tau_i[t]\}$, and $\mathcal{E}_i$ is a maximally serializable set of transitions in $\langle\mu_i,\tau_i-\tau_{i*}\rangle$ whose firing causes the state change $\langle\mu_i,\tau_i-\tau_{i*}\rangle\ [\mathcal{E}_i\rangle\!\!\Rightarrow \langle\mu_{i+1},\tau_{i+1}\rangle$, for $i = 0, ..., n-1$.

We stress that the *maximal serializability semantics* we have adopted is similar to but differs from the *maximal step semantics* of [13], which further requires $\mu \geq \sum_{t\in\mathcal{E}_f} D^-[\cdot,t]$, so that any permutation of all transitions in $\mathcal{E}_f$ is fireable. It is also similar to, but different from, the *maximal non-blocking semantics* of [63], which simply requires that $\mu + \sum_{t\in\mathcal{E}_f} D[\cdot,t] \geq 0$, i.e., that the cumulative effect on μ of all firings in $\mathcal{E}_f$ be such that the result is a legal marking. Our semantic is suitable to model concurrent systems which require serializability, e.g., transaction processing systems [9]. Maximal and interleaving semantic differ in a fundamental way: in the former, the firing time of a transition that is disabled for a zero amount of time (if we sequentialize the transitions in $\mathcal{E}_f$) is not reset.

For simplicity, we defined ITPNs with an initial set of markings $\mathcal{M}_0$ but their analysis requires an initial set of states $\mathcal{S}_0$, which we can then define as $\mathcal{S}_0 = \{\langle\mu,\tau\rangle : \mu \in \mathcal{M}_0 \text{ and } \tau[t] = \infty \text{ if } t \notin \mathcal{E}(\mu')\ , \tau[t] \in \mathcal{F}(t) \text{ otherwise}\}$. Indeed, we could even allow an arbitrary initial setting for the remaining firing times associated to any initial marking. The techniques we present would still be applicable, as long as the set of initial states $\mathcal{S}_0$ is finite.

The upper left part of Figure 5.2 shows our ITPN running example, with places a, b, and c and transitions α, β, and γ. The set of initial markings $\mathcal{M}_0$ contains a single marking, a^2 ($a^2 = a^2b^0c^0$, empty places are omitted). The firing time of all transitions can be either 1 or 2. So $\mathcal{S}_0 = \{\langle a^2,\alpha^1\beta^1\rangle, \langle a^2,\alpha^1\beta^2\rangle, \langle a^2,\alpha^2\beta^1\rangle, \langle a^2,\alpha^2\beta^2\rangle\}$, where the notation for the remaining firing times is analogous to the one for markings, except that we omit disabled transitions instead of empty places.

5.2 Timed reachability

The timed reachability problem aims at finding all markings where the model can be at a finite time $\theta_f \in \mathbb{N}$. Formally, we seek the set of markings

$$\{\mu' : \exists\langle\mu,\tau\rangle \in \mathcal{S}_0, \langle\mu,\tau@0\rangle \ [*\rangle\!\!\Rightarrow \langle\mu',\tau'@\theta\rangle,\ \theta \le \theta_f < \theta + \min_{t\in\mathcal{E}}\{\tau'[t]\}\}\,.$$

For ITPN models, this section presents an efficient solution that uses symbolic manipulations. We stress that, while the problem statement focuses on *markings*, the reachability algorithms must process *states*, since the remaining firing time information is essential in determining the future evolution of the ITPN.

5.2.1 Untimed Petri nets

First, we recall symbolic state-space generation approaches for untimed Petri nets under interleaving semantics. These nets correspond to ignoring the $\mathcal{F}$ component in the ITPN, thus identifying the state with the marking. We can think that each transition is an event, then apply state-space generation in Ch. 4.

The *transition relation* due to transition $t \in \mathcal{E}$ is $\mathcal{T}_t = \{(\mu, \mu') : \mu\ [t\rangle\!\!\Rightarrow \mu'\}$ and the *overall* transition relation is $\mathcal{T} = \bigcup_{t \in \mathcal{E}} \mathcal{T}_t$. The set of reachable markings $\mathcal{M} \subseteq \mathbb{N}^{\mathcal{P}}$ is the minimal set satisfying $\mathcal{M} \supseteq \mathcal{M}_0$ and $\mathcal{M} \supseteq Img(\mathcal{M}, \mathcal{T})$, where $Img(\mathcal{Y}, \mathcal{Z}) = \{\mu' : \exists \mu \in \mathcal{Y}, (\mu, \mu') \in \mathcal{Z}\}$ denotes the *image* of the set of markings $\mathcal{Y}$ under the binary relation $\mathcal{Z}$. We can start from $\mathcal{M}_0$ and repeatedly perform image computations, breadth-first or saturation, under $\mathcal{T}$, or $\mathcal{T}_t$ as long as each of them is applied often enough, until reaching a fixpoint.

We use an MDD on $\mathbf{x} = (x_L, ..., x_1)$, with $L = |\mathcal{P}|$ to encode $\mathcal{Y}$, where each variable encodes number of tokens in a different place. In practice, it turns out that it is more effective to reduce the number of variables by grouping multiple places together using a heuristic guaranteed not to increase the size of the MDD [20], but we do not consider this optimization for ease of exposition. In the approach we follow, we assume that a set of markings $\mathcal{Y}$ is encoded by an MDD defined on $\mathbf{x}$, with $L = |\mathcal{P}|$. The transition relation $\mathcal{T}_t$ is instead encoded by a $2L$-variable MDD defined on $(\mathbf{x}, \mathbf{x}') = (x_L, x'_L, ..., x_1, x'_1)$. As firing transition t only affects or is affected by its input and output places, we denote with *top*(t)

the maximum variable according to "$\succ$" on which $\mathcal{T}_t$ depends. Once $\mathcal{Y}$ and $\mathcal{Z}$ are encoded as MDDs, the MDD encoding $Img(\mathcal{Y}, \mathcal{Z})$ is obtained as the relational product of these two MDDs. As in Ch. 4, we use reduction rule vector $\boldsymbol{\rho}^{Q}$ for $\mathcal{Y}_t$ and $\boldsymbol{\rho}^{QFI}$ for $\mathcal{T}$, to maintain a compact encoding and maximize the utilization of the cache.

5.2.2 Preliminaries

We now analyze timed reachability for ITPNs. We use a global variable θ to keep track of time. If the ITPN is in tangible state $\langle \mu, \tau \rangle$ at time $\theta = 0$ and $\tau_* = \min_{t \in \mathcal{E}}\{\tau(t)\}$, the set $\mathcal{S}_{\theta_f}$ of tangible states in which the ITPN can be at time θ_f is determined as follows:

- If $\theta_f < \tau_*$, the ITPN will still be in the same marking at time θ_f.
- If $\theta_f = \tau_*$, the ITPN reaches vanishing state $\langle \mu, \tau - \tau_* \not@ \tau_* \rangle$ at time θ_f, then immediately moves to a tangible state by firing a nondeterministically chosen $\mathcal{E}_f$, i.e., $\mathcal{S}_{\theta_f} = \{\langle \mu', \tau' \rangle : \langle \mu, \tau @ 0 \rangle \; [\tau_*]\!\!\Rightarrow \langle \mu, \tau - \tau_* \not@ \tau_* \rangle \; [\mathcal{E}_f]\!\!\Rightarrow \langle \mu', \tau' \rangle$.
- If $\theta_f > \tau_*$, repeat these steps starting from each state in $\mathcal{S}_{\tau_*}$ at time $\theta = \tau_*$.

The first case is trivial, as it generates no new markings. For the second case, we need to generate the set of tangible states $\langle \mu', \tau' \rangle$ reachable from the vanishing state $\langle \mu, \tau - \tau_* \rangle$ by firing all possible $\mathcal{E}_f$. This is a fixpoint iteration analogous to the one for state-space generation of untimed nets, but restricted to applying each transition in $\mathcal{E}(\langle \mu, \tau - \tau_* \rangle)$ at most once. Furthermore, unlike ordinary state-space generation, we are only interested in collecting the *tangible frontier*, i.e., the states where any transition in $\mathcal{E}(\langle \mu, \tau - \tau_* \rangle)$ that has not yet been fired is now marking-disabled. The key ideas to apply the symbolic state-space generation algorithms of Sec. 4.1.3, in particular saturation, are to treat a remaining firing time of 0 as part of the enabling condition, so that only transitions in $\mathcal{E}(\langle\mu,\tau-\tau_*\rangle)$ can be enabled, and to temporarily set the remaining firing time of any fired transitions to ∞, thus ensuring they cannot fire twice.

5.2.3 Algorithm

From the above analysis, when $\theta = 0$, the model cannot change to a new marking before $\theta = \tau_b$, where we call $\tau_b = \min_{t\in\mathcal{E},\langle\mu,\tau\rangle\in\mathcal{S}_0}\{\tau(t)\}$ a *breakpoint*. At $\theta = \tau_b$, the model moves to a vanishing state, and we collect the set of possible states at $\theta = \tau_b$ immediately before the firing, which includes those vanishing states $\mathcal{S}^-_{\tau_b} = \{\langle\mu^-,\tau^-\rangle : \exists\langle\mu,\tau\rangle \in \mathcal{S}_0, \mu^- = \mu, \tau^- = \tau - \tau_b\}$. Then, we apply a fixpoint image computation *FixImg* on $\mathcal{S}^-_{\tau_b}$ using a modified version of the transition relations $\mathcal{T}_t$, so that $(\langle\mu,\tau\rangle, \langle\mu',\tau'\rangle) \in \mathcal{T}_t$ iff:

- $t \in \mathcal{E}(\langle\mu,\tau\rangle)$, i.e., t is marking-enabled in μ and $\tau(t) = 0$.
- $\mu\ [t\rangle\!\!\Rightarrow \mu'$, i.e, the firing of t in μ leads to μ'.
- $\tau'(t) = \infty$ and $\tau'(t') = \tau(t')$ for $t' \neq t$, which ensures that t can only fire once and its firing does affect the remaining time of other transitions.

Consider a state $\langle\mu^-,\tau^-\rangle \in \mathcal{S}^-_{\tau_b}$; if it is tangible, it is unaffected by the above fixpoint image computation, since no transition is state-enabled in it, i.e., $\mathit{FixImg}(\langle\mu^-,\tau^-\rangle) = \{\langle\mu^-,\tau^-\rangle\}$; if it is vanishing, $\mathit{FixImg}(\langle\mu^-,\tau^-\rangle)$ instead contains all states $\langle\mu',\tau'\rangle$ satisfying: (1) there is a subset $\mathcal{E}_f \subseteq \mathcal{E}(\langle\mu^-,\tau^-\rangle)$ such that $\mu^-\ [\mathcal{E}_f\rangle\!\!\Rightarrow \mu'$, (2) $\tau'[t] = \tau^-[t]$ if $t \notin \mathcal{E}_f$, (3) $\tau'[t] = \infty$ if $t \in \mathcal{E}_f$. Thus, $\mathit{FixImg}(\langle\mu^-,\tau^-\rangle)$ "almost" contains our desired tangible frontier, except for two problems. First, it also contains all intermediate vanishing states, recognized by having at least one zero component in the remaining firing time for a marking-enabled transition. Second, the remaining firing times of a tangible frontier state $\langle\mu',\tau'\rangle$ might be incorrect, as some transition marking-enabled in μ^- might be disabled in μ', or vice versa; these must be updated in a separate step.

5.2.4 Symbolic implementation

Following Sec. 5.2.1, we encode a set of states using an MDD over the L variables $(y_{|\mathcal{E}|}, ..., y_1, x_{|\mathcal{P}|}, ..., x_1)$, where y_t is used to encode the remaining firing time of transition t and x_p is used to encode the number of tokens in place p. The special value ∞ is stored in practice as the value $\max \mathcal{F} + 1$, although in the pseudocode we still write ∞ for clarity. The transition relation $\mathcal{T}_t$ is then encoded by a 2L-variable MDD.

Fig. 5.3 and Fig. 5.4 shows the pseudocode to solve the timed reachability problem. Procedure *TimedReach* computes $enc(\mathcal{S}_{\theta_f})$ from $\mathcal{S}_0$ by iteratively advancing to the next breakpoint, given the MDD encoding the current set of states encoded by MDD s at time θ. It first computes the minimal remaining time τ_b using procedure *MinNonzeroIndex*, then subtracts it from all remaining firing times τ of the states in $\mathcal{B}(s)$ to obtain $enc(\mathcal{S}^-_{\tau_b})$ using procedure *Elapse*, and increases θ by the same amount. If τ_b is ∞, this means that the model can only be in dead states by time θ. Next, it builds $enc(\mathcal{S}^+_{\tau_b})$ through a fixpoint image computation, using procedure *Saturate* discussed in Ch. 4. Then, it calls procedure *Reset* to reset the remaining firing times for all states in $\mathcal{S}^+_{\tau_b}$ and, finally, it calls procedure *ElimVan* to eliminate all vanishing states. The global computation ends when the (next) value of θ exceeds θ_f. At that point, $\mathcal{B}(s)$ encodes all the *states* in which the ITPN can be at time θ_f; the call *GetMarkings*(s) strips the remaining firing time information from s, and returns the MDD encoding the *markings* in which the ITPN can be at time θ_f.

The pseudocode assumes that the following MDDs have been generated prior to calling *TimedReach*, for $t \in \mathcal{E}$: the L-variable MDDs $s_{enabled(t)}$, encoding the set of states where t is marking-enabled, the $2L$-level MDD $r_{resample(t)}$, which changes $y_t = \infty$ into $y'_t \in \mathcal{F}_t$, and the $2L$-level MDD $r_{reset(t)}$, which sets $y'_t = \infty$ regardless of the value of y_t.

5.3 Earliest Reachability

The earliest reachability problem aims at finding the minimum time at which each reachable marking of the ITPN is entered. Formally, we seek the function

$$\epsilon : \mathbb{N}^{\mathcal{P}} \to \mathbb{N} \cup \{\infty\} \text{ satisfying } \epsilon(\mu') = \min\left\{\theta : \exists\langle\mu,\tau\rangle \in \mathcal{S}_0, \langle\mu,\tau@0\rangle \ [*\rangle\!\!\Rightarrow \langle\mu',\tau'@\theta\rangle\right\},$$

where the entries of τ' are strictly positive (possibly infinite), and $\epsilon(\mu') = \infty$ if the marking is unreachable (thus, this also gives us the reachable markings).

This is similar to finding the shortest path in reachability graphs and can be solved by exploring all markings and keeping track of the minimum time at which we first saw each reachable state, by comparing the original minimum time and the new time when it is reached. Following Sec. 5.2, we know the model can only reach a new marking at some breakpoint $\theta = \tau_b$. Thus we let the model evolve as for timed reachability and update the minimum time for each marking reached at every breakpoint. Unlike timed reachability, however, we need to generate *all* reachable markings, and stop only when no new marking

```
mdd TimedReach(integer θ_f)
1 $\theta \leftarrow 0$;  $s \leftarrow enc(\mathcal{S}_0)$;
2 repeat forever
3   $\tau_b \leftarrow MinNonzeroIndex(s)$;            • compute the next breakpoint
4   if $\tau_b = \infty$ then return $\mathbf{0}$;     • no transition is marking-enabled
5   $\theta \leftarrow \theta + \tau_b$;  $s \leftarrow Elapse(s, \tau_b)$;   • advance to the next breakpoint
6   if $\theta > \theta_f$ then return $GetMarkings(s)$;
7   $s \leftarrow Saturate(s)$;
8   $s \leftarrow Reset(s)$;                           • reset remaining firing times
9   $s \leftarrow ElimVan(s)$;
```

Figure 5.3: Timed reachability algorithm.

is found. To do so, we accumulate the states $\mathcal{S}^-_{\tau_b}$ at every breakpoint, including vanishing states, until we reach a fixpoint, ensuring that no new marking can be reached in the future evolution of the ITPN.

EV$^+$MDDs are very efficient for the joint computation of reachable states and their distance, using the *Minimum* operation of Sec. 2.3. We adopt them to encode the function ϵ we seek, so that $\epsilon(\mu)$ records the first time marking μ was encountered; by default, $\epsilon(\mu) = \infty$ for any μ not yet found in the exploration.

Fig. 5.5 shows the procedure to compute the EV$^+$MDD $\langle\rho,u\rangle$ encoding the desired earliest reachability function ϵ. It proceeds as for timed reachability, except that we extract the set $\mathcal{M}$ of markings that are part of tangible states after advancing to each breakpoint, with a call to *GetMarkings*. Then, $\mathcal{M}$, encoded in an MDD that only refers to the variables $x_{|\mathcal{P}|}, ..., x_1$, is used to build the EV$^+$MDD encoding the function defined by $f_{\langle\rho_c,u_c\rangle}(\mu) = \theta$, the current time, if $\mu \in \mathcal{M}$, and $f_{\langle\rho_c,u_c\rangle}(\mu) = \infty$ otherwise. After that, a call to *Minimum*

```
$integer\ MinNonzeroIndex(mdd\ s)$                                    • assume $s.var = x_k$
 1 $m \leftarrow \min\{i \in \mathcal{X}_k : s[i] \neq \mathbf{0}\}$ ;
 2 if $s.var = y_1$ then return $m$;                   • no need to visit nodes associated to places
 3 if $CacheHit(MNI,s,m)$ then return $m$;
 4 foreach $i \in \mathcal{X}_k$ s.t. $s[i] \neq \mathbf{0}$ do
 5   $u \leftarrow MinNonzeroIndex(s[i])$;  $m \leftarrow \min\{m, u\}$;
 6 $CacheAdd(MNI, s, m)$;
 7 return $m$;
```

```
$mdd\ Elapse(mdd\ s, integer\ b)$                                     • assume $s.var = x_k$
 1 if $s.var = x_{|\mathcal{P}|}$ then return $s$;                  • no need to visit nodes associated to places
 2 if $CacheHit(ELS, s, b, u)$ then return $u$;
 3 $u \leftarrow NewNode(x_k)$;
 4 foreach $i \in \mathcal{X}_k$ s.t. $s[i] \neq \mathbf{0}$ do $u[i-b] \leftarrow Elapse(s[i], b)$;
 5 $CacheAdd(ELS, s, b, UTInsert(u))$;
 6 return $q$;
```

```
$mdd\ Reset(mdd\ s)$
 1 foreach $t \in \mathcal{E}$ do
 2   $s_{enb} \leftarrow Intersect(s, s_{enabled(t)})$;         • states in $\mathcal{B}(s)$ where $t$ is marking-enabled
 3   $s_{dis} \leftarrow Difference(s, s_{enb})$;             • states in $\mathcal{B}(s)$ where $t$ is marking-disabled
 4 $u_1 \leftarrow RelProd(s_{enb}, r_{resample(t)})$;  $u_2 \leftarrow RelProd(s_{dis}, r_{reset(t)})$;
 5 return $Union(u_1, u_2)$;
```

```
$mdd\ ElimVan(mdd\ s)$                                                • assume $s.var = x_k$
 1 if $s.var = x_{|\mathcal{P}|}$ then return $s$;                  • no need to visit nodes associated to places
 2 if $CacheHit(ELV, s, u)$ then return $u$;
 3 $u \leftarrow NewNode(x_k)$;
 4 foreach $i \in \mathcal{X}_k \backslash \{0\}$ do $u[i] \leftarrow ElimVan(s[i])$;
 5 $CacheAdd(ELV, s, UTInsert(u))$;
 6 return $u$;
```

Figure 5.4: Timed reachability algorithm (continued).

ensures that, if some new marking μ was found, its earliest reachability time is recorded in the EV$^+$MDD $\langle\rho,u\rangle$, overwriting the old value $f_{\langle\rho,u\rangle}(\mu) = \infty$. The MDD g is used to accumulate all states before fixpoint image computation at every breakpoint. The global computation halts when g reaches a fixpoint or if the model has reached only dead states.

5.4 Decidability, Complexity, Related Research

Decidability. It is well known that the *marking reachability problem*: "given marking $\mu, \exists\langle\mu,\tau\rangle \in \mathcal{S}$?" is undecidable for arbitrary time Petri nets [40] but decidable for token-bounded TPNs [10], although boundedness is itself undecidable for TPNs. For ITPNs, the problem is different, given their essentially discrete nature. Nevertheless, we can show that their integer firing times can be used to enforce priorities between transitions, hence achieve the "test-for-zero" of counter machines, and Turing-equivalence. Thus, (earliest) reachability is undecidable for ITPNs (it is of course decidable for bounded ITPNs but, again, boundedness itself is not). Of course, timed reachability is instead decidable, since only a finite number of markings can be reached in a finite time horizon, given that all firing times are positive integers.

Complexity. The time complexity is $N_{\tau_b} \cdot T_s$, where N_{τ_b} counts the breakpoints and T_s is the average complexity for the work performed at each breakpoint. When $\mathcal{F}$ contains more firing times, the number of markings $|\mathcal{M}|$ tends to grow and approach the number of reachable markings in the underlying untimed PN.

Related work. [46] proposes a symbolic state-space generation for FIHTPNs, a type of TPNs with different semantics from our ITPNs. They translate the TPN into an untimed

```
idd EarliestReach()
 1 $\langle\rho,u\rangle \leftarrow MakeEvmdd(0, enc(\mathcal{M}_0))$          • initialize the EV$^+$MDD encoding $\epsilon$
 2 $s \leftarrow enc(\mathcal{S}_0)$;   $\tau_b \leftarrow MinNonzeroIndex(s)$;   $s \leftarrow Elapse(s, \tau_b)$;
 3 $g \leftarrow s$;                                                   • used to recognize the fixpoint
 4 repeat
 5   $s \leftarrow Saturate(s)$;   $s \leftarrow Reset(s)$;   $s \leftarrow ElimVan(s)$;
 6   $m \leftarrow GetMarkings(s)$;                                    • an MDD on $x_{|\mathcal{P}|}, ..., x_1$
 7   $\langle\rho_c,u_c\rangle \leftarrow MakeEvmdd(\theta, m)$;   $\langle\rho,u\rangle \leftarrow Minimum(\langle\rho,u\rangle, \langle\rho_c,u_c\rangle)$;
 8   $\tau_b \leftarrow MinNonzeroIndex(s)$;   $s \leftarrow Elapse(s, \tau_b)$;   $g \leftarrow Union(g, s)$);
 9 until $g$ does not change;
10 return $\langle\rho,u\rangle$;
```

Figure 5.5: Earliest reachability algorithm.

PN by defining a subnet for each timed transition, combine these subnets, and perform breadth-first symbolic state-space generation on the resulting PN. Our approach does not require this translation step and, more importantly, does not increment a counter (a PN place) one unit at a time; instead, we advance the global clock by the minimal remaining firing time τ_b.

5.5 Experimental Results

Running model. Fig. 5.6 shows the MDDs and EV$^+$MDDs built during timed and earliest reachability computation of our running model. Each row corresponds to a different time point θ and each column corresponds to a different stage of the iteration. Sets of states are encoded by a 6-variable MDD and the earliest reachability function is encoded by a 3-variable EV$^+$MDD. We start reachability computation from $enc(\mathcal{S}_0)$, at time $\theta = 0$, then

$enc(\mathcal{S}_\theta^-)$ is obtained after *Elapse* is called to move to the breakpoint $\tau_d = 1$, then *Saturation* is called on it to obtain $enc(\mathcal{S}_\theta^+)$, then *Reset* and *ElimVan* are called on the MDD encoding $\mathcal{S}_\theta$, the set of tangible states at time θ. Then *GetMarkings*, *MakeEvmdd*, and *Minimum* $enc(\mathcal{S}_\theta)$ generate the EV$^+$MDD encoding the earliest reachability time of each reachable marking found up to time θ. For $\mathcal{S}_\theta^-$, we use a white box in the node to emphasize when a transition has remaining firing time 0, i.e., when it is state-enabled. Our ITPN model can evolve up to $\theta = 6$, at which time it can only be in a dead state. The fixpoint for earliest reachability is obtained at time $\theta = 4$, not shown in the figure. Thus, at time $\theta = 3$, i.e., the last EV$^+$MDD gives the encoding of the earliest reachability times for *all* markings.

We implemented our algorithms in our model checker [17] and ran them on an Intel Xeon 3.0Ghz workstation with 16GB RAM under SuSE Linux 9.1.

Tab. 5.1 shows the running time and peak memory consumption to compute timed reachability (TR) at $\theta_f = 5$ or 100 and earliest reachability (ER), on a variety of models (referenced in the table). The possible firing times of each transition in each model are $\{1, 2, 3, 4, 5\}$. Parameter N denotes the initial number of tokens in certain places or the number of repeated subnets, and affects the size of state space. We record the number of reachable *states* at time θ_f (for TR) and of overall reachable *markings* (for ER). From the table, we can see that our algorithms can generate large set of states (up to 10^{38}) for TR and markings (up to 10^{55}) for ER with a small time and memory requirements, which demonstrates the applicability and efficiency of symbolic methods on state-space generation of synchronous timed systems, not just to untimed asynchronous systems, where they have already been extensively applied.

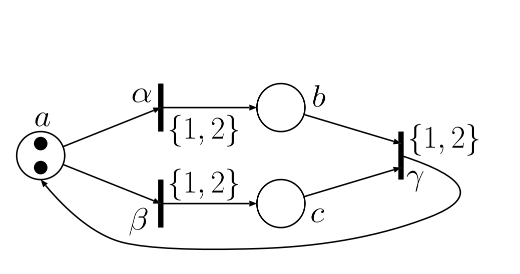
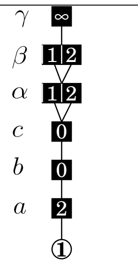
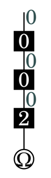
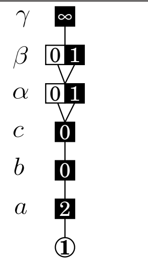
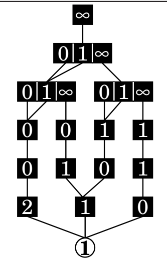
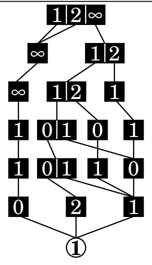
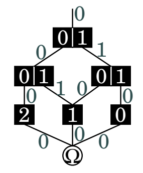
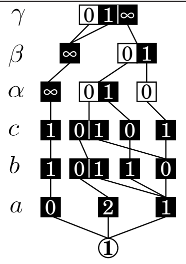
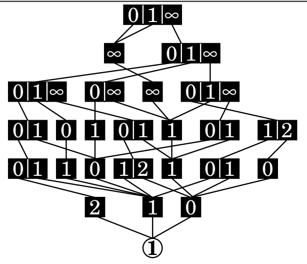
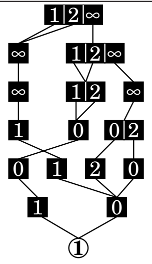
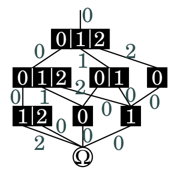
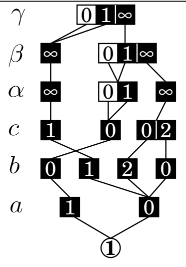
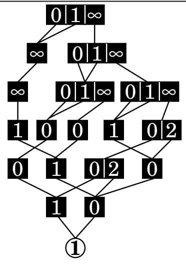
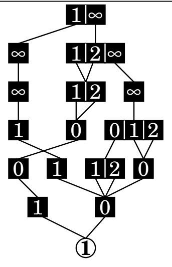
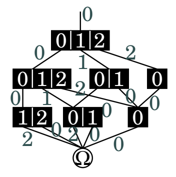
θ	$\mathcal{S}_\theta^-$	$\mathcal{S}_\theta^+$	$\mathcal{S}_\theta$	$\mathcal{M}_\theta$
0				
1				
2				
3				

Figure 5.6: MDDs and EV$^+$MDDs for timed and earliest reachability of our running model.

5.6 Conclusion

We considered two fundamental problems for a class of timed Petri net models where events have integer but not necessarily constant durations: timed reachability and earliest reachability. For these, we provided detailed symbolic algorithms that, through the use of both ordinary and edge-valued decision diagrams, can quickly explore very large state spaces. These algorithms are non-trivial extensions of known symbolic reachability algorithms for untimed nets, showing that the potential of these approaches goes well beyond strict logical analysis.

As a first extension of the present work, we envision exploring decision diagrams variants that can better tackle models with large ranges of durations.

		TR $\theta_{fin} = 5$			**TR** $\theta_{fin} = 100$			**ER**		
Model	N	$\lvert\mathcal{S}\rvert$	time	mem	$\lvert\mathcal{S}\rvert$	time	mem	$\lvert\mathcal{M}\rvert$	time	mem
bittoggle	10	$4.14 \cdot 10^{7}$	0.13	4.36	$1.00 \cdot 10^{8}$	1.42	34.28	$1.02 \cdot 10^{3}$	1.46	35.46
[39]	15	$2.27 \cdot 10^{11}$	0.91	21.90	$7.83 \cdot 10^{11}$	91.57	1415.63	-	-	-
bqueue	5	$2.37 \cdot 10^{7}$	1.08	24.53	$1.82 \cdot 10^{8}$	22.02	408.38	$1.58 \cdot 10^{4}$	22.47	414.68
[55]	10	$2.99 \cdot 10^{7}$	1.13	24.31	-	-	-	-	-	-
bubsort	5	$7.50 \cdot 10^{4}$	0.01	0.37	$7.50 \cdot 10^{4}$	0.01	0.37	$1.20 \cdot 10^{2}$	0.01	0.37
[28]	8	$3.15 \cdot 10^{9}$	5.72	51.29	$3.15 \cdot 10^{9}$	5.70	51.29	$4.03 \cdot 10^{4}$	6.06	29.89
fms	2	$3.42 \cdot 10^{7}$	2.78	58.41	$3.97 \cdot 10^{7}$	10.20	193.06	$3.44 \cdot 10^{3}$	10.29	195.95
[50]	4	$1.12 \cdot 10^{11}$	103.04	919.49	-	-	-	-	-	-
kanban	2	$1.86 \cdot 10^{7}$	0.33	8.44	$3.87 \cdot 10^{9}$	33.50	555.02	$4.60 \cdot 10^{3}$	62.19	353.39
[50]	4	$1.50 \cdot 10^{9}$	2.56	49.98	-	-	-	-	-	-
intshift	20	$5.30 \cdot 10^{2}$	0.01	0.23	$3.83 \cdot 10^{17}$	0.33	6.12	$4.02 \cdot 10^{16}$	0.37	6.51
[28]	70	$5.30 \cdot 10^{2}$	0.02	0.54	$2.84 \cdot 10^{26}$	0.72	17.43	$3.25 \cdot 10^{55}$	10.09	82.57
phils	5	$3.75 \cdot 10^{7}$	4.76	95.54	$3.77 \cdot 10^{7}$	7.33	146.35	$1.36 \cdot 10^{3}$	7.47	149.22
[50]	6	$1.22 \cdot 10^{9}$	28.89	503.04	$1.23 \cdot 10^{9}$	52.16	873.54	-	-	-
queen	10	$2.61 \cdot 10^{6}$	13.89	248.98	$1.27 \cdot 10^{4}$	30.71	519.94	$3.55 \cdot 10^{4}$	27.63	460.29
[34]	11	$1.41 \cdot 10^{7}$	35.08	594.04	-	-	-	$1.66 \cdot 10^{5}$	81.74	1300.12
robin	20	$8.07 \cdot 10^{2}$	0.08	2.84	$1.59 \cdot 10^{4}$	14.40	263.12	$4.00 \cdot 10^{2}$	13.95	259.11
[28]	30	$8.07 \cdot 10^{2}$	0.15	5.01	$2.38 \cdot 10^{4}$	57.22	894.75	$6.00 \cdot 10^{2}$	56.54	883.04
slot	4	$2.20 \cdot 10^{7}$	5.18	112.20	$2.55 \cdot 10^{7}$	13.71	282.11	$5.13 \cdot 10^{3}$	13.95	288.50
[28]	5	$1.51 \cdot 10^{9}$	80.21	1316.79	-	-	-	-	-	-
swapper	10	$2.20 \cdot 10^{18}$	1.25	26.25	$3.20 \cdot 10^{18}$	1.96	39.79	$1.67 \cdot 10^{5}$	2.04	28.01
[28]	20	$4.75 \cdot 10^{35}$	49.82	250.60	$2.38 \cdot 10^{38}$	126.43	250.60	$1.31 \cdot 10^{11}$	123.62	127.51

Table 5.1: Experimental results (time in sec, mem in MB); "–" means time > 600sec.

Chapter 6

Multiplicative Edge-valued Decision Diagrams

Kronecker-based encodings of transition rate matrices have been proposed to compute the *exact* or *approximate* stationary solution of ergodic CTMCs. These encodings are compact and effective but only applicable to models with a Kronecker decomposition, i.e., when we can express the transition rate matrix as the sum (over all events) of the Kronecker product (over all state variables) of *local* matrices [5, 12, 36]. While in principle this Kronecker form always exists, in practice it might be achieved only by splitting model events, resulting in an excessive number of events, or by merging state variables, resulting in excessively large local state spaces. To overcome this restriction and make our approach completely general, we propose a new class of edge-valued decision diagrams as an alternative to Kronecker matrices, namely multiplicative edge-valued multi-way decision diagrams (EV*MDDs) to retain their memory efficiency while working with arbitrary model decompositions.

This chapter is organized as follows. Sec. 6.1 introduces our new class of edge-valued decision diagrams and discusses their canonicity. Sec. 6.2 presents algorithms to canonize and manipulate them. Sec. 6.3 compares this new data structure with Kronecker matrices. Sec. 6.4 concludes the chapter.

6.1 A new class of edge-valued decision diagrams

6.1.1 Definition of EV*MDD

We now introduce the multiplicative edge-valued multi-way decision diagrams (EV*MDDs). Given L variables $\mathbf{x} = (x_L,..,x_1)$ with an order $x_L \succ ... \succ x_1$, each x_k taking value in a finite set $\mathcal{X}_k = \{0, 1, ..., n_k\} \subset \mathbb{N}$, an EV*MDD defined on $\mathbf{x}$ encodes a function of the form $f : \mathcal{X}_L \times ... \times \mathcal{X}_1 \to [0, +\infty)$ by associating values to the edges of the diagram. Formally, a (quasi-reduced) EV*MDD defined on $\mathbf{x}$ is a directed acyclic edge-labeled multi-graph where:

- Each nonterminal node p is associated to a variable $p.var = x_k \in \{x_L, ..., x_1\}$.
- Ω is the only *terminal* node. Let $\Omega.var = x_0$ and $x_k \succ x_0$, for $L \geq k \geq 1$.
- Each nonterminal node p associated with x_k has $|\mathcal{X}_k|$ edges, labeled with a different index $i \in \mathcal{X}_k$ and associated with a value in $[0, 1]$. We write $p[i] = \langle\omega,q\rangle = \langle p[i].v,p[i].d\rangle$ if the edge labeled by i points to node q and is associated with value ω, and require that $q = \Omega$ if $\omega = 0$ and $q.var = x_{k-1}$ otherwise. Also, at least one edge leaving the node must have an associated value equal to 1.

- There is a single *root* node associated to x_L, with an incoming *dangling* edge. If the root node is r and the dangling edge having associated value $\omega \in (0, +\infty)$, then the EV*MDD can be identified by $\langle\omega,r\rangle$.

- *Duplicate* nodes are not allowed: given two distinct nonterminal nodes p and q with $p.var = q.var = x_k$, there must be an index $i \in \mathcal{X}_k$ such that $p[i] \neq q[i]$.

Again, we can extend the edge notation to paths so that the node reached from p through a tuple $\alpha = (i_k, i_{k-1}, ..., i_h) \in \mathcal{X}_k \times ... \times \mathcal{X}_h$, for $L \geq k \geq h \geq 1$, is defined recursively as

$$p[\alpha].d = \begin{cases} (p[i_k].d)[i_{k-1}, ..., i_h].d & \text{if } p[i_k].d \neq \Omega, \\ \Omega & \text{otherwise,} \end{cases}$$

and the value associated to this path is

$$p[\alpha].v = \begin{cases} p[i_k].v \cdot p[i_{k-1}, ..., i_h].v & \text{if } p[i_k].v \neq 0, \\ 0 & \text{otherwise.} \end{cases}$$

Each edge $\langle\omega,r\rangle$ with $r.var = x_k \succ x_0$ encodes a function $f(\alpha) = \omega \cdot r[\alpha].v$ for any $\alpha \in \mathcal{X}_k \times ... \times \mathcal{X}_1$; The function encoded by edge $\langle\omega,p\rangle$, with $p.var = x_k$, is recursively defined by

$$f_{\langle\omega,p\rangle}(i_k, ..., i_1) = \begin{cases} \omega & \text{if } r = \Omega \\ \omega \times f_{p[i_k]}(i_{k-1}, ..., i_1) & \text{otherwise} \end{cases}$$

for any $(i_k, ..., i_1) \in \mathcal{X}_k \times ... \times \mathcal{X}_1$.

EV*MDDs can be considered the multiplicative counterpart of EV$^+$MDDs [25] and they provide a canonical encoding for functions of the form $f : \mathcal{X}_L \times ... \times \mathcal{X}_1 \to \mathbb{R}_{\geq} 0$. The proof of canonicity is in the Sec. 6.1.3 and we write the EV*MDD encoding f as $enc(f)$.

The root node is always associated with x_L, except for the constant function identically equal to 0, for which we can simply allow the special case $\langle 0,\Omega\rangle$.

6.1.2 Running example

We will use the running model in Ch. 2. Fig. 6.1 shows, on the left, the CTMC for our running example and, on the right, its EV*MDD encoding (only edges with nonzero value are displayed, and we hide all edge values except for a path). The path, highlighted in white, corresponds to $\mathbf{R}[0000, 0111] = 8 \cdot 1 \cdot 0.5 \cdot \mathbf{1} \cdots \mathbf{1} = 4.0$. Note that, this EV*MDD contains some "fictitious" rates, as it in fact encodes a *potential* transition rate matrix, i.e., $\mathbf{R}[\mathbf{i}, \mathbf{i}'] > 0$ does not necessarily imply $\mathbf{i} \in \mathcal{S}$.

6.1.3 Proof of canonicity

To prove that EV*MDDs are canonical representations, we must show that any non-negative function can be represented as an EV*MDD and this EV*MDD is unique.

The first part is proved by given a construction of the EV*MDD encoding an arbitrary non-negative function defined on a set of variables, we delay this part to Sec. 6.2.1 for EV*MDD operations. We first prove that this encoding is unique.

Theorem 6.1.1. *Given a tuple of variables* $\mathbf{x}$ *and functions* f, g *defined on* $\mathbf{x}$*, let* $f_{\langle \omega_f, r_f \rangle} = f, f_{\langle \omega_g, r_g \rangle} = g$ *then* $\langle \omega_f, r_f \rangle = \langle \omega_g, r_g \rangle \Leftrightarrow f \equiv g$.

Proof. $\Rightarrow$ is obvious. We only need to prove $\Leftarrow$. Since when $f \equiv g \equiv 0$, $\langle \omega_f, r_f \rangle = \langle \omega_g, r_g \rangle = \langle 0,\Omega\rangle$ and when $\mathbf{x}$ is an empty tuple $()$, $\langle \omega_f, r_f \rangle = \langle \omega_g, r_g \rangle = \langle f,\Omega\rangle$, we only need to consider the case $f \not\equiv 0$ and $\mathbf{x} \neq ()$.

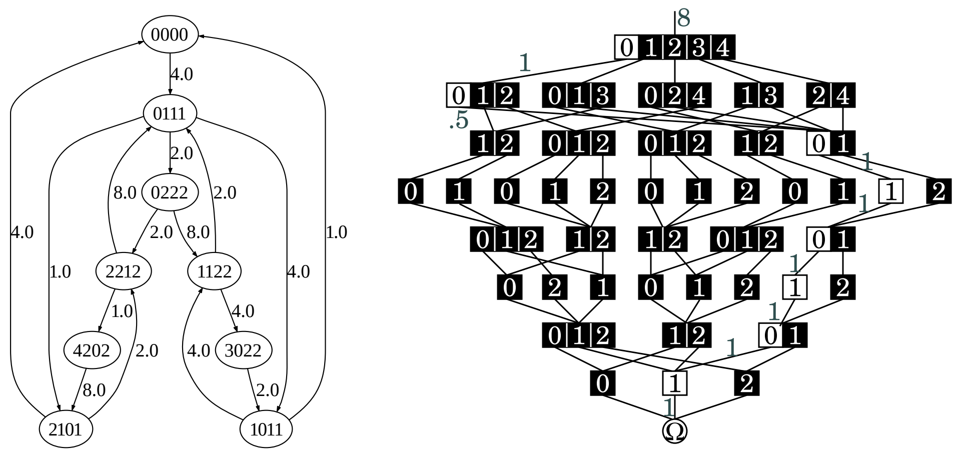


Figure 6.1: Running example: exact CTMC and the EV*MDD encoding it.

First, from the definition we have $\omega_f = \max(f)$ and $\omega_g = \max(g)$, so $\omega_f = \omega_g$.

Second, we now prove $r_f = r_g$. For the basis of the induction, when f is defined on just one variable, i.e., $L = 1$ and $\mathbf{x} = (x_1)$, for any $i \in \mathcal{X}_1$, $f(i) = \omega_f \cdot r_f[i].v = g(i) = \omega_g \cdot r_g[i].v \Rightarrow r_f[i].v = r_g[i].v$, and $r_f[i].d = r_g[i].d = \Omega$, so $r_f = r_g$. For the inductive step, assume the theorem holds when $L = l$ for some $l \geq 1$ and, then show it holds when $L = l+1$. For any tuple $\alpha = (i_{l+1}, i_l, ..., i_1) \equiv (i_{l+1}, \alpha') \in \mathcal{X}_{l+1} \times \mathcal{X}_l \times ... \times \mathcal{X}_1$, let f' be the function encoded by $r_f[i_{l+1}]$ and g' be the function encoded by $r_g[i_{l+1}]$, we have

$$f(\alpha) = \omega_f \cdot r_f[\alpha].v \equiv \omega_f \cdot r_f[i_{l+1}, \alpha'].v = \omega_f \cdot r_f[i_{l+1}].v \cdot (r_f[i_{l+1}].d)[\alpha'].v \equiv \omega_f \cdot f'(\alpha').$$

Similarly $g(\alpha) = \omega_g \cdot g'(\alpha')$. We already know $\omega_f = \omega_g$, so $f' \equiv g'$.

- if $f' \equiv 0$, we must have $r_f[i_{l+1}].v = 0$. Otherwise $(r_f[i_{l+1}].d).var = x_l \succ x_0$. Since each non-terminal node contains at least one edge valued 1, there exists at least one path $\beta \in \mathcal{X}_l \times ... \times \mathcal{X}_1$ such that $(r_f[i_{l+1}].d)[\beta].v = 1 \neq 0$, so $f'(\beta) = r_f[i_{l+1}].v > 0$; we get contradiction. So $r_f[i_{l+1}] = \langle 0,\Omega\rangle$. Similarly $r_g[i_{l+1}] = \langle 0,\Omega\rangle$, so $r_f[i_{l+1}] = r_g[i_{l+1}]$.

- if $f' \not\equiv 0$, since f' is an l-variable function, from the assumption of the induction, $f' \equiv g' \Rightarrow r_f[i_{l+1}] = r_g[i_{l+1}]$.

Since i_{l+1} is arbitrarily chosen from $\mathcal{X}_{l+1}$, we can conclude that $r_f = r_g$ and the theorem is proved. $\square$

6.2 Operations with EV*MDD

6.2.1 Building the EV*MDD encoding a function

We now show how to build the EV*MDD encoding a function defined on $\mathbf{x}$ in Fig. 6.2. In the pseudocode, we use type *rdd* for real-valued EV*MDD edges.

Procedure *Build*$(f, \mathbf{x})$ reads a function f defined on $\mathbf{x}$ and outputs the canonical EV*MDD encoding it. It first constructs a non-canonical EV*MDD that allows duplicate nodes and does not require the maximum value of the edges leaving a node to be 1, by enumerating nonzero values of f and inserting them in order using procedure *AddEntry*. Then, it appropriately scales its edge values with procedure *Normalize*, to ensure that the maximum value for each node is 1, using a bottom-up strategy. Finally, procedure *Reduce* removes duplicate nodes. A special case is when $\mathbf{x}$ is an empty tuple, which indicates that f is a constant function; in this case, the pseudocode just returns $\langle f,\Omega\rangle$.

```
$rdd\ Build(expr\ f, tuple\ \mathbf{x})$
  1 if $\mathbf{x} = ()$ return $\langle f,\Omega\rangle$;                    • constant function
  2 $\langle\omega,r\rangle \leftarrow \langle 0,\Omega\rangle$;
  3 foreach $\mathbf{i} \in \mathcal{X}_{l_1} \times ... \times \mathcal{X}_{l_n}$ do
  4     if $f(\mathbf{i}) \neq 0$ then $\langle\omega,r\rangle \leftarrow AddEntry(r, \mathbf{i}, f(\mathbf{i}), \mathbf{x})$;
  5 $\omega \leftarrow Normalize(\langle\omega,r\rangle)$;
  6 return $Reduce(\langle\omega,r\rangle)$;

$rdd\ AddEntry(rdd\ \langle\omega,r\rangle, tuple\ \mathbf{i}, real\ val, tuple\ (x_{l_n}, ..., x_{l_1})$
  1 if $\langle\omega,r\rangle = \langle 0,\Omega\rangle$ then $\langle\omega,r\rangle \leftarrow \langle 1,NewNode(x_{l_n})\rangle$;
  2 if $x_{l_n} = x_{l_1}$ then $r[i_l] \leftarrow \langle val,\Omega\rangle$;         • last variable in support
  3 else $r[i_l] \leftarrow AddEntry(r[i_l], \mathbf{i}, val, (x_{l_{n-1}}, ..., x_{l_1}))$;
  4 return $\langle\omega,r\rangle$;

$real\ Normalize(rdd\ \langle\omega,r\rangle)$          • for nodes not yet checked into the unique table
  1 if $r = \Omega$ return $\omega$;                                         • terminal case
  2 $scale \leftarrow 0$;
  3 foreach $i \in \mathcal{X}_k$ s.t. $r[i] \neq \langle 0,\Omega\rangle$ do
  4     $val \leftarrow Normalize(r[i])$;
  5     $scale \leftarrow \max\{scale, val \cdot r[i].v\}$;                      • find the max value
  6 foreach $i \in \mathcal{X}_k$ s.t. $r[i] \neq \langle 0,\Omega\rangle$ do                • assume $r.var = x_k$
  7     $r[i].v \leftarrow r[i].v/scale$;                                  • divide by the max value
  8 return $scale \cdot \omega$;

$rdd\ Reduce(rdd\ \langle\omega,r\rangle)$                                         • assume $r.var = x_k$
  1 if $r = \Omega$ or $r$ is already reduced then return $\langle\omega,r\rangle$;
  2 foreach $i \in \mathcal{X}_k$ do $r[i] \leftarrow Reduce(r[i])$;
  3 return $\langle\omega,UTInsert(r)\rangle$;
```

Figure 6.2: Building the EV*MDD encoding f.

```
$rdd\ Multiply(rdd\ \langle a,p \rangle, rdd\ \langle b,q \rangle)$                • assume $p.var = x_k$
 1 if $a = 0$ or $b = 0$ return $\langle 0,\Omega \rangle$;
 2 if $p = \Omega$ then return $\langle a \cdot b,q \rangle$;
 3 if $q = \Omega$ then return $\langle a \cdot b,p \rangle$;                    • trivial cases
 4 if $q.var \succ p.var$ then $Swap(\langle a,p \rangle, \langle b,q \rangle)$;                • commutativity
 5 if $CacheHit(MULTIPLY, p, q, \langle \omega,r \rangle)$ then return $\langle a \cdot b \cdot \omega,r \rangle$;   • check cache
 6 $r \leftarrow NewNode(p.var)$; $scale \leftarrow 0$;
 7 foreach $i \in \mathcal{X}_k$ do
 8   if $p.var \succ q.var$ then $r[i] \leftarrow Multiply(p[i], \langle b,q \rangle)$;
 9   else $r[i] \leftarrow Multiply(p[i], q[i])$;                          • $p.var = q.var$
10   $scale \leftarrow \max\{scale, r[i].v\}$;
11 if $scale \neq 1$ then
12   foreach $i \in \mathcal{X}_k$ s.t. $r[i] \neq \langle 0,\Omega \rangle$ do $r[i].v \leftarrow r[i].v/scale$;   • normalize
13 $r \leftarrow UTInsert(r)$;                                    • insert into unique table
14 $CacheAdd(MULTIPLY, p, q, \langle scale,r \rangle)$;                      • add to cache
15 return $\langle a \cdot b \cdot scale,r \rangle$;
```

```
$rdd\ Add(rdd\ \langle a,p \rangle, rdd\ \langle b,q \rangle)$                     • assume $p.var = x_k$
 1 if $a = 0$ return $\langle b,q \rangle$;
 2 if $b = 0$ return $\langle a,p \rangle$;
 3 if $p = q$ return $\langle a + b,p \rangle$;                              • trivial cases
 4 if $a > b$ then $Swap(\langle a,p \rangle, \langle b,q \rangle)$;                      • commutativity
 5 $a \leftarrow a/b$;
 6 if $CacheHit(ADD, a, p, q, \langle \omega,r \rangle)$ then return $\langle b \cdot \omega,r \rangle$;       • check cache
 7 $r \leftarrow NewNode(p.var)$; $scale \leftarrow 0$;
 8 foreach $i \in \mathcal{X}_k$ do
 9   $r[i] \leftarrow Add(\langle a \cdot p[i].v,p[i].d \rangle, q[i])$;
10   $scale \leftarrow \max\{scale, r[i].v\}$;
11 if $scale \neq 1$ then
12   foreach $i \in \mathcal{X}_k$ s.t. $r[i] \neq \langle 0,\Omega \rangle$ do $r[i].v \leftarrow r[i].v/scale$;   • normalize
13 $r \leftarrow UTInsert(r)$;                                    • insert into unique table
14 $CacheAdd(ADD, a, p, q, \langle scale,r \rangle)$;                        • add to cache
15 return $\langle b \cdot scale,r \rangle$;
```

Figure 6.3: Standard EV*MDD operations.

6.2.2 The *Multiply* and *Add* operations

Fig. 6.3 shows two representative operations for EV*MDDs, *Multiply* and *Add*, which take as input two EV*MDDs $enc(f)$ and $enc(g)$ and output $enc(f \cdot g)$ or $enc(f + g)$, respectively. *Multiply* can be applied to two EV*MDDs defined on two different tuples of variables, as long as these variables can be compared according to some predefined order; the resulting EV*MDD is defined on a tuple containing all these variables. *Add* instead assume the two inputs defined on the same tuple of variables. Of course, real functions defined on different tuple of variables can also be added, but we use this assumption to avoid details of reduction rules and we will show in the following sections that this framework is adequate for our needs. An operation cache is used to make both operations efficient, as always with decision-diagram manipulations.

6.3 Comparison to Kronecker matrices

When the model is Kronecker-consistent, $\mathbf{R}$ can be encoded either with Kronecker matrices or as an EV*MDD. The Kronecker encoding is a *disjunctive* encoding where each $\mathbf{R}_e$ is encoded as the Kronecker product of local matrices. This encoding is compact because we only need to store $|spt(\mathbf{R}_e)|$ relatively small matrices for event e. EV*MDDs also support this disjunctive encoding, and we can encode each $\mathbf{R}_e$ into a $|spt(\mathbf{R}_e)|$-variable EV*MDD.

In the worst case, the disjunctive EV*MDD encoding uses twice as many edges as there are nonzeroes in the Kronecker matrices because each nonzero element in a Kronecker matrix corresponds to two edges in an EV*MDD. On the other hand, EV*MDDs allow for node sharing, which can greatly reduce the number of edges, and is the main motivation to use decision diagrams instead of decision trees.

Table 6.1 shows a comparison of the size of the Kronecker and the EV*MDD encodings mentioned above for our running example. We consider two decompositions, the finest uses one place per variable, while the Kronecker decomposition requires to put places a, d, e together. N is the initial number of tokens in place a. We can see that, when N is large and the decomposition is Kronecker-consistent, a single-root encoding tends to use fewer edges than a disjunctive encoding. With the finer decomposition, the disjunctive encoding tends to be better memory-wise and either EV*MDD encoding is better than the Kronecker encoding when L is large enough.

We can see that, under Kronecker decomposition, disjunctive EV*MDDs uses slightly fewer than twice the nonzeroes of Kronecker matrices. The single-root EV*MDD, when N is small, use more than the disjunctive representation; when N grows, it outperforms disjunctive encoding due to more node and edge sharings. Finally, when we adapt to the finest decomposition, Kronecker encoding does not apply any more, but EV*MDDs still work well and not surprisingly, use much fewer edges due to the smaller variable domains.

6.4 Conclusion

We presented a new canonical form of edge-valued decision diagrams, which can be used to encode nonnegative real functions. It retains the compactness of a Kronecker encoding, while being more general, no longer requiring a Kronecker-consistent decomposition. We also presented algorithms for the manipulation of this new data structure and

			Kronecker		**EV*MDD**					
					Disjunctive		Single root		Actual **R**	
N	$\eta(\mathbf{R})$	Decomp.	matrs	nz	nodes	edges	nodes	edges	nodes	edges
1	5	Kron	10	9	16	18	17	23	21	25
		finest	x	x	23	26	31	42	28	32
2	16	Kron	10	24	30	47	26	49	52	67
		finest	x	x	34	50	42	72	66	81
3	33	Kron	10	45	50	88	37	83	97	129
		finest	x	x	45	74	53	102	116	148
4	56	Kron	10	72	76	141	50	125	156	211
		finest	x	x	56	98	64	132	178	233
5	85	Kron	10	105	108	206	65	175	229	313
		finest	x	x	67	122	75	162	252	336
6	120	Kron	10	144	146	283	82	233	316	435
		finest	x	x	78	146	86	192	338	457
7	161	Kron	10	189	190	372	101	299	417	577
		finest	x	x	89	170	97	222	436	596
8	208	Kron	10	240	240	473	122	373	532	739
		finest	x	x	100	194	108	252	546	753
9	261	Kron	10	297	296	586	145	455	661	921
		finest	x	x	111	218	119	282	668	928
10	320	Kron	10	360	358	711	170	545	804	1123
		finest	x	x	122	242	130	312	802	1121

Table 6.1: Running Model: Matrices/nonzeros, or nodes/edges, for Kronecker and EV*MDD encodings of **R**. “x” indicates not applicable.

provided numerical examples of its effectiveness. An immediate application of EV*MDDs will be seen in the next chapter.

Chapter 7

Approximate Numerical Solution For Large Ergodic Models

Markov models are extensively used in the modeling, performance and reliability analysis of discrete-state systems. In this chapter, we consider the stationary solution of ergodic continuous-time Markov chains (CTMCs) with a finite state space, i.e., the computation of the steady-state probability vector.

In practice, most models are compactly described using some high-level formalism, but their underlying CTMC may be so large that storing the transition rate matrix and the probability vector, as well as computing this vector, may overwhelm even the largest computers. To deal with this *state explosion* problem, approximate techniques have been proposed, where some smaller-scale CTMCs are solved, and the results somehow combined. The approach of [52], which we extend, is particularly appealing since, once the overall model is decomposed, it uses a *multi-way decision diagram* (MDD) [60] to encode the state-space. Then, by *aggregating* the states of the CTMC based on their MDD representation, approximate stationary probabilities can be computed. As these aggregated states are relatively few, the storage and computational cost are enormously reduced.

The method of [52] is empirically shown to be efficient and have fast convergence, but it has a major limitation. To apply this approximation, the original model must be *Kronecker-consistent*, i.e., the transition rate matrix must be expressed as the sum (over all events) of the Kronecker product (over all state variables) of *local* matrices [5, 12, 36]. While in principle this Kronecker form always exists, in practice it might be achieved only by splitting model events, resulting in an excessive number of events, or by merging state variables, resulting in excessively large *local* state spaces. This restriction may render the state-space *generation* inefficient or even unfeasible, while in the meantime, a finer non-Kronecker decomposition would greatly improve the time and memory performance when used in conjunction with state-of-the-art symbolic state-space generation algorithms [28,62].

To overcome this restriction, we propose to encode the transition rate matrix with EV*MDDs (Ch. 6), which can compactly encode the transition rate matrix under an arbitrary decomposition of the model.

This chapter is organized as follows. Sec. 7.1 reviews the required background on CTMC aggregation. Sec. 7.2 discusses how to obtain the symbolic data structure needed for our approximation. Sec. 7.3 details our approximation algorithm. Sec. 7.4 reports experimental results on a set of benchmarks. Finally, Sec. 7.5 concludes the chapter. For reference, Figure 7.1 summarizes the symbols we consistently use throughout the chapter.

Symbol	Definition or Meaning
$\mathbf{Q}$	infinitesimal generator matrix
$\boldsymbol{\pi}$	probability vector
$p[\alpha, \alpha']$	a path from p, interleaving α and α'
$\mathcal{A}(p)$	$\{\alpha : r^*[\alpha].d = p\}$
$[\![p, i_k]\!]$	$\mathcal{A}(p) \times \{i_k\} \times \mathcal{B}(p[i_k])$
$\mathbf{R}_k, \mathbf{Q}_k, \boldsymbol{\pi}_k$	aggregation of $\mathbf{R}, \mathbf{Q}, \boldsymbol{\pi}$ w.r.t. x_k
$\boldsymbol{\pi}[\mathcal{Y}]$	$\sum_{\mathbf{i}\in\mathcal{Y}} \boldsymbol{\pi}[\mathbf{i}]$
$\boldsymbol{\pi}[p]$	$\boldsymbol{\pi}[\mathcal{A}(p) \times \mathcal{B}(p)]$
$\boldsymbol{\pi}[p, \gamma]$	$\boldsymbol{\pi}[\mathcal{A}(p) \times \{\gamma\} \times \mathcal{B}(p[\gamma].d)]$
$\boldsymbol{\pi}[\gamma \vert p]$	$\boldsymbol{\pi}[p, \gamma]/\boldsymbol{\pi}[p]$

Figure 7.1: New symbols and notations in this chapter.

7.1 Decision-diagram-based aggregation

7.1.1 Model description

Consider a "structured" CTMC with a finite state-space $\mathcal{S} \subset \mathbb{N}^L$ and transition rate matrix $\mathbf{R} \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{S}|}$, whose state is a tuple $\mathbf{x} = (x_L, ..., x_1) \in \mathbb{N}^L$ of L variables with a predefined order $x_L \succ ... \succ x_1$ imposed on them.

We aim at the computation of $\boldsymbol{\pi} \in \mathbb{R}^{\mathcal{S}}$ solution of

$$\boldsymbol{\pi} \cdot \mathbf{Q} = 0 \quad \text{subject to} \quad \textstyle\sum_{\mathbf{i}\in\mathcal{S}} \boldsymbol{\pi}[\mathbf{i}] = 1,$$

where $\mathbf{Q} \in \mathbb{R}^{\mathcal{S}\times\mathcal{S}}$ is the infinitesimal generator matrix of the CTMC and $\boldsymbol{\pi}[\mathbf{i}]$ is the stationary probability of state $\mathbf{i}$. $\mathbf{Q}$ coincides with $\mathbf{R}$ except in its diagonal elements, which are given by $\mathbf{Q}[\mathbf{i},\mathbf{i}] = -\sum_{\mathbf{j}\in\mathcal{S}} \mathbf{R}[\mathbf{i},\mathbf{j}]$.

Running Example. We will use the fork-and-join model from Ch. 2 as our running example. We consider the following decompositions for this model: $L = 4$, $\{x_4 \equiv [\#(d), \#(e)], x_3 \equiv [\#(c)], x_2 \equiv [\#(b)], x_1 \equiv [\#(a)]\}$, which is not a Kronecker-consistent decomposition.

7.1.2 Basic aggregation

An obvious way to tackle the state explosion problem is to *reduce* the number of states involved in the computation. An example of this idea is the basic aggregation we describe next.

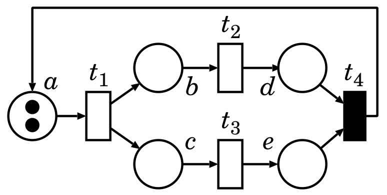
We can partition $\mathcal{S}$ into n disjoint sets of states $\{\mathcal{C}_1, ..., \mathcal{C}_n\}$. If we consider each set as a "macrostate", we can define an *aggregated* CTMC with state space $\mathcal{S}_{agg} = \{1, ..., n\}$ and transition rate matrix $\mathbf{R}_{agg}$ given by

$$\mathbf{R}_{agg}[i, i'] = \sum_{\mathbf{i}\in\mathcal{C}_i} \boldsymbol{\pi}[\mathbf{i}|\mathcal{C}_i] \cdot \sum_{\mathbf{i}'\in\mathcal{C}_{i'}} \mathbf{R}[\mathbf{i}, \mathbf{i}'], \tag{7.1}$$

where $\boldsymbol{\pi}[\mathbf{i}|\mathcal{C}_i]$ is the stationary conditional probability of being in state $\mathbf{i}$ given the state of CTMC is in set $\mathcal{C}_i$. It is well known that this aggregated CTMC is also ergodic if the original one is ergodic, and that $\boldsymbol{\pi}_{agg}[i] = \boldsymbol{\pi}[\mathcal{C}_i]$, if we let $\boldsymbol{\pi}[\mathcal{X}] = \sum_{\mathbf{i}\in\mathcal{X}i} \boldsymbol{\pi}[\mathbf{i}]$ for $\mathcal{X} \subseteq \mathcal{S}$.

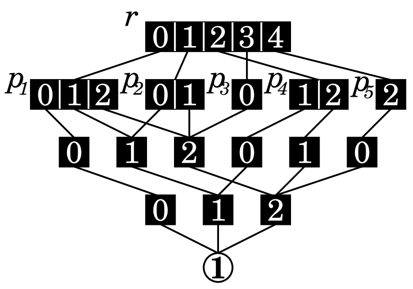
7.1.3 MDD-based aggregation

Let $r_{\mathcal{S}} = enc(\mathcal{S})$, following the partitioning strategy from [52], we can define L aggregated CTMCs, one for each variable in $\{x_L, ..., x_1\}$, based on the reachable state paths from node $r_{\mathcal{S}}$ to node $\mathbf{1}$ in the MDD $r_{\mathcal{S}}$. For each p where $p.var = x_k$ and $i_k \in \mathcal{X}_k$,



trans.	**rate**
t_1	$2.0 \cdot \#(a)$
t_2	$1.0 \cdot \#(b)$
t_3	$4.0 \cdot \#(c)$
t_4	immediate

Figure 7.2: Running example: the GSPN model of our fork-and-join system.



$$\mathcal{X}_4 = \{d^0e^0, d^0e^1, d^1e^0, d^0e^2, d^2e^0\} = \{0, ..., 4\}$$

$$\mathcal{X}_3 = \{c^0, c^1, c^2\} = \{0, 1, 2\}$$

$$\mathcal{X}_2 = \{b^0, b^1, b^2\} = \{0, 1, 2\}$$

$$\mathcal{X}_1 = \{a^2, a^1, a^0\} = \{0, 1, 2\}$$

Figure 7.3: Running example: the MDD encoding $\mathcal{S}$.

a macrostate of the aggregated CTMC for x_k, written as $[\![p, i_k]\!]$ corresponds to the set of states

$$[\![p, i_k]\!] \equiv \mathcal{A}(p) \times \{i_k\} \times \mathcal{B}(p[i_k]),$$

where $\mathcal{A}(p) \equiv \{\alpha : r_{\mathcal{S}}[\alpha] = p\}$ denotes all tuples leading from $r_{\mathcal{S}}$ to p, where $\mathcal{A}$ is mnemonic for "above". Note that $[\![p, i_k]\!] = \emptyset$ if $p[i_k] = \mathbf{0}$. As such, the state-space of the aggregated CTMC for x_k is $\{[\![p, i_k]\!] : p.var = x_k, p[i_k] \neq \mathbf{0}\}$, and we use $\boldsymbol{\pi}_k$ and $\mathbf{R}_k$ to denote the steady-state probability vector and the transition rate matrix of this aggregated CTMC, respectively.

Looking at the lower MDD in Fig. 7.3, we have $\mathcal{A}(p_1) = \{0\}$ and $\mathcal{B}(p_1[0]) = \{00\}$, therefore $[\![p_1, 0]\!] = \{0\} \times \{0\} \times \{00\} = \{0000\}$. The aggregated CTMC for x_3 has state space $\{[\![p_1, 0]\!], [\![p_1, 1]\!], [\![p_1, 2]\!], [\![p_2, 0]\!], [\![p_2, 1]\!], [\![p_3, 0]\!], [\![p_4, 1]\!], [\![p_4, 2]\!], [\![p_5, 2]\!]\}$.

7.1.4 Node and path probability

We define the probability of an MDD node p as $\boldsymbol{\pi}[p] = \boldsymbol{\pi}[\mathcal{A}(p) \times \mathcal{B}(p)]$, which can be viewed as the probability that a "random" path (chosen according to the stationary distribution $\boldsymbol{\pi}$) of the MDD passes through node p. Similarly, we define the probability of reaching node p and then following edges according to the tuple γ as $\boldsymbol{\pi}[p, \gamma] = \boldsymbol{\pi}[\mathcal{A}(p) \times \{\gamma\} \times \mathcal{B}(p[\gamma])]$, where $\boldsymbol{\pi}[p, i_k] = \boldsymbol{\pi}_k[[\![p, i_k]\!]]$ is a special case. We can then condition on tuples or nodes using the classical definition of conditional probability, e.g., $\boldsymbol{\pi}[\gamma|p] = \boldsymbol{\pi}[p, \gamma]/\boldsymbol{\pi}[p]$. In particular, we have that, for a state $\mathbf{i} = (\alpha, i_k, \beta) \in [\![p, i_k]\!]$, where $\alpha \in \mathcal{A}(p)$ and $\beta \in \mathcal{B}(p[i_k])$,

$$\boldsymbol{\pi}[\mathbf{i}|p, i_k] = \boldsymbol{\pi}[\alpha, i_k, \beta|p, i_k] = \boldsymbol{\pi}[\alpha|p, i_k] \cdot \boldsymbol{\pi}[\beta|\alpha, p, i_k] = \boldsymbol{\pi}[\alpha|p, i_k] \cdot \boldsymbol{\pi}[\beta|\alpha, i_k]. \quad (7.2)$$

7.2 From transition relations to transition rate matrices

Similarly to using an interleaved-order $2L$-variable MDD to encode a transition relation, we can use a $2L$-variable EV*MDD $\langle \omega_{\mathbf{R}}, r_{\mathbf{R}} \rangle$ to encode the transition rate matrix $\mathbf{R}$ such that

$$\forall \mathbf{i}, \mathbf{i}' \in \mathcal{S}, \mathbf{R}[\mathbf{i}, \mathbf{i}'] = \omega_{\mathbf{R}} \cdot r_{\mathbf{R}}[\mathbf{i}, \mathbf{i}']. \tag{7.3}$$

We then give the algorithm to build this symbolic encoding efficiently.

We adopt the state-space generation method of Ch. 4 so that, for each event e, the MDD $enc(\mathcal{T}_e)$ is either given a priori, or it is built on-the-fly during the iterations. Then we need translate the reduction rule vector of this MDD to ρ^Q.

We can treat $\mathcal{T}_e$ as a Boolean matrix $\mathbf{R}^b_e$, where $(\mathbf{i}, \mathbf{i}') \in \mathcal{T}_e \Leftrightarrow \mathbf{R}^b_e(\mathbf{i}, \mathbf{i}') = 1$. The EV*MDD $enc(\mathbf{R}^b_e)$ can be obtained from the MDD $enc(\mathcal{T}_e)$ by letting (1) all edges pointing to $\mathbf{1}$ point to Ω, (2) all edges pointing to $\mathbf{0}$ point to Ω and have an associated valued 0, (3) all edges not pointing to $\mathbf{0}$ have an associated value 1, and (4) adding a dangling edge pointing to the root node with an associated value 1.

Each $enc(\mathbf{R}_e)$ is then computed as the product of $enc(\mathbf{R}^b_e)$, $enc(f_e)$ and $enc(\varphi_e)$ according to Eq. 2.3, where $enc(f_e)$ is computed as $\prod_{c=1}^{C} enc(f^c_e)$ according to Eq. 2.2. Each f^c_e is encoded into an l^c-variable EV*MDD, where $l^c = |spt(f^c_e)| \leq L$, by invoking procedure $Build(f^c_e, (x_{l^c}, ..., x_1))$, where $\{x_{l^c}, ..., x_1\} = spt(f^c_e)$. Similarly we can build the encoding of φ_e as an $|spt(\varphi_e)|$-variable EV*MDD.

With the above strategy, we only need to enumerate the possible values of $(x_{l^c}, ..., x_1)$ for each f^c_e, and evaluate f^c_e on them; this is usually computationally feasible, unlike the full enumeration of the possible values for the tuple $(x_L, ..., x_1)$; a similar argument holds for φ_e. A finer decomposition which results in smaller local state spaces is also favorable for this phase.

The complete process is shown in Fig. 7.4. Procedure *BuildFunction* builds an EV*MDD encoding f_e or φ_e by first decomposing this input function and then invoking *Multiply* on the EV*MDDs encoding the sub-functions. Procedure *BuildTransitionRate* builds $\mathbf{R}_e$ by applying *Multiply* on EV*MDDs encoding $\mathbf{R}_e^b$, f_e and φ_e. Finally, procedure *BuildR* uses *Add* to sum $enc(\mathbf{R}_e)$ over all event e and obtains $enc(\mathbf{R})$, which is the second input of our approximation algorithm, discussed in Sec. 7.3.

We would like to mention that a disjunctive encoding (see Sec. 6.3) can also be used in our approximation algorithm, and this can be important, since the disjunctive encoding can in principle require many fewer nodes that the single-root MDD encoding, as in the worst case, a single-root EV*MDD generated after summing n EV*MDDs can be exponentially larger (in n) than the original EV*MDDs being summed. The algorithm in the next section then needs only minor adaption to use such a disjunctive encoding, traversing down from the root of each EV*MDD for each event, instead of from the only root.

In this chapter, we use a single-root EV*MDD to encode $\mathbf{R}$ instead of a disjunctive encoding for illustration purposes and also because by adopting it, we may avoid many additions of rates during the computation.

```
$BuildR()$
    1 $\langle \omega_{\mathbf{R}}, r_{\mathbf{R}} \rangle \leftarrow \langle 0, \Omega \rangle$;
    2 foreach $e \in \mathcal{E}$ do
    3     $\langle \rho, r \rangle \leftarrow BuildTransitionRate(e)$;
    4     $\langle \omega_{\mathbf{R}}, r_{\mathbf{R}} \rangle \leftarrow Add(\langle \omega_{\mathbf{R}}, r_{\mathbf{R}} \rangle, \langle \rho, r \rangle)$;

$rdd\ BuildTransitionRate(event\ e)$
    1 $\langle \rho, r \rangle \leftarrow MakeEvmdd(1, enc(\mathcal{T}_e))$;
    2 $\langle a, p \rangle \leftarrow BuildFunction(f_e)$;
    3 $\langle a, p \rangle \leftarrow Multiply(\langle a, p \rangle, BuildFunction(\varphi_e))$;
    4 return $Multiply(\langle \rho, r \rangle, \langle a, p \rangle)$;

$rdd\ BuildFunction(expr\ f)$                      • expressed as products of expressions
    1 $\langle \rho, r \rangle \leftarrow \langle 1, \Omega \rangle$;
    2 $exps \leftarrow GetProducts(f)$;                      • break into products
    3 foreach $f_c$ in $exps$ do
    4     $\langle a, p \rangle \leftarrow Build(f_c, (x_{l^c}, ..., x_1))$;
    5     $\langle \rho, r \rangle \leftarrow Multiply(\langle \rho, r \rangle, \langle a, p \rangle)$;
    6 return $\langle \rho, r \rangle$;
```

Figure 7.4: Building EV*MDD encoding **R**.

7.3 Our approximation algorithm

Once we have built an EV*MDD encoding of the transition rate matrix $\mathbf{R}$ and an MDD encoding of the state space $\mathcal{S}$ for the CTMC under study, we have all we need to carry out the proposed approximation algorithm. First let us examine the exact aggregation using EV*MDDs.

7.3.1 Exact aggregation

For clarity, we recall some equations obtained in the previous sections, which will be used in this section, namely Eq. 7.1 (basic aggregation):

$$\mathbf{R}_{agg}[i, i'] = \sum_{\mathbf{i} \in \mathcal{C}_i} \boldsymbol{\pi}[\mathbf{i}|\mathcal{C}_i] \cdot \sum_{\mathbf{i}' \in \mathcal{C}_{i'}} \mathbf{R}[\mathbf{i}, \mathbf{i}'],$$

Eq. 7.2 (node and path conditional probability): letting $\mathbf{i} = (\alpha, i_k, \beta)$ and $p = r_{\mathcal{S}}[\alpha]$,

$$\boldsymbol{\pi}[\mathbf{i}|p, i_k] = \boldsymbol{\pi}[\alpha|p, i_k] \cdot \boldsymbol{\pi}[\beta|\alpha, i_k],$$

and Eq. 7.3 (EV*MDD encoding $\mathbf{R}$):

$$\forall \mathbf{i}, \mathbf{i}' \in \mathcal{S}, \mathbf{R}[\mathbf{i}, \mathbf{i}'] = \omega_{\mathbf{R}} \cdot r_{\mathbf{R}}[\mathbf{i}, \mathbf{i}'].$$

We then compute the rate from macrostate $[\![p, i_k]\!]$ to macrostate $[\![q, i'_k]\!]$ in $\mathbf{R}_k$ as

$$\begin{aligned}
\mathbf{R}_k[[\![p, i_k]\!], [\![q, i'_k]\!]] &= \sum_{\mathbf{i} \in [\![p, i_k]\!]} \boldsymbol{\pi}[\mathbf{i} | [\![p, i_k]\!]] \cdot \sum_{\mathbf{i}' \in [\![q, i'_k]\!]} \omega_{\mathbf{R}} \cdot r_{\mathbf{R}}[\mathbf{i}, \mathbf{i}'] \\
&= \omega_{\mathbf{R}} \cdot \sum_{\substack{\alpha \in \mathcal{A}(p) \\ \beta \in \mathcal{B}(p)}} \boldsymbol{\pi}[\alpha, i_k, \beta | p, i_k] \cdot \sum_{\substack{\alpha' \in \mathcal{A}(q) \\ \beta' \in \mathcal{B}(q)}} r_{\mathbf{R}}[(\alpha, i_k, \beta), (\alpha', i'_k, \beta')].v \\
&= \omega_{\mathbf{R}} \cdot \sum_{\alpha \in \mathcal{A}(p)} \boldsymbol{\pi}[\alpha | p, i_k] \cdot \sum_{\alpha' \in \mathcal{A}(q)} r_{\mathbf{R}}[\alpha, \alpha'].v \cdot r[i_k, i'_k].v \\
&\quad \cdot \sum_{\beta \in \mathcal{B}(p[i_k])} \boldsymbol{\pi}[\beta | \alpha, i_k] \cdot \sum_{\beta' \in \mathcal{B}(q[i'_k])} r'[\beta, \beta'].v, \qquad (7.4)
\end{aligned}$$

where $r = r_{\mathbf{R}}[\alpha, \alpha'].d$ and $r' = r[i_k, i'_k].d$. This exact aggregation equation unfortunately cannot be applied in practice to reduce the size of the CTMC under study unless the quantities $\boldsymbol{\pi}[\alpha | p, i_k]$ and $\boldsymbol{\pi}[\beta | \alpha, i_k]$ can be computed exactly. In general, these quantities cannot be determined without first determining $\boldsymbol{\pi}[\mathbf{i}]$ for all states $\mathbf{i}$ in the original CTMC, which is the computation we are hoping to avoid. As such, we will instead use an *approximate* aggregation, in which we will use estimates for $\boldsymbol{\pi}[\alpha | p, i_k]$ and $\boldsymbol{\pi}[\beta | \alpha, i_k]$.

7.3.2 Approximate aggregation

We use $\boldsymbol{\pi}[\alpha | p]$ as an estimate of $\boldsymbol{\pi}[\alpha | p, i_k]$ and $\boldsymbol{\pi}[\beta | p[i_k]]$ as an estimate of $\boldsymbol{\pi}[\beta | \alpha, i_k]$ to simplify Eq. 7.4. This idea is also adopted in [52] and has an important justification: if the estimate probability is 0, then the actual probability must also be 0, and vice versa. When the estimate and the actual value are equal, our algorithm produces an exact solution since it is the only source of approximation.

These estimates can be computed recursively as

$$\boldsymbol{\pi}[\alpha|p] \;=\; \boldsymbol{\pi}[p', i_{k+1}|p] \cdot \boldsymbol{\pi}[\gamma|p'], \tag{7.5}$$

$$\boldsymbol{\pi}[\beta|p[i_k]] \;=\; \boldsymbol{\pi}[i_{k-1}|p[i_k]] \cdot \boldsymbol{\pi}[\delta|p[i_k, i_{k-1}]], \tag{7.6}$$

where $\alpha = (\gamma, i_{k+1}), p' = r_{\mathcal{S}}[\gamma]$, and $\beta = (i_{k-1}, \delta)$.

Then an estimate of $\mathbf{R}_k$ is

$$\begin{aligned}\mathbf{R}_k[[\![p, i_k]\!], [\![q, i'_k]\!]] \;\approx\; & \omega_{\mathbf{R}} \cdot \sum_{\alpha \in \mathcal{A}(p)} \boldsymbol{\pi}[\alpha|p] \cdot \sum_{\alpha' \in \mathcal{A}(q)} r_{\mathbf{R}}[\alpha, \alpha'].v \cdot r[i_k, i'_k].v \\ & \cdot \sum_{\beta \in \mathcal{B}(p[i_k])} \boldsymbol{\pi}[\beta|p[i_k]] \cdot \sum_{\beta' \in \mathcal{B}(q[i'_k])} r'[\beta, \beta'].v. \end{aligned} \tag{7.7}$$

We can decompose the right-hand side of Eq. 7.7 into three parts by letting

$$\mathbf{A}[r, p, q] \equiv \omega_{\mathbf{R}} \cdot \sum_{\substack{\alpha \in \mathcal{A}(p), \alpha' \in \mathcal{A}(q) \\ r_{\mathbf{R}}[\alpha, \alpha'].d = r}} \boldsymbol{\pi}[\alpha|p] \cdot r_{\mathbf{R}}[\alpha, \alpha'].v,$$

and

$$\mathbf{B}[r, p, q] \equiv \sum_{\substack{\beta \in \mathcal{B}(p), \beta' \in \mathcal{B}(q), \\ r[\beta, \beta'].d = \Omega, r[\beta, \beta'].v \neq 0}} \boldsymbol{\pi}[\beta|p] \cdot r[\beta, \beta'].v,$$

then Eq. 7.7 becomes

$$\mathbf{R}_k[[\![p, i_k]\!], [\![q, i'_k]\!]] \approx \sum_{\forall r: r.var = x_k} \mathbf{A}[r, p, q] \cdot r[i_k, i'_k].v \cdot \mathbf{B}[r[i_k, i'_k].d, p[i_k], q[i'_k]]. \tag{7.8}$$

A stands for "above", since $\mathbf{A}[r, p, q]$ does not depend on $\mathcal{B}(p)$ or $\mathcal{B}(q)$ and **B** stands for "below" since $\mathbf{B}[r, p, q]$ does not depend on $\mathcal{A}(p)$ and $\mathcal{A}(q)$.

Substituting Eq. 7.5, $\mathbf{A}[r,p,q]$ can be computed top-down using the recurrence

$$\mathbf{A}[r,p,q] = \sum_{\substack{\forall r',p',q',i_k,i'_k: \\ p=p'[i_k],\, q=q'[i'_k],\, r=r'[i_k,i'_k].d}} \mathbf{A}[r',p',q'] \cdot \boldsymbol{\pi}[p',i_k|p] \cdot r'[i_k,i'_k].v, \tag{7.9}$$

terminated with $\mathbf{A}[r_{\mathbf{R}}, r_{\mathcal{S}}, r_{\mathcal{S}}] = \omega_{\mathbf{R}}$.

Similarly, substituting Eq. 7.6, $\mathbf{B}[r,p,q]$ can be computed bottom-up using the recurrence

$$\mathbf{B}[r,p,q] = \sum_{\forall i_k,i'_k \in \mathcal{X}_k} \mathbf{B}[r[i_k,i'_k].d, p[i_k], q[i'_k]] \cdot \boldsymbol{\pi}[i_k \,|\, p] \cdot r[i_k,i'_k].v, \tag{7.10}$$

terminated with $\mathbf{B}[\Omega, \mathbf{1}, \mathbf{1}] = 1$.

7.3.3 The algorithm

The key step of the algorithm is to compute matrices $\mathbf{A}$ and $\mathbf{B}$ (we can think of them as 3-dimensional matrices). A nonzero entry $\mathbf{A}[r,p,q]$ should fulfill $r.var = p.var = q.var$ and $\mathbf{A}[r,p,q] > 0$ only if (1) $[r,p,q] = [r_{\mathbf{R}}, r_{\mathcal{S}}, r_{\mathcal{S}}]$ or (2) there exist nodes r', p', q' with $.var = x_k$ and integers $i, j \in \mathcal{X}_k$ such that $\mathbf{A}[r',p',q'] > 0$ and $r'[i,j].d = r, p'[i] = p, q'[j] = q$. Similarly, $\mathbf{B}[r,p,q] > 0$ only if (1) $[r,p,q] = [\Omega, \mathbf{1}, \mathbf{1}]$ or (2) there exist node r', p', q' with $.var = x_k$ and integer $i, j \in \mathcal{X}_k$ such that $\mathbf{B}[r',p',q'] > 0$ and $r[i,j].d = r', p[i] = p', q[j] = q'$. So the positions of the nonzero elements of $\mathbf{A}$ and $\mathbf{B}$ can be found a priori by concurrently traversing EV*MDD $r_{\mathbf{R}}$ and MDD $r_{\mathcal{S}}$ top-down and bottom-up, respectively.

Another important observation is that for nodes r, p, q, where $r.var = p.var = q.var$ and $x_L \succ r.var \succ x_0$, if either $\mathbf{A}[r,p,q]$ or $\mathbf{B}[r,p,q]$ is zero according to the above traversal, we do not need to store the other. Since $\mathbf{A}[r,p,q]$ is only used in a product with some $\mathbf{B}[r', p', q']$, where there exist i, j such that $r[i, j].d = r', p[i] = p', q[j] = q'$, but if $\mathbf{B}[r, p, q]$ is zero, any of these $\mathbf{B}[r', p', q']$ must also be zero. This further reduces the number of nonzero elements in $\mathbf{A}$ and $\mathbf{B}$, which are stored as sparse matrices, so that the elements for $\mathbf{A}$ and $\mathbf{B}$ have the same set of indices except for the terminal cases. More precisely, only those entries $[r, p, q]$ should be kept where there exist paths α, α', such that $r_{\mathbf{R}}[\alpha, \alpha'].d = \Omega, r_{\mathbf{R}}[\alpha, \alpha'].v > 0, r_{\mathcal{S}}[\alpha] = r_{\mathcal{S}}[\alpha'] = \mathbf{1}$, and r lies on path (α, α'), p lies on α, q lies on α', which leads to the following algorithm.

In Fig 7.5, procedure *BuildCorr* finds all *corresponding* MDD node pairs for each EV*MDD node, such that $(p, q) \in r.corr$ means $[r, p, q]$ is an element we need to compute in $\mathbf{A}$ and $\mathbf{B}$, by traversing down from the root node. We also compute *r.corr* if r is associated with a primed variable, which might at worst double the size of $\mathbf{A}$ and $\mathbf{B}$ but can greatly simplify the computation and better utilize the cache. Storing those extra elements for $\mathbf{A}$ and $\mathbf{B}$ is also efficient when computing $\mathbf{A}[r, p, q]$ or $\mathbf{B}[r, p, q]$ from another $\mathbf{A}[r', p', q']$ or $\mathbf{B}[r', p', q']$ according to Eq. 7.9 and Eq. 7.10, which allows a recursive computation with the least information, shown later.

Note that *BuildCorr* returns a Boolean value to indicate whether we should put (p, q) into *r.corr*. It returns *false* when we get to r from (α, α') in $r_{\mathbf{R}}$ and the corresponding α is not a valid path in $r_{\mathcal{S}}$. This can happen because each $enc(\mathcal{T}_e)$ encodes the *potential* transitions [22], i.e., $(\mathbf{i}, \mathbf{i}') \in \mathcal{T}_e$ does not necessarily indicate $\mathbf{i} \in \mathcal{S}$; this also happens in a Kronecker representation and is a common strategy to encode transition relations as it usually results in a much more compact encoding. Since $r_{\mathbf{R}}$ is obtained from $enc(\mathcal{T}_e)$, it, too, might contain some “fictitious” rates. Of course, we could use $\mathcal{S}$ as a filter to eliminate these fictitious rates, and this preprocessing can be seamlessly integrated to *BuildCorr* as shown in the pseudocode. Sec. 7.4 lists the number of nodes and edges of the EV*MDD encoding the actual $\mathbf{R}$ after filtering these fictitious rates; this is not used in our approach and is presented for comparison only.

Fig. 7.6, Fig. 7.7 and Fig. 7.8 show the pseudocode of our approximation. Procedure *ComputeAbove*(x_k) computes $\mathbf{A}$ for all elements $[r, p, q]$ with $r.var = x'_{k+1}$ and then $r.var = x_k$, by traversing all elements $[r', p', q']$ with $r.var = x_{k+1}$ and then $r.var = x'_{k+1}$ and applying Eq. 7.9 appropriately. Procedure *ComputeBelow*(x_k) computes all elements $[r, p, q]$ of $\mathbf{B}$, with $r.var = x'_k$ and then $r.var = x_k$, by traversing all elements $[r', p', q']$ with $r.var = x'_k$ and then $r.var = x_k$, based on Eq. 7.10. These two procedures benefit from a sparse representation of matrices $\mathbf{A}$ and $\mathbf{B}$, since they traverse only the nonzero elements. During the computation, $r[i, i'].v$ is broken into $r[i].v \cdot r[i][i'].v$; $\boldsymbol{\pi}[p', i_k|p]$ is computed as $\boldsymbol{\pi}_k[[\![p', i_k]\!]]/\boldsymbol{\pi}[p]$; and$\boldsymbol{\pi}[i_k|p]$ is obtained from $\boldsymbol{\pi}_k[[\![p, i_k]\!]]/\boldsymbol{\pi}[p]$. We can see that storing *r.corr* for r associated with a primed variable simplifies the computation, since now we can compute $\mathbf{A}[r, p, q]$ from another $\mathbf{A}[r', p', q']$ by traversing only r's outgoing edges and either p or q's outgoing edges (depending on $r.var$); the same holds for $\mathbf{B}$. Thus, in the actual data structures set up for $\mathbf{A}$ and $\mathbf{B}$, only one EV*MDD node pointer and one MDD node pointer need to be stored for each nonzero element.

Procedure *SolveVariable*(x_k) is used to build and solve the aggregated CTMC for x_k. The probability vector $\boldsymbol{\pi}_k$ satisfying $\boldsymbol{\pi}_k \cdot \mathbf{Q}_k = \mathbf{0}$ can be determined using an iterative method such as Jacobi or Gauss-Seidel [61]. As the probability vectors change, so do the rates between these macrostates, and the vector $\boldsymbol{\pi}_k$ must be recomputed. To accelerate this process, we use the previous solution to $\boldsymbol{\pi}_k$ as the initial vector for the iterative method when recomputing $\boldsymbol{\pi}_k$. Doing so causes the number of iterations required to solve $\boldsymbol{\pi}_k$ to decrease as the overall fixpoint computation converges. When $\boldsymbol{\pi}_k$ is computed at each global iteration, the probability of each node p, $\boldsymbol{\pi}[p]$, with $p.var = x_k$, is updated by summing all $\boldsymbol{\pi}_k[[\![p, i]\!]]$ and used for the next global iteration; in the meantime, $\boldsymbol{\pi}[p]$, with $p.var = x_{k-1}$, is also updated, as it is used for *ComputeAbove*(x_{k-1}) in the current global iteration, also improving convergence.

Procedure *Solve* performs the overall fixpoint computation, assuming that the MDD encoding $\mathcal{S}$ and the EV*MDD encoding $\mathbf{R}$ have been built, and that the vectors $\boldsymbol{\pi}_k$ for each x_k are set to some initial distribution (we use the uniform distribution). A stopping criterion must be used to determine when the overall computation has reached a fixpoint. We stop the iterations for every aggregated CTMC when the probability vector does not change anymore.

7.3.4 Computing measures

Once we have obtained an estimate for $\boldsymbol{\pi}$, we typically wish to compute one or more measures of interest defined via a reward function $\theta : \mathcal{S} \to \mathbb{R}$, for example $m = \sum_{\forall \mathbf{i} \in \mathcal{S}} \theta(\mathbf{i}) \cdot \boldsymbol{\pi}[\mathbf{i}]$. Since computing $\boldsymbol{\pi}[\mathbf{i}]$ requires a product according to the tuple of L nodes visited along the path $\mathbf{i}$, the time to compute m according to this sum is $O(L \cdot |\mathcal{S}|)$, which can be extremely large.

However, if the reward function can be expressed as the product of functions on state variables, $\theta(\mathbf{x}) = \theta_L(x_L) \cdots \theta_2(x_2) \cdot \theta_1(x_1)$, we can then compute m recursively, by

```
$boolean\ BuildCorr(rdd\ node\ r, mdd\ p, mdd\ q)$                    • assume $p.var = x_k$
1  if $r = \Omega$ then return $true$;                                   • terminal case
2  if $CacheHit(CORR, r, p, q, answer)$ then return $answer$;            • check cache
3  $answer \leftarrow false$;
4  if $r.var = p.var$ then                                 • $r$ associated to unprimed variable
5    foreach $i \in \mathcal{X}_k$ s.t. $p[i] \neq \mathbf{0}$ and $r[i] \neq \langle 0,\Omega \rangle$ do
6      if $BuildCorr(r[i].d, p[i], q) = true$ then
7        $answer \leftarrow true$;
8  else                                                      • $r$ associated to primed variable
9    foreach $i \in \mathcal{X}_k$ s.t. $q[i] \neq \mathbf{0}$ and $r[i] \neq \langle 0,\Omega \rangle$ do
10     if $BuildCorr(r[i].d, p, q[i]) = true$ then
11       $answer \leftarrow true$;
12 if $answer = true$ then              • $[r, p, q]$ can be an nonzero element for $\mathbf{A}$ and $\mathbf{B}$
13   $r.corr \leftarrow r.corr \cup \{(p, q)\}$;
14 $CacheAdd(CORR, r, p, q, answer)$;
15 return $answer$;
```

Figure 7.5: Setting up nonzeroes for **A** and **B**.

```
$Solve()$
1  $BuildCorr(r_{\mathbf{R}}, r_{\mathcal{S}}, r_{\mathcal{S}})$;
2  while not converged do
3    $\mathbf{A} \leftarrow \mathbf{0}$; $\mathbf{B} \leftarrow \mathbf{0}$; $\mathbf{R} \leftarrow \mathbf{0}$;
4    for $k \leftarrow 0$ to $L$ do $ComputeBelow(x_k)$;
5    for $k \leftarrow L$ to 1 do
6      $ComputeAbove(x_k)$;
7      $SolveVariable(x_k)$;
```

Figure 7.6: The global iteration.

```
$ComputeAbove(variable\ x_k)$
1 if $k = L$ then $\mathbf{A}[r_{\mathbf{R}}, r_{\mathcal{S}}, r_{\mathcal{S}}] \leftarrow \omega_{\mathbf{R}}$; return;                • terminal case
2 foreach $rdd\ node\ r$ where $r.var = x_{k+1}$ do
3   foreach $(p, q) \in r.corr$ do
4     foreach $i \in \mathcal{X}_{k+1}$ s.t. $p[i] \neq \mathbf{0}$ and $r[i] \neq \langle 0{,}\Omega\rangle$ do
5       $r' \leftarrow r[i].d$; $p' \leftarrow p[i]$;
6       $\mathbf{A}[r', p', q] \leftarrow \mathbf{A}[r', p', q] + \mathbf{A}[r, p, q] \cdot \boldsymbol{\pi}_{k+1}[[\![p, i]\!]] \cdot r[i].v$;
7 foreach $rdd\ node\ r$ where $r.var = x'_{k+1}$ do
8   foreach $(p, q) \in r.corr$ do
9     foreach $i \in \mathcal{X}_{k+1}$ s.t. $q[i] \neq \mathbf{0}$ and $r[i] \neq \langle 0{,}\Omega\rangle$ do
10      $r' \leftarrow r[i].d$; $q' \leftarrow q[i]$; $\mathbf{A}[r', p, q'] \leftarrow \mathbf{A}[r', p, q'] + \mathbf{A}[r, p, q] \cdot r[i].v/\boldsymbol{\pi}_k[p]$;
```

```
$ComputeBelow(variable\ x_k)$
1 if $k = 0$ then $\mathbf{B}[\Omega, \mathbf{1}, \mathbf{1}] \leftarrow 1$; return;                • terminal case
2 foreach $rdd\ node\ r$ where $r.var = x'_k$ do
3   foreach $(p, q) \in r.corr$ do
4     foreach $i \in \mathcal{X}_k$ s.t. $q[i] \neq \mathbf{0}$ and $r[i] \neq \langle 0{,}\Omega\rangle$ do
5       $r' \leftarrow r[i].d$; $q' \leftarrow q[i]$; $\mathbf{B}[r, p, q] \leftarrow \mathbf{B}[r, p, q] + \mathbf{B}[r', p, q'] \cdot r[i].v$;
6 foreach $rdd\ node\ r$ where $r.var = x_k$ do
7   foreach $(p, q) \in r.corr$ do
8     foreach $i \in \mathcal{X}_k$ s.t. $p[i] \neq \mathbf{0}$ and $r[i] \neq \langle 0{,}\Omega\rangle$ do
9       $r' \leftarrow r[i].d$; $p' \leftarrow p[i]$;
10      $\mathbf{B}[r, p, q] \leftarrow \mathbf{B}[r, p, q] + \mathbf{B}[r', p', q] \cdot r[i].v \cdot \boldsymbol{\pi}_k[[\![p, i]\!]]/\boldsymbol{\pi}_k[p]$;
```

Figure 7.7: Steps in the global iteration.

```
$SolveVariable(variable\ x_k)$
1  foreach $rdd\ node\ r$ where $r.var = x_k$ do
2    foreach $(p,q) \in r.corr$ do
3      foreach $i \in \mathcal{X}_k$ s.t. $p[i] \neq \mathbf{0}$ and $r[i] \neq \langle 0,\Omega \rangle$ do
4        foreach $j \in \mathcal{X}_k$ s.t. $q[j] \neq \mathbf{0}$ and $r[i][j] \neq \langle 0,\Omega \rangle$ do
5          $r' \leftarrow r[i][j].d$; $p' \leftarrow p[i]$; $q' \leftarrow q[i]$; $val \leftarrow r[i].v \cdot r[i][j].v$;
6          $\mathbf{R}_k[[\![p,i]\!],[\![q,j]\!]] \leftarrow \mathbf{R}_k[[\![p,i]\!],[\![q,j]\!]] + \mathbf{A}[r,p,q] \cdot \mathbf{B}[r',p',q'] \cdot val$;
7  $\boldsymbol{\pi}_k \leftarrow$ solution of $\boldsymbol{\pi}_k \cdot \mathbf{Q}_k = \mathbf{0}$;          • Gauss-Seidel or Jacobi
8  foreach $p$ where $p.var = x_k$ do
9    $\boldsymbol{\pi}[p] \leftarrow \sum_{i \in \mathcal{X}_k} \boldsymbol{\pi}_k[[\![p,i]\!]]$;          • Adjust node sum of $\boldsymbol{\pi}_k$
10 if $k > 1$ then
11   foreach $p$ where $p.var = x_{k-1}$ do $\boldsymbol{\pi}_{k-1}[p] \leftarrow 0$;
12   foreach $[\![p,i]\!]$ where $p.var = x_k$ do
13     $\boldsymbol{\pi}[p[i]] \leftarrow \boldsymbol{\pi}[p[i]] + \boldsymbol{\pi}_k[[\![p,i]\!]]$;          • Adjust node sum of $\boldsymbol{\pi}_{k-1}$
```

Figure 7.8: Steps in the global iteration (continued).

defining the value of the measure for a node p with $p.var = x_k$ as

$$m(p) \;=\; \sum_{\forall i_k : p[i_k] \neq 0} \theta_k(i_k) \cdot \boldsymbol{\pi}[i_k|p] \cdot m(p[i_k]),$$

so that $m(r_{\mathcal{S}})$ is equal to m. We can then compute $m(r_{\mathcal{S}})$ in a bottom-up fashion, by visiting each MDD node p only once and scanning the nonzero edges of p, with a computational cost proportional to the number of nonzero edges in the MDD, normally *much* smaller than $O(L \cdot |\mathcal{S}|)$. Related statistics are listed in Sec. 7.4.

7.3.5 Complexity, accuracy, and convergence

The overall time complexity of our approximate solution is $O(iters \cdot (\eta(\mathbf{A}) + T))$, where *iters* is the number of global iterations, $\eta(\mathbf{A})$ is the number of nonzero elements of matrix $\mathbf{A}$ and T is the average complexity to compute each $\boldsymbol{\pi}_k$ using the Gauss-Seidel or

Jacobi method at each global iteration, which is of course related to each $\eta(\mathbf{R}_k)$.

The number of triples for x_k at worst case is $N_k \cdot N_k \cdot M_k$, where N_k is the #mddnodes for x_k and M_k is #evmddnodes for x_k; but in practice it is usually much smaller, shown in Table. 7.5.

We recall that the approximation in our algorithm is due to using $\boldsymbol{\pi}[\alpha|p]$ as an estimate of $\boldsymbol{\pi}[\alpha|p, i_k]$ and $\boldsymbol{\pi}[\beta|p[i_k]]$ as an estimate of $\boldsymbol{\pi}[\beta|\alpha, i_k]$. Thus, the accuracy of our algorithm depends on how well this assumption holds.

Using an under-relaxation parameter (e.g., 0.95) with the Gauss-Seidel or Jacobi methods, the computation of each $\boldsymbol{\pi}_k$ is guaranteed to converge. However, global convergence is not guaranteed in general, thus we can only set a bound on the number of global iterations, to guard against models and decompositions that do not converge (for Gauss-Seidel method, we have not found such a case in our experiments so far)

7.3.6 Results for our running model

We conclude this section by applying our approximation to the running model. Table 7.1 shows the results from the exact solution and from our approximation, specifically the average number of tokens in each place for our running example. The approximation is quite accurate for this model: the maximum observed relative error is within 0.1%.

7.4 Experimental results

In this section, we report experimental results and related statistics on a set of benchmarks. We will mainly discuss results of a flexible manufacturing system (FMS)

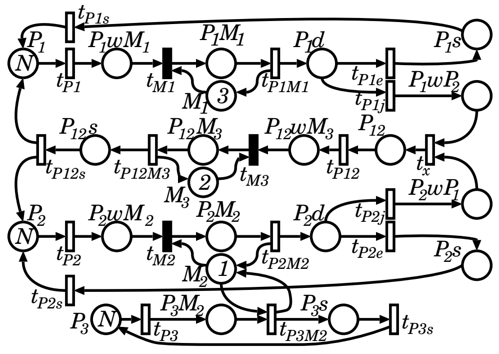
		Approx	
Place	**Exact**	Decomp.1	Decomp.2
a	0.656393	0.656393	0.656393
b	1.31279	1.31278	1.31278
c	0.328196	0.328196	0.328196
d	0.0308214	0.308214	0.308214
e	1.01541	1.01541	1.01541

Table 7.1: Running example: E[number of tokens] in each place, when $N = 2$.

model from [27], but also list results for other models to show the wide applicability of our methods. All algorithms are implemented in smart [17] and all experiments are run on an Intel Xeon 3.0Ghz workstation with 4GB RAM under SuSE Linux 9.1. For all experiments, we set up the time bound of 3600 seconds, a memory bound of 4GB and the maximum number of global iterations of 10,000.

7.4.1 The Flexible Manufacturing System

Fig. 7.9 models a flexible manufacturing system (FMS) [27] with N pallets to move different types of parts to various machines. Transitions t_{P1}, t_{P2}, t_{P3}, and t_{P12} share these pallets in a "processor sharing" fashion; for example, the rate of t_{P1} is $\#(P_1) \cdot \mu$, where $\#(P_1)$ is the number of tokens in the input place of t_{P1}, and $\mu = \min\left\{1, \frac{N}{\#(P_1)+\#(P_2)+\#(P_3)}\right\}$ is a slowdown factor evaluating to 1 if the total number of pallets in use is at most N, or to less than 1 if the requests exceed the pallets. Due to this dependency, under the Kronecker



trans.	**rate**	**trans.**	**rate**
t_{P1}	$\#(P_1) \cdot \mu$	t_{P2}	$\#(P_2) \cdot \mu$
t_{P3}	$\#(P_3) \cdot \mu$	t_{P12}	$\#(P_{12}) \cdot \mu$
t_{P1M1}	$\#(P_{1M1})/4$	t_{P2M2}	$1/6$
t_{P3M2}	$1/2$	t_{P12M3}	$\#(P_{12M3})$
t_{P1s}	$\#P_{1s}/60$	t_{P2s}	$\#P_{2s}/60$
t_{P3s}	$\#P_{3s}/60$	t_{P12s}	$\#P_{12s}/60$
t_{P1e}	80	t_{P1j}	20
t_{P2e}	60	t_{P2j}	40
t_x	100		

Figure 7.9: Benchmark – the FMS model.

restriction, P_1, P_2 and P_3 must be encoded by one variable and due to the presence of immediate transitions, $\{P_1wM_1, M_1, P_1M_1\}$, $\{P_2wM_2, M_2, P_2M_2\}$, and $\{P_{12}wM_3, M_3, P_{12}M_3\}$ must also be put into one variable each; each other place can be encoded by one distinct variable, which consists the finest Kronecker decomposition. So, for this model, the finest decomposition uses 21 variables while the finest Kronecker decomposition uses 13 state variable. We measure the probability $\Pr \#(P_{12}s) = 0$.

We compare four different solution approaches.

- The *Explicit* solution stores $\mathbf{R}$ explicitly and computes $\boldsymbol{\pi}$ using Gauss-Seidel for the numerical solution, with a relative stopping criterion set to 10^{-4}, i.e., the iterations stop when $\max_{\mathbf{i}\in\mathcal{S}} |\boldsymbol{\pi}[\mathbf{i}] - \boldsymbol{\pi}^{old}[\mathbf{i}]|/\boldsymbol{\pi}[\mathbf{i}] \leq 10^{-4}$. For this solution, we use the general state-space generation technique from [62].

- Discrete-event *Simulation* computes the given measure to within an interval of relative width 2% with 99% confidence, i.e., the number of simulation runs is such that, with probability 0.99, the exact value of the measure is the computed point estimate plus or minus 1%. Of course, simulation does not require state-space generation.

- The *Kronecker-based* approximation uses the approximate technique in [52], where $\mathbf{R}$ is stored as a Kronecker expression, and is applicable only when the decomposition of the model is Kronecker-consistent. When solving each aggregate CTMC, we use Gauss-Seidel or Jacobi with a relative precision of 10^{-4}, and the global fixpoint iterations stop when every aggregate CTMC solution converges within one Gauss-Seidel or Jacobi iteration. To guarantee convergence in the aggregate CTMC solutions, we use a relaxation parameter of 0.95 for both Gauss-Seidel and Jacobi. This solution uses the state-space generated from [19].

- Our *EV*MDD-based* approximation method, always applicable, uses the same stopping criteria as the Kronecker-based solution. The state-space generation technique from [62] is employed.

7.4.2 Result tables

We report the results, runtime and number of global iterations (when applicable) of each approach in Table 7.2. The runtime of each approach is composed of two part, except for simulation: T_{init} is the initialization time, including state-space generation, generation of the transition rate matrix or its symbolic encoding, and setting up related computational data structures; T_{solve} under Gauss-Seidel or Jacobi is the time elapsed until convergence after initialization.

			Explicit Solution				**Simulation**		**Kronecker-based Approximation**						**EV*MDD-based Approximation**					
				Gauss-Seidel						Gauss-Seidel		Jacobi				Gauss-Seidel		Jacobi		
Model	N	Decomp.	T_{init}	T_{solve}	iters	meas.	time	meas.	T_{init}	T_{solve}	iters	T_{solve}	iters	meas.	T_{init}	T_{solve}	iters	T_{solve}	iters	meas.
fms	5	Kron	64.97	33.91	785	0.3621	18.64	0.3688	0.23	0.47	22	0.56	23	0.3630	0.34	0.24	20	0.31	23	0.3630
		finest	/	/	/	/	/	/	x	x	x	x	x	x	0.08	0.18	21	0.04	58	0.3630
		break							2.30	3.42	22	3.64	23	0.3630	1.03	0.25	20	0.33	23	0.3630
fms	6	Kron	334.03	200.25	946	0.2952	21.21	0.3040	0.39	0.90	20	1.05	22	0.2964	0.60	0.51	20	0.64	22	0.2964
		finest	/	/	/	/	/	/	x	x	x	x	x	x	0.11	0.24	22	0.03	19	0.2964
		break							5.69	8.31	20	9.20	22	0.2964	2.10	0.56	23	0.67	22	0.2964
fms	10	Kron	-	-	-	-	120	0.1431	2.57	6.38	22	8.15	23	0.1319	4.61	4.44	22	5.35	23	0.1319
		finest	/	/	/	/	/	/	x	x	x	x	x	x	0.32	4.31	24	0.15	17	0.1319
		break							123	182	22	191	23	0.1319	22.52	4.48	22	5.38	23	0.1319
fms	20	Kron	-	-	-	-	452	0.0191	124	558	144	591	106	0.0178	140	359.08	100	454	106	0.0178
		finest	/	/	/	/	/	/	x	x	x	x	x	x	1.87	45.42	98	18.03	217	0.0178
		break							-	-	-	-	-	-	1899	370	105	-	-	0.0178

Table 7.2: Experimental results: Time in sec, limit 3600sec, "x": not applicable; "–": out of time/memory; "/": no need.

		Explicit Solution				**Simulation**		**Kronecker-based Approximation**						**EV*MDD-based Approximation**					
			Gauss-Seidel						Gauss-Seidel		Jacobi				Gauss-Seidel		Jacobi		
Model	N	T_{init}	T_{solve}	iters	meas.	time	meas.	T_{init}	T_{solve}	iters	T_{solve}	iters	meas.	T_{init}	T_{solve}	iters	T_{solve}	iters	meas.
kanban-4	5	122.38	43.13	267	0.6344	0.27	0.6288	0.06	0.00	10	0.00	10	0.6346	0.09	0.00	10	0.00	10	0.6346
kanban-4	15	-	-	-	-	0.65	0.5282	0.93	0.12	8	0.20	8	0.5319	22.08	0.09	8	0.14	8	0.5319
kanban-16	5	/	/	/	/	/	/	0.03	0.01	11	0.01	12	0.6344	0.04	0.00	11	0.00	12	0.6344
kanban-16	15	/	/	/	/	/	/	0.37	0.08	8	0.12	9	0.5318	0.26	0.06	8	0.07	9	0.5318
polling	5	2.43	0.15	58	0.8889	0.98	0.8890	0.03	0.00	6	0.00	16	0.8889	0.02	0.00	6	0.00	38	0.8889
polling	30	-	-	-	-	1.23	0.9815	1.02	0.19	31	-	-	0.9815	4.36	0.09	31	-	-	0.9815
robin	5	0.06	0.00	15	0.2048	0.53	0.2046	0.03	0.01	16	-	-	0.2046	0.02	0.00	16	-	-	0.2046
robin	15	-	-	-	-	2.29	0.0682	0.09	0.17	47	-	-	0.0682	0.11	0.02	45	-	-	0.0682
slot	5	1.86	0.10	49	0.9282	0.25	0.9284	0.02	0.01	25	0.01	25	0.9249	0.02	0.00	24	0.01	25	0.9249
slot	10	-	-	-	-	0.50	0.9326	0.05	0.10	64	0.13	63	0.9281	0.06	0.03	61	0.06	60	0.9281

Table 7.3: Experimental results for other models (Time in seconds, limit is 1800 secs), "–" indicates out of memory/time or exceed maximum number of global iteration (10,000)."/" indicates no need to carry on the experiment.

				Kronecker		**EV*MDD**					
						Disjunctive		Single root		Actual **R**	
Model	N	$\eta(\mathbf{R})$	Decomp.	matrs	nz	nodes	edges	nodes	edges	nodes	edges
fms	5	6.60×10^6	Kron	51	1587	1392	2942	1046	3342	39939	63587
			finest	x	x	924	1642	2374	4527	5481	8454
fms	6	3.23×10^7	Kron	51	2528	2202	4693	1330	4767	70673	116604
			finest	x	x	1087	2055	2904	5780	7637	12202
fms	10	4.25×10^9	Kron	51	9972	8642	18577	3164	15205	393995	715570
			finest	x	x	1969	4439	6071	13454	24615	43222
fms	20	7.33×10^{12}	Kron	51	71142	61842	132947	14259	92260	4992281	10073732
			finest	x	x	5059	14224	18916	46649	144515	284812

Table 7.4: Matrices/nonzeros, or nodes/edges, for Kronecker and EV*MDD encodings of **R**. "x" indicates not applicable.

Model	N	$\|\mathcal{S}\|$	Decomp.	nodes	$\sum_1^L \|\boldsymbol{\pi}_k\|$	$\max_k \|\boldsymbol{\pi}_k\|$	$\eta(\mathbf{A})$	$\sum_1^L \eta(\mathbf{R}_k)$	$\max_k \eta(\mathbf{R}_k)$
fms	5	8.52×10^5	Kron	1669	4779	756	5712	26335	4590
			finest	236	431	52	6531	1007	133
fms	6	3.84×10^6	Kron	2612	8414	1372	8979	48543	8673
			finest	301	579	74	9629	1433	202
fms	10	4.14×10^8	Kron	9824	45694	7986	33937	291245	55055
			finest	631	1421	202	37424	4007	628
fms	20	5.91×10^{11}	Kron	66634	547974	101871	229962	3795790	754110
			finest	1946	5626	1371	280144	17932	4848

Table 7.5: Memory statistics for for EV*MDD-based approximation.

We also report the results for some other models in Table 7.3: *kanban* and *polling* from [50], while *robin* and *slot* from [28]. *kanban*-4 and *kanban*-16 are the same model with different decompositions: the first one is decomposed into 4 partitions and the second one 16.

We report the number of nonzero elements of the overall transition rate matrix, the number of matrices and nonzero elements of a Kronecker representation, and the number of nodes and nonzero edges of an EV*MDD encoding in Table 7.4 for the FMS model; these can be seen as the memory requirements for explicit solution, Kronecker-based approximation and EV*MDD-based approximation, respectively. For the EV*MDD encoding, we compare the disjunctive and the single-root representations for a potential $\mathbf{R}$ as discussed in 6.3; we also build the EV*MDD encoding actual $\mathbf{R}$ and list its statistics in the last two columns as discussed in Sec. 7.3.3 for comparison.

In Table 7.5, we report detailed statistics for data structures used in our EV*MDD approximation for the FMS model, including size of $\mathcal{S}$, number of nodes in $r_{\mathcal{S}}$, total size ($\sum_{k=1}^{L} |\boldsymbol{\pi}_k|$) and maximum size ($\max_k |\boldsymbol{\pi}_k|$) for all aggregated CTMCs, number of nonzeroes in matrix $\mathbf{A}$, and total number and the maximum number of nonzeroes for the transition rate matrices of the aggregated CTMCs.

7.4.3 Evaluation.

Columnwise comparison. From Tables 7.2 and 7.5, we see that the explicit solution is feasible only when the number of states is in the millions, and that our EV*MDD-based technique requires much less time and its results are quite accurate, most of them well within 1%.

As N increases, we can only compare our approximation results with those from simulation. We realize that comparing the runtime of simulation and of our approximation is somewhat arbitrary, since either can be made very fast or very slow by appropriately changing the stopping criteria, but the conclusion remains that our approximation is quite accurate, within 10%, if we assume that the simulation results are reliable.

Finally, comparing with our previous Kronecker-based approximation when possible, we observe that the initialization time for the new method is usually a little worse but the computation time is much better in our more general setting, because using a single-root representation instead of a disjunctive encoding saves numerous additions of rates during the computation and because the implementation has improved. When both methods can be run, they produces essentially the same numerical results. We also observe that T_{solve} under Gauss-Seidel and Jacobi are different because they require different number of *global* iterations to finish and because, in each global iteration, they required different numbers of *local* iterations to converge. We run both of them since one is not always better than the other; however, for the explicit solution of all models, Gauss-Seidel always uses less time than Jacobi, thus we only listed the former.

From Table 7.4, we can see that both Kronecker and EV*MDD encodings are quite compact, improving by several magnitude on number of nonzeroes for explicit sparse storage. The disjunctive EV*MDD approach uses less than twice as many edges as the nonzeroes for Kronecker matrices, as discussed in Sec. 6.3. The single-root EV*MDD encoding we adopt for this chapter can use more or fewer nonzero edges than the disjunctive encoding and the memory usage of the three (or the latter two when the Kronecker is not applicable) encodings are in the same order of magnitude for a given decomposition. When the model becomes larger, the single-root encoding tends to perform better and use fewer edges than the disjunctive encoding, due to greater node sharing. The EV*MDD encoding the actual matrix **R** (i.e., removing fictitious entries) always use more edges, at times by several orders of magnitude, than when encoding the potential **R** (e.g., for FMS with $N = 20$, 10^7 versus 10^5 edges), which makes us adopt the latter.

Comparison between different decompositions. First, we can see that different decompositions do not appear to affect the results from the approximation algorithms, which attests to the stability of the approach.

For the finest decomposition, the Kronecker-based approximation is not applicable but this finer decomposition improves our new EV*MDD-based approximation by a large factor. For the same benchmark with the same parameter N, using a finer decomposition requires much less time in the initialization stage than the coarser decomposition, by up to several orders of magnitude (e.g., FMS with $N = 20$). This is mainly because a finer decomposition results in smaller local state spaces, which benefits both state-space generation and transition-rate-matrix generation (see Ch. 4 and Sec. 7.2), which counts for the most part of the initialization. Under a finer decomposition, T_{solve} is also greatly reduced, for both the Gauss-Seidel and Jacobi methods. The difference for Jacobi seems more apparent; we believe this to be model-dependent and when the model become larger, the improvement tends to be greater. This is mainly because each global iteration takes less time to finish, e.g., FMS with $N = 20$, 217 iterations require 18.03 seconds compared to 106 iterations requiring 453.94 seconds with a coarser decomposition. Furthermore, according to the complexity discussion in Sec. 7.3.5, the size of aggregated CTMCs and the number of nonzeroes for each $\mathbf{R}_k$ are greatly reduced with a finer decomposition, while the number of nonzeroes in $\mathbf{A}$ still has the same order of magnitude (see Table 7.5).

A finer decomposition also greatly reduces the memory required to store $\mathbf{R}$ for our EV*MDD encoding (see Table 7.4). When the model becomes larger, a disjunctive or single-root EV*MDD encoding is better than the Kronecker encoding with a coarser decomposition, e.g., FMS with $N = 20$, 14,224 and 46,649 edges compared to 71,142 nonzeroes.

Besides the "Kron" and "finest" decompositions, we also study the possibility to make the Kronecker decomposition finer by breaking transitions to see if this will help the Kronecker-based approximation. For example, since the rate of t_{P_1} depends on the sum of tokens in P_1, P_2 and P_3 and each of these places can have tokens from 0 to N, we need to break t_{P_1} into $(N+1)^3$ transitions where each transition corresponds to a possible value of the sum, similar for t_{P_2}, t_{P_3} and $t_{P_{12}}$. After this transfer, P_1, P_2 and P_3 no long needs to be merged to achieve Kronecker consistency, so we get a "finer" Kronecker decomposition by adding many new transitions, which actually changes the model but we still list it under the same model, in the "break" section of Decomposition column for convenience of comparison. We can see from the table that this change does not help the Kronecker-based approximation; it makes both T_{init} and T_{solve} much worse compared to the "Kron" decomposition due to excessive number of transitions. For the EV*MDD-based approximation, it is worth noticing that the computation time T_{solve} is not affected much because we use single-root EV*MDDs.

In summary, from the above observations we can conclude that (1) our new method provides quite accurate results, less than 10% relative error in our experiments; (2) when Kronecker-based and EV*MDD-based approximation are both applicable for the same decomposition, the EV*MDD-based method uses less time to produce the same results; (3) a finer decomposition improves the time and memory requirements of our new method, which can be much better than the Kronecker-based approximation with a coarser decomposition, while still producing results of similar accuracy; (4) since our method is applicable to any decomposition, we can always choose the "finest" decomposition; (5) our EV*MDD encoding of the transition rate matrix allows a seamless and efficient construction from the transition relation MDDs built during state-space generation, which makes our methods eminently applicable when paired with a symbolic state-space generation such as that of Ch. 4; furthermore, (6) symbolic generation of the state-space and transition rate matrix also benefits from a finer decomposition.

7.5 Conclusion

We presented an approximate numerical solution for the computation of stationary measures for a large structured Markov model described in a compositional way. Unlike the previous method based on a Kronecker representation of the transition rate matrix, the new approach is completely general and can be applied to any model, regardless of how it is structured. At the same time, the new approach is at least as efficient as the old one, when both can be applied.

Our approximation algorithm is accurate and can be used to study large models whose exact numerical solution is infeasible. Since it uses the exact state space encoded as an MDD and computes L aggregated CTMCs defined upon macro-states corresponding to the edges at each level of the MDD, the runtime and memory costs are ultimately limited by the number of these macro-states. This limit is enormously less stringent than for an exact solution in practical models, as we demonstrated on a benchmark suite. Future applications of this work will target model checking of stochastic models with an underlying CTMC.

Chapter 8

The Decision Diagram Library

We have seen a few applications which show the effectiveness of our newly-defined and generalized decision diagrams. Decision diagrams (DDs) can be used to encode arbitrary sets of structured data or functions defined on them and reduced the size of the problem by a large factor. The operations between data sets, between sets and relations, between matrices, or anything encoded by decision diagrams, then turn to be operations applied on decision diagram nodes. With the use of operation caches, those symbolic operations can become much more efficient than their explicit correspondences as shown in numerous experimental results.

This chapter introduces the Decision Diagram Library. It consists of two parts, user's manual and developer's reference. This library is still in the alpha phase and will be publicly released soon. This chapter describe the current version of the library. Please refer to the manual file in the release package for up-to-date information.

8.1 User's Manual

A user of the library is defined as a person who needs to manipulate decision diagrams only through the exported functions or the interfaces of the DDL. The rich set of functions included in the DDL package allows many applications to be written in this way. The following illustration assume you are familiar with C++ basics, templates, and some STL [3]. We use some abbreviations for convenience, `lhs` denotes the left-hand side of an assignment while `rhs` denotes the right-hand side and we use `type` and `class` interchangeably.

8.1.1 C++ interface

To build an application that uses the DDL C++ interface, you should add

```
#include "ddl.h"
using namespace DDL;
```

to your source files and link `libddl.a` to your executable. So far the library is tested under Linux, FreeBSD, and MacOS, and work with both 32 and 64 bit systems.

8.1.2 Basic manipulation

The following fragment of code is an example to create an MDD node, modify its arcs, and insert it into the unique table.

```
mdd p = g_mddf.new_node(2,2);           //p.lvl = 2, p.size = 2
p.set_child(0,nodeONE);                 //p[0] = terminal ONE
p = g_mddf.insert_ut(p);                //insert into unique table
bigint n = g_mddf.num_paths(p);         //count number of paths
                                        //leading to terminal ONE
cout << "num paths = " << n << endl;
p.print(cout);                          // print to screen
```

```
The output is:
num paths = 1
Node(0x99f0270)<0C0> L2 I1 [(0:0x99f0218)]
/* the first 0x.. is p’s address,  <0C0> meaning p is canonical node,
    more details exposed later */
```

`bigint` is the type for arbitrary size integers in DDL, it can be just used the same way as `int`.

Note that all functions are in lower case, which is the convention in DDL, as it is in STL. We define an object instead of an pointer to an object. This is because we use smart pointers with a reference count scheme, similar to that in the Boost [1] library. The memory allocation and deallocation for nodes is done automatically. The assignment `=` will not cause copying of the `rhs` but just increase the reference count of the real object under smart pointer by 1, please refer to the developer’s manual for more details.

8.1.3 Nodes

Nodes are the least unit defined in DDL. The base class for DD nodes is `nodeptr`; TDDs, EV$^+$MDDs, and EV*MDDs are handled by type `mdd`,`idd`, and `rdd`, respectively which inherit `nodeptr`.

Here are predefined node constants:

- `nodeONE` terminal node **1** for MDDs.
- `inodeONE` terminal node Ω for EV$^+$MDDs.
- `rnodeONE` terminal node Ω for EV*MDDs.

Three different terminal nodes are required to enforce *type check*. These three node types cannot be explicit or implicit converted to one another.

We do not explicitly store edges pointing to δ and δ itself for TDDs, edges with value ∞ for EV$^+$MDDs or edges with value 0 for EV*MDDs; those edges are `NULL` pointers instead.

In the following description, "edge" i meaning the edge label with i, where i is the *index* and "child" i means the destination node of edge i.

Common interface functions for type `nodeptr`

Here we list the common interface function for `nodeptr`, so they can be used by all DD node types including `mdd`, `idd` and `rdd`.

- `int level() const` level (lvl) of node; $p.lvl = 2$ means $p.var = x_2$, and negative level is used to denote primed variables, e.g., $p.lvl = -2$ means $p.var = x'_2$.
- `int size() const` size of the node; setting outgoing edges with an index larger or equal than the size is forbidden.
- `int min() const` return the minimal non-`NULL` child index.
- `int max() const` return the maximum non-`NULL` child index.
- `bool is_canonical() const` return `true` iff the node is canonical (Sec. 3.2.1).
- `nodeptr copy() const` make a copy of the node.
- `void print(ostream &os) const` print the node to `os` in a predefined format.

- `nodeptr first(int &idx)` return the first non-`NULL` child and assign its index to `idx`, return `NULL` if all its children are `NULL`.

- `nodeptr next(int &idx)` return the next non-`NULL` child and assign its index to `idx`, return `NULL` if it already reach the last child or all following children are `NULL`.

  These two functions are not `const` because they change a built-in forward-iterator. The following fragment shows the preferred way to iterate over all non-`NULL` children

  ```
  int idx;
  mdd ch = p.first(idx);  //p:type mdd; idd, rdd are similar
  while(ch){
      //do something
      ch = p.next(idx);
  }
  ```

- `nodeptr child(int idx) const` return the child `idx`.

- `void set_child(int idx, nodeptr ch)` set edge `idx` to node `ch`.

Function only for type `mdd`

- `static mdd new_terminal(int v)` create a terminal with value `v`

- `mdd ext()` return the [$*$] part of the node (Sec. 3.1.1).

- `void set_ext(mdd ch)` set [$*$] to be `ch`.

The following functions can be used to simulate BDDs

- `mdd child0() const` return child 0.

- `mdd child1() const` return child 1.

- `void set_child0(mdd ch)` set child 0 to `ch`.

- `void set_child1(mdd ch)` set child 1 to `ch`.

Function only for types `idd` and `rdd`

`evnodeptr` is the template type for all EVDDs, which inherit `nodeptr`. `idd` is actually `evnodeptr<int>`, while `rdd` is `evnodeptr<double>`.

`T` is either `double` or `int` in the following code.

- `evnodeptr<T> first(int &idx, T &v)` return the first non-`NULL` child, assign its index to `idx`, assign edge `idx`'s value to `v`.

- `evnodeptr<T> next(int &idx, T &v)` return next non-`NULL` child, assign its index to `idx`, assign edge `idx`'s value to `v`.

- `T value(int idx) const` return edge `idx`'s value.

- `void set_value(int idx, T v)` set edge `idx`'s value to be `v`.

8.1.4 Forests

A *forest* is a group of DD nodes of the same type. A forest takes charge of node creation/deletion, unique table check-in, node type-check and many other operations that require input of DD nodes of the corresponding type. A forest has its own unique table and operation cache, so nodes in different forest are not shared.

The base template class for forests is `forestptr<T>`, where `T` is the node type. There are two forest types for `mdd`. `mddf` inherits `forestptr<mdd>`, which does not store the reduction rule vector, thus does not support *CheckIn* and *Translate* operations of Sec. 3.2. `grmddf` inherits `mddf` to include the reduction rule vector and a vector to store $|\mathcal{X}_k|$, in order to apply the above two operations.

The forest type for handling `idd` and `rdd` is `iddf` and `rddf`, respectively, which inherits `forestptr<idd>` and `forestptr<rdd>`, respectively.

We provide three predefined global forest objects (actually smart pointers, too) , one for each node type, but also provide the flexibility to create a new forest. This is especially useful when it is undesirable to share nodes of with those of another forest with the same type. The global forest are `g_mddf`, `g_iddf` and `g_rddf`.

Supported functions from common `forestptr<T>` interface

Here are the function supported by all forest types, some functions not covered here are deferred to later subsections for cache and output.

- `static forest<T> new_forest()` create a new forest.
- `int height()` return L.
- `void set_height()` set L, not allowed in `grmddf`.
- `T new_node(int k [,int sz])` create a node with level `k` and size `sz`. If `sz` is not set, the function will check if $|\mathcal{X}_k|$ is set (for `grmddf`) and then use $|\mathcal{X}_k|$ for the size.
- `T insert_ut(T p)` insert temporal node `p` into the unique table and return the canonical node. For `grmddf`, it is actually a *CheckIn* function, which takes care checking whether the node is redundant or singular and breaks the reduction rules,

as well as for duplicate nodes. For other forest types, it is just the *UTInsert* function used to avoid duplicate nodes.

- `T reduce(T p)` the *Reduce* function.
- `T transfer(nodeptr p)` this function transfers node `p` to the same type as nodes in the forest if it is not. We discuss the transfer from `mdd` to `idd` and `rdd` in Ch. 5 and Ch. 7 respectively. The transfer from `idd` and `rdd` to `mdd` is the reverse operation, and the transfer between `idd` and `rdd` is equivalent to transferring one to `mdd` then transferring the resulting `mdd` to the other.

Functions only for type `mddf`

- `mdd or_qq(mdd p, mdd q)` the union of MDDs p and q , assume $\boldsymbol{\rho}^{Q}$ (Fig. 2.7).
- `mdd or_ff(mdd p, mdd q)` the union of MDDs p and q , assume skipped variables are fully-reduced.
- `mdd or_fi(mdd p, mdd q)` the union of MDDs p and q , assume $\boldsymbol{\rho}^{FI}$ or $\boldsymbol{\rho}^{QFI}$ (Fig. 4.10).
- `mdd or_zs(mdd p, mdd q)` the union of MDDs p and q , assume skipped variables are 0-reduced (zero-suppressed).
- `mdd minus_qq(mdd p, mdd q)` the *Difference* function, assume $\boldsymbol{\rho}^{Q}$ (Fig. 2.7).
- `mdd and_ff(mdd p, mdd q)` the intersection of p and q, , assume skipped variables are fully-reduced (Fig. 4.8).

- `mdd and_zs(mdd p, mdd q)` the intersection of p and q, , assume skipped variables are 0-reduced (zero-suppressed).
- `mdd relprod(mdd p, mdd q)` the *RelProd* function (Fig. 4.4).
- `mdd ewor_zs(mdd p, mdd q)` *elementwise* union for zero-suppressed BDDs

Fig. 8.1 shows the algorithm for elementwise union ($EWOr$). We use type *bdd* for BDD nodes in the pseudocode. An elementwise operator $\cdot$ on two tuples $\mathbf{i} = (i_L, ..., i_1) \in \mathbb{N}^L$, $\mathbf{j} = (j_L, ..., j_1) \in \mathbb{N}^L$ is defined as $\mathbf{i} \odot \mathbf{j} = \{i_L \odot j_L, ..., i_1 \odot j_1\}$. An elementwise operation on two sets $\mathcal{Y}$ and $\mathcal{W}$ according a generic elementwise operator $\odot$ is defined as $\mathcal{Y} \odot \mathcal{U} = \{\mathbf{i} \odot \mathbf{j} : \forall \mathbf{i} \in \mathcal{Y}, \mathbf{j} \in \mathcal{U}\}$. For $EWOr$, the operator is logical *or* or $\vee$. More details in Ch. 9.

Functions only for type `grmddf`

For `grmddf`, a reduction rule vector is explicit, and for fully-reduced variable x_k, $\mathcal{X}_k$ must be set explicitly. A `grmddf` is fixed to a certain $\boldsymbol{\rho}$ and a selection of $\mathcal{X}_k$, so the function encoded by a node is fixed. This is unlike the reduction-rule vector in `mddf`, which is implicit, so that the function encoded by a node can be depending on $\boldsymbol{\rho}$ we use to interpret it. The advantage of `grmddf` is that, it can support any $\boldsymbol{\rho}$, and can support the generic *Apply* function. Instead, `mddf` support only a limited combination of reduction rules, and might need to define a particular set of operations to support to a specific $\boldsymbol{\rho}$. `grmddf` use strict *CheckIn* while `mddf` use *UTInsert* which might cost less. Between these two, the user should choose one according to his needs If it is known that $\boldsymbol{\rho}$ is some predefined vector such as $\boldsymbol{\rho}^{QFI}$, then `mddf` is a good choice. If the main concern is to reduce memory usage,

```
$bdd\ EWOr(bdd\ p, bdd\ q)$
if $p = \mathbf{0}$ or $q = \mathbf{0}$ then return $\mathbf{0}$;
if $p = q$ then return $p$;                                   • *trivial cases*
if $CacheHit(EWOR, p, q, r)$ return $r$;                      • *check cache*
if $q.var \succ p.var$ then $Swap(p, q)$;                     • *commutativity*
$r \leftarrow NewNode(p.var)$;
if $p.var \succ q.var$ then                                   • $q.var$ *is skipped at this level*
  $r[0] \leftarrow EWOr(p[0], q)$;
  $r[1] \leftarrow EWOr(p[1], q)$;
else
  $r[0] \leftarrow EWOr(p[0], q[0])$;
  $r_{01} \leftarrow EWOr(p[0], q[1])$;
  $r_{10} \leftarrow EWOr(p[1], q[0])$;
  $r_{11} \leftarrow EWOr(p[1], q[1])$;
  $r[1] \leftarrow Or(r_{01}, r_{10})$;
  $r[1] \leftarrow Or(r[1], r_{11})$;                          • $or(0,1) = or(0,1) = or(1,1) = 1$
$r \leftarrow UTInsert(r)$;
$CacheAdd(EWOR, p, q, r)$;
return $r$;
```

Figure 8.1: Elementwise union for zero-suppressed BDDs

then probably one should adopt `grmddf` and switch to a $\boldsymbol{\rho}$ that result in less nodes (this is an ongoing research).

When a `grmddf` forest is created, L needs to be set in order to initialize vector $\boldsymbol{\rho}$:

```
grmddf f = grmddf::new_forest(l); // some l is required here
```

Type `reduction_rule` is an enumeration type for reduction rules:

```
enum reduction_rule {Q,F,I};
```

After initialization of `grmddf`, $\boldsymbol{\rho}$ is initialized to $\boldsymbol{\rho}^Q$ and $|\mathcal{X}_k|$ is initialized to 0 for every k, $1 \leq k \leq L$ (this will not be a problem if there is no fully-reduced variable, and function `new_node` have size specified, otherwise, DDL will generate error message if you forget to set it).

- `set_rr(int k, int c)` set $\rho_k = c \in \mathbb{N}$.
- `set_rr(int k, reduction_rule r)` set $\rho_k = r$.
- `set_sz(int k, int sz)` set $|\mathcal{X}_k| = sz$.

  Note that, the above functions can only be called once per variable per forest , and must be invoked before any node creation. To change $\boldsymbol{\rho}$ or $|\mathcal{X}_k|$, one must use `translate` below.
- `mdd translate(mdd p, grmddf &f)` the *Translate* function in Sec. 3.2.2, it will translate p into q in another `grmddf f` (so conforming to `f`'s $\boldsymbol{\rho}$).

Functions only for type `iddf`

- `idd min(int vp, idd p, int vq, idd q, int &v)` the *Minimum* function (Fig. 2.8),

$\langle vp,p\rangle$ and $\langle vq,q\rangle$ are the two input edges, set `v` to the value of the result edge and return the result node.

Functions only for type `rddf`

- `rdd multiply(rdd p, rdd q, double &v)` *Multiply* operation (Sec. 6.2.2).
- `rdd add(double vp, rdd p, double vq, rdd q, double &v)` *Add* operation (Sec. 6.2.2).
- `rdd transpose(rdd p)` return the `rdd` node encoding the transpose of the original matrix encoded by `p`. This function assume p encodes a matrix, so every path from p should be interleaved with nodes associated to an unprimed variable and nodes associated to a primed variable; otherwise, the function aborts and generates an error message.
- `double normalize(rdd p, rdd q, double &v)` *Normalize* (Fig. 6.2).
- `double filter(rdd p, mdd q, double &v)` use a `mdd` q to filter out those transition rates that are not actually possible, but encoded by p in a potential encoding (Ch. 6). Used to obtain the `rdd` encoding actual $\mathbf{R}$.

8.1.5 Cache

Operations of decision diagrams are usually performed in a recursive manner, which requires the use of an *operation cache* to store computed results. The cache check and add steps are almost in every forest functions in the library package, using a built-in cache for each forest. Users can use this built-in cache as well as build their own caches.

Too small a cache will cause frequent overwriting of useful results. Too large a cache will cause overhead, because the entire cache is scanned every time garbage collection takes place. In DDL, we use a scheme where the cache can automatically grow or shrink. Users can control the maximum grown size by setting parameters introduced in the next subsection.

The type for cache is `cacheptr`. A cache entry is a tuple where the first component is an `OperatorCode`, usually one for each operation. Any operator can be registered using function

`static cache::add_operator(OperatorCode, int numOps, bool commu)`

The latter two parameters for number of operands and whether it is commutative (when `numOps` is 2, otherwise it is not used).

The following cache manipulating functions are member functions of `forestptr`

- `cacheptr new_cache(long n)` build a cache with initial size `n`
- `void delete_cache(cacheptr t)` delete cache `t` created by user. It is not a problem if a user creates but forget to delete a cache, since those caches will be destroyed when the forest is destroyed, both are done automatically for smart pointers.
- `void clear_cache(cacheptr t)` clear all entries in the cache, e.g., when the entries are no longer valid. This might be useful for certain functions. Built-in caches cannot be cleared or deleted, so if there is such function to implement, use `new_cache`.

- `cache_add(...,[,cacheptr t])` add the cache entry into the built-in cache (default) or user's cache `t`.

- `cache_hit(..., bool &hit [,cacheptr t])` search if there is a match in the cache, and set `hit` to `true` and return the result, or assign `hit` to false and return `NULL`.

8.1.6 Setting environment parameters

There are several DDL parameters can be set to control the execution of an application. Below are the most important global functions for this purpose,

- `set_membound(size_t m)` set the memory bound in megabytes, this memory bound is for the total memory usage, including nodes and caches.

- `set_clupthresh(long n)` set clean up threshold. Details can be found in Sec. 8.2.7. Usually, when `n` is larger, the application uses less time but consumes more memory (so memory bound needs to be enlarged accordingly). The default threshold of 1000 is adopted if it is not set. Users can experiment with them for applications with different scale.

8.1.7 Output

This subsection introduces library functions for outputting data. It is possible to generate a graph of the DD structure, report statistics such as node usage, number of paths (states), and even see an "movie" showing how a DD forest grows and shrinks during manipulations.

Statistics

The following functions are member functions of `forestptr<T>`:

- `int num_nodes(T p)` count the total number of nonterminal nodes reachable from `p`.
- `void num_nodesnarcs(T p, long &nds, size_t &arcs)` count the total number of nodes and arcs of the DD rooted at p.
- `bigint num_paths(T p)` count the total number of different paths leading from p to **1** for `mddf`, different non-∞ paths from p to Ω for `iddf`, or different non-0 paths from p to Ω for `rddf`. Often used to compute number of states when $p = enc(\mathcal{S})$.
- `report(ostream &s)` print statistics report, below is a sample report showing the memory and node usage for both unique table and operation cache.

```
-----------------------------
Unique Table:
-----------------------------
Current         memory: 4768    size: 863       #elements: 9
Maximum         memory: 115044  size: 1733      #elements: 1724
Worst search length: 8
Average search length: 1.3864
-----------------------------
Operation Cache:
-----------------------------
Current         memory: 4984    size: 1239      #elements: 7
Maximum         memory: 13980   size: 2541      #elements: 954
Worst search length: 6
Average search length: 1.04681
Number of Cleanups 2
```

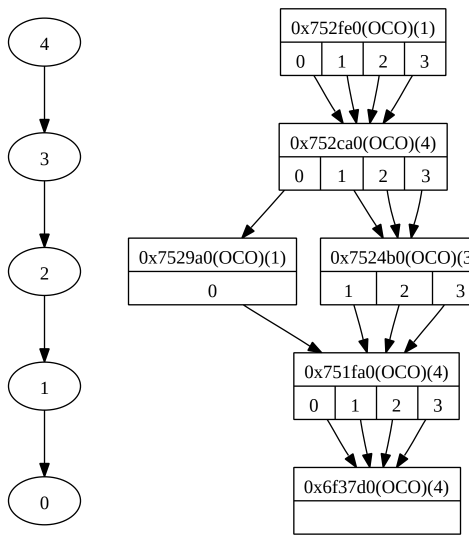


Figure 8.2: A sample MDD graph generated by `dot`

Decision diagram graph

One can print out the DD nodes and DD structure using of following functions (also member function of `forestptr<T>`):

- `print_nodes(ostream &os [,T p])` print p and all its derivative nodes in a top down manner, in a predefined format; if p is omitted, will print out all nodes in the unique table.

- `print_nodesdot(char *file [,T p])` print p and all its derivative nodes into a dot file; if p is omitted, print all nodes in the unique table. One can further use tool `dot` [2] to generate a `.ps` file. Fig. 8.2 is a sample output file with levels listed on the left.

The Movie module

We are building a movie module based on OpenGL, which can show the growing and shrinking of DDs during execution of your application. More details will be presented with the formal release of the software.

8.2 Developer's Reference

DDL is designed to be an open library. A Developer of the library is defined as a person who has the source code of DDL and is willing to extend or complement the functions of the library, or port the library to a new platform.

Sometimes, when writing a sophisticated application based on decision diagrams, efficiency might dictate that some functions be implemented as direct recursive manipulation of the diagrams, instead of being written in terms of existing primitive functions. This section gives insights into the implementation of DDL for reference and to help adding your new DD functions to the library.

8.2.1 Compile and linking

The library source is in a tar ball, `ddl.tar.gz`, extract all the files and run `./Make` in the `src` folder, it will generate the lib file `libddl.a` in `lib` folder. The `doc` folder contains up-to-date documents for DDL.

Below are the other options for compiling.

```
Make gdb        //debugging mode
Make pg         //debugging model with profile
Make test       //generate test executable
Make install    // install the lib and header files,
                // use ./configure --path= to change destination
```

8.2.2 Dependency tree

Fig. 8.3 shows the dependency tree for the classes in DDL. These classes defines real objects while interface classes including `nodeptr`, `forestptr`, and `cacheptr` wrap a

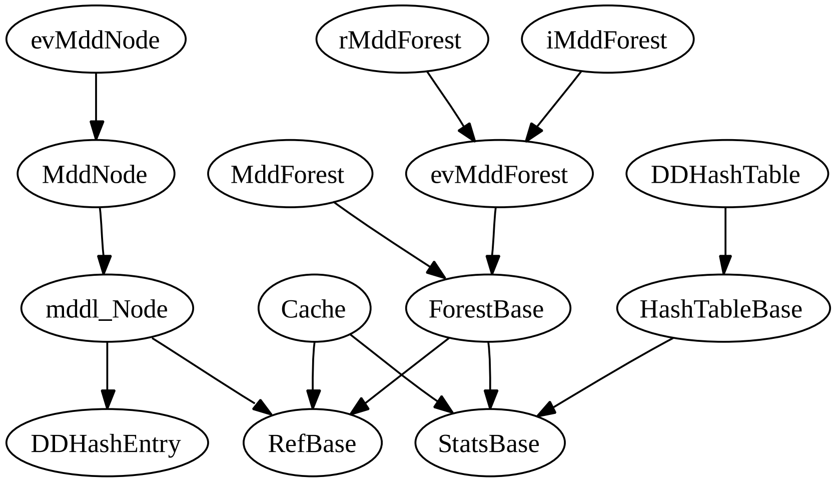


Figure 8.3: The dependency tree for classes in DDL

real pointer to an actual object, to hide implementation details and greatly simplify use of DDL. For example, `nodeptr` wraps a `mddl_Node` pointer. For this section, when we say node, we mean the actual node, and refer to smart pointer as pointer to the node, although it looks like an object. You can refer to the Boost library for more details on smart pointers.

`class StatsBase` is the interface for statistics report. Classes inheriting it share similar statistics computation and output format. `class RefBase` is the interface for a basic reference count scheme. It enables its inherited class to be wrapped by a smart pointer. `class DDHashEntry` is the interface for any type that needs be hashed in `class DDHashTable`. It adds a `next` pointer and makes the type linkable. Besides that, it defines functions required by `DDHashTable`, so they must be implemented in the derived type: function `Signature` is used to compute the hash value and function `Equals` decide whether two entries are equal.

Now we introduce the four main components of DDL: node, forest, unique table, and cache.

8.2.3 Node

The base class for DD nodes is `mddl_Node`, which inherits `DDHashEntry` and `RefBase` so that it is hash-able and can be wrapped by a smart pointer (`nodeptr`).

A `mddl_Node` has following notable fields:

- `inc` the reference count.

- `num_marked` a `static` data member to record the number of nodes marked for deletion, required for garbage collection.

- `Stat` node status, a 4 bit struct, each bit corresponds to `marked, canonical, sparse`, and `dead` respectively. `marked` and `dead` are used for garbage collection, `canonical` is used for check-in. If `canonical` is not set, then this node is a temporal node. All newly created nodes are temporal nodes.

  A node is an array of pointers to its children. There are two storage methods for this array: `sparse` just stores non-`NULL` children in a sparse array, while `truncate-full` stores a full array. If the `sparse` bit is not set, then this node uses `truncate-full` storage. All temporal node are `truncate-full`. Canonical nodes can use either of these two storage methods; this is decided when a temporal node is checked into the unique table.

Sparse storage is perfect for an forward-iterator scheme, but not good for random access, and the opposite is true for truncate-full. So one should avoid using random access function such as `mdd::child(idx)` for sparse nodes, and use `first()-next() scheme` instead for efficiency.

8.2.4 Forest

The base class for forests is a template `class Forest<T>`, which inherits `RefBase` and `StatsBase`, so that it can be wrapped into a smart pointer (`forestptr`) and have standard statistics report (function `report()`)

The most important functions of a forest are providing type check, providing node manipulation functions and storing the unique table and cache.

A forest class which inherit from `Forest<T>` can only work on node type of `T`, thus avoiding errors due to mixing different types of nodes. A forest has its own unique table to allow node sharing only inside the forest.

A forest keeps a built-in cache and a link-list of user-defined caches; when a cleanup (garbage collection) is invoked or the forest is destroyed, the forest is responsible to traverse its caches and then the unique table, to delete all entries and nodes, or destroy them entirely.

8.2.5 Unique table

The unique table (UT) is a `DDHashTable` with real DD node as entries, not pointers or smart pointers. It is implemented as an array of linked-lists. A unique table is affiliated with a forest and it behaves as follows

- When there is *UTInsert* call (`forestptr::insert_ut`), it computes the hash value of the temporal node, checks if there is a duplicate in the UT already; if yes, it returns the one already in the UT; otherwise, it make a copy of the node and changes the storage type to an optimal one, then puts the copy into UT and returns the pointer to it.

- When there are too many nodes or too few nodes, UT will grow or shrink accordingly.
- It provides a `RemoveSatisfying()` function to remove obsolete nodes (with `marked` bit set) during cleanup.

8.2.6 Cache

The base class for caches is a `class Cache`, which inherits `RefBase` and `StatsBase`, so that it can be wrapped into a smart pointer (`cacheptr`) and has standard statistics report (function `report()`)

A cache is basically a set of hash tables, each to deal with a type of cache entries. A cache entry consists of three parts, `OperatorCode`, a list of operands and a list of results. The variance in the number and type of operands as well as of results makes for different cache entry types.

One can define new cache entry types and for each new type, a overloaded `RemoveSatisfying()` in the `Forest` is required class to ensure obsolete entries are deleted correctly during cleanup.

8.2.7 Garbage Collection

The garbage collection in DDL is performed through reference counting with a "lazy" policy. With the adoption of smart pointers, increasing and decreasing reference count as well as clean up is done automatically. Below are the details.

- When a node is referenced, either through assignment (`=`), or setting an edge (`nodeptr::set_child`), its reference count increases by 1. If its `marked` bit is set, then this node is *revived* and the `marked` bit is unset, and `mddl_Node::num_marked` decreases by 1.

- When a node is dereferenced, e.g., the `nodeptr` holding it now assigned to a new node, or edges pointing to it pointing to other nodes, its reference count decrease by 1. If the reference count become 0, the `marked` bit is set and `mddl_Node::num_marked` increases by 1.

- Before a forest creates a new node, it checks if `mddl_Node::num_marked` exceeds the clean up threshold (`set_clupthresh`). If so, the clean up is invoked in this forest.

- Then, the forest traverses UT, checks all nodes that are marked, and recursively dereferences its children and sets the `dead` bit for any node that has 0 reference count.

- Then, the forest traverse all its caches and deletes entries that have one operand or result with the `dead` bit set (obsolete entries).

- Finally, the forest deletes all nodes in the unique table with dead bit set ; since those nodes are not reference by other node or external pointers, their deletion is safe.

8.2.8 A sample function

Now we illustrate code of the `or_qq` function in DDL, as a guide for writing a library functions. All library functions share some common properties, note the comments.

```
// Union assuming no skipping variables
mdd
//=============================
mddf::or_qq(
//=============================
        mdd p,
        mdd q)
{
    // terminal cases are always the first to check
    if(!p) {
        MDDL_ASSERT(!q || q.is_canonical());
        return q;
    } else if(!q){
        MDDL_ASSERT(!p || p.is_canonical());
        return p;
    }

    // canonical nodes are required for all operations that need cache
    MDDL_ASSERT(p.is_canonical() && q.is_canonical());

    // trivial cases
    if (p == q) return p;

    // use assertions to ensure correctness
    int k = p.lvl();
    int kq = q.lvl();
    MDDL_ASSERT(k == kq);

    // commutativity,
    if (p > q) SWAP(p,q);

    //check cache
    bool hit;
    mdd answer = cache_hit(MDD_UNION_QQ, p, q, hit);
    if (hit) return answer;

    // use the maximum size for union
    int sz = MAX(p.size(),q.size());
    answer = new_node(k,sz);

    // the standard forward access
    int chdp_idx,chdq_idx;
    mdd chdp = p.first(chdp_idx);
    mdd chdq = q.first(chdq_idx);
    while(chdp && chdq){
        // consider all possible cases
        if(chdp_idx < chdq_idx){
            answer.set_child(chdp_idx,chdp);
            chdp = p.next(chdp_idx);
        } else if (chdp_idx > chdq_idx){
            answer.set_child(chdq_idx,chdq);
            chdq = q.next(chdq_idx);
        } else {
            // recursive invoke on children
            mdd chd = or_qq(chdp,chdq);
            answer.set_child(chdp_idx,chd);
            chdp = p.next(chdp_idx);
            chdq = q.next(chdq_idx);
        }
    }
    // don't forget the rest of node
    while(chdp){
        answer.set_child(chdp_idx,chdp);
        chdp = p.next(chdp_idx);
    }
    while(chdq){
        answer.set_child(answer,chdq_idx,chdq);
        chdq = q.next(&chdq_iter, &chdq_idx);
    }

    //ut insertion, we should always return a canonical node
    answer = insert_ut(answer);

    // add to cache!
    cache_add(MDD_UNION_QQ, p, q, answer);

    // done
    return answer;
}
```

Chapter 9

Summary and Future Research

Decision diagrams are key data structures for symbolic verification. In this thesis, we proposed a new canonical form of decision diagrams, namely finite ordered multi-way terminal-valued decision diagrams (TDDs), which generalize traditional multi-terminal multi-way decision diagrams by associating each variable with a reduction rule and allow an infinite variable domain. We have illustrate the suitability and effectiveness of TDDs through a new state-space generation framework for asynchronous models and its seamless adaption on a type of timed synchronous models.

To efficiently encode large transition rate matrices, we devised a new type of edge-valued decision diagrams that retains the compactness of previous Kronecker encoding, while being applicable to arbitrary CTMC decompositions. We integrated this new data structure into an approximate steady-state solution for large ergodic models.

These decision diagram algorithms make logic and timed verification more efficient and more tractable for large-scale systems. For users who would like to adopt these algorithms or developers who would like to explore more applications of this extrinsically simple but intrinsically powerful data structure, we also provide the Decision Diagram Library, with a friendly interface to manipulate decision diagrams.

Although the applications in thesis are limited to verification, decision diagrams are suitable to encode any structured data set or function and are potentially useful beyond verification, as an alternative to ordinary explicit algorithms.

In the following, we suggest some promising future research directions or topics, which would extend the work of thesis, and hopefully would provide some inspiration for computer science researchers.

Automatically discovery of proper reduction rule vectors for TDDs, mentioned in Ch. 3 and Ch. 8. We know a different $\boldsymbol{\rho}$ may result in a different TDD size (i.e., number of nodes). When memory is the main concern, we might want to use as few nodes as possible. This would require a monitor that keeps checking the TDD, discovers the good vector for current time point, and updates it periodically.

Use infinite domain beyond transition relation encodings. Transition relation encoded by MDDs paired with the domain $\mathbb{N}$ is a perfect match for on-the-fly state-space generation, as shown in Ch. 4. We limited our use of infinite domain to encoding a transition relation in this thesis and we assume the reachable state space $\mathcal{S}$ is finite. When $\mathcal{S}$ is infinite, it is possible that we can adopt this infinite domain and use a finite MDD to encode it. This suggests a new way to model checking a restrict class of infinite-state systems.

Reachability analysis for a timed system with an infinite set of possible firing times. This is an extension of work done in Ch. 5. If, for a transition t, $\mathcal{F}(t)$ has infinite number of possible fire times, but there exists a integer n_t such that $\{k \in \mathbb{N} : k > n_t\} \subseteq \mathcal{F}(n)$, then $\mathcal{F}(n)$ can be encoded in an MDD where the variable corresponding to t has domain $\mathbb{N}$.

A similar approach might work when $\mathcal{F}(t)$ is a *regular* infinite set, for example, all even numbers greater than some n_t. Another extension is to allow $\mathcal{F}$ to be a dense set, this requires extending MDDs to encode dense sets and could be an interesting new class of decision diagrams.

Approximation methods for CSL [7] model checking. Steady-state measures are important performance properties that can be expressed by continuous stochastic logic (CSL) [7]. However, the explicit computation of this measure might not be feasible when the CTMC is large. Our EV*MDD-based approximation technique may be applied in this field when an approximate steady-state measure is adequate.

Applications of elementwise operations. We mentioned the elementwise union operation (*EWOr*) in Fig. 8.1. Distinguished from the *Apply* type of operations, this operation applies to each component of the elements and on all pairs of elements (Cartesian product) of two sets. If the size of the two sets are N and M , then complexity of the explicit computation is $O(N \cdot M \cdot L)$, but a symbolic operation based on BDDs/MDDs would reduce this complexity to $O(\sum_{k=1}^{L} N_k \cdot M_k)$, where N_k and M_k is the number of nonzero arcs at level k. In practice, due to the node sharing and the operation cache, the symbolic operation

```
bdd KeepMinimal(bdd p)
if $p = \mathbf{1}$ then return $\mathbf{1}$;
if $p = \mathbf{0}$ then return $\mathbf{1}$;                    • terminal cases
if $CacheHit(MINIMAL, p, r)$ then return $r$;
$r_0 \leftarrow KeepMinimal(p[0])$;
$r_1 \leftarrow KeepMinimal(p[1])$;
if $r_0 = r_1$ then return $r_0$;                    • early return
$r \leftarrow NewNode(p.var)$;
$r[0] \leftarrow r_0$;
$u \leftarrow EWOr(r_0, r_1)$;
$r[1] \leftarrow Difference(r_1, u)$;                    • delete non-minimal elements
$r \leftarrow UTInsert(r)$;
$CacheAdd(MINIMAL, p, r)$;
return $r$;
```

Figure 9.1: The *KeepMinimal* algorithm.

would be much more efficient than its explicit correspondent.

An immediate application for *EWOr* is to find the *minimal* elements/tuples in a set. We define "$\geq$" on two tuples $\mathbf{i} = (i_L, ..., i_1) \in \mathbb{N}^L$, $\mathbf{j} = (j_L, ..., j_1) \in \mathbb{N}^L$, such that $\mathbf{i} \geq \mathbf{j} \Leftrightarrow \forall k, 1 \leq k \leq L, i_k \geq j_k$. A tuple $\mathbf{i} \in \mathcal{Y}$ is minimal if there does not exist $\mathbf{j} \in \mathcal{Y}, \mathbf{j} \neq \mathbf{i}$, such that $\mathbf{i} \geq \mathbf{j}$. Then, following Algorithm *KeepMinimal* in Fig. 9.1 keeps only the minimal elements for a set of binary tuples encoded by BDD p. The algorithm uses a divide-and-conquer paradigm, typical of decision diagram recursions, based on the observation that if $\mathbf{j} \neq \mathbf{i}$, then $\mathbf{i} \geq \mathbf{j} \Leftrightarrow or(\mathbf{i}, \mathbf{j}) = \mathbf{i}$.

EWOr can be extended to MDDs, by letting the elementwise operator to be *max* instead of *or*. More symbolic elementwise operations and their applications are to be discovered.

Bibliography

[1] Boost C++ libraries. http://www.boost.org.

[2] Graphviz - graph visualization software. http://www.graphviz.org.

[3] Standard Template Library. http://www.sgi.com/tech/stl.

[4] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, New York, 1995.

[5] 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.

[6] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206, Apr. 1997.

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

[8] S. Barner and I. Rabinovitz. Efficient symbolic model checking of software using partial disjunctive partitioning. In *Proc. CHARME 2003*, LNCS 2860, pages 35–50. Springer-Verlag, 2003.

[9] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[10] B. Berthomieu and M. Menasche. An enumerative approach for analyzing time petri nets. *Proceedings IFIP* 1983, 41–46.

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

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

[13] H.-D. Burhard. On priorities of parallelism: Petri nets under the maximum firing strategy. In *Logics of Programs and Their Applications*, LNCS 148, pages 86–97. Springer-Verlag, 1983.

[14] G. Ciardo. Petri nets with marking-dependent arc multiplicity: properties and analysis. In R. Valette, editor, *Proc. 15th Int. Conf. on Applications and Theory of Petri Nets*, LNCS 815, pages 179–198, Zaragoza, Spain, June 1994. Springer-Verlag.

[15] G. Ciardo. Discrete-time Markovian stochastic Petri nets. In W. J. Stewart, editor, *Computations with Markov Chains*, pages 339–358. Kluwer, Boston, MA, 1995.

[16] G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. Logical and stochastic modeling with SMART. In P. Kemper and W. H. Sanders, editors, *Proc. Modelling Techniques and Tools for Computer Performance Evaluation*, LNCS 2794, pages 78–97, Urbana, IL, USA, Sept. 2003. Springer-Verlag.

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

[18] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In M. Nielsen and D. Simpson, editors, *Proc. 21th Int. Conf. on Applications and Theory of Petri Nets*, LNCS 1825, pages 103–122, Aarhus, Denmark, June 2000. Springer-Verlag.

[19] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In T. Margaria and W. Yi, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 2031, pages 328–342, Genova, Italy, Apr. 2001. Springer-Verlag.

[20] G. Ciardo, G. Lüttgen, and A. J. Yu. Improving static variable orders via invariants. In J. Kleijn and A. Yakovlev, editors, *Proc. 28th International Conference on Application and Theory of Petri nets and Other Models of Concurrency (ICATPN)*, LNCS 4546, pages 83–103, Siedlce, Poland, June 2007. Springer-Verlag.

[21] G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In H. Garavel and J. Hatcliff, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 2619, pages 379–393, Warsaw, Poland, Apr. 2003. Springer-Verlag.

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

[23] G. Ciardo and A. S. Miner. A data structure for the efficient Kronecker solution of GSPNs. In P. Buchholz, editor, *Proc. 8th Int. Workshop on Petri Nets and Performance Models (PNPM'99)*, pages 22–31, Zaragoza, Spain, Sept. 1999. IEEE Comp. Soc. Press.

[24] G. Ciardo, A. S. Miner, M. Wan, and A. J. Yu. Approximating stationary measures of structured continuous-time Markov models using matrix diagrams. *ACM SIGMETRICS Perf. Eval. Rev.*, 35(3):16–18, Dec. 2007.

[25] G. Ciardo and R. Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In M. D. Aagaard and J. W. O'Leary, editors, *Proc. Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, LNCS 2517, pages 256–273, Portland, OR, USA, Nov. 2002. Springer-Verlag.

[26] G. Ciardo and R. Siminiceanu. Structural symbolic CTL model checking of asynchronous systems. In W. Hunt, Jr. and F. Somenzi, editors, *Computer Aided Verification (CAV'03)*, LNCS 2725, pages 40–53, Boulder, CO, USA, July 2003. Springer-Verlag.

[27] G. Ciardo and K. S. Trivedi. A decomposition approach for stochastic reward net models. *Perf. Eval.*, 18(1):37–59, 1993.

[28] G. Ciardo and A. J. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In D. Borrione and W. Paul, editors, *Proc. CHARME*, LNCS 3725, pages 146–161, Saarbrücken, Germany, Oct. 2005. Springer-Verlag.

[29] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new symbolic model checker. *Software Tools for Technology Transfer*, 2(4):410–425, 2000.

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

[31] D. Daly, D. D. Deavours, J. M. Doyle, P. G. Webster, and W. H. Sanders. Möbius: An Extensible Tool for Performance and Dependability Modeling. In B. R. Haverkort, H. C. Bohnenkamp, and C. U. Smith, editors, *Proc. 11th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation, Schaumburg, IL*, LNCS 1786, pages 332–336. Springer-Verlag, Mar. 2000.

[32] M. Davio. Kronecker products and shuffle algebra. *IEEE Trans. Comp.*, C-30:116–125, Feb. 1981.

[33] D. Dolev, M. Klawe, and M. Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *J. of Algorithms*, 3(3):245–260, Sept. 1982.

[34] H. E. Dudeney. *Amusements in Mathematics.* Dover, 1970.

[35] P. Fernandes and B. Plateau. Modeling finite capacity queueing networks with Stochastic Automata Networks. In *Proc. 4th Int. Workshop on Queueing Networks with Finite Capacity*, pages 16/1–16/12, Ilkley, West Yorkshire, UK, July 2000.

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

[37] M. Fujita, P. C. McGeer, , and J. C.-Y. Yang. Multi-terminal binary decision diagrams: an efficient data structure for matrix representation. *Formal Methods in System Design*, 10:149–169, 1997.

[38] S. Graf, B. Steffen, and G. Lüttgen. Compositional minimisation of finite state systems using interface specifications. *Journal of Formal Aspects of Computing*, 8(5):607–616, 1996.

[39] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[40] N. D. Jones, L. H. Landweber, and Y. E. Lien. Complexity of some problems in Petri nets. *Theoretical Computer Science*, 4:277–299, 1977.

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

[42] 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.

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

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

[45] K. Lampka and M. Siegle. MTBDD-based activity-local state graph generation. In *Proc. Sixth Int. Workshop on Performability Modeling of Computer and Communication Systems (PMCCS-6)*, pages 15–18, Monticello, IL, USA, Sept. 2003.

[46] M. Magnin, P. Molinaro, and O. Roux. Decidability, expressivity and state-space computation of stopwatch Petri nets with discrete-time semantics. In *8th International Workshop on Discrete Event Systems (WODES)*, pages 33–38, 2006.

[47] P. M. Merlin. *A study of the recoverability of computing systems*. PhD thesis, Department of Information and Computer Science, University of California, Irvine, 1974.

[48] S.-i. Minato. Zero-suppressed BDDs and their applications. *Software Tools for Technology Transfer*, 3:156–170, 2001.

[49] A. S. Miner. Efficient solution of GSPNs using canonical matrix diagrams. In R. German and B. Haverkort, editors, *Proc. 9th Int. Workshop on Petri Nets and Performance Models (PNPM'01)*, pages 101–110, Aachen, Germany, Sept. 2001. IEEE Comp. Soc. Press.

[50] A. S. Miner. Implicit GSPN reachability set generation using decision diagrams. *Perf. Eval.*, 56(1-4):145–165, Mar. 2004.

[51] A. S. Miner. Saturation for a general class of models. In G. Franceschinis, J.-P. Katoen, and M. Woodside, editors, *Proc. QEST*, pages 282–291, Enschede, The Netherlands, Sept. 2004.

[52] A. S. Miner, G. Ciardo, and S. Donatelli. Using the exact state space of a Markov model to compute approximate stationary measures. In J. Kurose and P. Nain, editors, *Proc. ACM SIGMETRICS*, pages 207–216, Santa Clara, CA, USA, June 2000. ACM Press.

[53] T. Murata. Petri nets: properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–579, Apr. 1989.

[54] E. Pastor, O. Roig, J. Cortadella, and R. M. Badia. Petri net analysis using boolean manipulation. In R. Valette, editor, *Proc. International Conference on Applications and Theory of Petri Nets (ICATPN)*, LNCS 815, pages 416–435, Zaragoza, Spain, June 1994. Springer-Verlag.

[55] B. Plateau, P. Fernandes, and W. Stewart. The PEPS software tool. In P. Kemper and W. H. Sanders, editors, *Proc. Modelling Techniques and Tools for Computer Performance Evaluation*, LNCS 2794, pages 98–115, Urbana, IL, USA, Sept. 2003. Springer-Verlag.

[56] O. Roig, J. Cortadella, and E. Pastor. Verification of asynchronous circuits by BDD-based model checking of Petri nets. In G. De Michelis and M. Diaz, editors, *Proc. 16th Int. Conf. on Applications and Theory of Petri Nets*, LNCS 935, pages 374–391, Turin, Italy, June 1995. Springer-Verlag.

[57] R. Siminiceanu and G. Ciardo. Formal verification of the NASA Runway Safety Monitor. *Software Tools for Technology Transfer*, 9(1):63–76, Feb. 2007.

[58] M. Solé and E. Pastor. Traversal techniques for concurrent systems. In M. D. Aagaard and J. W. O'Leary, editors, *Proc. FMCAD*, LNCS 2517, pages 220–237, Portland, OR, USA, Nov. 2002. Springer-Verlag.

[59] F. Somenzi. CUDD: CU Decision Diagram Package, Release 2.3.1. http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html.

[60] A. Srinivasan, T. Kam, S. Malik, and R. K. Brayton. Algorithms for discrete function manipulation. In *Int. Conference on CAD*, pages 92–95. IEEE Comp. Soc. Press, 1990.

[61] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains.* Princeton University Press, 1994.

[62] M. Wan and G. Ciardo. Extensible decision diagrams for symbolic state-space generation of asynchronous systems. In *Proc. 35th Int. Conf. Current Trends in Theory and Practice of Computer Science (SOFSEM)*, Špindlerův Mlýn, Czech Republic, Jan. 2009. Springer-Verlag. To appear.

[63] Y. Zhang. *Non-blocking Synchronization: Algorithms and Performance Evaluation.* PhD thesis, Chalmers University of Technology, 2003.

[64] W. L. Zuberek. Timed Petri nets definitions, properties, and applications. *Microelectronics and Reliability*, 31:627–644, 1991.