Informatik

Johann Schuster

Towards faster numerical solution of Continuous Time Markov Chains stored by symbolic data structures

der Bundeswehr
Universität München

Fakultät für Informatik
Institut für Technische Informatik

Towards faster numerical solution of Continuous Time Markov Chains stored by symbolic data structures

Johann Schuster

Dissertation

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften (Dr. rer. nat.)

1. Berichterstatter:	Prof. Dr.-Ing. Markus Siegle Universität der Bundeswehr München
2. Berichterstatter:	Prof. Dr.-Ing. Holger Hermanns Universität des Saarlandes
Tag der Prüfung:	16.12.2011

Neubiberg, Januar 2012

Druck gefördert aus Haushaltsmitteln der Universität der Bundeswehr München

Abstract

This work considers different aspects of model-based performance- and dependability analysis. This research area analyses systems (e.g. computer-, telecommunication- or production-systems) in order to quantify their performance and reliability. Such an analysis can be carried out already in the planning phase, without a physically existing system. All aspects treated in this work are based on finite state spaces (i.e. the models only have finitely many states) and a representation of the state graphs by Multi-Terminal Binary Decision Diagrams (MTBDDs).

Currently, there are many tools that transform high-level model specifications (e.g. process algebra or Petri-Net) to low-level models (e.g. Markov chains). Markov chains can be represented by sparse matrices.

For complex models very large state spaces may occur (this phenomenon is called *state space explosion* in the literature) and accordingly very large matrices representing the state graphs. The problem of building the model from the specification and storing the state graph can be regarded as solved: There are heuristics for compactly storing the state graph by MTBDD or Kronecker data structure and there are efficient algorithms for the model generation and functional analysis. For the quantitative analysis there are still problems due to the size of the underlying state space.

This work provides some methods to alleviate the problems in case of MTBDD-based storage of the state graph. It is threefold:

- For the generation of smaller state graphs in the model generation phase (which usually are easier to solve) a symbolic elimination algorithm is developed.
- For the calculation of steady-state probabilities of Markov chains a multilevel algorithm is developed which allows for faster solutions.
- For calculating the most probable paths in a state graph, the mean time to the first failure of a system and related measures, a path-based solver is developed.

Zusammenfassung

Diese Arbeit beschäftigt sich mit verschiedenen Aspekten der modellbasierten Leistungs- und Zuverlässigkeitsbewertung. Hierbei werden Systeme (z.B. Computer-, Telekommunikations- oder Produktionssysteme) analysiert, um deren Leistungsvermögen und Zuverlässigkeit zu quantifizieren. Eine solche Analyse ist bereits in der Planungsphase möglich, noch bevor das System physikalisch existiert.

Alle in dieser Arbeit behandelten Aspekte basieren auf einem endlichen Zustandsraum (d.h. die Modelle haben nur endlich viele Zustände) und einer Repräsentation des Zustandsgraphen durch Multi-Terminale Binäre Entscheidungsdiagramme (MTBDDs).

Derzeit gibt es viele Werkzeuge, die eine hochsprachliche Modell-Spezifikation (z.B. Prozess-Algebra oder Petri-Netz) in ein niedersprachliches Modell (z.B. eine Markov-Kette) transformieren können. Diese Arbeit beschäftigt sich mit Markov-Ketten, die aus Prozess-Algebren generiert und als dünn besetzte Matrizen dargestellt werden.

Für komplexe Modelle ergeben sich oft sehr große Zustandsräume (dieses Phänomen wird in der Literatur als *Zustandsraumexplosion* bezeichnet) und entsprechend große Matrizen für die Zustandsgraphen. Das Problem der Modellgenerierung und Speicherung der Zustandsgraphen kann auch für sehr große Zustandsräume als gelöst betrachtet werden: Es existieren Heuristiken zur kompakten Speicherung mit Hilfe von MTBDDs oder Kronecker Matrizen sowie effiziente Algorithmen zur Modell-Generierung und funktionalen Analyse. Allerdings ergeben sich bei der quantitativen Analyse oft Probleme durch die Größe der generierten Zustandsgraphen.

Die vorliegende Arbeit stellt Methoden zur Verfügung, um diese Probleme im Falle von MTBDD-basierter Speicherung zu verringern. Sie besteht aus drei Teilen:

- Zur Erzeugung kleinerer Zustandsgraphen in der Modellgenerierungsphase (die in der Regel leichter – oder überhaupt erst – analysiert werden können) wird ein Eliminations-Algorithmus entwickelt.
- Zur Ermittlung der Gleichgewichtsverteilung von Markov-Ketten wird ein Multi-Level-Lösungsansatz entwickelt, der schnellere Lösungen ermöglicht.
- Um die wahrscheinlichsten Pfade in einem Zustandsgraphen, die mittlere Zeit bis zum ersten Systemausfall und weitere verwandte Maße zu ermitteln, wird eine pfadbasierte Lösungskomponente entwickelt.

Contents

1. Introduction

1.1. Model-based performance and dependability analysis

Model-based performance and dependability analysis is a research area that analyses systems (e.g. computer-, telecommunication- or production-systems) in order to quantify their performance and reliability.

We introduce the scope of the methods developed in this work by means of a small example, using a modification of the model presented in [11]. The system is a satellite on a certain mission that has to carry out two experiments. The progress of the mission is shown in Fig. 1.1. Once the satellite starts working, its mission is to gather experimental data from two experiments successively. Each experiment has to be carried out for some random period of time, until all measurements are taken. After the second experiment has finished, the mission is successfully accomplished. However, the experimental equipment in the satellite is assumed to be unreliable, so there are different kinds of uncritical and critical failures possible. Whenever a critical failure occurs, the satellite turns into the error state and the mission collapses.

In this work we follow an analytical model-based approach. That means in the satellite example that a model of the satellite is developed from the specification. With the help of software tools, some characteristics of the satellite model can be analytically obtained *before* the system physically exists. This approach is much more cost-effective than testing the existing system: Changing the model is much easier and faster than changing existing hardware. A further advantage is that analytical methods are capable of quantifying even very rare events that are usually harder to detect by simulative model-based approaches. The following list shows some questions on the example model that can be raised:

1. What is the probability that both experiments can be completed successfully
   a) without any uncritical failure?
   b) in the presence of uncritical failures?
2. Which are the most probable critical failures that lead to an error of the satellite? (this kind of analysis can be used to identify weak spots)
3. What is the probability that the mission can be successfully accomplished within at most t time units?

If the answers to these questions have to be calculated from the model specification, additional information about the unreliability of the satellite's components has to be annotated.

There are many different ways of modelling such a system. The modelling world used in this work supports finitely many states and two types of actions, *timed* and *immediate*. A timed action is driven by an exponentially distributed random variable and is specified by a rate. Therefore, the timed behaviour of a model containing timed actions only is that of a Continuous Time Markov Chain (CTMC). Timeless decisions are driven by immediate actions (e.g. if the failure of one component induces a failure of another component). They are specified by weights that are normalised to probabilities once the closed model has been constructed. The restriction to exponentially distributed delays is in theory no limitation as phase-type distributions (i.e. distributions where every phase is exponentially distributed) are dense in the field of positive-valued distributions. In practice however, the phase-type approach suffers

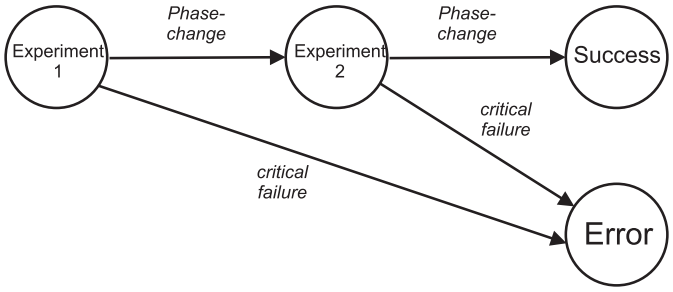


Figure 1.1.: Satellite on a mission

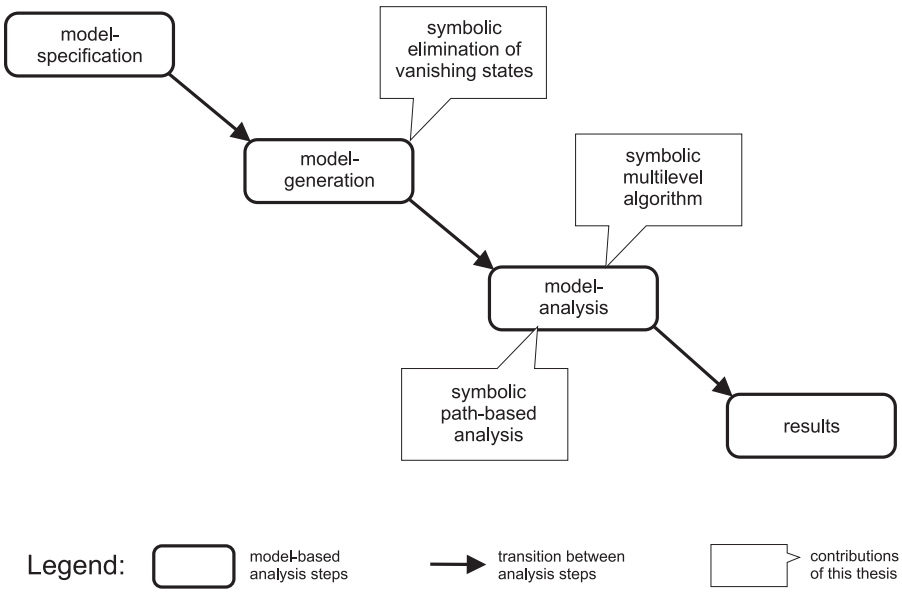


Figure 1.2.: Workflow of model-based analysis

from the *state space explosion* problem, which means that models get too large to analyse. The mathematically elegant attribute of the restriction to the above case is that the given questions can be transformed into systems of linear differential equations (or even systems of linear equations, depending on the type of analysis) or properties of graphs.

The typical model-based performance- and dependability evaluation process is depicted in Fig. 1.2. The different phases will be explained in the sequel. In the model specification phase the physical system is abstracted and transformed into some high-level modelling language (e.g. stochastic Petri Net, stochastic process algebra). The model should be complex enough to answer all the questions above, but also sufficiently small such that it can still be analysed. From the high-level specification, a lower-level model (e.g. a CTMC) is generated automatically. This tool-based transformation approach is by far less error-prone than specifying the low-level model by hand. Moreover many high-level formalisms allow for compositional modelling. That means that larger models can be built from smaller models. When the low-level model has been obtained, it can be analysed. Usual types of analysis are computations of performance measures through the derivation of the transient or steady-state probability vector. Also graph-based algorithms can be used. Finally, the analysis results are displayed to the user.

In this process, several problems may occur. One issue that is practically solved is the compact storage of the state graph of the model. Current tools deal with this problem by using

special data structures, such as Kronecker matrices or Multi-Terminal Binary Decision Diagrams (MTBDDs) and efficient algorithms based on this data structure. Another problem is that there is usually a tradeoff between compact storage of the state graph and performance of the numerical analysis. Normally extracting subsets of the state graph from the compact data structure is time-consuming and therefore slows down numerical calculations. From the compositional modelling paradigm, another issue arises: Unless specially suited algorithms are used (e.g. for Nearly Completely Decomposable systems), the structure of the model is hardly exploited by numerical analysis.

This work tries to address the problems above at different stages of the process as indicated in Fig. 1.2. All methods are based on the MTBDD data structure. The three different main topics of this work are explained in the next section.

1.2. Contributions

The contribution of this work is in the context of symbolical methods for model-based performance and dependability evaluation. It is threefold: Efficient elimination of vanishing states during the model generation, faster steady-state analysis with a symbolic multilevel algorithm and symbolic path-based analysis of a model.

1.2.1. Elimination of vanishing states

Often it is convenient to specify timeless behaviour in addition to timed behaviour. Before standard numerical algorithms for CTMCs can be used, all timeless behaviour has to be eliminated. A straight-forward method to specify such systems by means of stochastic process algebra is shown and a symbolic elimination algorithm is developed. The algorithm consists of two steps: The first phase is the fully-symbolic elimination approach. This phase provides a fast means for elimination. As typical for symbolic algorithms, a set-based scheme is presented for this phase thus running considerably faster than a state-by-state approach. As the first phase does not necessarily eliminate all the vanishing states, a second phase is used to eliminate the remaining vanishing states in a state-by-state manner. Thus the second phase is used as a post-processor for the fully-symbolic phase.

1.2.2. Symbolic multilevel algorithm

The major part of this work is devoted to the steady-state solution of Markov chains. To speed up iterative solution algorithms, multilevel methods have been proposed to get a faster reduction of the error terms. This work presents a fast adaptation of the multilevel principle to Markov chains stored by symbolic data structures. The current version of the algorithm stores the state graph as a (modified) MTBDD and the iteration vectors as normal double vectors. A scalable parallel version of the algorithm designed for multi-core architectures is developed.

1.2.3. Symbolic path-based analysis

For calculating path probabilities and approximative analysis of a system's unreliability and related measures, a path-based analysis component has been realised. The k-most probable paths can be calculated exactly (up to the roundoff error set in the MTBDD package). A major application of path-based analysis is the counter-example generation in model checking. It can also be used to debug models (e.g. if some paths are theoretically known). By the k-most probable path algorithm additional measures like Mean Time To First Failure (MTTFF) can be calculated. For a certain class of models, these approximations fit quite well to the accurate result.

1.3. Acknowledgements

I would like to thank everyone who helped me to make progress in this work. I would like to mention only a few of them explicitly: Peter Buchholz who shared his experience on multilevel methods in the Kronecker-context, Ivana Pultarova for some interesting discussions on the convergence behaviour of multilevel methods and my colleagues Alexander Gouberman and Martin Riedl for many interesting discussions on the subject. Further I would like to thank Holger Hermanns who gave me the chance to discuss my results with his working group and came up with very interesting questions that helped a lot to improve this work. Special and cordial thanks to my PhD supervisor Markus Siegle. His working group was a fertile soil where ideas could grow. Expecially his unresting demand for both fast and memory-efficient numerical methods lead to the current version of the hybrid multilevel algorithm. His door was literally always open and I appreciate very much his suggestions and the discussions we had on the whiteboard or on the writing desk.

2. Foundations

The two main topics of this chapter are the CASPA modelling language and the Multi-Terminal Binary Decision Diagram (MTBDD) data structure. We start with some graph definitions, give an introduction to the CASPA syntax and semantics, reason about the compositionality of the CASPA language, introduce the MTBDD data structure and operations thereon. Finally we will introduce some standard numerical algorithms for solving linear systems of equations. As we consider all models to be finite, we assume every set in the sequel to be finite (unless stated otherwise).

2.1. Definitions on graphs

The transition systems defined in this section have some similarities with graphs. The MTBDD data structure is defined using graph terminology. Before talking about graphs, we make a definition on the direct product of sets.

Definition 1. *Given a n-fold direct product of sets $V_1 \times \ldots \times V_n$, we define the* projection *function $proj_i$, $1 \leq i \leq n$ as:*

$$\begin{array}{rccc} proj_i: & V_1 \times \ldots \times V_n & \rightarrow & V_i \\ & (v_1, \ldots, v_n) & \mapsto & v_i \end{array}$$

Now we give the basic definition of a directed graph and some related terminology.

Definition 2. *A* directed graph *G is a pair (V, E) where V is a set of vertices and $E \subseteq V \times V$ is the set of* directed edges*. A directed edge is represented as an ordered pair (h, t) of vertices. For an edge $(h, t) \in E$ we call h the* head *and t the* tail*. In this case we say that t is a* direct successor *of h. We define the head (tail) function of an edge by $H := proj_1$ ($T := proj_2$). A tuple $p = (v_1, \ldots, v_n) \subset V^n$, $n \in \mathbb{N}$ with the property that $\forall i \in \{1, \ldots, n-1\} : (v_i, v_{i+1}) \in E$ is called a* finite path of length $n-1$ *in the graph. The set of all possible paths in the graph G is denoted by $Paths(G)$. We define the* head *(* tail*) of the path as $H(p) := proj_1(p) = v_1$ ($T(p) := proj_n(p) = v_n$). The set of* successors *of a vertex v is defined as $succ(v, E) := \{w \mid \exists (v, w) \in E\}$. If $|succ(v, E)| = 1$ we also write $v \rightarrow E$ for $succ(v, E)$. A graph is called* finite *if V and E are finite sets.*

In the sequel all graphs will be finite. For the definition of a BDD we need the following definition:

Definition 3. *A* directed acyclic graph *G is a directed graph $G = (V, E)$ with the property that for every path $p = (v_1, \ldots, v_n) \in Paths(G)$ it holds that $\forall i, j \in \{1, \ldots, n\}, i \neq j : v_i \neq v_j$.*

Definition 4. *A* rooted directed acyclic graph *$G = (V, E)$ is a* directed acyclic graph *with a vertex* root $\in V$*, such that for every node $v \in V$, $v \neq root$, there exists a path $p \in Paths(G)$ with $H(p) = root$, $T(p) = v$ (cf. [34]).*

Remark 1. *In a rooted directed acyclic graph the root vertex is uniquely determined (otherwise, if there was a vertex root$'$, then according to the definition a cycle from root via root$'$ to root would exist, which would be a contradiction).*

2.2. Transition systems

A model in our context can be seen as a set of states S with an initial state $s_0 \in S$ and labelled transitions between them. To make it more explicit, we give the following definitions:

Definition 5. *A* Labelled Transition System *(LTS) is a tuple $(S, L, \rightarrow, s_0)$ where S is a finite set of states, L is a finite set of labels, $s_0 \in S$ is the* initial state *and $\rightarrow$ is a relation $\rightarrow \subseteq S \times L \times S$. An element $(s_1, a, s_2) \in \rightarrow$ is called a* transition *from s_1 to s_2 with the action label a. Alternatively we write $s_1 \overset{a}{\rightarrow} s_2$.*

Remark 2. *An LTS can also be seen as a directed graph (S, E) together with a starting vertex $s_0 \in V$ and additional label annotations given by a labelling function $l : E \rightarrow L$. In the sequel we will use graph-theoretic definitions (like succ etc.) also for LTS by means of the corresponding graph.*

Definition 6. *A* finite path *σ in an LTS $(S, L, \rightarrow, s_0)$ is a sequence*

$$s_1 \overset{a_1}{\rightarrow} s_2 \overset{a_2}{\rightarrow} s_3 \ldots \overset{a_{n-1}}{\rightarrow} s_n$$

where $n \geq 2$.

We will assume similar definitions for paths in the other types of labelled transition systems that are defined in the sequel.

Definition 7. *A* Stochastic Labelled Transition System *(SLTS) is tuple $(S, L, \rightarrow, s_0)$ where S, L and s_0 are the same as in Def. 5. The relation $\rightarrow$ is given as $\rightarrow \subseteq S \times L \times \mathbb{R}^{>0} \times S$. An element $(s_1, a, \lambda, s_2) \in \rightarrow$ is called a transition from s_1 to s_2 with the action label l and the rate λ (describing a negative exponentially distributed random variable). The transitions are also called* Markovian *transitions. Alternatively we write $s_1 \overset{a,\lambda}{\rightarrow} s_2$.*

Remark 3. *Similar to the alternative interpretation of an LTS, an SLTS can be seen as a graph. In addition to the labelling function there is a* rate function *$r : E \rightarrow \mathbb{R}$.*

Remark 4. *In the sequel it is assumed that "parallel" transitions carry different labels, i.e. if $(x, a, \lambda, y) \in \rightarrow$ and $(x, a', \lambda', y) \in \rightarrow$, then $a \neq a'$ must hold. This is no restriction as the minimum of some exponentially distributed random variables is, again, exponentially distributed (with the sum of the rates).*

Definition 8. *For the calculation of the transition probabilities and the average time for the traversal of a path in an SLTS $(S, L, \rightarrow, s_0)$, the notation of the* exit rate *of a state $x \in S$ is important. It is defined as*

$$\lambda(x) := \sum_{(x,a,\lambda,y) \in \rightarrow} \lambda,$$

that means the cumulated outgoing rates of state x.

Definition 9. *A Continuous Time Markov Chain (CTMC) is a SLTS with only one action (or equivalently: where the actions are ignored).*

Definition 10. *A* Weighted Extended Stochastic Labelled Transition System *(WSLTS) is tuple $(S, L_M, L_I, \rightarrow, \dashrightarrow, s_0)$ where S, s_0 are the same as in Def. 7. Two sets of labels are present: L_M and L_I, $L_M \cap L_I = \emptyset$. A special label $\tau \in L_I$ will be used for the internal tau transition. The relations are given as $\rightarrow \subseteq S \times L_M \times \mathbb{R}^{>0} \times S$ and $\dashrightarrow \subseteq S \times L_I \times \mathbb{R}^{>0} \times S$. The relation $\rightarrow$ is interpreted as in the SLTS case. An element $(s_1, a, w, s_2) \in \dashrightarrow$ is called a transition from s_1 to s_2 with the action label l and the* weight *w. Alternatively we write $s_1 \overset{a,w}{\dashrightarrow} s_2$. Transitions in $\rightarrow$ ($\dashrightarrow$) are called timed (immediate) transitions. A state with at least one outgoing immediate transition is called* vanishing*, otherwise it is called* tangible*.*

Remark 5. *Usually immediate transitions in an* Extended Stochastic Labelled Transition System *(ESLTS) are given without any weight. In our setup we use weights to generate transition probabilities. Like for exponentially distributed transitions it is assumed that no "parallel" immediate transitions with the same action exist. They are grouped together by summing up the parallel weights.*

The next definition covers the special case where the weights in a WSLTS already form a probability distribution for every vanishing state.

Definition 11. *A* Probabilistic Extended Stochastic Labelled Transition System *(PSLTS) is tuple* $(S, L_M, L_I, \rightarrow, \dashrightarrow, s_0)$ *where* S, L_M, L_I, s_0 *and* $\rightarrow$ *are the same as in Def. 10. The relation* $\dashrightarrow$ *is given as* $\dashrightarrow \subseteq S \times L \times (0,1] \times S$ *with the property that for every vanishing* $s \in S$ $\sum_{(s,a,w,s') \in \dashrightarrow} w = 1$, *i.e. for every vanishing state there has to be a probability distribution between the possible immediate transitions.*

2.3. Syntax of the CASPA language

The current version of the CASPA input language supports both immediate and Markovian actions. It is used to generate WSLTSs. The current version of the CASPA language is as an extension of the Markovian CASPA language given in [36]. The syntax given here in EBNF (Extended Backus-Naur-Form [24]) is based on the CASPA handbook [37].

We give some basic syntax definitions in EBNF that cover the current language features. The following EBNF elements are used:

`|`	definition-separator-symbol
`=`	defining-symbol
`"`	second-quote-symbol
`[`	start-option-symbol
`]`	end-option-symbol
`{`	start-repeat-symbol
`}`	end-repeat-symbol

The option symbols specify terms that can be omitted, the repeat symbols specify terms that can be omitted or repeated (finitely often). In the following it is clear when a new definition starts, so the terminator-symbol (`;`) is omitted. Every terminal symbol is put in double quotes (" "). It is assumed that the following terminal symbols are already defined:

VARIABLE variable name (identifier), may contain digits, characters and underscore. Digits only are not allowed.

FLOAT floating point number where the decimal point is optional.

INT integer value.

ACTION, PROCESS, PARAMETER all denote an identifier, they are thus synonyms for `VARIABLE`, used for better readability.

Some general purpose definitions are

$$
\begin{aligned}
\text{Expression} &= \text{Expression “+” Term} \\
&\mid \text{Expression “-” Term} \\
&\mid \text{Term} \\
\text{Term} &= \text{Term “*” Factor} \\
&\mid \text{Term “/” Factor} \\
&\mid \text{Factor} \\
\text{Factor} &= \text{“(” Expression “)”} \\
&\mid \text{VARIABLE} \mid \text{INT} \mid \text{FLOAT} \\
\text{Labels} &= \text{ACTION } \{\text{“,” ACTION }\}
\end{aligned}
$$

2.3.1. Definition of a model

A model consists of several definitions. Either integer-, rate-, weight-, process- or measure definitions.

$$
\begin{aligned}
\text{Model} &= \text{Definition } \{ \text{ Definition } \} \\
\text{Definition} &= \text{Integerdefinition} \\
&\mid \text{RateDefinition} \\
&\mid \text{WeightDefinition} \\
&\mid \text{ProcessDefinition} \\
&\mid \text{MeasureDefinition}
\end{aligned}
$$

2.3.2. Constant definitions

For a better readability of the model specification, constants for integer values, rates and weights can be defined.

$$
\begin{aligned}
\text{Integerdefinition} &= \text{“int” VARIABLE “=” Expression “;”} \\
\text{RateDefinition} &= \text{“rate” VARIABLE “=” FLOAT “;”} \\
\text{WeightDefinition} &= \text{“weight” VARIABLE “=” FLOAT “;”}
\end{aligned}
$$

2.3.3. Measure definitions

Several performance measures can be defined: A `StateMeasure` specifies a set of states. The probability of being in this set can be calculated by the `statemeasure` keyword. The `targetstate` keyword uses the subset for path-based analysis (e.g. for approximating the probability of reaching this set). For parameterised processes, the mean value of a certain process parameter can be calculated by `meanvalue`. The throughput of a certain Markovian action can be calculated by `throughputmeasure`.

MeasureDefinition = "statemeasure" VARIABLE StateMeasure
| "targetstate" VARIABLE StateMeasure
| "meanvalue" VARIABLE PROCESS
"(" PARAMETER ")" ["{" INT "}"]
| "throughputmeasure" VARIABLE ACTION

StateMeasure = StateMeasure "&" StateMeasure
| StateMeasure "|" StateMeasure
| "!" StateMeasure
| "(" StateMeasure ")"
| PROCESS ["("Conditions")"] ["{" INT "}"]

The optional parameter `INT` is used to access several instances of the same process.

2.3.4. Process definition

Processes can be specified either with or without parameters. A parameterised process can only have successors of type SequentialProcess. Conditions, that determine for which process parameter a certain SequentialProcess successor is active, are also called guards.

ProcessDefinition = PROCESS ":=" GeneralProcess
| PROCESS ["(" Parameters ")"] ":="
GuardedSequentialProcess, { GuardedSequentialProcess }

Parameters = Parameter { "," Parameter }

Parameter = PARAMETER "[" (INT | VARIABLE) "]"

GuardedSequentialProcess = "[" Conditions "]" "->" SequentialProcess

Conditions = Condition {"," Condition }

Condition = Expression "=" Expression
| Expression "<" Expression
| Expression ">" Expression
| Expression ">=" Expression
| Expression "<=" Expression
| Expression "!=" Expression
| "*"

A `GeneralProcess` is a process that may contain parallel composition and/or hide operators.

GeneralProcess = GeneralProcess "|[" Labels "]|" GeneralProcess
| "hide" Labels "in" GeneralProcess
| "(" GeneralProcess ")"
| SequentialProcess

A `SequentialProcess` may only contain choice or prefix operators, i.e. only sequential behaviour.

SequentialProcess = "stop"
| PROCESS ["(" Arguments ")"]
| "(" ACTION "," Expression ");" SequentialProcess
| "(*" ACTION "," Expression "*);" SequentialProcess
| SequentialProcess "+" SequentialProcess
| "(" SequentialProcess ")"

Arguments = Expression { "," Expression }

2.4. Structural operational semantics

In this section some basic structural operational semantics (SOS) rules are defined. There is a special action τ, specified by action name `tau`, which is called the *internal immediate action*. This action cannot be used for synchronisation or for hiding.

2.4.1. Prefix operators

The easiest rules are the prefix rules for Markovian and immediate actions:

$$\frac{}{(a,\lambda);P \xrightarrow{(a,\lambda)} P}\ a \in L_M$$

$$\frac{}{(*a,\omega*);P \overset{(a,\omega)}{\dashrightarrow} P}\ a \in L_I$$

2.4.2. Choice operator

Four different cases for the choice operator exist:

	Markovian	immediate
left	$\dfrac{P \xrightarrow{(a,\lambda)} P'}{P+Q \xrightarrow{(a,\lambda)} P'}$	$\dfrac{P \overset{(a,\omega)}{\dashrightarrow} P'}{P+Q \overset{(a,\omega)}{\dashrightarrow} P'}$
right	$\dfrac{Q \xrightarrow{(a,\lambda)} Q'}{P+Q \xrightarrow{(a,\lambda)} Q'}$	$\dfrac{Q \overset{(a,\omega)}{\dashrightarrow} Q'}{P+Q \overset{(a,\omega)}{\dashrightarrow} Q'}$

2.4.3. Unsynchronised parallel composition

Either the left or the right process may move. Definitions are similar for Markovian and immediate transitions.

	Markovian	immediate
left	$\dfrac{P \xrightarrow{(a,\lambda)} P'}{P\|[]\|Q \xrightarrow{(a,\lambda)} P'\|[]\|Q}$	$\dfrac{P \overset{(a,\omega)}{\dashrightarrow} P'}{P\|[]\|Q \overset{(a,\omega)}{\dashrightarrow} P'\|[]\|Q}$
right	$\dfrac{Q \xrightarrow{(a,\lambda)} Q'}{P\|[]\|Q \xrightarrow{(a,\lambda)} P\|[]\|Q'}$	$\dfrac{Q \overset{(a,\omega)}{\dashrightarrow} Q'}{P\|[]\|Q \overset{(a,\omega)}{\dashrightarrow} P\|[]\|Q'}$

2.4.4. Synchronised parallel composition

Now we assume $S \subseteq (L_M \cup L_I)$ and $\tau \notin S$. Any action that is not in the set of synchronised actions can be performed as in the unsynchronised case:

	Markovian	immediate
left	$\dfrac{P \xrightarrow{(a,\lambda)} P'}{P\|[S]\|Q \xrightarrow{(a,\lambda)} P'\|[S]\|Q}\ a \notin S$	$\dfrac{P \overset{(a,\omega)}{\dashrightarrow} P'}{P\|[S]\|Q \overset{(a,\omega)}{\dashrightarrow} P'\|[S]\|Q}\ a \notin S$
right	$\dfrac{Q \xrightarrow{(a,\lambda)} Q'}{P\|[S]\|Q \xrightarrow{(a,\lambda)} P\|[S]\|Q'}\ a \notin S$	$\dfrac{Q \overset{(a,\omega)}{\dashrightarrow} Q'}{P\|[S]\|Q \overset{(a,\omega)}{\dashrightarrow} P\|[S]\|Q'}\ a \notin S$

A synchronised action must be performed by both parallel processes:

$$\text{Markovian} \qquad \frac{P \xrightarrow{(a,\lambda)} P' \qquad Q \xrightarrow{(a,\mu)} Q'}{P|[S]|Q \xrightarrow{(a,\lambda\cdot\mu)} P'|[S]|Q'}\, a \in S$$

$$\text{immediate} \qquad \frac{P \overset{(a,\omega)}{\dashrightarrow} P' \qquad Q \overset{(a,\sigma)}{\dashrightarrow} Q'}{P|[S]|Q \overset{(a,\omega\cdot\sigma)}{\dashrightarrow} P'|[S]|Q'}\, a \in S$$

Note that the resulting rate/weight is the product of the synchronising rates/weights. There are many other policies known for the synchronised rate [21]. As an immediate consequence of the semantics one sees, that synchronisation can only occur between actions of the same type (i.e. both immediate or both Markovian). It is *not* required that $S \subseteq L_I$. From a practical point of view it is often sufficient to use immediate actions only for a synchronisation in the style of Interactive Markov Chains (IMCs) [29]. In this case, the synchronising actions have all weights equal to 1 and they do not concur with any other immediate actions.

2.4.5. Hiding

For the hiding operator it is assumed that $S \subseteq L_I$ and $\tau \notin S$. Hidden immediate actions are mapped to the internal immediate action τ,

$$\frac{P \overset{(a,\omega)}{\dashrightarrow} P'}{\texttt{hide}\ S\ \texttt{in}\ P \overset{(\tau,\omega)}{\dashrightarrow} \texttt{hide}\ S\ \texttt{in}\ P'}\, a \in S$$

the immediate actions that are not hidden remain the same.

$$\frac{P \overset{(a,\omega)}{\dashrightarrow} P'}{\texttt{hide}\ S\ \texttt{in}\ P \overset{(a,\omega)}{\dashrightarrow} \texttt{hide}\ S\ \texttt{in}\ P'}\, a \notin S$$

All Markovian actions remain unchanged:

$$\frac{P \xrightarrow{(a,\lambda)} P'}{\texttt{hide}\ S\ \texttt{in}\ P \xrightarrow{(a,\lambda)} \texttt{hide}\ S\ \texttt{in}\ P'}$$

2.4.6. Parameterised processes

Parameterised processes are only abbreviations for sequential processes and therefore no separate semantics is given. A parameterised process can be converted into equivalent process definitions (e.g. by a preprocessor) as follows: Assume $n \in \mathbb{N}$ and $n_i \in \mathbb{N}$, $i \in \{1, 2, \ldots, n\}$ are arbitrary but fixed numbers. Then in process $P(var_1[n_1], var_2[n_2], \ldots, var_n[n_n])$ the i-th parameter can range from 0 to n_i, so this process can be represented by $\prod_{i=1}^{n}(n_i + 1)$ separate process definitions P_i, $i \in I$ in a canonical way: $P_0 \simeq P(0, 0, \ldots, 0)$, $P_1 \simeq P(0, 0, \ldots, 0, 1)$, …, $P_{(\prod_{i=1}^{n}(n_i+1))-1} \simeq P(n_1, n_2, \ldots, n_n)$. Transitions for each P_i are those where for the corresponding parameter set $(i_1, i_2, \ldots, i_n)$, $i_i \in \{0, 1, \ldots, n_i\}$ the condition of the successor specification is fulfilled.

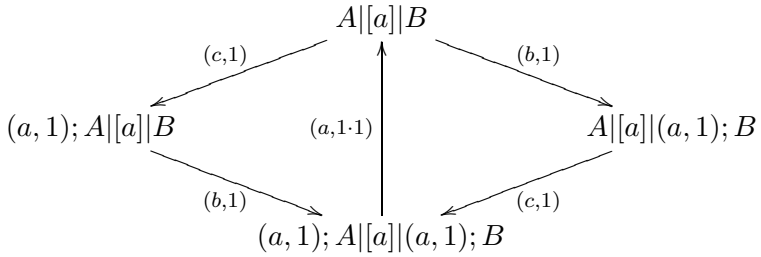
2.4.7. A parallel composition example

The syntax definition in Sec. 2.3 shows that the basic building blocks of the language are sequential processes, i.e. processes without parallelism. These basic building blocks can be combined successively by the parallel composition operator to larger models.

Suppose there are given two sequential processes:

```
A:=(c,1);(a,1);A
B:=(b,1);(a,1);B
```

Then, by SOS rules, the process $A|[a]|B$ has the following transitions:



This example will be used later for encoding parallel compositions by Multi-Terminal Binary Decision Diagrams (MTBDDs).

2.5. Operations on the closed model

Once the *closed model* is obtained (i.e. no further parallel compositions can take place), we use the *maximal progress* assumption: As timed actions are driven by exponentially distributed random variables (with expectation > 0) while immediate actions happen without any delay, one assumes that whenever there is a race condition between timed and immediate actions, the timed action never can be taken and therefore it can be omitted.

On the closed model we transform the specified WSLTS to a PSLTS by normalising for every state the sum of its outgoing weights to 1.

Once the PSLTS is obtained, some of the vanishing states with outgoing tau transitions may be eliminated. One has to take care that such a vanishing state has no other outgoing immediate transitions. For this purpose the following definition can be made [57]:

Definition 12. *Given a PSLTS $(S, L_M, L_I, \rightarrow, \dashrightarrow, s_0)$. A compositionally vanishing state is a vanishing state $s \in S$ with the following properties:*

1. *At least one outgoing immediate tau transition exists:*
   $\exists c \in (0,1], \exists s' \in S : s \overset{\tau,c}{\dashrightarrow} s'$

2. *There are no immediate non-tau transitions emanating from s:*
   $\forall c \in (0,1], \forall s' \in S, \forall a \in L \setminus \{\tau\} : s \overset{a,c}{\not\dashrightarrow} s'$

Compositionally vanishing states can be eliminated without side-effects to other immediate transitions, i.e. without losing information.

The elimination can be done according to the following rules (similar to the elimination of vanishing states in a Generalised Stochastic Petri Net): Firstly, for every vanishing state P' its immediate self-loops are eliminated by normalising the probabilities of the other outgoing immediate transitions to a sum of 1 and removing the loop from the set of immediate transitions. An exception occurs, if there are no other outgoing immediate transitions than the self-loop. This is a timeless trap situation (cf. Sec. 3.2.1). Secondly, there are two redirection rules for a given *vanishing state* P':

$$\frac{P \overset{(a,q)}{\dashrightarrow} P' \qquad P' \overset{(\tau,p)}{\dashrightarrow} Q}{P \overset{(a,q\cdot p)}{\dashrightarrow} Q} \not\exists P' \dashrightarrow P'$$

$$\frac{P \overset{(a,\lambda)}{\longrightarrow} P' \qquad P' \overset{(\tau,p)}{\dashrightarrow} Q}{P \overset{(a,\lambda\cdot p)}{\longrightarrow} Q} \not\exists P' \dashrightarrow P'$$

After any incoming transition into P' has been redirected to all possible outgoing transitions, the incoming and the outgoing transitions attached to P' are removed from the set of transitions and P' is removed from the set of vanishing states.

2.6. Classification of CASPA models

2.6.1. Problem statement

The following examples show some situations where the CASPA language could lead to undesired model behaviour, i.e. behaviour that the modeller didn't intend. Although the CASPA language does not support priorities and passive actions (nondeterminism) as supported by EMPA [8], similar issues as in [9] may arise. Therefore we admit that the CASPA language is not compositional in a strict sense.

We will define a criterion for strictly compositional (cf. [35]) CASPA models that can be reduced to CTMCs. This criterion is rather abstract and cannot be calculated easily, as a certain weak bisimulation equivalence has to be found. We will show a stronger sufficient criterion that can be checked more easily. Unfortunately, both characterisations cannot be checked on the syntax level, but the state space has to be considered.

We proceed as follows. Firstly, we show some examples where pitfalls can arise. Secondly, we give an alternative semantics for the CASPA language that uses nondeterminism to overcome the pitfalls. Finally we reconsider the examples in the light of the alternative semantics.

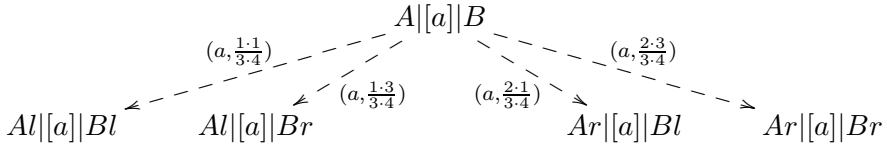
2.6.1.1. Synchronised parallel composition

Take the two processes (disregarding for the moment that CASPA does not permit vanishing initial states):

```
A:=(*a,1*);Al+(*a,2*);Ar
B:=(*a,1*);Bl+(*a,3*);Br
```

The normalised processes are

In the parallel process `A|[a]|B` there are four possible transitions (after normalising weights to probabilities), i.e.



One observes that in this case the resulting probabilities are the products of the probabilities of the local transitions. In other words, the following diagram commutes:

$$\begin{array}{ccc} PR \times PR & \xrightarrow{||[a]||} & PR \\ {\scriptstyle norm \times norm}\downarrow & & \downarrow{\scriptstyle norm} \\ PR \times PR & \xrightarrow{||[a]||} & PR \end{array}$$

Here, PR is the set CASPA processes, $||[a]||$ is the parallel composition operation and *norm* the normalisation operation (as weights are dynamically normalised depending on the state space, this operation may in general lead to more complicated CASPA processes — symmetries can be broken). From the diagram it follows that, if `A` is left unchanged and `B` is substituted by

```
B':=(*a,2*);Bl+(*a,6*);Br
```

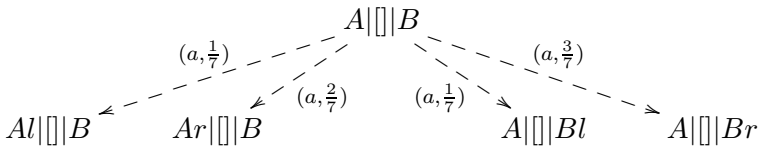
the result is the same (it is clear that the image after $norm \times norm$ must be the same, so the parallel composition is the same. By the commutativity equality holds also for normalisation after parallel composition. The commutativity of the diagram above is desirable but unfortunately does not hold in general, as the following examples show.

2.6.1.2. Problem: unsynchronised parallel composition

Look again at the processes (as before):

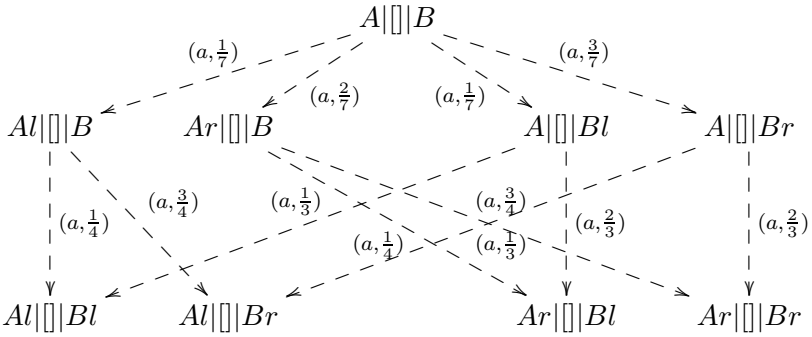
```
A:=(*a,1*);Al+(*a,2*);Ar
B:=(*a,1*);Bl+(*a,3*);Br
```

In the parallel process `A|[]|B` there are four possible transitions (with the weights normalised to probabilities):

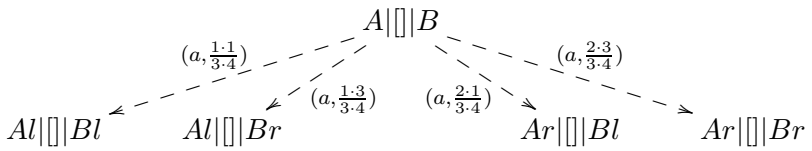


The problem is that it is not evident that the possibly non-deterministic decision which component moves first can be resolved by a probabilistic decision [21]. By the specification of weights, in CASPA one probabilistic trace for this situation can be specified.

The situation improves if it is known that the intermediate states that resolve the non-determinism are *not recognised* by the environment (i.e. no other transition in the closed model disturbs the current non-determinism). If we are only interested in what happens after *both* `A` and `B` have made some step, the situation is as follows:



So, ignoring the intermediate steps, the resulting probability distribution is always the same (scale-free).



The reason behind this phenomenon is that for parameterised weights a_l, a_r, b_l, b_r for the left and right branches we get $\frac{a_l}{a_l+a_r+b_l+b_r}\frac{b_l}{b_l+b_r} + \frac{b_l}{a_l+a_r+b_l+b_r}\frac{a_l}{a_l+a_r}$ for the branch to `Al|[]|Bl`. This

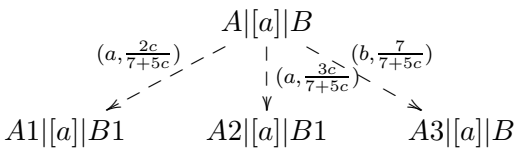
can be simplified to $\frac{a_l b_l}{(b_l+b_r)(a_l+a_r)}$, i.e. it only depends on the ratios $\frac{a_l}{a_l+a_r}$ and $\frac{b_l}{b_l+b_r}$. In other words: If it is known that both components have made one step, the distribution of the possible outcomes is known. Problems arise when synchronisation comes into play, disabling some of the transitions and therefore breaking the symmetry. This is not forbidden in the CASPA language. We show an example in the next subsection.

2.6.1.3. Problem: synchronised parallel composition

The next problem arises when synchronised and unsynchronised transitions occur, as in the following example [4]. Let $c \in \mathbb{R}_{>0}$ and take the process definitions:

```
A:=(*a,2*);A1+(*a,3*);A2+(*b,7*);A3
B:=(*a,c*);B1
```

Then the composition `A|[a]|B` leads to



(2.1)

which clearly depends on c, i.e. is not scale-free. The deeper meaning of the problem is that the (possibly nondeterministic) choice whether a synchronised or an unsynchronised transition occurs cannot be resolved by probabilistic transitions.

Definition 13. *A CASPA model consisting of N submodels, denoted by $\mathcal{S}_i$, $i \in \{1, \ldots, N\}$, is called* scale-free *if for all $i \in \{1, \ldots, N\}$ the weights in $\mathcal{S}_i$ can be rescaled by any constant $r_i \in \mathbb{R}$, $r > 0$, without changing the resulting PSLTS.*

One can also state a weaker definition that restricts the notion of scale-freeness to the PSLTS after elimination of vanishing states (cf. Ch. 3). The next section gives a characterisation for scale-freeness in the weaker sense by means of Markov Automata and a sufficient criterion in the stronger sense by observations of the state space.

2.6.2. CASPA models as Markov Automata

We want to characterise CASPA models that are strictly compositional in the sense of [35] *and* can be transformed to a CTMC up to lumpability in order to analyse it. For this purpose we will use Markov Automata that provide strict compositionality (by allowing for nondeterminism) to characterise the compositionality of CASPA models. The basic setup is given in Fig. 2.1: The term CASPA models describes the set of CASPA models defined in Sec. 2.3 and Sec. 2.4. This will be referred to as the *standard interpretation*. In the semantics for CASPA models no nondeterminism can occur, as — similar to Generalised Stochastic Petri Nets (GSPNs) — every (possibly) nondeterministic choice is cast by definition to a probabilistic choice according to some weights chosen in the model (i.e. a probabilistic execution fragment [56]). In order to characterise situations, where a CASPA model can be regarded as *strictly compositional*, we introduce a different interpretation of a CASPA model — called CASPA-ND — in terms of Markov Automata that also includes nondeterminism. As other limitations apply (e.g. no synchronisation over timed transitions is allowed), CASPA-ND is not a superset of the standard CASPA models. CASPA models that allow for a CASPA-ND interpretation are called *strictly compositional* and can be characterised in greater detail. CASPA models without CASPA-ND interpretation are called *dubiously compositional*. The property of having a timeless trap is derived from the state space of a CASPA model's standard interpretation (using the semantics given in Sec. 2.4) and is orthogonal to the other characterisations. In the following we target on the *nice* CASPA models which are strictly compositional and allow for CTMC representation.

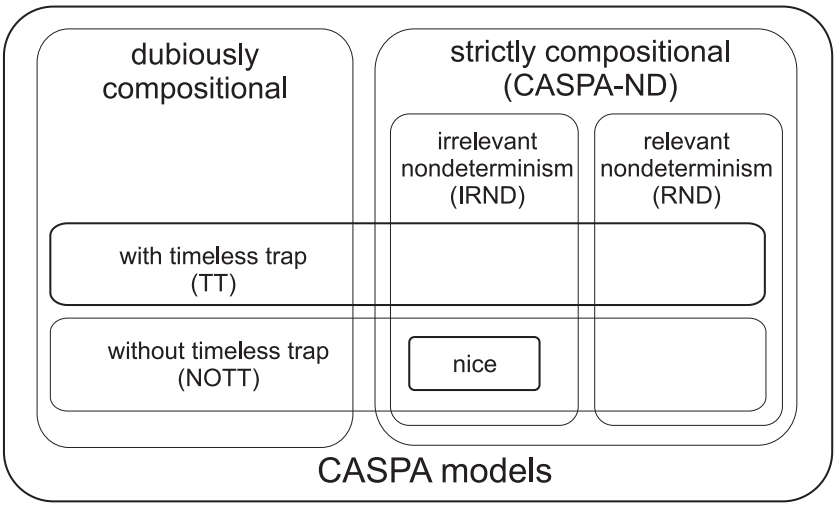


Figure 2.1.: Different classes of CASPA models

2.6.2.1. Markov Automata

We use Markov automata (MA) [23] — because they are strictly compositional — in order to define when CASPA models are strictly compositional.

Definition 14. *A Markov automaton MA is a tuple* $(S, Act, \multimap, \rightarrowtail, s_0)$*, where*

- S *is a nonempty finite set of states*
- Act *is a set of actions containing the internal action* τ
- $\multimap \subset S \times Act \times Dist(S)$ *a set of action-labelled probabilistic transitions (also called PT)*
- $\rightarrowtail \subset S \times \mathbb{R}_{\geq 0} \times S$ *a set of Markovian timed transitions (also called MT) and*
- $s_0 \in S$ *the initial state*

Hereby, $Dist(S)$ *denotes all distributions over* S*.*

There is a weak bisimulation relation $\approx$ defined for MA [23], but we will not go into the details here. Let $\Delta_{s'}$ be the *Dirac distribution* on s' (i.e. $\Delta_{s'}(s') = 1$ and 0 otherwise). As in [23] we will shortly write $s \xrightarrow{a} s'$ for $(s, a, \Delta_{s'}) \in \multimap$. For $s \xrightarrow{\tau} s'$ we will simply write $s \to s'$. The authors in [23] describe a canonical transformation from GSPN to MA: Every GSPN can be considered as an MA $G = (S, \{\tau\}, \multimap, \rightarrowtail, s_0)$ in the sense that:

- S is the set of reachable markings.
- $\multimap$ and $\rightarrowtail$ do not overlap in their first components (where they do, the corresponding element in $\rightarrowtail$ is removed). This is the *maximal progress* assumption.
- As the reachability graph is a WSLTS, we can speak of vanishing and tangible states.
- It is a deterministic MA, i.e. every vanishing state has exactly one outgoing τ transition.

The transformation from GSPN to MA as defined in [23] will be denoted by i_{GSPN}. Note that by abstraction of the transition names, different GSPNs may be mapped to the same MA.

2.6.2.2. Standard transformation

For every CASPA model, a reachable state graph can be constructed using the semantics from Sec. 2.4. We note that this state graph can be transformed into a GSPN:

Remark 6. *For a CASPA model C, the following steps lead to a unique (state machine) GSPN associated to C:*

1. *apply the maximal progress assumption.*
2. *abstract from all action labels (summing up weights/rates of coinciding transitions).*
3. *every reachable state is transformed to a place and there is only one token in the GSPN.*
4. *transitions between states are transitions between places (weights and rates remain the same as in the abstraction above).*

The transformation from a CASPA model to a GSPN consists of uniquely defined operations, therefore a mapping is defined:

Definition 15. *Let C be a CASPA model. The GSPN constructed by Remark 6 will be denoted by C^{GSPN}.*

The following definition associates a unique MA to a given CASPA model.

Definition 16. *There is a canonical mapping $i_{CASPA} : CASPA \to MA$ from CASPA models to MA defined by the following diagram:*

$$\begin{array}{ccc} CASPA & & \\ {\scriptstyle (.)^{GSPN}}\downarrow & \searrow^{i_{CASPA}} & \\ GSPN & \xrightarrow{i_{GSPN}} & MA \end{array}$$

The dashed arrow here means that the mapping is induced by the other mappings. As $(.)^{GSPN}$ and i_{GSPN} are uniquely defined, so is i_{CASPA}.

Principally MA allow for nondeterministic models, but it is easy to observe that i_{CASPA} only leads to deterministic MA:

Remark 7. *According to the definition of the CASPA semantics and the transformation i_{CASPA} it is clear that for any CASPA model C the image $i_{CASPA}(C)$ is an MA that does* not *include any nondeterministic choice.*

Lemma 1. *The following diagram commutes for any CASPA model without timeless traps (for CASPA models we require to have a tangible initial state), where γ_{el} is the elimination of vanishing states (cf. Chapter 3), i_{CTMC} is the canonical inclusion and $\exists \approx$ means that there exists a suitable weakly bisimilar MA.*

$$\begin{array}{ccc} CASPA & \xrightarrow{\gamma_{el}} & CTMC \\ {\scriptstyle i_{CASPA}}\downarrow & & \downarrow{\scriptstyle i_{CTMC}} \\ MA & \xrightarrow{\exists \approx} & MA \end{array}$$

Proof. It is shown in [23] (Th. 7) that the following diagram commutes for any GSPN without timeless traps and with tangible initial marking, where γ_{el} is the elimination of vanishing states

(in the sense of [23]), i_{CTMC} is the canonical inclusion and $\exists \approx$ means that there exists a suitable weakly bisimilar MA.

$$\begin{array}{ccc} GSPN & \xrightarrow{\gamma_{el}} & CTMC \\ {\scriptstyle i_{GSPN}}\downarrow & & \downarrow{\scriptstyle i_{CTMC}} \\ MA & \xrightarrow{\exists\approx} & MA \end{array}$$

Further — in situations without timeless traps — by definition the elimination of vanishing states of a CASPA model C (denoted by γ_{el}, defined in Chapter 3) coincides with the elimination of vanishing states in the state space of the GSPN C^{GSPN} (also denoted by γ_{el}, cf. [23]), i.e. the following diagram commutes:

$$\begin{array}{ccc} CASPA & & \\ {\scriptstyle (.)^{GSPN}}\downarrow & \searrow^{\gamma_{el}} & \\ GSPN & \xrightarrow{\gamma_{el}} & CTMC \end{array}$$

Therefore, the entire diagram commutes. □

2.6.2.3. Nondeterministic approach: $CASPA_{ND}$

The next thing to do is to define a transformation $i_{CASPA_{ND}}$ from CASPA to MA that preserves the (potentially) non-deterministic choices that are transformed to probabilities by i_{CASPA}.

A CASPA model can be regarded as a tree structure, where all the leaves are sequential processes. So the transformation $i_{CASPA_{ND}}$ will be defined recursively over the CASPA language.

As we have to leave the immediate action labels as long as necessary in the corresponding MA (due to further synchronisations), we propose a two-level approach. The first step generates a MA, where immediate action labels other than τ may occur (recall that the maximal progress for MA is only defined for timed transitions competing with τ actions). This step will be called $i^{pot}_{CASPA_{ND}}$ (index *pot* is to be read in the sense of a *potential* transition system: the resulting MA may contain non-τ actions that could disappear after further synchronisation and timed transitions that could disappear due to maximal progress after some non-τ actions are hidden). The second step hides all remaining non-τ immediate action labels, considers the maximal progress assumption and restricts the model to the reachable subset.

Definition 17. *Given a CASPA model C. We define $Act(C)$ as the function that returns a comma-separated list of all non-τ immediate action names in C and mp as the function that removes all MA transitions that violate the maximal progress assumption and performs a reachability analysis afterwards in order to remove superfluous transitions. Then we define*

$$i_{CASPA_{ND}}(C) := mp(i^{pot}_{CASPA_{ND}}(\texttt{hide}\ Act(C)\ \texttt{in}\ C))$$

We see that it remains to define $i^{pot}_{CASPA_{ND}}$ for CASPA models, which is done by the next definitions. Before we can start doing this, we introduce an additional annotation for immediate actions in CASPA models, that concretises the modeller's intension on locality and globality of transitions.

Definition 18 (Local and global transitions)**.** *A CASPA model is called* locally/globally annotated*, if every immediate action $a \in Act \setminus \{\tau\}$ has the suffix* `_loc` *for local transitions, i.e. transitions that only affect this submodel or* `_glob` *if the transition is used for synchronisation with other components. The internal action τ is always assumed to be local, i.e. it cannot be used for further synchronisations (τ can always be read as τ*`_loc`*). We define Act_{loc} (Act_{glob}) as the set of local (global) immediate actions defined in C — with suffix* `_loc` *(*`_glob`*) omitted — and require that $Act_{loc} \cap Act_{glob} = \emptyset$. Immediate transitions that perform local (global) actions are called* local *(*global*)* transitions*.*

In the following we always assume that the CASPA models are locally/globally annotated.

Remark 8. *The local/global assumption is no restriction. The sets Act_{loc} and Act_{glob} can be constructed for every CASPA model if it is closed, i.e. no further parallel composition takes place. For a submodel, without knowing the rest of the system specification, it is impossible to determine Act_{loc} and Act_{glob}. As an example, regarding $R(IDLE)$ from Sec. 2.6.2.6 (Fig. 2.9) as a closed model, Act_{loc} consists of* all *immediate action labels, $Act_{glob} = \emptyset$ (We ignore for the moment, that this would not be a valid CASPA model, as the initial state is vanishing). The picture changes when looking at $R(IDLE)$ in the context of the system specification sys in Sec. 2.6.2.6. In this case $Act_{loc} = \emptyset$ and all immediate action labels are in Act_{glob}.*

Definition 19 (Sequential models)**.** *A* sequential CASPA model *is a model that does not include any parallel composition or hiding operator (i.e. of type* `SequentialProcess` *or parameterised process defined in Sec. 2.3.4). For a sequential CASPA model C let $Trans_I$ be the relation that characterises reachable immediate transitions. We define $i^{pot}_{CASPA_{ND}}(C)$ as the MA $M = (S, Act, PT, MT, s_0)$ where:*

- *S is the set of states reachable from s_0, disregarding maximal progress assumption.*
- *$Act = Act_{glob} \cup \{\tau\}$.*
- *PT is the set of probabilistic transitions (with possible nondeterminism).*
- *MT the set of Markovian transitions.*

We shall characterise PT in greater detail: For every state s with outgoing immediate transitions one calculates

$$prob_{loc}(s \overset{(a\_loc,w)}{\dashrightarrow} s') := \frac{w}{\sum_{\{s \overset{(b\_loc,w)}{\dashrightarrow} s'' \in Trans_I | b \in Act_{loc}\}} w}.$$

and for all global actions

$$prob_{glob}(s \overset{(a\_glob,w)}{\dashrightarrow} s') := \frac{w}{\sum_{\{s \overset{(a\_glob,w)}{\dashrightarrow} s'' \in Trans_I\}} w}.$$

So we can define the local and global transitions:

$$\begin{aligned}
PT_{loc} := \quad & \{(s, \tau, \mu_s) | \quad (\exists a \in Act_{loc} : s \overset{(a_loc,w)}{\dashrightarrow} s' \in Trans_I) \wedge \\
& (\mu_s = \oplus_{\{s \overset{(b_loc,w)}{\dashrightarrow} s' \in Trans_I | b \in Act_{loc}\}} prob_{loc}(s \overset{(b_loc,w)}{\dashrightarrow} s') \cdot \Delta_{s'})\} \\
PT_{glob} := \quad & \{(s, a, \mu_s) | \quad (\exists a \in Act_{glob} : s \overset{(a_glob,w)}{\dashrightarrow} s' \in Trans_I) \wedge \\
& (\mu_s = \oplus_{s \overset{(a_glob,w)}{\dashrightarrow} s' \in Trans_I} prob_{glob}(s \overset{(a_glob,w)}{\dashrightarrow} s') \cdot \Delta_{s'})\} \\
PT := \quad & PT_{loc} \cup PT_{glob}
\end{aligned}$$

Here $\oplus$ means the canonical addition of subdistributions (cf. [23]). The difference between PT_{loc} and PT_{glob} is the normalisation. Local transitions are normalised with respect to all *other concurring local transitions (regardless which local action name they have), while for global transitions only global transitions with the* same *global action name are considered.*

A few remarks on this definition are in order. The basic setup is nondeterministic:

Remark 9 (Nondeterminism)**.** *For a state s in a sequential model, potential nondeterminism is introduced between local actions (label τ) and every global action (label a_i for some $a_i \in Act_{glob}$) emanating from s.*

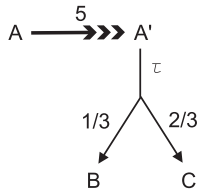


Figure 2.2.: MA for sequential process A

(a) Nondeterminism (b) Resolved after Hiding

Figure 2.3.: Nondeterminism and hiding

But there are special sequential cases like the following:

Remark 10 (Determinism)**.** *For a sequential CASPA model C without global actions and without timeless traps, $i_{CASPA_{ND}}(C)$ is weakly bisimilar to a MA that can be regarded as a CTMC, especially it does not have nondeterministic choices.*

The standard interpretation of CASPA models is not scale-free (cf. Sec. 2.4), but the CASPA-ND semantics is:

Remark 11 (Scale-free)**.** *For a submodel C, by $i_{CASPA_{ND}}(C)$ only weights from* this *submodel are used to calculate transition probabilities. The $CASPA_{ND}$ transformation will be constructed in a way that multiplying all weights within a submodel by some constant $c_0 > 0$ will not change the transformation result (in contrast to the standard CASPA interpretation, cf. the example in Sec. 2.6.1.3).*

Example 1. *As an easy example we transform the CASPA model:*

```
A:=(a,5);A'
A':=((*b_loc,1*);B+(*c_loc,2*);C)
```

The result can be seen in Fig. 2.2. Note that the local transitions are converted into a common τ transition, while the global action label remains.

The hiding operator will simply relabel the hidden action labels to τ labels.

Definition 20 (Hiding operator)**.** *The MA of a hidden CASPA model* `hide a in C` *is defined in the following way: Let $i^{pot}_{CASPA_{ND}}(C) = (S, Act, PT, MT, s_0)$, then*

$$i^{pot}_{CASPA_{ND}}(\texttt{hide a in } C) := (S, (Act \setminus \{a\}) \cup \tau, PT^{\hat{a}}, MT, s_0),$$

where

$$PT^{\hat{a}} := \{(s, \tau, \mu) | (s, a, \mu) \in PT\} \cup \{(s, b, \mu) | b \neq a \wedge (s, b, \mu) \in PT\}.$$

Remark 12. *Hiding does not change nondeterministic situations a priori, as it only does a relabelling. By the relabelling process some additional actions are rendered as τ actions, so the MA weak bisimulation can be applied. The example in Fig. 2.3a shows a nondeterministic situation that can be resolved to the deterministic one in Fig. 2.3b after hiding actions* `a` *and* `b`*.*

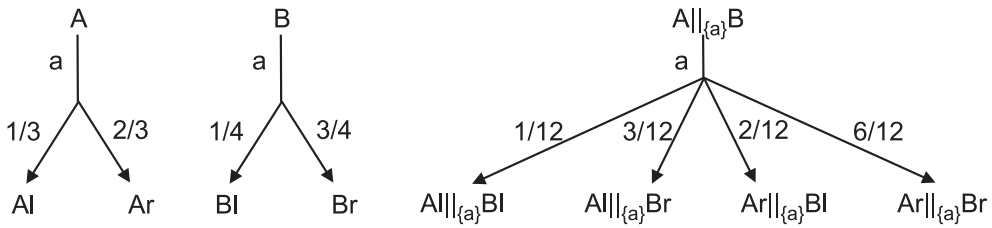


Figure 2.4.: MA for synchronised parallel composition example

In the following we assume that the modeller knows, when a global action is no longer needed and abstracts from it by the hide operation. It remains to define the parallel composition in $i^{pot}_{CASPA_{ND}}$.

Definition 21. *As long as the synchronisation set does not include Markovian actions, the parallel composition in $CASPA_{ND}$ is defined as*

$$i^{pot}_{CASPA_{ND}}(C1 \,|\, [\mathtt{S}] \,|\, C2) := \left(i^{pot}_{CASPA_{ND}}(C1)\right) ||_S \left(i^{pot}_{CASPA_{ND}}(C2)\right),$$

where $||_S$ is the MA parallel composition (cf. [23]).

The synchronised parallel composition in CASPA-ND is deterministic, as the following example shows:

Example 2. *This is the example from Sec. 2.6.1.1, adapted to the local/global notation:*

```
A:=(*a_glob,1*);Al+(*a_glob,2*);Ar
B:=(*a_glob,1*);Bl+(*a_glob,3*);Br
```

The result can be seen in Fig. 2.4. As expected, no relevant nondeterminism occurs (as long as the successor processes do not have relevant nondeterminism).

The idea now is that the potential nondeterminism introduced in the definition $i^{pot}_{CASPA_{ND}}$ for sequential models gets reduced to determinism by synchronisation constraints. We can observe:

Remark 13. *No nondeterminism can occur when*

1. *every nondeterministic choice is reduced to a deterministic choice by synchronisation constraints.*

2. *the resulting MA is weakly bisimilar to a MA without nondeterminism.*

The prior case is easier to check but is not exhaustive (e.g. the situation in Remark 12 is not regarded as nondeterministic). The latter case is more complicated to check but exhaustive.

In the sequel we will target on the latter case.

Definition 22. *A CASPA model C is called (cf. Fig. 2.1)*

- strictly compositional, *if $i_{CASPA_{ND}}(C)$ is defined, i.e. C contains* no *synchronisation over Markovian actions.*

- dubiously compositional, *if $i_{CASPA_{ND}}(C)$ is not defined, i.e. C contains synchronisation over Markovian actions.*

- *with* irrelevant nondeterminism (IRND), *if it is strictly compositional and there exists* $M \approx i_{CASPA_{ND}}(C)$, $M \neq \emptyset$, *such that* $i_{GSPN}^{-1}(M)$ *exists. That means, there is an induced function* $\gamma_{det}^{\approx}$ *in Eq. 2.2.*

$$\begin{array}{ccc} CASPA & \overset{\gamma_{det}^{\approx}}{\dashrightarrow} & GSPN \\ {\scriptstyle i_{CASPA_{ND}}}\downarrow & & \uparrow{\scriptstyle i_{GSPN}^{-1}} \\ MA & \xrightarrow{\exists \approx} & MA \end{array} \tag{2.2}$$

  Note that in this case M *must be deterministic, as* i_{GSPN} *only leads to deterministic MA.*

- *with* relevant nondeterminism (RND) *if it is strictly compositional but does not have the (IRND) property.*

- nice, *if it is strictly compositional and there exists* $M \approx i_{CASPA_{ND}}(C)$, $M \neq \emptyset$, *such that* $i_{CTMC}^{-1}(M)$ *exists. That means, there is an induced function* $\gamma_{ind}^{\approx}$ *in Eq. 2.3.*

$$\begin{array}{ccc} CASPA & \overset{\gamma_{ind}^{\approx}}{\dashrightarrow} & CTMC \\ {\scriptstyle i_{CASPA_{ND}}}\downarrow & & \uparrow{\scriptstyle i_{CTMC}^{-1}} \\ MA & \xrightarrow{\exists \approx} & MA \end{array} \tag{2.3}$$

  Note that in this case M *must be deterministic and must not contain timeless traps.*

- universally nice *if it is nice and it remains nice for all possible choices of Markovian rates in the model.*

To relate the definition of IRND CASPA models to the scale-free terminology we show:

Lemma 2. *Universally nice CASPA models are scale-free in the weak sense, i.e. the corresponding CTMCs do not depend on rescalings of submodels*[1]*.*

Proof. Let C be a universally nice CASPA model. Note that in $i_{CASPA_{ND}}(C)$ no nondeterministic choices between τ transitions leading to bisimilar Markovian successors may occur (for otherwise this nondeterminism could not be resolved for different choices of the rates). First observe that all distributions occurring in $i_{CASPA_{ND}}(C)$ do not depend on any rescaling of submodels in C. But then it is clear that also in a weakly bisimilar MA the distributions cannot depend on any rescaling of submodels in C, especially the one without nondeterministic decisions, as assumed by the IRND property. By the nice property all states with outgoing τ transitions can be eliminated. So we conclude that the resulting CTMC does not depend on any rescaling of submodels. □

We show that $\gamma_{ind}^{\approx}$ actually does not depend on the choice of the equivalent MA in the definition.

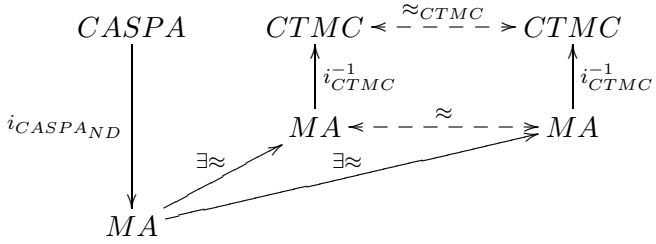
Lemma 3. *Let* C *be a strictly compositional CASPA model. If* $\gamma_{ind}^{\approx}(C)$ *in Eq. 2.3 exists, it is unique up to lumping equivalence, i.e. the mapping*

$$\gamma_{ind} : CASPA \to {}^{CTMC}/_{\approx_{CTMC}}$$

is well-defined for nice CASPA models.

[1] The converse is not true. Already in the case of two processes scale-freeness is possible also for models with relevant nondeterminism

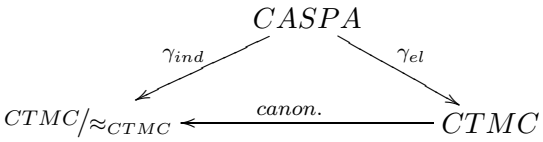
Proof. Suppose there are two possible choices for the equivalent MAs as shown in the following diagram, then by transitivity of $\approx$, they are weakly bisimilar (this is indicated by the dashed line between the MA).



As lumpability corresponds in the CTMC case to weak bisimilarity [22], the dashed line between the CTMCs is also clear. Therefore, γ_{ind} is well-defined up to lumpability, which was to be shown. □

In IRND situations the given weights for the nondeterministic choices do not play a role (as it actually does not matter, which nondeterministic branch is taken). If additionally it holds that no timeless trap is present, the following Lemma can be stated.

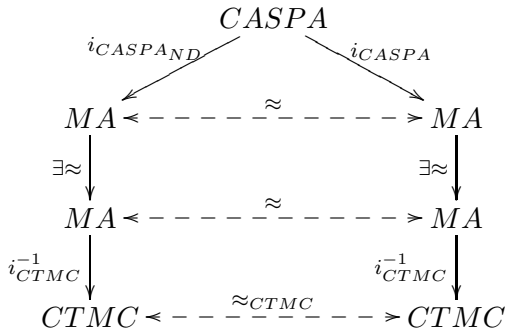
Lemma 4. *If a CASPA model C is nice, the following diagram commutes*



(2.4)

where canon. is the canonical mapping into the quotient.

Proof. Let C be a nice CASPA model. Recall that $\gamma_{ind} = i^{-1}_{CTMC} \circ \exists \approx \circ i_{CASPA_{ND}}$ for every suitable weakly bisimilar MA (without nondeterminism and vanishing states) and that by Lemma 1 $\gamma_{el} = i^{-1}_{CTMC} \circ \exists \approx \circ i_{CASPA}$ for one suitable weakly bisimilar MA. So it is sufficient to show that the following diagram commutes



(2.5)

By definition, the initial marking of C does not vanish. As C is nice, C must have the IRND property, so nondeterminism does not play a role for the model, i.e. it is independent of the choice of weights, when the non-determinism is converted to a probabilistic choice. Especially one can choose the weights as done by i_{CASPA}. Therefore, weak bisimilarity of the first step is guaranteed. In the second step, the suitable weak bisimulations on MA level are calculated in order to remove all probabilistic and non-deterministic decisions. As MA weak bisimulation coincides with lumpability on CTMCs [22], the equivalence in the last step is shown. Because of transitivity of $\approx$ the lemma is proven. □

We see that in this special case the elimination of vanishing states actually calculates a weakly bisimilar MA.

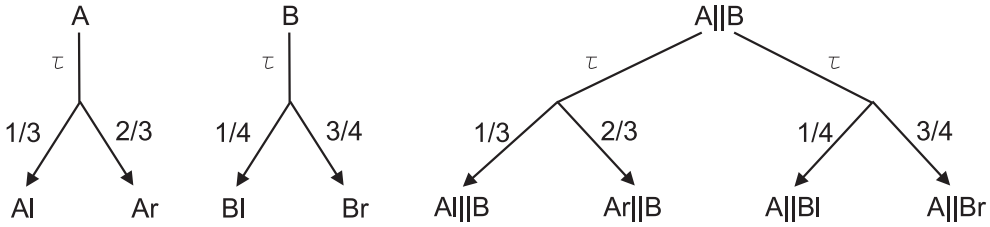


Figure 2.5.: MA for unsynchronised parallel composition example

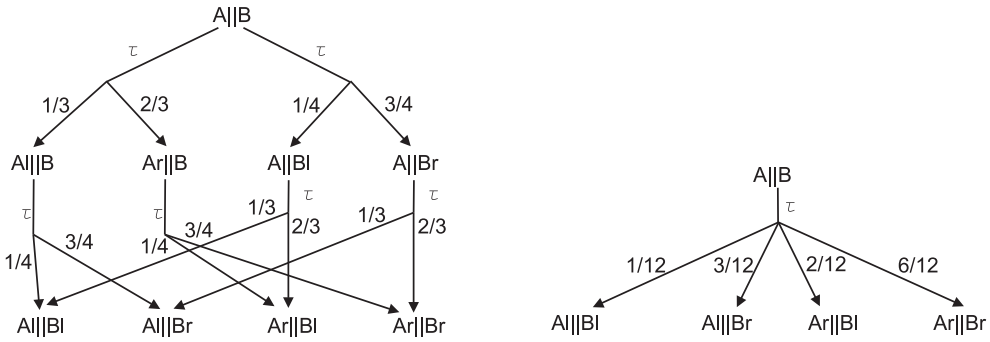


Figure 2.6.: Weakly bisimilar MA's for two steps

In the following subsections we give some examples of the current machinery.

2.6.2.4. Unsynchronised parallel composition

The example in Sec. 2.6.1.2, adapted to local/global notation is:

```
A:=(*a_loc,1*);Al+(*a_loc,2*);Ar
B:=(*a_loc,1*);Bl+(*a_loc,3*);Br
```

The result can be seen in Fig. 2.5. In contrast to the interpretation by the CASPA semantics, nondeterminism is introduced by the CASPA-ND interpretation. In some lucky cases, when the resulting processes finally lead to the same behaviour, there exists a weak bisimulation that removes the nondeterminism. In the current example (assuming Al, Ar, Bl, Br cannot perform immediate steps), one sees this by the looking at the next step. There it is clear that only probabilistic choices can occur. No matter, which τ-branch has been taken, the same distributions on {`Al||Bl`, `Al||Br`, `Ar||Bl`, `Ar||Br`} result, therefore both MA in Fig. 2.6 are weakly bisimilar. If no relevant nondeterminism occurs in the successor states, this model is IRND

2.6.2.5. Synchronised parallel action concurring with local action

This is the example from Sec. 2.6.1.3, annotated with local/global extensions. For $c \in \mathbb{R}_{>0}$ we have:

```
A:=(*a_glob,2*);A1+(*a_glob,3*);A2+(*b_loc,7*);A3
B:=(*a_glob,c*);B1
```

The MA of `A|[a]|B` is given in Fig. 2.7. In contrast to the MA that would result by i_{CASPA} from the state graph in Sec. 2.6.1.3 we see that the result here does no longer depend on c. If the successor states are not weakly bisimilar, this model is a RND case.

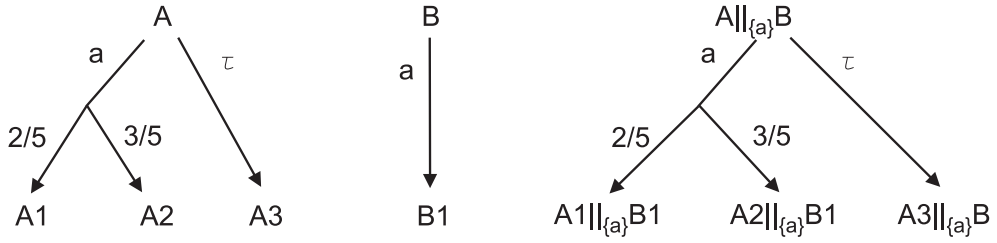


Figure 2.7.: MA for processes A, B and $A||_{\{a\}}B$

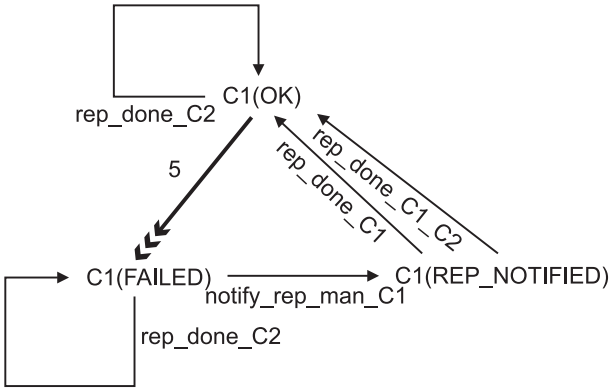


Figure 2.8.: MA representing C1(OK) (interpreted by $CASPA_{ND}$)

2.6.2.6. A large example

Suppose we have the following three parameterised processes C1, C2, R that model a reliable system that consists of two unreliable components and a repairman. The repairman is able to repair up to two failed components at once.

```
/* state definitions for unreliable components */
int OK = 0;
int FAILED = 1;
int REP_NOTIFIED = 2;
/* state definitions for unreliable components */
int IDLE = 0;
int BUSY = 1;
```

In the following, we assume that all immediate transitions are global, therefore we omit the required `_glob` suffix. The components are defined as follows:

```
C1(state [2]):= [state!=REP_NOTIFIED] -> (*rep_done_C2,1*); C1(state)
                [state=OK] -> (Fail,5); C1(FAILED)
                [state=FAILED] -> (*notify_rep_man_C1,1*); C1(REP_NOTIFIED)
                [state=REP_NOTIFIED] -> (*rep_done_C1,1*); C1(OK)
                [state=REP_NOTIFIED] -> (*rep_done_C1_C2,1*); C1(OK)

C2(state [2]):= [state!=REP_NOTIFIED] -> (*rep_done_C1,1*); C2(state)
                [state=OK] -> (Fail,5); C2(FAILED)
                [state=FAILED] -> (*notify_rep_man_C2,1*); C2(REP_NOTIFIED)
                [state=REP_NOTIFIED] -> (*rep_done_C2,1*); C2(OK)
                [state=REP_NOTIFIED] -> (*rep_done_C1_C2,1*); C2(OK)
```

As C1 and C2 are quite similar, we only show the MA $i_{CASPA_{ND}}(C1(OK))$ in Fig. 2.8

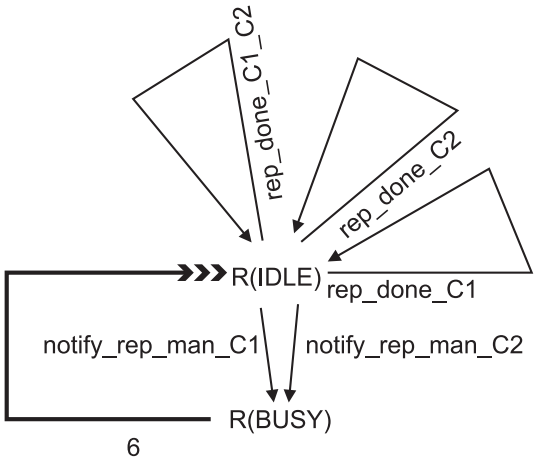


Figure 2.9.: MA representing R(IDLE) (interpreted by $CASPA_{ND}$)

```
R(state [1]):= [state=IDLE] -> (*notify_rep_man_C1,1*); R(BUSY)
               [state=IDLE] -> (*notify_rep_man_C2,1*); R(BUSY)
               [state=IDLE] -> (*rep_done_C1,1*); R(IDLE)
               [state=IDLE] -> (*rep_done_C2,1*); R(IDLE)
               [state=IDLE] -> (*rep_done_C1_C2,1*); R(IDLE)
               [state=IDLE] -> (*Repair,6*); R(IDLE)
```

The system is defined by the following CASPA code:

```
sys:= (
        (
           C1(OK)
           |[rep_done_C1. rep_done_C2, rep_done_C1_C2]|
           C2(OK)
        )
        |[rep_done_C1. rep_done_C2, rep_done_C1_C2,
          notify_rep_man_C1, notify_rep_man_C2]|
        R(IDLE)
      )
```

and the corresponding state space (assuming maximal progress, but leaving the action names in the model for better readability) is shown in Fig. 2.10. There the states

$$(C1(FAILED)||_{S1}C2(OK))||_{S2}R(IDLE))$$

and

$$(C1(OK)||_{S1}C2(FAILED))||_{S2}R(IDLE))$$

have been copied to avoid crossing transitions. Fig. 2.11 shows a model that is weakly bisimilar to the system model (after hiding all action names): As the exit rates of the different successor states of

$$(C1(REP\_NOTIFIED)||_{S1}C2(FAILED))||_{S2}R(IDLE)$$

and

$$(C1(FAILED)||_{S1}C2(REP\_NOTIFIED))||_{S2}R(IDLE)$$

do not match, there is relevant nondeterminism in the model. This nondeterminism cannot be seen easily from the CASPA specification. Probably the modeller didn't intend this behaviour.

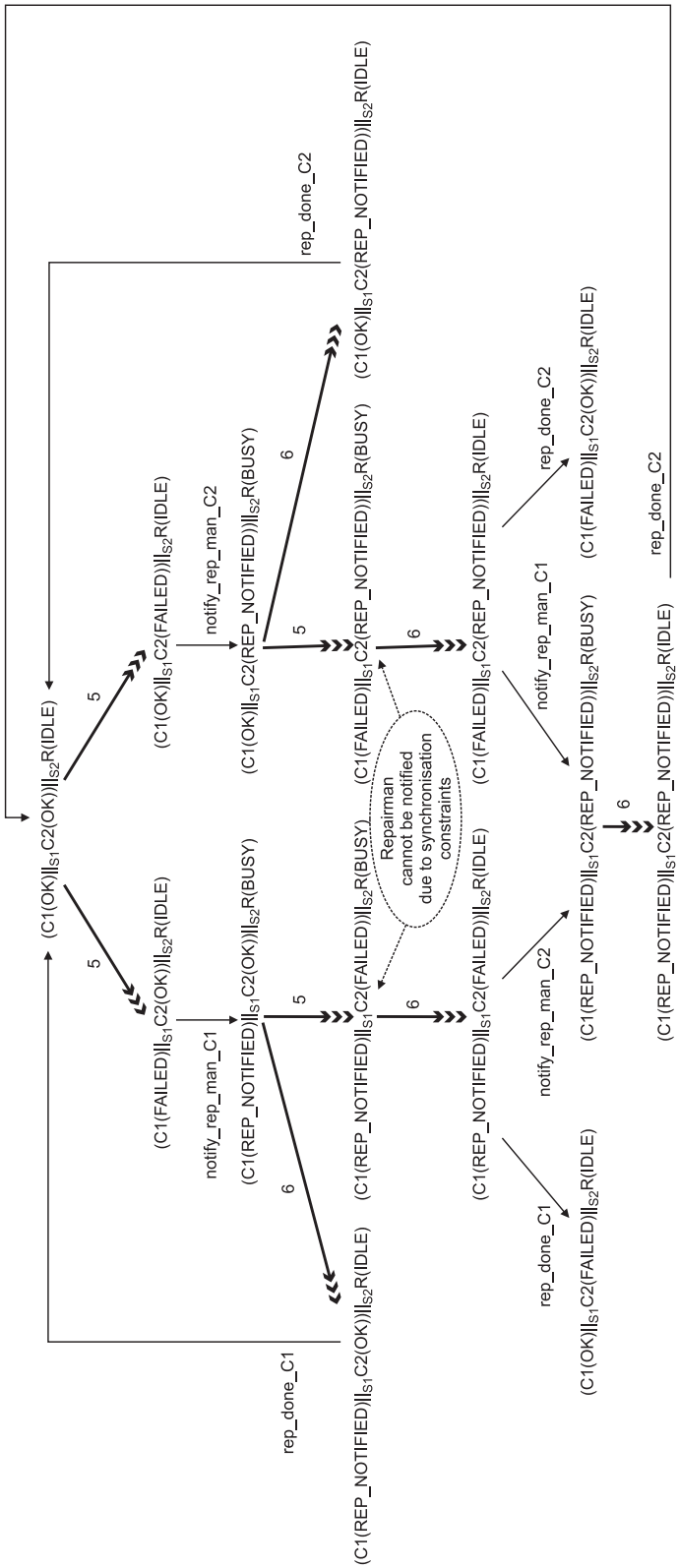


Figure 2.10.: MA representing system process (interpreted by $CASPA_{ND}$)

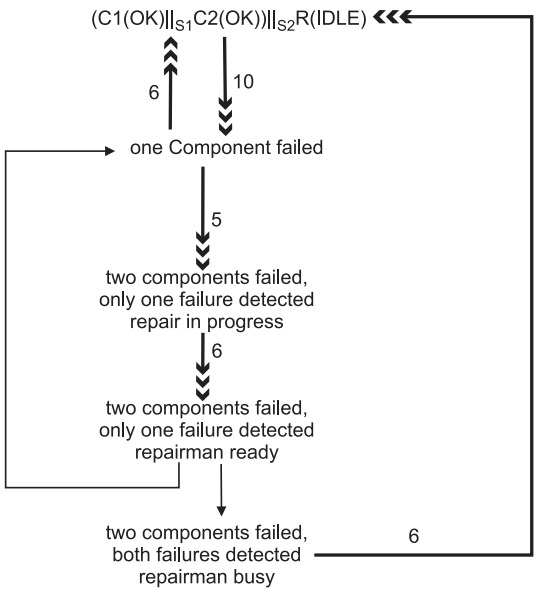


Figure 2.11.: MA representing system process (interpreted by $CASPA_{ND}$)

2.6.3. A sufficient criterion for nice CASPA models

Without using MA bisimulation we can state the following remark for CASPA models. All properties can be checked on the state space (possibly adding label suffixes for the current submodel). We will consider the case 1 in Remark 13. It may be reformulated to:

Remark 14. *If after maximal progress assumption for every vanishing state it holds that*

1. a) *only one global action and no local action emanates or*
   b) *arbitrary many local actions coming from the* same *submodel and no global actions emanate.*
2. *the elimination procedure (cf. Chapter 3) does not detect a timeless loop.*

then the CASPA model is nice.

This is a compromise that can be calculated with moderate effort. As we don't see the global picture given by the nice property in terms of MA weak bisimulation, but only the local view, we can say that: If there is at least one problem that violates the conditions in Remark 14 then the model might not be nice (but it could be). So actually we might produce a lot of false negatives but we cannot produce false positives. For the calculations we need some further information:

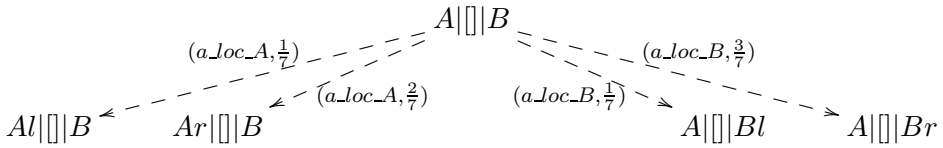
Definition 23. *A locally/globally annotated CASPA model is called* partitioned*, if every local action has an additional suffix that determines the submodel a local action acts in. As before A_{loc} still covers only action names without any suffix (neither* `loc` *nor the submodel suffix).*

As for the local/global annotation we would like to state that for closed CASPA models the submodel annotation can always be performed. We come back to our examples which we assume to be partitioned.

2.6.3.1. Unsynchronised parallel composition

```
A:=(*a_loc_A,1*);Al+(*a_loc_A,2*);Ar
B:=(*a_loc_B,1*);Bl+(*a_loc_B,3*);Br
```

Obviously `A|[]|B` has four possible transitions.



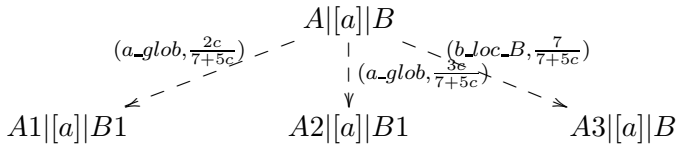
As there are local transitions from A and B in conflict, the conditions of Remark 14 are not fulfilled and we suspect the model of not being nice.

2.6.3.2. Synchronised parallel action concurring with local action

This is the example from Sec. 2.6.1.3, annotated with local/global extensions. For $c \in \mathbb{R}_{>0}$ we have:

```
A:=(*a_glob,2*);A1+(*a_glob,3*);A2+(*b_loc_B,7*);A3
B:=(*a_glob,c*);B1
```

The composition `A|[a]|B` leads to



We suspect the model of not being nice.

2.6.3.3. Large example reviewed

In this example there are concurring global actions in two different situations:

- `rep_done_C1` and `notify_rep_man_C2` in state

$$(C1(REP\_NOTIFIED)||_{S1}C2(FAILED))||_{S2}R(IDLE)$$

- `rep_done_C2` and `notify_rep_man_C1` in state

$$(C1(FAILED)||_{S1}C2(REP\_NOTIFIED))||_{S2}R(IDLE)$$

therefore we suspect the model of not being nice. Here, the modeller could be warned that there are states that could potentially lead to nondeterminism.

2.6.4. Discussion

We have defined a notion of *nice* CASPA models which are strictly compositional and can be reduced to CTMCs. In general nice CASPA models are hard to find, as MA weak bisimulations have to be calculated. A much simpler (but by far not exhaustive) approach has been presented that only makes use of some properties of the state space of a CASPA model and calculates the elimination of vanishing states (if existent).

2.7. MTBDD data structure

The MTBDD data structure provides means for compactly representing a special kind of mapping called generalised switching functions (GSFs). MTBDD software packages (e.g. CUDD

[20], JINC [33]) allow for efficiently manipulating the MTBDD data structure. Firstly the GSFs will be introduced. Secondly, by a small example, the connection between GSFs and matrices is shown and then MTBDDs are introduced.

2.7.1. Generalised switching functions

As usual, a mapping is called embedding if it is injective, onto if it is surjective and bijective (or one-to-one) if it is both injective and surjective.

Definition 24. *A* generalised switching function *is a function $f : \mathbb{B}^n \to M$, where $n \in \mathbb{N}$, $\mathbb{B} := \{0,1\}$ is the (two-element) Boolean set and M is a set. The set of all generalised switching functions $f : \mathbb{B}^n \to M$ will be denoted by $GSF(n, M)$.*

For the abstraction and apply operation it is assumed that $(M, *)$ is a law of composition, (that is a set M with an operation $M \times M \longrightarrow M$). We will assume this for the rest of this section.

Remark 15. *Even if the set M is infinite, for a certain function $f \in GSF(n, M)$ the image $Im(f)$ will be finite (at most 2^n distinct values).*

Remark 16. *Setting $M := \mathbb{B}$ with an arbitrary binary operation, the generalised switching functions specialise to switching functions in the usual sense.*

The aim of Binary Decision Diagrams (BDDs) is to compactly encode switching functions [13] in a canonical way. One well-known problem of BDDs is that integer multiplication can only be handled by BDDs of exponential size in the variables representing the multiplicands [14]. Multiplication is often needed in the context of matrix representations. This has led to Multi-Terminal Binary Decision Diagrams (MTBDDs), where multiplication (or some other operation) is handled by operations on the terminal nodes rather than by a binary encoding of the numbers.

Remark 17. *There is a one-to-one correspondence between MTBDDs over M (in n variables) and generalised switching functions $GSF(n, M)$. Similarly there is a one-to-one correspondence between BDDs (in n variables) and switching functions $GSF(n, \mathbb{B})$. We will introduce some basic operations on MTBDDs by means of operations on switching functions.*

Definition 25. *The restriction on index $i \in \{1, \ldots, n\}$ with value x of $f : \mathbb{B}^n \to M$ is defined by*

$$\begin{array}{rccc} |_{i=x}\colon & GSF(n,M) & \longrightarrow & GSF(n,M) \\ & f & \mapsto & f\,|_{i=x} \end{array}$$

where $\begin{array}{rccc} f\,|_{i=x}\colon & \mathbb{B}^n & \to & M \\ & b & \mapsto & f(b_1, \ldots, b_{i-1}, x, b_{i+1}, \ldots, b_n) \end{array}$.

$f\,|_{i=x}$ is also written as $RESTRICT(f, i, x)$.

Remark 18. *Operations on GSFs will be defined in a pointwise manner. Therefore the set $GSF(n, M)$ can also be equipped with a law of composition by the operation*

$$\begin{array}{rccc} *^*: & GSF(n,M) \times GSF(n,M) & \longrightarrow & GSF(n,M) \\ & (f,g) & \mapsto & f *^* g \end{array}$$

where

$$\begin{array}{rccc} f *^* g: & \mathbb{B}^n & \longrightarrow & M \\ & (b_1, \ldots, b_n) & \mapsto & f(b_1, \ldots, b_n) * g(b_1, \ldots, b_n). \end{array}$$

This construction gives $GSF(n, M)$ the structure of a law of composition $(GSF(n, M), *^*)$. We say that $*^*$ is induced by $*$. If the context is clear, we simply write $*$ instead of $*^*$. The operation $*^*$ can be calculated pointwise on the images of f and g. This is the motivation for the APPLY operator in the MTBDD setup.

Definition 26. *Let $(M, *)$ be a law of composition and let $(GSF(n, M), *^*)$ be the induced law of composition. The abstraction of the switching function $f : \mathbb{B}^n \to M$, with respect to index $i \in \{1, \ldots, n\}$ and the operation $*$ is defined as*

$$\begin{array}{rccc} A(i,*): & GSF(n,M) & \longrightarrow & GSF(n,M) \\ & f & \mapsto & f\,|_{i=0} *^* f\,|_{i=1} \end{array}$$

$A(i,*)(f)$ is also written as $ABSTRACT(f, i, *)$.

One can now ask which conditions ensure that the following diagram is commutative ($i, j \in \{1, 2, \ldots, n\}$, $i \neq j$)?

$$\begin{array}{ccc} GSF(n,M) & \xrightarrow{A(i)} & GSF(n,M) \\ {\scriptstyle A(j)}\downarrow & & \downarrow{\scriptstyle A(j)} \\ GSF(n,M) & \xrightarrow[A(i)]{} & GSF(n,M) \end{array}$$

This question was answered incorrectly in the literature [5] and has now been corrected [26]. Commutativity of the diagram holds for so-called medial (also called alternation, transposition, interchange, bi-commutative, bisymmetric, surcommutative, entropic in the literature) laws of composition [60, 61]. For this work it is enough to say that all associative and commutative laws of composition lead to the commutativity of the diagram above and therefore the abstraction of a set of indices is well-defined.

Remark 19. *From a switching function $f \in GSF(n, M)$ new switching functions can be constructed using permutations of the arguments. The permutation group S_n defines all permutations of n elements. For every $\sigma \in S_n$, a function $f^\sigma := f \circ \sigma$ can be defined ($f^{id} = f$).*

$$\begin{array}{ccc} \mathbb{B}^n & \xrightarrow{\sigma} & \mathbb{B}^n \\ & \searrow^{f^\sigma} & \downarrow{\scriptstyle f} \\ & & M \end{array}$$

Remark 20. *Suppose $m \geq n$ and let $\{k, \ldots, l\} := \{k, k+1, \ldots, l-1, l\} \subset \mathbb{N}$ with the induced order. Consider the order-preserving embeddings*

$$I^{ord}_{n \hookrightarrow m} := \{e : \{1, \ldots, n\} \longrightarrow \{1 \ldots m\} | e \ \texttt{strictly monotonic increasing}\}.$$

One has $|I^{ord}_{n \hookrightarrow m}| = \binom{m}{n}$. An embedding $e \in I^{ord}_{n \hookrightarrow m}$ induces an embedding of generalised switching functions in the canonical way:

$$\begin{array}{rccc} \tilde{}: & GSF(n,M) & \longrightarrow & GSF(m,M) \\ & f & \mapsto & \tilde{f} \end{array}$$

where $\tilde{f}$ is defined as

$$\begin{array}{rccc} \tilde{f}: & \mathbb{B}^m & \longrightarrow & M \\ & (b_1, \ldots, b_m) & \mapsto & f(b_{e(1)}, \ldots, b_{e(n)}) \end{array}$$

Note: Whenever a function f is to be interpreted as a function $\tilde{f}$, the embedding e has to be given (unless the canonical embedding $id|_{\{1,\ldots,n\}}$ is used). The general embeddings $I_{n \hookrightarrow m}$ are not necessarily order-preserving. A general embedding is given as a permutation of an order-preserving embedding. As $|S_n| = n!$, the total number of general embeddings $f \xrightarrow{\sigma} f^\sigma \hookrightarrow \tilde{f^\sigma}$ is $\binom{m}{n} \cdot n! = \frac{m!}{(m-n)!}$.

Definition 27. *A switching function $f \in GSF(n, M)$ is called* independent of the i-th compo-

nent, *if*

$$\forall (b_1, \ldots, b_{n-1}) \in \mathbb{B}^{n-1} : f(b_1, \ldots, b_{i-1}, 0, b_i, \ldots, b_{n-1}) = f(b_1, \ldots, b_{i-1}, 1, b_i, \ldots, b_{n-1}).$$

Switching functions which are independent of the i-th component will be denoted by $GSF^{\hat{i}}(n, M)$.

Remark 21. *There is a well-defined mapping (assuming* $GSF(0, M) := M$*)*

$$\begin{array}{rccc} ' : & GSF^{\hat{i}}(n, M) & \longrightarrow & GSF(n-1, M) \\ & f & \mapsto & f' \end{array}$$

with

$$\begin{array}{rccc} f' : & \mathbb{B}^{n-1} & \longrightarrow & M \\ & (b_1, \ldots, b_{n-1}) & \mapsto & f(b_1, \ldots, b_{i-1}, 0, b_i, \ldots, b_{n-1}) \end{array}$$

The mapping introduced in Remark 20 with the embedding

$$\begin{array}{rccc} i : & \{1, \ldots, n-1\} & \longrightarrow & \{1, \ldots, n\} \\ & x & \mapsto & \begin{cases} x & \texttt{for } x < i \\ x+1 & \texttt{for } x \geq i \end{cases} \end{array}$$

is for $n > 1$ *the inverse mapping to '. For* $n = 1$ *one uses*

$$\begin{array}{rccc} i_0 : & M = GSF(0, M) & \longrightarrow & GSF(1, M) \\ & m & \mapsto & f(x) = m \end{array}$$

(the mapping of m to the constant function with value m) and therefore gets canonical identifications of switching functions that are independent of some of their parameters with switching functions of fewer parameters.

Definition 28. *Suppose* $f \in GSF(n, M)$ *and* $(M, <)$ *is an ordered set and* $t \in M$*, then the threshold function can be defined as*

$$\begin{array}{rccc} T(t) : & GSF(n, M) & \to & GSF(n, \mathbb{B}) \\ & f & \mapsto & f_{01} \end{array}$$

where

$$\begin{array}{rccc} f_{01} : & \mathbb{B}^n & \longrightarrow & \mathbb{B} \\ & b & \mapsto & \begin{cases} 1 & \texttt{if } f(b) > t \\ 0 & \texttt{otherwise} \end{cases} \end{array}$$

$T(t)(f)$ *will also be called THRESHOLD(f,t).*

Definition 29. *The usual if-then-else construct for switching functions is defined as follows:*

$$\begin{array}{rccc} ITE : & GSF(n, \mathbb{B}) \times GSF(n, M) \times GSF(n, M) & \longrightarrow & GSF(n, M) \\ & (f, g, h) & \mapsto & ITE(f, g, h) \end{array}$$

where

$$\begin{array}{rccc} ITE(f, g, h) : & \mathbb{B}^n & \longrightarrow & M \\ & b & \mapsto & \begin{cases} g(b) & \texttt{if } f(b) = 1 \\ h(b) & \texttt{otherwise} \end{cases} \end{array}$$

Definition 30. *Assume now that* $(\mathbb{B}, \wedge, \neg)$ *is the usual Boolean Algebra. The Shannon (Boole) expansion of a switching function* $f : \mathbb{B}^n \to (\mathbb{B}, *)$ *with respect to* b_i *is defined as*

$$f = b_i \cdot f|_{b_i=1} + \bar{b}_i \cdot f|_{b_i=0}$$

If the Shannon expansion has been applied successively to all variables and the terms with function values 0 are omitted, the function f is in disjunctive normal form (DNF).

Remark 22. *Also for generalised switching functions there are unique generalised DNFs: With the help of indicator functions f_c for all $c \in M$, the generalised switching function f can be converted to $|M|$ (maybe trivial) switching functions ($f_c(b_1, \ldots, b_n) := 1$ if $f(b_1, \ldots, b_n) = c$, 0 otherwise). All these switching functions have unique DNF representations. Then $f = \sum_{c \in M} c \cdot f_c$ is the unique generalised DNF of f (assuming $c \cdot 1 := c$, $c \cdot 0 = 0$).*

2.7.1.1. Example: Matrix multiplication

This small example shows how matrix multiplication can be done by GSFs. Suppose the matrices given are $A := \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ and $B := \begin{pmatrix} 3 & 4 \\ 1 & 2 \end{pmatrix}$ and we want to calculate $A \cdot B = \begin{pmatrix} 5 & 8 \\ 13 & 20 \end{pmatrix}$. In the sequel, we assume that $\mathbb{N}$ is equipped with both an additive $(+)$ and a multiplicative $(\cdot)$ structure. Firstly, a mapping from the $n \times n$ matrices over $\mathbb{N}$ ($Mat(n \times n, \mathbb{N})$) to $GSF(2 \cdot \lceil \log_2 n \rceil, \mathbb{N})$ is defined. The index function maps a row- or column index to the corresponding bitstring:

$$\begin{array}{rccc} \iota: & \{0, \ldots, n-1\} & \longrightarrow & \mathbb{B}^{\lceil \log_2 n \rceil} \\ & x & \mapsto & Euclid(x) \end{array}$$

Here *Euclid* is the usual Euclidean algorithm to perform the base change to base 2. For the mapping to be correct, the indices must be counted starting with 0. Suppose a matrix $M = (m_{ij})_{i,j=\{1,\ldots,n\}}$ should be encoded. For row-index r and column-index c, $\iota(r)$ and $\iota(c)$ are bit strings of length $\lceil \log_2 n \rceil$. Now every tuple $(\iota(r)_1, \iota(c)_1, \iota(r)_2, \ldots, \iota(c)_n)$ is mapped to the corresponding matrix entry m_{rc}. Note that the row and column bits are used in an interleaved fashion.

Coming back to the example above, the situation is as follows: The two parameters are interpreted as row- and column index (Remember that any other permutation of the parameters would also do).

x	$f_A(x)$	$f_B(x)$
(0,0)	1	3
(0,1)	2	4
(1,0)	3	1
(1,1)	4	2

Now, the functions are mapped to $GSF(3, \mathbb{N})$ by the following embeddings: $e_A := id|_{\{1,2\}}$, $e_B : \{1,2\} \to \{1,2,3\}, e_B(1) := 2, e_B(2) := 3$. The scalar multiplication is done by two steps: Firstly, the products are calculated. This is done by $\tilde{f}_A \cdot \tilde{f}_B$ The result is given in the following table:

x	$\tilde{f}_A(x)$	$\tilde{f}_B(x)$	$(\tilde{f}_A \cdot \tilde{f}_B)(x)$
(0,0,0)	1	3	3
(0,0,1)	1	4	4
(0,1,0)	2	1	2
(0,1,1)	2	2	4
(1,0,0)	3	3	9
(1,0,1)	3	4	12
(1,1,0)	4	1	4
(1,1,1)	4	2	8

Secondly, the abstraction over index 2 is done by operation $+$: $ABSTRACT(\tilde{f}_A \cdot \tilde{f}_B, 2, +)$. The result is independent of the second component and will be interpreted as an element of

$GSF(2, (\mathbb{R}, +, \cdot))$

x	$ABSTRACT(\tilde{f}_A \cdot \tilde{f}_B, 2, +)(x)$
(0,0)	3+2=5
(0,1)	4+4=8
(1,0)	9+4=13
(1,1)	12+8=20

that is, with the initial interpretation of the row and column variable, the matrix we expected.

2.7.2. (Multi-Terminal) Binary Decision Diagrams

MTBDDs are used as a compact graphical (also called *symbolic*) representation of GSFs (cf. Sec. 2.7.1). In this sections the basic definitions for (MT)BDDs are given following [13, 19].

Definition 31. *An ordered non-reduced Binary Decision Diagram (BDD) over an ordered set of variables $Vars$ (ordered by a fixed ordering relation $<\subset Vars \times Vars$) is a tuple $(NT, T, var, then, else, root)$ that consists of a finite set of non-terminal nodes (NT), a set of terminal nodes ($T = \{0, 1\}$), the variable labelling function $var : NT \cup T \to Vars$, two directed acyclic graphs ($NT \cup T$,then) and ($NT \cup T$,else) and the root node of the graph with the following properties:*

1. $\forall v_{NT} \in NT : |succ(v, then)| = |succ(v, else)| = 1$
2. *($NT \cup T$,then$\cup$else) is a rooted directed acyclic graph*
3. *For every path $p = (v_1, \ldots, v_n) \in Paths(NT, then \cup else)$ starting at the root node ($H(p) = root$) there must be an extension p^* of p with $p^* = (v_1, \ldots, v_n, v^*_{n+1}, \ldots, v^*_m)$, $m \geq n$ with $T(p^*)$ is a terminal node.*
4. $\exists c \in Vars\ \forall v_T \in T : var(v_T) = c$.
5. $\forall v_{NT} \in NT$, $v_T \in T$: $var(v_{NT}) < var(v_T)$.
6. *The paths have to be compatible with the variable ordering function, i.e. $\forall p \in Paths(NT \cup T, then \cup else)$, $p = (v_1, \ldots, v_n) : \forall i \in \{1, \ldots, n-1\} var(v_i) < var(v_{i+1})$.*

Remark 23. *An ordered set of variables $\{v_1, \ldots, v_n\}$, $v_1 < \ldots < v_n$ will often be written as a tuple $(v_1, \ldots, v_n)$.*

Definition 32. *An ordered reduced BDD is an ordered BDD $(NT, T, var, then, else, root)$ that fulfills the following additional properties:*

1. *then$\cap$else$= \emptyset$ (no* don't care nodes *exist).*
2. $\forall v_1, v_2 \in NT : (succ(v_1, then) \neq succ(v_2, then) \vee succ(v_1, else) \neq succ(v_2, else)$ *(no isomorphic nodes exist).*

In the sequel mainly ordered reduced BDDs are used, therefore the terms *ordered* and *reduced* are omitted. The only exception is the description of the hybrid multilevel algorithm (cf. Sec. 4.4). There reduction rule 1 is violated, as don't care nodes have to be inserted explicitly for the offset labelling.

In a similar way BDDs were defined, one can define Multi-Terminal Binary Decision Diagrams (MTBDDs). The only difference is that the set of terminal nodes is extended to $T \subset \mathbb{R}$, $|T| < \infty$. As T is a set, the terminal elements will be pairwise distinct. To shorten notation, the paths to terminal zero nodes are omitted in the graphs of BDDs/MTBDDs. An example of a MTBDD over the variable set (s_1, t_1, s_2, t_2) is given in Fig. 2.12c.

2.7.3. (MT)BDD operations

Let the given MTBDD rely on a fixed variable ordering. We define the following operations on BDDs/MTBDDs that we will use later in this work. Except the threshold function (defined e.g. in the CUDD [20] library) all operations are quite standard nowadays. They perform the same operations as the functions given for GSFs. By the canonical correspondence between GSFs and MTBDDs one sees that the following schematic diagram must commute:

$$\begin{array}{ccc} GSF & \xrightarrow{\simeq} & MTBDD \\ {\scriptstyle <OP>_{GSF}}\downarrow & & \downarrow{\scriptstyle <OP>_{MTBDD}} \\ GSF & \xrightarrow{\simeq} & MTBDD \end{array}$$

To ensure the reducedness of resulting MTBDDs in the algorithms, we assume that there exists a function `newNode` that creates a new node, if the same node is not present in the MTBDD so far and returns a reference to the node if it is already present. Such a function is provided by every MTBDD package, often realised by a so-called *unique table*.

- $\langle$MTBDD1$\rangle\langle$OP$\rangle\langle$MTBDD2$\rangle$:= APPLY($\langle$MTBDD1$\rangle$, $\langle$MTBDD2$\rangle$,$\langle$OP$\rangle$), the general apply operator. It is motivated by the observation that for $f, g \in GSF(n, M)$ $(f *^* g)(x) = f(x) * g(x)$ can be calculated pointwise. The operator can be described recursively by the code given in Alg. 1. Line 1 and 2 handle the terminal case. The apply operation of two terminal nodes can be calculated directly by connecting these nodes with the operator. In the second case (line 3 to 6) the variable orders of the two nodes do not fit, so the node with the smaller variable order has to be replaced by its child nodes (the dual case $var(node2) > var(node1)$ is treated in line 7-10). When the variable orderings are the same, a synchronous recursive descent is performed (line 11-14).

- $\langle$MTBDD$\rangle|_{\langle VAR\rangle=\langle VAL\rangle}$, returns $\langle$MTBDD$\rangle$ with $\langle$VAR$\rangle$ set to $\langle$VAL$\rangle$ (Restriction).

- $ABSTRACT(\langle$MTBDD$\rangle, \langle$VAR$\rangle, \langle$OP$\rangle)$:= $\langle$MTBDD$\rangle|_{\langle VAR\rangle=0}\langle$OP$\rangle\langle$MTBDD$\rangle|_{\langle VAR\rangle=1}$. That means the abstraction operation removes a certain variable by relating the different assignments of this variable by operation $\langle$OP$\rangle$.

- $THRESHOLD(\langle$MTBDD$\rangle, \langle$VAL$\rangle)$ generates a 0-1-MTBDD with value 1 where the function represented by the MTBDD is $>\langle$VAL$\rangle$, value 0 elsewhere. The corresponding algorithm ist given in Alg. 2. The operation processes all terminal nodes and sets the new values according to the given threshold. The result of Alg. 2 might not be reduced. In order to get a reduced MTBDD, a reduction has to be performed.

- $ITE(\langle$MTBDD$_{01}\rangle, \langle$MTBDDT$\rangle, \langle$MTBDDE$\rangle)$, the general if-then-else operator. Parameter MTBDD$_{01}$ is a 0-1 MTBDD and whenever it is equal to 1, the value of MTBDDT is returned, the value of MTBDDE otherwise.

- $\langle$MTBDD$\rangle_{\langle VAR1\rangle\rightarrow\langle VAR2\rangle}$ returns a MTBDD where all VAR1 variables are mapped to variable set VAR2. It is required that $\langle$MTBDD$\rangle$ does not depend on $\langle VAR2\rangle$. and that both variable sets have the same size, i.e. $|VAR1| = |VAR2|$ (Renaming).

Abstraction and restriction of more than one variable is defined in the canonical recursive way.

For the general APPLY operation, the following operators are used in this work:

- $\langle$ MTBDD1$\rangle$ == $\langle$ MTBDD2$\rangle$ returns 1 whenever MTBDD1 and MTBDD2 coincide, 0 otherwise.

- $\langle$ MTBDD1$\rangle$ > $\langle$ MTBDD2$\rangle$ returns 1 whenever MTBDD1>MTBDD2, 0 otherwise.

Algorithm 1 APPLY(node1, node2, op)

```
1: if (node1 ∈ T AND node2 ∈ T) then
2:    return newNode(node1 ⟨op⟩ node2)
3: else if var(node1)>var(node2) then
4:    e_node = APPLY(node1, node2 → else)
5:    t_node = APPLY(node1, node2 → then)
6:    return newNode(e_node, t_node)
7: else if var(node1)<var(node2) then
8:    e_node = APPLY(node1→ else, node2 )
9:    t_node = APPLY(node1→ then, node2 )
10:   return newNode(e_node, t_node)
11: else if var(node1)=var(node2) then
12:   e_node = APPLY(node1→ else, node2 → else)
13:   t_node = APPLY(node1→ then, node2 → then)
14:   return newNode(e_node, t_node)
15: end if
```

Algorithm 2 THRESHOLD(MTBDD, VAL)

```
1: for all node ∈ T do
2:    if value(node) > VAL then
3:       value(node) = 1
4:    else
5:       value(node) = 0
6:    end if
7: end for
```

- ⟨ MTBDD1⟩ + ⟨ MTBDD2⟩ returns MTBDD1+MTBDD2

Remark 24. *In some algorithms we will use a depth first search (DFS) traversal. This algorithm is shown in Alg. 3. In line 1 the recursion bottoms out if a terminal node is found. For non-terminal nodes, the else-successor is processed prior to the then-successor (line 4 and 5). The DFS traversal induces an ordering on the MTBDD nodes (visit number).*

Algorithm 3 DFS(node)

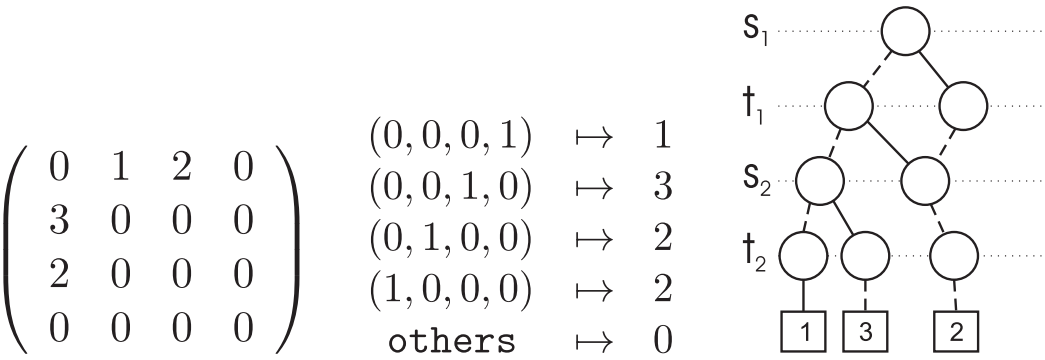
```
1: if node ∈ T then
2:    return
3: else
4:    DFS(node → else)
5:    DFS(node → then)
6: end if
```

2.8. Compact encodings by (MT)BDDs

This section shows how objects such as sets, matrices, labelled transition systems, etc. can be encoded by symbolic data structures.

2.8.1. Encoding matrices as MTBDDs

The description of the algorithms presented in this work is built upon a standard MTBDD representation of a labelled transition system. As this representation is commonly accepted heuristics, the algorithms can be used for every tool that codes finite transition matrices in the same way. A real valued quadratic matrix $M = (a_{ij})_{i,j=1...k}$ of dimension $k \times k$, $k \in \mathbb{N}$, can be



(a) example matrix (b) bit strings (c) MTBDD representation

Figure 2.12.: MTBDD representation of a matrix

$$\begin{pmatrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \\ 9 & 10 & 13 & 14 \\ 11 & 12 & 15 & 16 \end{pmatrix}$$

Figure 2.13.: Ordering of matrix elements induced by DFS traversal

seen as a function $f : \{1, \ldots, k\}^2 \to \mathbb{R}; (s,t) \mapsto a_{st}$ (cf. 2.7.1.1). The function f maps a row index s and column index t to the corresponding element a_{st} of the matrix. The mapping to a MTBDD representation works as follows. Firstly, the sets $\mathbb{N}_{1\ldots k}$ of row- and column indices have to be encoded by boolean variables ($m := \lceil log_2 k \rceil$ variables for each index). To use as few bits as possible, row- and column-index are decremented by 1 before encoding them, i.e. $\underbrace{(0,0,\ldots,0)}_{m-times}$ corresponds to index 1. The corresponding variables for the indices will be called `s` and `t` variables. Secondly, the variables are ordered from the most significant bit to the least significant bit (i.e. $s_1, \ldots, s_m$ and $t_1, \ldots, t_m$) and bit-strings are generated in an interleaved way $(s_1, t_1, s_2, t_2, ..., s_m, t_m)$. In this way, every combination of row and column indices can be encoded as a set of MTBDD variables and the corresponding function value is encoded as the value of a terminal node. An encoding of a small matrix is shown in Fig. 2.12. The matrix to be encoded is shown in Fig. 2.12a. With the notation introduced before one has $k = 4$, $m = 2$. The corresponding bitstrings (s_1, t_1, s_2, t_2) are given in Fig. 2.12b and the MTBDD representation is given in Fig. 2.12c. As an example, the leftmost path represents the matrix element $a_{12} = 1$.

Remark 25. *With the chosen variable ordering, by a Depth First Traversal (DFS) an ordering of the matrix elements is induced. Assuming a full matrix of dimension* 2×2*, the index of the element induced by a DFS is given as the matrix entries in Fig. 2.13. So a recursive block structuring is induced on the matrix. This will be exploited in Sec. 4.4.*

2.8.2. MTBDD representations of transition systems

An LTS $(S, L, \to, s)$ can be interpreted as a set of $|L|$ matrices in the following way: Fix a bijection $\mathbb{N}_{1\ldots|S|} \to S$ to identify every state with a number. For every label $a \in Act$ one defines a matrix $M^a := (m^a_{ij})_{i,j=1\ldots|S|}$ by

$$m^a_{ij} = \begin{cases} 1 & \text{if } i \xrightarrow{a} j \\ 0, & \text{otherwise} \end{cases}$$

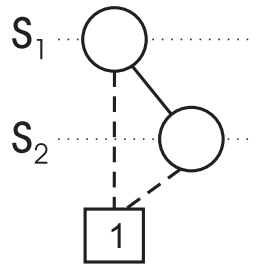


Figure 2.14.: 0-1-MTBDD representation of reachable states

The LTS $(S, L, \rightarrow, s)$ can be identified by the set of matrices $\{M^a | a \in Act\}$ and a number in $\mathbb{N}_{1...|S|}$ that represents the initial state. Of course, every matrix M^a can be encoded as an MTBDD with the method given in Sec. 2.8.1. It remains to encode the set of matrices as one single MTBDD. This is done by introducing another fixed bijection $\mathbb{N}_{1...|Act|} \rightarrow Act$, such that similar to the encoding of a row and column index in Sec. 2.8.1 the actions are encoded by a tuple of $n_a := \lceil log_2 |Act| \rceil$ action bits. The action bits are sorted from the most to the least significant bit $(a_1, \ldots, a_{n_a})$. In the resulting MTBDD the action variables are located on top of the state variables, so a variable ordering $(a_1, \ldots, a_n, s_1, t_1, s_2, t_2, ..., s_m, t_m)$ is induced. Let the tuple of action bits corresponding to action a be denoted by $\tilde{a}$. Now the representation of the LTS is calculated by the following APPLY operations:

$$M := \sum_{a \in Act} \tilde{a} \cdot M^a.$$

For the encoding of a SLTS the approach is quite similar. The only difference is that M^a is not only 0-1 valued, but the matrix entries are the rates of the corresponding transitions. An WSLTS can be encoded either by two MTBDDs (one for the immediate part, one for the timed part) or by one MTBDD (distinguishing immediate and timed transitions by their action names).

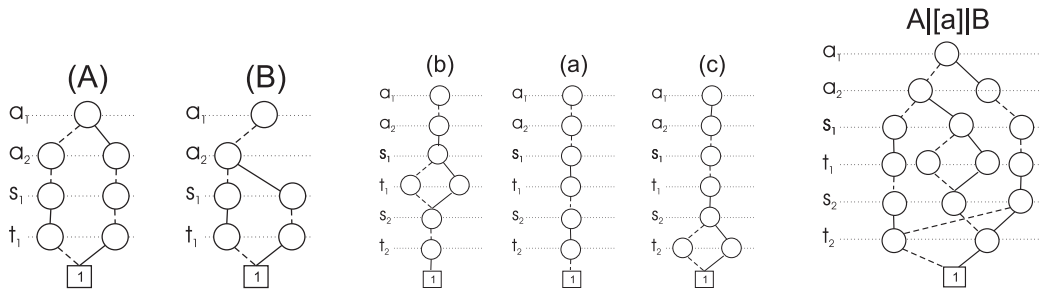
2.8.3. Reachable states and state probabilities

Current symbolic modelling tools encode the state space as BDDs (or 0-1-MTBDDs). The variable set corresponds to the variable set used for encoding the row/column variables in a matrix (cf. 2.8.1). A reachable state is encoded by a path in the BDD leading to the terminal 1 node, while unreachable states correspond to paths leading to the terminal 0 node. The set of reachable states in the matrix given in Fig. 2.12 (assuming one of the states with index $\{1, 2, 3\}$ as initial state) can be determined by symbolic reachability analysis [47]. The reachable indices $\{1, 2, 3\}$ are decremented before encoding them as MTBDD paths (in practice one would inherit the decremented indices from the matrix when performing symbolic reachability analysis). The MTBDD M, encoding the reachable states, can be seen in Fig. 2.14. Three bit-strings $\{00, 01, 10\}$ lead to reachable states. For the first two bit-strings there is one don't care node that has been removed in the figure. In the same way a vector of state probabilities can be represented by allowing only MTBDDs that represent a probability distribution, i.e. all terminal nodes are greater or equal to zero and ABSTRACT(M, $\{\mathtt{s}_1, \mathtt{s}_2\}$, +)=1.

2.8.4. Compositional modelling

A basic concept for process algebra is the parallel composition. With this operation large systems can be built out of small subsystems (cf. Sec. 2.4.7). By means of the following tiny example the basic concept of parallel composition by symbolic operations is sketched, for details we refer to [58]. Suppose there are given two sequential processes:

```
A:=(c,1);(a,1);A
```



(a) MTBDD for A and B (b) Common set of variables (c) MTBDD for $A|[a]|B$

Figure 2.15.: Example aggregation procedure - continued

```
B:=(b,1);(a,1);B
```

Assume the encodings of actions $a \simeq 00$, $b \simeq 01$, $c \simeq 10$ are fixed. The initial state of each process is encoded by 0, the second state of each process by 1. According to Sec. 2.8.2, the processes have the MTBDD representations given in Fig. 2.15a. Now, as the states of processes A and B have to be distinct, a new pair of variables ($\mathtt{s}_2$, $\mathtt{t}_2$) is introduced and the state variables of B are renamed: $B := B_{(\mathtt{s}_1,\mathtt{t}_1)\to(\mathtt{s}_2,\mathtt{t}_2)}$.

From this the MTBDD, representing the parallel composition $A|[a]|B$, can be deduced: The synchronous parallel composition consists of three parts:

1. A performs a non-a step, B is idle
2. B performs a non-a step, A is idle
3. both A and B perform an a-step

The three situations are shown in Fig. 2.15b. When a component is inactive in an unsynchronised step, it remains in its current state. This is encoded by the diamond structures for the b and c actions. A diamond encodes the identity matrix (here $0 \mapsto 0$, $1 \mapsto 1$ for pairs ($\mathtt{s}_1$, $\mathtt{t}_1$) and ($\mathtt{s}_2$, $\mathtt{t}_2$)). From this, the MTBDD representing $A|[a]|B$ can be achieved by adding up all branches and reducing the result according to Def. 32 (cf. Fig. 2.15c).

We would like to stress that the parallel composition of two processes can be calculated by using only MTBDD operations. Therefore, MTBDD-based process algebra tools are capable of directly encoding the transition systems specified by the input language into a MTBDD representation without having to generate transition matrices explicitly. It is notable that the MTBDD size of the product only grows linear in the sizes of the submodels while the explicit matrices for the transition systems grow multiplicatively [58].

2.8.5. Maximal progress and calculating probabilities from weights

After the MTBDD representing the whole WSLTS has been built from the input language, the maximal progress assumption is applied, i.e. all Markovian transitions that are in a race condition with at least one immediate transition are removed. The following lines show how this is done symbolically. Assume that MTBDD M^M (M^I) encodes the Markovian (immediate) transitions:

- Source states of immediate transitions are determined by $S^I = THRESHOLD(ABSTRACT(M^I, (\vec{a}, \vec{t}), +), 0)$, the resulting MTBDD S^I is a 0-1 MTBDD, i.e. it has only 0 or 1 terminal nodes.
- Markovian transitions concurring with immediate transitions are removed by $M^M = M^M \cdot (1 - S^I)$, where $(1 - S^I)$ denotes the complement of S^I. After the apply

operation only those Markovian transitions remain that have no source state in common with some immediate transition.

Another preprocessing step is to normalise the weights of immediate transitions to probabilities. Now, for every vanishing state, the weights of all outgoing immediate transitions are normalised to probabilities. This is achieved by the following symbolic operations:

- Abstract from the immediate action labels (ST indicates that this MTBDD only depends on `s` and `t` variables): $M^{I_{ST}} = ABSTRACT(M^I, (\vec{a}), +)$
- Sum up the exit weights for every state (S indicates that this MTBDD only depends on `s` variables): $M^{I_S} = ABSTRACT(M^{I_{ST}}, (\vec{t}), +)$
- Divide the outgoing weights by the sum of exit weights to get the new immediate transition system: $M^{I_{new}} = M^{I_{ST}} / M^{I_S}$

In the rest of the work we are interested in probabilities, so we use M^I as a short hand for $M^{I_{new}}$.

2.9. Numerical algorithms for steady state solutions

The steady state equation of a CTMC is given as $\pi \cdot Q = 0$, where Q is the infinitesimal generator matrix of dimension $n \times n$ (cf. [59]). So, after transposing, an equation system $Ax = 0$ has to be solved.

2.9.1. Splitting methods

Standard solution methods are the so-called splitting methods where $A = (a_{ij})_{i,j=1,\ldots,n}$ is split into $A = B + (A - B)$, such that B is a matrix that can be easily inverted. An iteration method $x_{m+1} = Mx_m$ can be defined therefrom with $M := B^{-1}(B - A)$. Let A=L+D+R with

$$D = (d_{ij})_{i,j=1,\ldots,n}, \quad \texttt{where } d_{ij} := \begin{cases} a_{ij} & \text{if } i = j \\ 0, & \text{otherwise} \end{cases}$$

$$L = (l_{ij})_{i,j=1,\ldots,n}, \quad \texttt{where } l_{ij} := \begin{cases} a_{ij} & \text{if } i > j \\ 0, & \text{otherwise} \end{cases}$$

$$R = (r_{ij})_{i,j=1,\ldots,n}, \quad \texttt{where } r_{ij} := \begin{cases} a_{ij} & \text{if } i < j \\ 0, & \text{otherwise} \end{cases}$$

i.e. A is split into strictly lower triangular, diagonal and strictly upper triangular part. Two common splitting methods are:

Jacobi method B:=D, the matrix of the diagonal elements of A, which can be easily inverted by inverting the diagonal elements.

Gauss-Seidel method B:=D+L. Looking at the iteration equation $(D + L)x_{m+1} = -Rx_m$ an easy solution can be achieved by forward substitution ($D+L$ is a lower triangular matrix), therefrom the elementwise iteration scheme can be deduced.

2.9.2. Relaxation

For an iterative solution method that produces a new approximation x_{m+1} starting from a given approximation x_m, linear relaxations with parameter $\omega \in \mathbb{R}^+$ are defined as follows:

$$x'_{m+1} := \omega x_{m+1} + (1 - \omega) x_m$$

The initial value for the next step of the iterative solution method is set to $x_{m+1} := x'_{m+1}$. Depending on the relaxation parameter one speaks of an *overrelaxation* ($\omega > 1$) or an *underrelaxation* ($\omega < 1$). In the PRISM tool (cf. [48]) all relaxation methods are called overrelaxation. We use PRISM's standard relaxation parameter $\omega = 0.9$ and also call it overrelaxation due to the PRISM terminology. The relaxed version of Gauss-Seidel only converges for $0 < \omega < 2$ [62].

2.9.3. Pseudo Gauss-Seidel

This method is a compromise between the methods of Jacobi and Gauss-Seidel and lends itself well to an MTBDD-based (or hybrid symbolic-explicit) implementation, as first described in [47]. The generator matrix is divided into blocks and in every block the Jacobi algorithm is used, while on the block level Gauss-Seidel iterations are performed. In the sequel the overrelaxed variant of the algorithm (again with $\omega = 0.9$) will also be called Pseudo Gauss-Seidel, even if the correct PRISM terminology would be Pseudo Successive Overrelaxation.

2.10. Experimental setup

Two different computers were used for the experiments. They will be referenced in the sequel by *Xeon* and *Altix*.

Xeon Intel dual Xeon 3.06 GHz machine with 2 GB of main memory running SuSE Linux version 9.1 (i586).

Altix SGI Altix with 10 Itanium 2 processors (1.5 GHz cpu speed) running SuSE Linux Enterprise Server 10 (ia64). It has 20 GByte of *distributed shared memory*.

Most experiments (except those for the parallel speedup and symbolic elimination without pre-reachability) have been carried out on the Xeon machine.

3. Symbolic elimination algorithm

This chapter describes an algorithm for the elimination of vanishing states in a PSLTS that is stored by MTBDD data structure. The PSLTS assumption is no restriction, as after the precalculations in Sec. 2.8.5 the WSLTS defined by the CASPA language is transformed to a PSLTS. We would like to stress that all calculations are done on the *closed model*, i.e. no further parallel composition is possible.

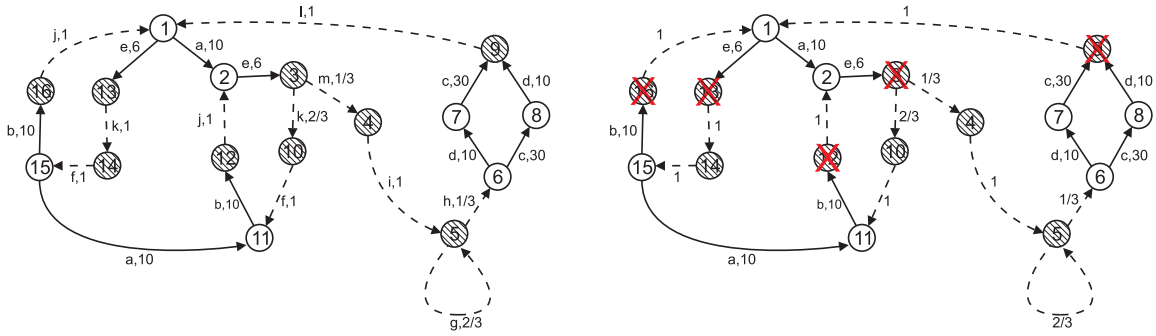
The elimination algorithm used in CASPA is a combination of an adaptation of the fully symbolic approach sketched in [58] and an adaptation of a semi-symbolic algorithm which was first presented in [25] leading to a two phase approach. The first phase, whose MTBDD version will be called fully symbolic algorithm, is a fast round-based method to eliminate per round a set of vanishing states that do not have vanishing predecessor states. The second phase, whose MTBDD version will be called semi-symbolic algorithm, eliminates the remaining vanishing states (i.e. loops or cycles and successors of those).

We require that the initial state is not vanishing, otherwise the model will not be regarded as valid. We start with an informal example, then set the theoretical basis for the elimination that justifies arbitrary orderings of the states to be eliminated. Next, we give a set-theoretic representation of the two phases for the elimination of all vanishing states, followed by the MTBDD versions and some experimental results. Finally we show an adaptation to the case where all immediate τ transitions can be eliminated (this is not always the case, c.f. the discussion on compositionally vanishing states in [58]).

3.1. Example

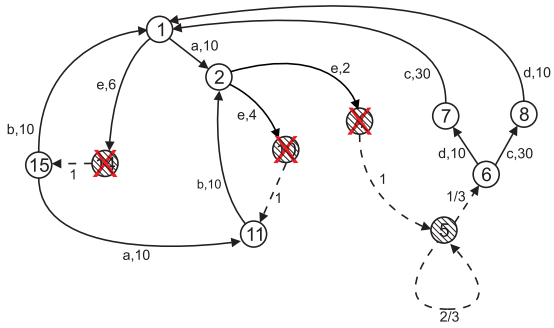
As an intuitive introduction, an example can be seen in Fig. 3.1. Vanishing states are drawn shaded, tangible states are drawn without shading. Markovian (immediate) transitions are solid (dashed) arrows. The initial PSLTS is given in Fig. 3.1a. The transition system after abstraction from immediate action labels and with marked states to be eliminated in round one of phase one (i.e. vanishing states without vanishing predecessor transitions can be eliminated) is given in Fig. 3.1b.

The result of this elimination (which scales the incoming rates into vanishing states with the outgoing probabilities) is given in Fig. 3.1c (where the states to be eliminated in the second round of phase one are already marked). The first elimination changed the transition system such that states 4, 10 and 14 can now be eliminated as they do not have vanishing predecessors anymore. The result of the second elimination round is given in Fig. 3.1d. Here, phase one terminates as no more vanishing states without vanishing predecessors are present. In phase two, only state 5 has to be eliminated. This is done in two steps. Firstly, the self-loop is deleted by rescaling the outgoing probabilities by $\frac{1}{1-p_l}$, where p_l is the loop probability. This leads to the probability distribution function for the successor state of state 5. The rescaling is indicated in Fig. 3.1e, the elimination result is shown in Fig. 3.1f. We would like to stress, that if state 5 had no other outgoing transitions than the loop transition, this would be a *timeless trap* [6].

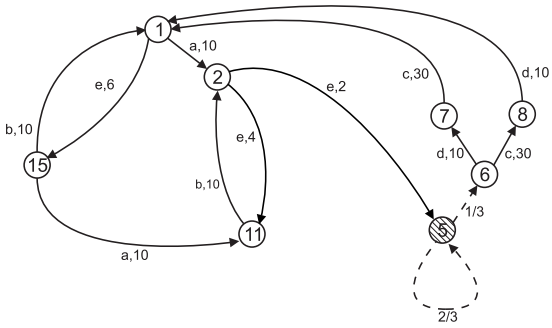


(a) Initial configuration

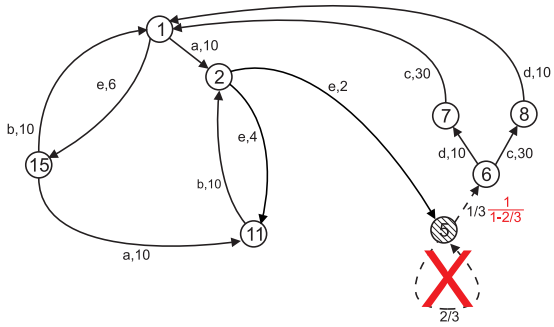
(b) Symb. elimination (1st round)



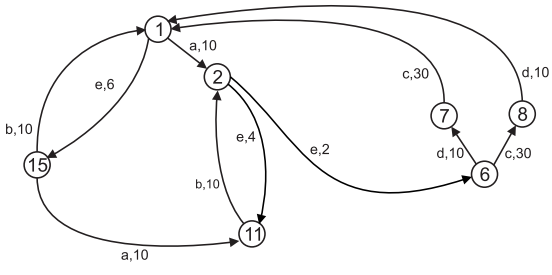
(c) Symb. elimination (2nd round)



(d) After symb. elimination



(e) Semi-symb. elimination



(f) After semi-sym. elimination

Figure 3.1.: Small elimination example

3.2. Foundations

3.2.1. Introduction

Suppose we are given two matrices M^M and M^I for the Markovian and immediate transitions that describe a PSLTS. For simplicity, let us assume that there is only one Markovian action and one immediate action. In the general case one has to work with one Markovian matrix for every Markovian action, while different immediate actions are collected into one immediate action. The elements of M^M (M^I) are the rates (probabilities) of the corresponding transitions. We assume that the resulting state space is deadlock-free (this can be checked in advance).

The set of transition splits the state space into two parts: States with outgoing immediate transitions are called vanishing states, otherwise they are called tangible. As vanishing states are immediately left as soon as they are entered, the probability of observing the corresponding process in a vanishing state is zero.

Similarly to the situation in GSPNs, vanishing states can be eliminated (a vanishing state is left immediately, so the probability of observing a vanishing state is equal to zero). If only tangible states remain, the steady state probability can be calculated from this form by standard methods.

Without loss of generality we leave the rates in the matrix, i.e. we do not look at the embedded Markov Chain as in [6]. Further we may assume that after reordering into vanishing (V) and tangible (T) states, the matrix M^M can be written as $M^M := \left(\begin{array}{c|c} 0 & 0 \\ \hline M^M_{TV} & M^M_{TT} \end{array}\right)$. In the same way one gets $M^I := \left(\begin{array}{c|c} M^I_{VV} & M^I_{VT} \\ \hline 0 & 0 \end{array}\right)$. In the following, we identify a state by its corresponding index in the state space given by M^M and M^I. The 2-tuple (M^M, M^I) defines a PSLTS $M := \left(\begin{array}{c|c} M^I_{VV} & M^I_{VT} \\ \hline M^M_{TV} & M^M_{TT} \end{array}\right)$. We will use (M^M, M^I) and M depending on the context. Denote all PSLTS with $|V|$ vanishing and $|T|$ tangible states by $PSLTS(|V|, |T|)$. Let e be the column vector of ones of appropriate size. According to [39] we give the following definition of a timeless trap:

Definition 33. *Let $p := M^I_{VT} \cdot e$ be the probability vector of leaving the set of vanishing states, and define the sets $J_0 := \{i | p_i = 0\}$ and $J_{>0} := \{i | p_i > 0\}$. States in J_0 will in the next step enter a vanishing state with probability 1, while states in $J_{>0}$ have a positive probability to enter a tangible state. We say that M^I* has no timeless trap *(or alternatively $M = (M^M, M^I)$ has no timeless trap), if for all states in J_0 there exists a path that ends up in a state of $J_{>0}$.*

As we only look at finite state spaces, it is known from [39] that M^I does not have a timeless trap if and only if $(I - M^I_{VV})^{-1} = \sum_{i=0}^{\infty} (M^I_{VV})^i$ exists.

In order to describe the elimination process, for a subset S of the state space, the following Matrix will be required:

$$I|_S := (a_{ij})_{ij} \texttt{ with } a_{ij} = \begin{cases} 1 & \texttt{if } i = j \texttt{ and } i \in S \\ 0 & \texttt{otherwise} \end{cases}$$

As usual, we use $\bar{S}$ to denote the complement of S. If the context is clear we will omit parentheses (e.g. we write $I|_x$ for $I|_{\{x\}}$). With this notation, $I|_S \cdot M$ are the transitions that emanate from states in S, $M \cdot I|_S$ are transitions that lead to states in S. It holds that $(I|_S)^n = I|_S$ for all $n \in \mathbb{N}$, $n \geq 1$. The elimination of a vanishing state can be defined as follows:

Definition 34. *Let $M := \left(\begin{array}{c|c} M^I_{VV} & M^I_{VT} \\ \hline M^M_{TV} & M^M_{TT} \end{array}\right) \in PSLTS(n, m)$ be a PSLTS. Assume that $M^M = (m^M_{ij})$, $M^I = (m^I_{ij})$ and that M^I_{VV} does not have a* timeless loop *with loop-probability 1 at v,*

i.e. $m^I_{vv} \neq 1$. *Then the elimination of a vanishing state* $v \in V$ *is a mapping*

$$\begin{array}{rccc} \hat{v}: & PSLTS(|V|,|T|) & \to & PSLTS(|V|,|T|) \\ & M & \mapsto & M^{\hat{v}} \end{array}$$

where $M^{\hat{v}} = (m^{\hat{v}}_{ij})$ *with*

$$m^{\hat{v}}_{ij} := \begin{cases} m^M_{iv} \cdot m^I_{vj} \cdot \frac{1}{1-m^I_{vv}} + m^M_{ij} & i \in T,\ j \neq v \\ m^I_{iv} \cdot m^I_{vj} \cdot \frac{1}{1-m^I_{vv}} + m^I_{ij} & i \in V,\ i,j \neq v \\ 0 & \texttt{otherwise} \end{cases}$$

We also use $\hat{v}$ *as annotation for submatrices of* M *to indicate the corresponding part of* $M^{\hat{v}}$.

Remark 26. *Let* $M \in PSLTS(|V|,|T|)$. *After elimination of one state* $v \in V$ *it is clear, that the row and column corresponding to state* v *is equal to zero. Therefore* $M^{\hat{v}}$ *can be regarded in a natural way as an element of* $PSLTS(|V|-1,|T|)$. *If all states have been eliminated, one has*

$$M^{\hat{v}_1,\ldots,\hat{v}_{|V|}} = \left(\begin{array}{c|c} 0 & 0 \\ \hline 0 & M_{TT}^{M\ \hat{v}_1,\ldots,\hat{v}_{|V|}} \end{array}\right).$$

This can be regarded in a canonical way as a Markov chain with the rate matrix $M_{TT}^{M\ \hat{v}_1,\ldots,\hat{v}_{|V|}}$.

3.2.2. Prerequisites

Before presenting the algorithm we show a statement that it is not important in which order the states are eliminated as long as no timeless trap is detected. Even though there are many papers on elimination of vanishing states of GSPNs in the literature (e.g. [44, 39, 17, 43]), to our best knowledge, Lemma 5 and Lemma 6 have not been published so far.

Lemma 5. *If no immediate loop with loop-probability 1 is detected during the elimination process, the result of the elimination does not depend on the order of the eliminated states, i.e. in terms of Def. 34, the following diagram commutes:*

$$\begin{array}{ccc} PSLTS(|V|,|T|) & \xrightarrow{\hat{v}} & PSLTS(|V|,|T|) \\ \downarrow{\scriptstyle \hat{u}} & & \downarrow{\scriptstyle \hat{u}} \\ PSLTS(|V|,|T|) & \xrightarrow[\hat{v}]{} & PSLTS(|V|,|T|) \end{array}$$

Proof. We compute the matrix entries of the transition matrices after the elimination of two states and show that they are independent of the order of elimination. It suffices to look at two states, as the general result follows from that (the permutation group of n elements, S_n, is generated by transpositions). Suppose the matrices are given as $M^M = (m^M_{ij})$, $M^I = (m^I_{ij})$. Elimination of state v leads to the following matrix entries ($i,j \notin \{v\}$, all other entries are zero):

$$m^{M^{\hat{v}}}_{ij} = m^M_{iv} \cdot m^I_{vj} \cdot \frac{1}{1-m^I_{vv}} + m^M_{ij}$$

$$m^{I^{\hat{v}}}_{ij} = m^I_{iv} \cdot m^I_{vj} \cdot \frac{1}{1-m^I_{vv}} + m^I_{ij}$$

Now the elimination of state u leads to the following results ($i,j \notin \{v,u\}$, all other entries are

zero):

$$\begin{aligned} m_{ij}^{M^{\hat{v}\hat{u}}} = \quad & (m_{iv}^M \cdot m_{vu}^I \cdot \tfrac{1}{1-m_{vv}^I} + m_{iu}^M) \\ & \cdot (m_{uv}^I \cdot m_{vj}^I \cdot \tfrac{1}{1-m_{vv}^I} + m_{uj}^I) \\ & \cdot \tfrac{1}{1-(m_{uv}^I \cdot m_{vu}^I \cdot \frac{1}{1-m_{vv}^I} + m_{uu}^I)} \\ & + m_{iv}^M \cdot m_{vj}^I \cdot \tfrac{1}{1-m_{vv}^I} + m_{ij}^M \end{aligned} \tag{3.1}$$

Now let $c := \frac{1}{(1-m_{vv}^I)\cdot(1-m_{uu}^I)-m_{uv}^I \cdot m_{vu}^I}$. One gets after some calculations

$$m_{ij}^{M^{\hat{v}\hat{u}}} = c(m_{iv}^M \cdot m_{vu}^I \cdot m_{uj}^I + m_{iu}^M \cdot m_{uv}^I \cdot m_{vj}^I + (1-m_{vv}^I) m_{iu}^M \cdot m_{uj}^I + (1-m_{uu}^I) \cdot m_{iv}^M \cdot m_{vj}^I) + m_{ij}^M$$

which is clearly symmetric in v and u. From Eq. 3.1 one immediately sees that if in the first or second elimination a timeless loop occurs, the matrix entry $m_{ij}^{M^{\hat{v}\hat{u}}}$ is not defined (division by zero). The calculations for M^I are similar, so we do not carry them out here. □

Lemma 6. *If M^I does not have a timeless trap and v is a vanishing state, then it holds that:*

1. *v does not have a self-loop with loop-probability 1*
2. *$M^{I^{\hat{v}}}$ does not have a timeless trap.*

Proof. It is clear that v cannot have a self-loop probability of 1 for otherwise this would be a timeless trap. As for every vanishing state in M, there is a path to a tangible state, we can show that this also holds for the eliminated system $M^{\hat{v}}$. Suppose that for every $u \in V$, $u \neq v$, there is a path $(v_1, \ldots, v_n, t)$ in M to a tangible state. Two cases have to be considered:

1. $\forall\, i \in \{1, \ldots, n\} : v_i \neq v$ then the path is also a path in $M^{\hat{v}}$.
2. Otherwise, as v does not have a loop probability of 1, one can omit every occurrence of v in the path and use the corresponding transitions in $M^{\hat{v}}$. Thus we found a path in $M^{\hat{v}}$ leading to a tangible state.

Therefore it follows, that $M^{I^{\hat{v}}}$ does not have a timeless trap. □

Corollary 1. *If M^I does not have timeless traps, it will still not have timeless traps after any number of elimination steps.*

Lemma 7. *The following statements are equivalent:*

1. *M^I has a timeless trap*
2. *During the elimination there is one state that has a timeless loop with loop-probability 1*

Proof. Assume first that M^I has a timeless trap. According to Def. 33 we may assume that there is one state v_0 that has no path into a tangible state and there is a certain non-empty set S_{v_0} of vanishing states that are reachable from v_0 (no tangible state is reachable). We have to distinguish two cases:

1. v_0 is recurrent (i.e. the probability of returning to v_0 is 1). Suppose that all vanishing states that are reachable from v_0 are eliminated. Then the last eliminated state within this set must have a probability-one-self-loop (otherwise there would be a positive probability to some state that is not reachable from v_0, which is a contradiction).
2. v_0 is transient. Now there must be a recurrent state v_0' in S_{v_0}. Choose v_0' and proceed as in the recurrent case.

The other way around, we assume that there is no timeless trap in M^I. Pick an arbitrary vanishing state v and assume that for $v = v_1$ there is a path $(v_1, \ldots, v_n, t)$ leading to a tangible state. By Lemma 6 and Corollary 1 we can eliminate all vanishing states in this path without the occurrence of a probability-one-self-loop and that the remaining transitions $M^{I'}$ still have no timeless trap. By Lemma 5 it is clear that the result of this elimination does not depend on the ordering of eliminated states. Inductively one can proceed until all vanishing states have been eliminated.

□

From Lemma 5 and Lemma 7 we can conclude the Corollary that shows the validity of the algorithms presented in Sec. 3.4.2 and Sec. 3.4.3.

Corollary 2. *If a timeless loop with loop-probability 1 occurs during the elimination process of (M^M, M^I), then M^I has a timeless trap. Otherwise, M^I does not have a timeless trap and the result of the elimination algorithm does not depend on the order of the eliminated states.*

Remark 27. *This result also holds for eliminations of subsets of vanishing states. As long as no timeless loop occurs, the result is independent of the ordering of eliminated states.*

As due to Corollary 2 the elimination can be done in an arbitrary ordering of the eliminated states, we may divide the elimination into two phases. The first phase eliminates in every round all vanishing states without immediate predecessor (there, no timeless trap will be detected), the second phase eliminates the remaining states in a state-by-state manner. Note that the first phase cannot eliminate cycles and loops of immediate transitions (and successors of those).

3.3. Set-theoretic representation of the elimination phases

The following representation of the two phases is the basis for the MTBDD-based implementations in Sec. 3.4.2 and Sec. 3.4.3. Phase one eliminates subsets of vanishing states simultaneously, the remaining vanishing states (those in cycles and loops of immediate transitions and in successors of those) are eliminated by phase two. We will use the term M^M (M^I) to denote Markovian (immediate) transition matrices.

3.3.1. Phase one

Before an elimination can take place, the vanishing states that can currently be eliminated have to be determined for every round (as the vanishing states are gradually eliminated). The probability of leaving a vanishing state (towards a vanishing or a tangible state) is equal to $M^I \cdot e$ (e is the column vector of ones of appropriate size). All states with exit probability greater than 0 are called S_I^{exit}. We will denote this by $S_I^{exit} := Threshold(M^I \cdot e, 0)$. In the same way, one calculates the probabilities of entering a vanishing state (coming from a vanishing state) by $e^T \cdot M_{VV}^I$. As a tangible state cannot have an outgoing immediate transition, this is the same as $e^T \cdot M^I$ (padding the tangible part of the vector with zeros for the smaller matrix). All states with this probability larger than 0 are called S_I^{target}. The algorithm for eliminating in every round all vanishing states without immediate predecessors is given in Algorithm 4. Line 2-4 calculate the set of vanishing states that can be eliminated in the current round. In line 5 the termination criterion is checked and the main loop terminates in line 6. Line 8 (9) calculate the unchanged Markovian (immediate) transitions (as every Markovian transition emanates from a tangible state, $I|_{\overline{S_I^{el}}} \cdot M^M \cdot I|_{\overline{S_I^{el}}}$ can be reduced to $M^M \cdot I|_{\overline{S_I^{el}}}$).

The redirected Markovian transitions are calculated in line 10. Line 11 (12) calculates the new set of Markovian (immediate) transitions after the current elimination.

Algorithm 4 eliminateParallel(M^M, M^I)

1: **while** TRUE **do**
2: $S_I^{exit} := Threshold(M^I \cdot e, 0)$
3: $S_I^{target} := Threshold(e^T \cdot M_{VV}^I, 0)$
4: $S_I^{el} = S_I^{exit} \cap \overline{S_I^{target}}$
5: **if** $S_I^{el} == \emptyset$ **then**
6: **break**
7: **end if**
8: $M^{M_{left}} = M^M \cdot I|_{\overline{S_I^{el}}}$
9: $M^{I_{left}} = I|_{\overline{S_I^{el}}} \cdot M^I \cdot I|_{\overline{S_I^{el}}}$
10: $M^{M_{redir}} = M^M \cdot I_{S_I^{el}} \cdot M^I$
11: $M^M = M^{M_{redir}} + M^{M_{left}}$
12: $M^I = M^{I_{left}}$
13: **end while**

3.3.2. Phase two

The elimination algorithm for single states is given in Algorithm 5. It is able to eliminate all vanishing states if and only if there are no timeless traps. If the algorithm cannot eliminate all vanishing states that should be eliminated, it is clear that there are timeless traps in the PSLTS. In line 1 the set of vanishing states that still have outgoing immediate transitions is determined. Now the main loop proceeds as long as there are states to be eliminated and no timeless trap has been detected. Line 3 chooses a vanishing state s and removes it from S_I^{el}. In line 4 the matrix with the only non-zero element m_{ss}^M is computed in order to get the probability of a loop in line 5. The transitions emanating from state s (without the loop) are calculated in line 6 (they will be rescaled in line 11 and 12 unless a timeless trap is present). Line 7 detects timeless traps. The remaining statements calculate the new matrices M^M and M^I similar to the redirections in Algorithm 4. Redirected immediate transitions are those (without the loop transition) that lead to state s and then proceed with the redirections given by $M_{current}^I$. This leads to $M_{redir}^I = (M^I - loopmatrix) \cdot I|_s \cdot M_{current}^I$, which can be reduced using the definition of $M_{current}^I$ and $I|_s^2 = I|_s$ leading to the expression in line 12.

3.4. Elimination of all vanishing states by MTBDD operations

We will now give the MTBDD representations of the algorithms defined in Sec. 3.3. In the following we assume the encoding of Markovian and vanishing states as in Sec.2.8.2, i.e. a set of Markovian (immediate) transitions is interpreted as a function $f : \mathbb{N}_{1\ldots k}^3 \to \mathbb{R}_{\geq 0}; (a, s, t) \mapsto \lambda$ ($g : \mathbb{N}_{1\ldots k}^3 \to [0, 1]; (a, s, t) \mapsto p$). The convention is used that whenever a tuple is mapped to 0, no transition exists. As we only deal with finite state spaces, both functions have *finite support*, i.e. only a finite number of tuples (a, s, t) are mapped to non-zero values.

The following variables will be used for the description of the algorithm: $\vec{a}$ (action labels), $\vec{s}$ (source states), $\vec{t}$ (target states), $\vec{u}$ (temporary states). The variable ordering in the MTBDD is $a_1 \prec \ldots \prec a_n \prec s_1 \prec t_1 \prec u_1 \ldots \prec s_m \prec t_m \prec u_m$ according to commonly accepted heuristics [58]. The (MT)BDD operations used are described in Sec. 2.7.3.

The next subsections describe the steps to eliminate all the vanishing states in the given PSLTS. In the following, let M^M be the MTBDD encoding the Markovian transitions and M_{st}^I the MTBDD encoding the vanishing states. We assume in the sequel, that the precalculations from Sec. 2.8.5 have already been applied (i.e. maximal progress assumption, probability calculations and abstraction of immediate action labels).

Algorithm 5 eliminateStateByState(M^M, M^I)

1: $S_I^{exit} := Threshold(M^I \cdot e, 0)$
2: **while** $S_I^{exit} \neq \emptyset$ **do**
3: remove s from S_I^{exit}
4: $loopmatrix = I|_s \cdot M^I \cdot I|_s$
5: $loopprob = e^T \cdot loopmatrix \cdot e$
6: $M^I_{current} = I|_s \cdot (M^I - loopmatrix)$
7: **if** $M^I_{current} == 0$ **then**
8: trap detected
9: **break**
10: **end if**
11: $M^M_{redir} = M^M \cdot M^I_{current} \cdot \frac{1}{1-loopprob}$
12: $M^I_{redir} = (M^I - I|_s \cdot M^I) \cdot M^I_{current} \cdot \frac{1}{1-loopprob}$
13: $M^{M_{left}} = M^M \cdot I|_{\overline{s}}$
14: $M^{I_{left}} = I|_{\overline{s}} \cdot M^I \cdot I|_{\overline{s}}$
15: $M^M = M^{M_{redir}} + M^{M_{left}}$
16: $M^I = M^{I_{redir}} + M^{I_{left}}$
17: **end while**

3.4.1. Pre-reachability

In practice it turned out that for bigger models it is often faster to do some reachability analysis before starting the elimination, cf. the experiments in Sec. 3.4.4. For the so-called pre-reachability the entire transition system $M^I + M^M$ is taken as input for reachability analysis. The resulting set of reachable states S_{reach}, encoded as s-variables is used to reduce the set of transitions to be eliminated by $M^I := S_{reach} \cdot M^I$, $M^M := S_{reach} \cdot M^M$. Some results are given in Sec. 3.4.4. For very large potential state spaces it is usually beneficial to use pre-reachability, even if the MTBDD size grows as symmetries are broken.

3.4.2. Fully-symbolic elimination step

The algorithm is a round-based scheme where in every round all immediate transitions without vanishing predecessor transition are eliminated at once. The number of rounds is determined by the maximum length of sequences of vanishing states (without cycles), and *not* by the number of vanishing states. Such a round-based scheme is typical for symbolic algorithms and works very efficiently. The motivation for taking the elimination candidates as the states without incoming immediate transitions is that in this case the different eliminations do not interfere with each other and therefore can be performed in the same MTBDD operation - without having to synchronise. The elimination procedure is repeated successively until only immediate transitions with at least one vanishing predecessor remain.

The set of vanishing states that do not have vanishing predecessor states is calculated by Alg. 6: In line 1 the source states of vanishing states $Immediate^{exit}_{s01}$ are calculated from M^I_{st} by abstracting over the target variables. The threshold function with parameter 0 converts all nonzero terminal nodes to terminal node 1. The resulting 0-1-MTBDD has value 1 for states that have outgoing vanishing states and value 0 for tangible states. Line 2 calculates the target states of vanishing states. Abstraction of M^I_{st} over the $\vec{s}$ variables leads to the incoming probabilities for all target states. Again, the threshold function is used to distinguish between zero and nonzero probabilities. $Immediate^{target}_{t01}$ depends only on $\vec{t}$ variables. The MTBDD $Immediate^{exit}_{s01}$ that depends only on $\vec{s}$ variables is swapped to $\vec{t}$ variables in line 3, i.e. s_i is mapped to t_i for all possible i. Finally, in line 4 the vanishing states without vanishing predecessors are calculated.

Alg. 7 shows the elimination procedure as a loop from line 1 to 13. Call by reference is assumed,

Algorithm 6 getVanishingSet(M^I_{st})

1: $Immediate^{exit}_{s01}$=THRESHOLD(ABSTRACT(M^I_{st}, t, +), 0)
2: $Immediate^{target}_{t01}$=THRESHOLD(ABSTRACT(M^I_{st}, s, +), 0)
3: $Immediate^{exit}_{t01}$=$(Immediate^{exit}_{s01})_{s\to t}$
4: **return** $Immediate^{exit}_{t01} \cdot (1 - Immediate^{target}_{t01})$

Algorithm 7 eliminateFullySymbolic(M^I_{st},M^M)

1: **while** TRUE **do**
2: $VanishingSet_{t01}$=getVanishingSet(M^I_{st})
3: **if** $VanishingSet_{t01} == 0$ **then**
4: **break**
5: **end if**
6: $VanishingSet_{s01} = (VanishingSet_{t01})_{t\to s}$
7: $M^{M_{left}}$=$M^M \cdot (1 - VanishingSet_{t01})$
8: $M^{I_{left}}_{st}$=$M^I_{st} \cdot (1 - VanishingSet_{s01})$
9: $M^{I_{temp}}_{tu}$=$((M^I_{st})_{t\to u})_{s\to t}$
10: $M^{M_{redir}}_{st}$=$(ABSTRACT(VanishingSet_{t01} \cdot M^M \cdot M^{I_{temp}}_{tu}, t, +)_{u\to t}$
11: $M^M = M^{M_{redir}} + M^{M_{left}}$
12: $M^I_{st} = M^{I_{left}}_{st}$
13: **end while**

i.e. whenever the parameters M^I_{st},M^M are redefined, the references are changed. In line 2 the set of vanishing states without predecessors is calculated. Line 3 checks for the termination criterion. If no more states can be eliminated, the loop is terminated in line 4. Just a variable swapping from target to source states is performed in line 6. Lines 7 and 8 calculate the Markovian and vanishing states that are left unchanged in the current round. The most important steps are line 9 where the vanishing states are swapped to an alternative variable set and line 10 that performs all the redirections and rescalings for this round via the $\vec{t}$ variables. After abstraction of the $\vec{t}$ variables, the $\vec{u}$ variables are swapped back to $\vec{t}$ and the resulting MTBDD only depends on $\vec{s}$ and $\vec{t}$ variables. Finally, in line 11 the new set of Markovian transitions is calculated and in line 12 the vanishing states are reduced to the set that has not been eliminated. This procedure is repeated until $VanishingSet_{t01}$ is equal to zero.

3.4.3. Semi-symbolic elimination step

It remains to eliminate the loops and cycles of vanishing states and successors of those. The elimination has to be done in a semi-symbolic way as vanishing states within a cycle have to be eliminated one after another. If a vanishing state has an immediate self-loop, the loop first has to be resolved before the elimination is possible. Self-loops are removed by a geometric series argument, i.e. before exiting a state with a self-loop, it can be taken from zero times to infinitely many times. If the probability for the self-loop is p, then the probability for finally leaving the self-loop in the direction of an immediate transition with probability q can be written as $\sum_{i=0}^{\infty} p^i \cdot q = q \cdot \frac{1}{1-p}$. As the scaling factor $\frac{1}{1-p}$ is equal for all concurring non-loop outgoing immediate transitions, they are simply scaled up to a probability distribution. Note that in the case that there is no non-loop outgoing immediate transition, there must be a timeless trap (cf. Sec. 3.2.1).

The semi-symbolic algorithm uses an MTBDD containing the remaining vanishing states as a trigger. This MTBDD is calculated symbolically by $Vanishing_s = ABSTRACT(M^I_{st}, t, +)$. The initial call for the elimination algorithm is eliminateSemiSymbolic(0, $Vanishing_s$, 1, M^I_{st}, M^M), where 1 means a MTBDD representing the constant 1. Again, we assume call by reference,

i.e. M_{st}^I and M^M are modified by the subroutine. The eliminateSemiSymbolic subroutine is shown in Alg. 8. The parameter *bit* encodes a source state variable or constant level in the

Algorithm 8 eliminateSemiSymbolic(*bit*, $Trigger$, $Minterm_s$, M_{st}^I,M^M)

```
1: if Trigger is not constant node then
2:   eliminateSemiSymbolic(bit + 2, Trigger_{v_bit=0}, Minterm_s · v̄_bit, M_st^I,M^M)
3:   eliminateSemiSymbolic(bit + 2, Trigger_{v_bit=1}, Minterm_s · v_bit, M_st^I,M^M)
4: else
5:   if NOT value(Trigger) == 0 then
6:     if bit <NumStateVariables then
7:       eliminateSemiSymbolic(bit + 2, Trigger, Minterm_s · v̄_bit, M_st^I,M^M)
8:       eliminateSemiSymbolic(bit + 2, Trigger, Minterm_s · v_bit, M_st^I,M^M)
9:     else
10:      Minterm_t=(Minterm_s)_{s→t}
11:      M_st^{I,curr}=M_st^I · Minterm_s
12:      M_t^{I,curr} = ABSTRACT(M_st^{I,curr}, s, +)
13:      Loop_t = M_t^{I,curr} · Minterm_t
14:      M_t^{I,curr} = M_t^{I,curr}·(1-THRESHOLD(Loop_t, 0))
15:      Loop = ABSTRACT(Loop_t, t, +)
16:      if NOT value(Loop)==0 then
17:        M_t^{I,curr} = M_t^{I,curr} · 1/(1 − value(Loop))
18:      end if
19:      redirect(M^M, Minterm_t, M_t^{I,curr})
20:      redirect(M_st^I, Minterm_t, M_t^{I,curr})
21:    end if
22:  end if
23: end if
```

MTBDD, *Trigger* is the node in the trigger MTBDD that is currently processed and $Minterm_s$ is a MTBDD used to encode source states of vanishing states. In the following, the notation v_i means the MTBDD variable corresponding to the i-th bit of the state variables in the MTBDD, $\overline{v}_i$ denotes its negation. Line 1 checks whether *Trigger* is a non-constant node. As long as no constant node of *Trigger* is reached, the function eliminateSemiSymbolic is called recursively with the restrictions of *Trigger* and $Minterm_s$ to the possible assignments of v_{bit} in lines 2 and 3. When a non-zero constant node is reached, but not all state variable bits are already processed, these *don't care* levels are resolved recursively in lines 7 and 8. Once the source state of a certain immediate transition is encoded in $Minterm_s$ and the corresponding terminal node of the trigger MTBDD is not 0, the elimination of $Minterm_s$ takes place in lines 10-21. In line 10 $Minterm_s$ is encoded as target state. Line 11 calculates the vanishing states $M_t^{I,curr}$ emanating from $Minterm_t$. Immediate loops may occur during the elimination process. They are calculated in lines 12-13, eliminated from $M_t^{I,curr}$ in line 14. In line 14 one has to check whether $M_t^{I,curr}$ is equal to the zero MTBDD (then a timeless trap has occurred and the elimination fails). This has been omitted for the sake of a compact representation. Finally, $M_t^{I,curr}$ is rescaled to a probability distribution in lines 15-18. Lines 19 and 20 redirect the Markovian and immediate transition systems according to $M_t^{I,curr}$.

The redirect subroutine is given in Alg. 9. Line 1 determines the transitions in M that end at the currently processed state and therefore have to be redirected. Here the restriction sets all $\vec{t}$-variables in M to the assignment given by state $Minterm_t$. If there are transitions to change, line 3 removes the transitions leading to state $Minterm_t$ from M. Note that $M_{as/s}^{changed}$ does no longer depend on `t` variables and therefore has to be multiplied by $Minterm_t$: Line 4 adds the redirected transitions to M. Note that I depends only on `t` variables whereas $M_{as/s}^{changed}$ only

Algorithm 9 redirect(M, $Minterm_t$, I)

1: $M_{as/s}^{changed} = M|_{\vec{t}=Minterm_t}$
2: **if** $M_{as/s}^{changed}$!= 0 **then**
3: $M = M \cdot$(1-THRESHOLD($M_{as/s}^{changed} \cdot Minterm_t$, 0))
4: $M = M + M_{as/s}^{changed} \cdot I$
5: **end if**

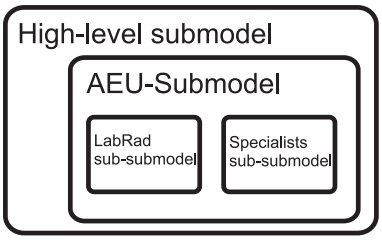


Figure 3.2.: Hierarchical structure of the AED model

depends on s (a and s) variables in the case of immediate (Markovian) transitions. Therefore the product of both again depends on s and t variables or a s and t variables.

3.4.4. Experimental results

In this section, we present some empirical results which we obtained analysing an Accident and Emergency Department (AED) of a large hospital. The model that we used is based on the model given in [2]. Its CASPA implementation is described in [4]. For the model, a hierarchical approach has been taken. Steady-state solutions of submodels are used to get the constants for higher-level models. The hierarchical structure is shown in Fig. 3.2.

Unless stated otherwise, all experiments in this section have been carried out on the Xeon machine. For the steady-state analysis of the high-level model and the submodels, the Pseudo Gauss-Seidel method (with relaxation parameter $\omega = 0.9$) was used. Table 3.1 shows the experimental results. Column s_{total} gives the size of the reachable state space including the vanishing states. Column s_{Markov} gives the size of the state space after the vanishing states have been eliminated. Column t_{gen} gives the time for generating the symbolic representation from the process algebraic description, and columns t_{reach} and t_{elim} show the times for reachability analysis and for the elimination of the vanishing states (where a "–" in column t_{elim} indicates that the model does not contain any vanishing states). The last two columns provide information about the number of iterations the Pseudo Gauss-Seidel algorithm needed to converge (n_{iter}) and the total solution time (t_{sol}). All timing information is given in seconds.

To show the different settings possible in CASPA, we give some alternative results for the elimination of the AEU submodel in Tab. 3.2. The *standard* row is the experiment as used above (only the elimination part), pre-reachability row is the variant using pre-reachability. Finally the *semi-symbolic only* row uses only the semisymbolic phase of the elimination algorithm (without the fully-symbolic phase, but with pre-reachability). For this model, pre-reachability does not

Model	s_{total}	s_{Markov}	t_{gen}	t_{reach}	t_{elim}	n_{iter}	t_{sol}
Specialists	14,641	14,641	0.02	0.02	–	340	0.64
LabRad	156,849	38,283	0.17	0.21	0.04	948	5.33
AEU	10,936,950	1,409,286	0.04	1.64	1.10	859	181.38
high-level	3,812,364	697,160	0.08	0.57	1.30	1013	101.90

Table 3.1.: Empirical results for the AED model (all times given in seconds)

Algorithm	pre-reachability s	elim. s	# el. rounds	reach. s	memory kB
standard	-	1.10	6	1.64	14,670.17
pre-reachability	6.50	0.52	5	1.64	28,404.00
semi-symbolic only	6.55	48992.63	-	1.80	58,439.94

Table 3.2.: Different elimination strategies for the AEU submodel

Algorithm	pre-reachability s	elim. s	# el. rounds	reach. s	memory kB
standard	-	(33,750.96)	31	(6.05)	(1,808,421.69)
pre-reachability	0.91	1.21	14	0.06	36,304.82
semi-symbolic only	0.90	77.28	-	0.06	48,674.82

Table 3.3.: Different elimination strategies for the powerline model

pay off - the time saved by the faster elimination step is annihilated by the time used for the prereachability step. Looking at the potential state space one sees that the AEU submodel has about 532.435.712 states (after maximal progress). The ratio $\frac{S_{pot}}{S_{reach}} \simeq 49$.

Looking at a different case, the picture changes. The powerline model (cf. Sec. 5.6) is an example where it pays off to use prereachability. All immediate transitions are eliminated by the three different elimination strategies as above. This model has $\sim 1.7 \cdot 10^{13}$ (16,909,910,750,560) potential states, while only $\sim 1.5 \cdot 10^4$ (15,159) states remain after prereachability, so $\frac{S_{pot}}{S_{reach}} \simeq 1.1 \cdot 10^9$. After elimination and reachability analysis, the model has only 1,136 states. The results are shown in Tab. 3.3. Here, in the *standard* case was cancelled after 15 minutes as the CASPA tool's memory grew towards 2 GB and the Xeon machine started swapping. The values in braces indicate the time taken for the calculations on the Altix machine, where the job terminated and the number of elimination rounds needed became apparent. Note that also the memory demand on the Altix machine cannot be compared directly to the Xeon machine due to 32/64 bit pointers (e.g. the pre-reachability case on the Altix machine used 55679.91 kB instead of 36304.82 kB on the Xeon machine). After this elimination, the resulting MTBDD still has $\sim 1.1 \cdot 10^{13}$ (10,809,732,344,482) potential states. Due to the long runtime and large memory demand the *standard* case has no practical relevance, as the eliminations using pre-reachability are by orders of magnitude faster. As there are relatively few states (15,159) that remain after pre-reachability, the semi-symbolic only calculations are only 63 times slower than the fully symbolic calculations.

3.5. Elimination of compositionally vanishing states by MTBDD operations

In the context of path-based analysis (cf. Ch. 5) it is useful to eliminate only a subset of the vanishing states. If in this case all vanishing states were eliminated, one would lose information about explicit on-demand-failure probabilities. In contrast, when some immediate actions are used for synchronisation purpose only, they are of no interest in a resulting path, so they can be hidden to internal immediate tau actions. As an example, the path $A \overset{fail\_A,0.001}{\dashrightarrow} B \overset{1,\tau}{\dashrightarrow} C \overset{fail\_C,0.001}{\dashrightarrow} D$ can be reduced to $A \overset{fail\_A,0.001}{\dashrightarrow} C \overset{fail\_C,0.001}{\dashrightarrow} D$. With respect to the number of reachable states it is better to eliminate vanishing states with outgoing immediate transitions instead of ignoring them in the path-based algorithms when reading the action labels.

In this section an algorithm is presented that safely can eliminate all vanishing states that have subsequent immediate tau transitions if they all are compositionally vanishing (cf. [58]). If there are vanishing states with outgoing tau transitions that are not compositionally vanishing,

no elimination is performed. The algorithm is used after the model has been built and before path-based analysis thus leading to smaller transition systems to be analysed by path-based analysis.

3.5.1. Applicability check & Algorithm

It is only allowed to remove an immediate tau transition from the PSLTS if it does not concur with another immediate non-tau transition. The corresponding algorithm simply checks for an empty section of the source states of vanishing states and the source states of immediate non-tau transitions. The decision can only be made if a reachability analysis is performed prior to the elimination step (it is very likely that in unreachable parts of the transition system such concurrent transitions do exist).

Assume that Act_M (Act_I) encodes all Markovian (immediate) actions and that $iTau$ encodes the immediate tau action (encoded as $\vec{a}$ variables). The basic idea is to treat non-tau immediate transitions in the same way as Markovian transitions, i.e. redirect them if necessary but to not eliminate the corresponding states (as they in this case are no longer vanishing). After a successful applicability check, immediate tau transitions can be eliminated safely, so the elimination algorithm from Sec. 3.4 can be applied. In greater detail, the algorithm for the

Algorithm 10 TauElimination(M^I, M^M)

1: $M_{st}^{I,\texttt{tau}}$ = ABSTRACT(ITE($iTau$, M^I, 0),a,+)
2: $M^{I,\texttt{noTau}}$ = ITE($iTau$, 0, M^I)
3: $M_s^{I,\texttt{tau}}$ = ABSTRACT($M_{st}^{I,\texttt{tau}}$, t,+)
4: $M_s^{I,\texttt{noTau}}$ = ABSTRACT($M^{I,\texttt{noTau}}$, a∪t,+)
5: intersect$_s$ = $M_s^{I,\texttt{tau}} \cdot M_s^{I,\texttt{noTau}}$
6: **if** intersect$_s$==0 **then**
7: redirect = $M^{I,\texttt{noTau}} + M^M$
8: eliminateFullySymbolic($M_{st}^{I,\texttt{tau}}$, redirect)
9: eliminateSemiSymbolic(0, $ABSTRACT(new, t, +)$, 1, $M_{st}^{I,\texttt{tau}}$, redirect)
10: M^M =redirect·Act_M
11: M^I =redirect·Act_I
12: **end if**

tau elimination is given in Alg. 10: As always, we assume M^M and M^I to be the MTBDDs of reachable transitions (as calculated in Sec. 3.4.1). Line 1-6 perform the applicability check: From the reachable vanishing states M^I, the tau and non-tau fractions are calculated in line 1 and 2. For the elimination, the immediate tau actions have to be independent of the action bits, therefore the abstraction is performed in line 1. The source states of tau and non-tau transitions are calculated in line 3 and 4 by abstraction over the $\vec{a}$ and $\vec{t}$ variables. The intersection between both sets of source states is calculated in line 5. Only if the intersection is empty (i.e. the zero-MTBDD), which is checked in line 6, all tau transitions can be safely eliminated. In the other case an error message can be generated. The actual tau-elimination works as follows: Line 7 calculates the transitions that should not be eliminated (i.e. redirected if necessary) and the actual fully symbolic elimination step is done in line 8 with the algorithm from Sec. 3.4.2, followed by the semi-symbolic algorithm from Sec. 3.4.3 in line 9. The semi-symbolic step only has to be performed as long as $M^{I,\texttt{tau}} \neq 0$. In both cases it is assumed that, as before, the functions do not have return values but directly change their arguments. After the elimination it remains to recover the Markovian and immediate fraction of the redirected transitions. This is done by a conjunction with the actions belonging to Markovian (immediate) transitions in line 10 (line 11).

Model	pot. states	constr. s	pre-reach. s	pre-reach. states	elim. s	# el. rounds	reach. s	reach. states
Powerline	$\approx 1.7 \cdot 10^{13}$	0.35	0.90	15,159	0.86	5	0.14	3,604
PMS	$\approx 8.2 \cdot 10^{9}$	0.02	0.24	605	0.13	8	0.01	117

Table 3.4.: Examples for tau-elimination

3.5.2. Experimental results

The following two examples show a class of models where the potential state space is extremely large in contrast to the reachable state space. The powerline model is described in Sec. 5.6, the phased mission system (PMS) is described in Sec. 5.7. Both models pass the applicability check, i.e. in the reachable state space (after maximal progress assumption, before elimination) there are no concurring tau and non-tau transitions. For both models the fully symbolic elimination step suffices, i.e. no cycles or loops of immediate tau transitions are present. Tab. 3.4 reads from left to right: Model name, number of potential states, time for construction of the MTBDD representing the model, time for reachability analysis after maximal progress assumption has been made, number of reachable states after maximal progress assumption, time for elimination of immediate tau transitions, number of symbolic elimination rounds taken for the elimination, time for reachability analysis after elimination, number of reachable states after elimination. Compared to the reachability analysis, the consumed time for the elimination is in the same order of magnitude as the first reachability analysis.

4. Symbolic multilevel algorithm

This chapter is devoted to the multilevel algorithm for the steady state solution of CTMCs. We start with the description of the multilevel method and give a short sketch of convergence properties of the algorithm. Next, the connection of MTBDD data structure and the multilevel algorithm is established. It follows the pure MTBDD approach, relying completely on the MTBDD data structure. A hybrid approach of the algorithm is developed, where the iteration vectors are stored as arrays of double values, the transition system is still stored by some (adapted) MTBDD structure. For further speedup, an orthogonal approach is introduced that substitutes blocks of MTBDD variables by sparse matrices. In order to show the applicability to multithreaded environments, a parallel version of the algorithm is proposed. Experimental results conclude this chapter.

4.1. Description and basic properties of the multilevel algorithm

4.1.1. The multilevel method

As an adaptation of multigrid techniques for the steady-state solution of Markov chains, the so-called multilevel algorithm has been presented in [31]. The idea of this algorithm is to successively reduce large Markov chains to smaller ones. From the solutions of the smaller Markov chains, the current iteration vector of the original Markov chain is corrected.

For a given CTMC $\mathcal{M}$, let R denote its transition rate matrix and Q its generator matrix, respectively. Let π denote the vector of steady-state probabilities of the reachable states of $\mathcal{M}$. A certain level, i.e. a horizontal dotted line in Fig. 4.1, of the multilevel solution scheme is denoted by an integer $l \in \{0, \ldots, N\}$. The original system of linear equations (also called the *fine system*) will be labelled as aggregation N, i.e. $Q^{(N)} := Q$, $\pi^{(N)} := \pi$. The system of linear equations obtained by the aggregation of the system of level l is denoted by level $l-1$.

For a given Markov chain of level l, the aggregated chain of level $l-1$ is calculated as follows: Let $\mathcal{S}$ be the finite state space of the chain at level l, and $\tilde{\pi}^{(l),pre}$ be an approximation of the solution-vector of the steady-state equation $\pi^{(l)} \cdot Q^{(l)} = 0$ with $\sum_i \pi_i^{(l)} = 1$. Let the partition $\{\mathcal{S}_i | i \in \{0, \ldots, M-1\}\}$ of $\mathcal{S}$ define the next aggregation step. Then the initial probability vector $\tilde{\pi}^{(l-1),pre}$ of the aggregated Markov chain is given by the partial sums $\tilde{\pi}_i^{(l-1),pre} = \sum_{j \in S_i} \tilde{\pi}_j^{(l),pre}$. Using $\tilde{\pi}^{(l),pre}$, the non-diagonal part of the $M \times M$ generator matrix of the aggregate is defined as follows:

$$Q_{ij}^{(l-1)} = \frac{\sum_{v \in S_i} \left(\tilde{\pi}_v^{(l),pre} \cdot \sum_{w \in S_j} q_{vw}^{(l)} \right)}{\sum_{v \in S_i} \tilde{\pi}_v^{(l),pre}} = \frac{\sum_{v \in S_i} \left(\tilde{\pi}_v^{(l),pre} \cdot \sum_{w \in S_j} q_{vw}^{(l)} \right)}{\tilde{\pi}_i^{(l-1),pre}}, \quad \texttt{for } i \neq j$$

That means that the non-diagonal entries of $Q^{(l-1)}$ consist of cumulative rates from single states into certain aggregates, weighted with the conditional probabilities of being in the corresponding source state. This can be seen as the analogon of the ideal aggregate for Discrete Time Markov Chains (cf. [51]). Note that Q^{l-1} in general depends on the current probability distribution in the Markov chain to be aggregated ($\tilde{\pi}^{(l),pre}$). The diagonal elements $Q_{ii}^{(l-1)}$ are given as negative row sums of the non-diagonal elements of $Q^{(l-1)}$.

After solving

$$\pi^{(l-1)} \cdot Q^{(l-1)} = 0 \quad \text{with} \quad \sum_i \pi_i^{(l-1)} = 1$$

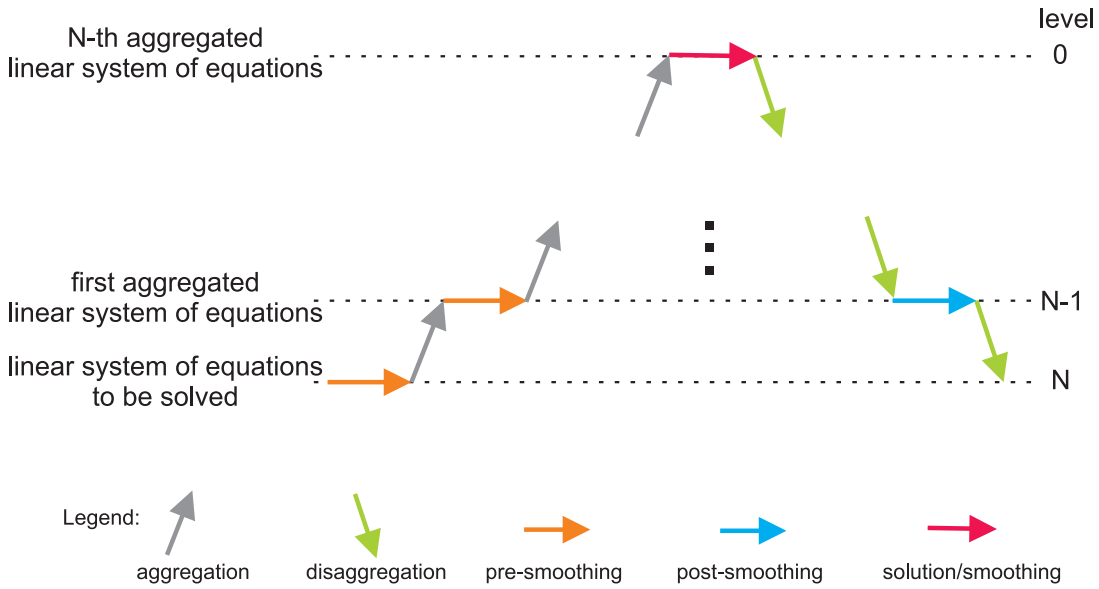


Figure 4.1.: Scheme of a multilevel V-cycle

by $\pi^{(l-1),post}$, the correction

$$\tilde{\pi}_j^{(l),post} \overset{j \in S_i}{=} \frac{\pi_i^{(l-1),post}}{\tilde{\pi}_i^{(l-1),pre}} \cdot \tilde{\pi}_j^{(l),pre} \tag{4.1}$$

is applied, i.e. all the states belonging to a certain aggregate are scaled by the same factor. Instead of directly solving level $(l-1)$ by $\pi^{(l-1),post}$, it is a canonical extension to approximate the solution of level $(l-1)$ by $\tilde{\pi}^{(l-1),post}$ through further recursive aggregation of level $(l-1)$. Then $\tilde{\pi}^{(l-1),post}$ is used instead of $\pi^{(l-1),post}$ in the disaggregation equation (4.1). The successive application of this approximation concept leads to the multilevel algorithm as indicated by a V-cycle in Fig. 4.1. On every level a certain number of *smoothing steps*, i.e. steps of an ordinary iterative solution algorithm, is performed.

In the sequel, for the smoothing steps Jacobi OverRelaxation steps with overrelaxation parameter $\omega = 0.9$ are used unless stated otherwise. The horizontal arrows in Fig. 4.1 are the smoothing steps (or alternatively a direct solution step for level 0). The arrows from level l up to $l-1$ are the aggregations while the arrows from level l down to $l+1$ are disaggregation steps.

4.1.2. Convergence results in the literature

There are several results on the convergence of the multilevel method. The first systematical treatment can be found in [41]. In this work, power iterations are used as smoothing algorithm for a two-level V-cycle. A comprehensive extension to different multilevel-cycles and smoothing algorithms can be found in [16], but the results therein are rather weak, as the matrices used for the calculations change from iteration to iteration. Basically there are two main concerns: Local and global convergence. In the following, let n be the number of variables in the linear system of equations to be solved.

local convergence means that there is some ϵ-environment $B_\epsilon(\pi) := \{x \in [0,1]^n : |x - \pi| < \epsilon\}$ around the exact solution of $\pi \cdot Q = 0$ where convergence can be shown. Usually the actual value of ϵ is not known.

global convergence means that for *every* initial state probability vector, convergence can be shown.

Of course, local convergence follows from global convergence. It seems to be an open question (at least no counterexample is known) whether the implication holds also in the other direction. For proving local convergence, [41, 49] consider the spectral radii of certain matrices. If they are lower than one, local convergence is achieved. Obtaining these matrices usually requires the inversion of a matrix of dimension n. In Sec. 4.1.3 we show an alternative approach based on functional analysis that only requires inversions of matrices of dimension less than n.

In order to prove global convergence, two approaches are common. Following [41], the number of smoothing steps (pre- and/or post-smoothing) has to be increased (Note that there are matrices known where increasing the number of smoothing steps is not beneficial [50]). In the limit, this approach turns a multilevel-cycle into an ordinary iteration, as the influence of the multilevel correction decreases. Another method has been presented in [32]. There, only one aggregation group with more than one element is allowed. We will show a third approach that—albeit for a very special case—can show global convergence using Möbius transformations.

The multilevel convergence considerations can also be seen in the wider range of *iteration of rational functions in dimension n*, where Fatou and Julia sets are considered. A survey on the one-dimensional theory is given in [7]. Not much is known for the n-dimensional case. This would help to get a deeper insight into the relation between local and global convergence of the multilevel algorithm.

4.1.3. Functional analysis approach

Here we give an intuitive and straight-forward approach to get a sufficient criterion for local convergence. It is basically of the same quality as the constant obtained in [41, 49], but for the calculation it suffices to invert smaller matrices than in those approaches.

Looking at the consecutive iterates, one V-cycle of the multilevel algorithm can be seen as a mapping $F : [0,1]^n \to [0,1]^n$ (even with the auxiliary condition $\sum_{i=1}^{n} x_i = 1$). By construction of the iteration scheme, one fixed point of this iteration scheme is the steady-state distribution [16].

Before stating a well-known theorem from nonlinear functional analysis, we give the following definition.

Definition 35. *Let $D \subseteq \mathbb{R}^n$. A mapping $F : D \to D$ is called a* contraction *if there exists an $0 < \alpha < 1$, such that for every $x, y \in D$ it holds that*

$$|F(x) - F(y)| \leq \alpha \cdot |x - y|.$$

A value $x \in D$ is called a fixed point *if $F(x) = x$. A fixed point x is called*

- attracting, *if there is an ϵ-environment $B_\epsilon(x)$ such that for all $y \in B_\epsilon(x)$ it holds that $\lim_{n\to\infty} F^n(y) = x$.*
- repelling, *if there exists an ϵ-environment $B_\epsilon(x)$ such that for all $y \in B_\epsilon(x)$ there exists $n_0(y) \in \mathbb{N}$ such that $F^{n_0(y)}(y) \notin B_\epsilon(x)$.*
- indifferent, *if there exists an ϵ-environment $B_\epsilon(x)$ such that for all $y \in B_\epsilon(x)$ it holds that $|\underbrace{F(x)}_{x} - F(y)| = |x - y|$.*

We try to look for contractions in order to apply the Banach fixed point theorem [52]:

Theorem 1. *For $D \subseteq \mathbb{R}^n$, $D \neq \emptyset$, D closed and a contraction $F : D \to D$ it holds that*

- *there is exactly one fixed point in D.*
- *for all initial values $x_0 \in D$ the sequence $F^n(x_0)$ converges to the fixed point.*

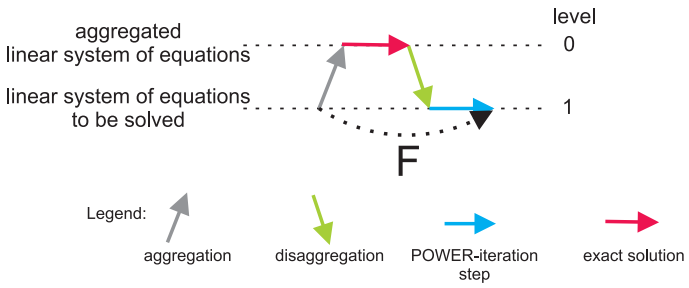


Figure 4.2.: Simple iteration scheme

In our case we need a considerably weaker form (that uses the same proof):

Corollary 3. *For $D \subseteq \mathbb{R}^n$, D open and a contraction $F : D \to D$ where the solution for $F(x) = x$ is in D it holds that*

- *the solution in D is unique.*
- *for all initial values $x_0 \in D$ the sequence $F^n(x_0)$ converges to the fixed point.*

We only need a convenient way to calculate the contraction constant α near the fixed point. An easy method for this is to use the Jacobi matrix. Using n-dimensional Taylor expansion one gets $|F(x) - F(y)| = |J(x)(x - y) + O(|x - y|^2)|$. Using triangle inequality one can conclude that $|F(x) - F(y)| \leq |J(x) \cdot (x - y)|$. Regarding $|J(x)|$ as a linear operator and taking the corresponding operator norm (cf. [1]), one gets $|F(x) - F(y)| \leq |J(x)| \cdot |(x - y)|$. So the operator norm of the Jacobi matrix can be used as an upper bound for the contraction constant α. In the following, we use the operator norm induced by the maximum norm on $\mathbb{R}$, which is the row sum norm $|M| = max_{i=1,\ldots,n} \sum_{j=1}^{n} |m_{ij}|$. As the functions F belonging to multilevel iterations do have poles it is clear that the norm of the Jacobi matrix is unbounded on the closed set of probability vectors, but it is clear for irreducible Markov chains that the solution will be a positive vector, i.e. all probabilities are greater than zero. So looking at the Jacobian is a convenient means to show *local* (in the sense of Corollary 3) but not *global* (in the sense of Theorem 1) convergence.

We will show a very special case, where global convergence can also be proven using Möbius transformations. Following [49, 40] we look at aggregations of the embedded Markov chain. We assume that B is a row stochastic matrix and it is aggregated with respect to a given partition. The aggregated linear system of equations is solved exactly and the disaggregated solution is smoothed by one power iteration step. The schematic iteration is given in Fig. 4.2. The mapping F covers an entire V-cycle.

In contrast to the functional analysis approach we also want to look at some properties of certain matrices. Therefore we define:

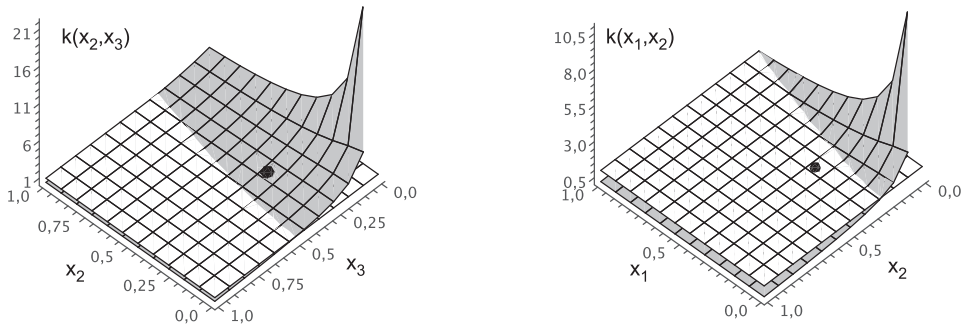
Definition 36. *Let M be a square matrix. If there exists a vector v and $\lambda \in \mathbb{R}$ such that the equation*

$$M \cdot v = \lambda \cdot v$$

holds, v is called an eigenvector *and λ is called the corresponding* eigenvalue. *The set of all eigenvalues of M is called the* spectrum *of M. The maximum of the absolute values of the elements in the spectrum is called* spectral radius *of M, denoted by $\rho(M)$.*

4.1.4. An example that strongly depends on the partition

In this section we show that the convergence of the multilevel method is not always guaranteed. This example can be found in [49], but there only the partitions $\{1\}, \{2, 3\}$ and $\{1, 2\}, \{3\}$ have



(a) Partition $\{1\}$, $\{2,3\}$ (b) Partition $\{1,2\}$, $\{3\}$

Figure 4.3.: Upper bound for the contraction constant

been treated.

$$B := \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 1 \\ \frac{1}{2} & \frac{1}{2} & 0 \end{pmatrix}$$

For the sake of completeness we present the missing partition $\{1,3\}$, $\{2\}$ that even converges within one step. Here are the results for the three different partitions:

1. The first partition $\{1\}$, $\{2,3\}$ leads to the following function:

$$\begin{array}{rccc} F: & [0,1]^3 & \to & [0,1]^3 \\ & (x_1,x_2,x_3) & \mapsto & (\frac{x_3}{x_2+2x_3}, \frac{x_3}{x_2+2x_3}, \frac{x_2}{x_2+2x_3}) \end{array}$$

The 1-Norm of the Jacobian is equal to

$$k(x_2,x_3) = \frac{2(x_2+x_3)}{(x_2+2x_3)^2}$$

The plot of this upper bound is given in Fig. 4.3a. The white plane marks the constant 1, the grey surface is the value of k. The fixed point $(\frac{1}{3},\frac{1}{3})$ is marked by a black dot. At the fixed point we have $k(\frac{1}{3},\frac{1}{3}) = \frac{4}{3}$, so it seems to be a *repelling* fixed point. As it is only an upper bound for the contraction constant, this does not mean that there is no convergence. To go on, we omit x_1 as it does not occur in the image. Further the image can be restricted to the last two components, as the first and the second component coincide. Looking at the restricted function:

$$\begin{array}{rccc} F_{23}: & [0,1]^2 & \to & [0,1]^2 \\ & (x_2,x_3) & \mapsto & (\frac{x_3}{x_2+2x_3}, \frac{x_2}{x_2+2x_3}) \end{array}$$

one has for $(n \in \mathbb{N}_0)$

$$F_{23}^{2n+1}(a,b) = (\frac{b}{a+2b}, \frac{a}{a+2b}) \tag{4.2}$$

$$F_{23}^{2(n+1)}(a,b) = (\frac{a}{b+2a}, \frac{b}{b+2a}) \tag{4.3}$$

$$\tag{4.4}$$

Setting $F^1_{23} = F^2_{23}$ leads to $a^2 = b^2$, so it is clear that F^n_{23} oscillates for initial values $a^2 \neq b^2$, and so does F^n for initial values that lead in the first step to (x_1, x_2, x_3) with $x_2^2 \neq x_3^2$.

Alternatively to calculating k, one could calculate the spectral radius ρ of J at the fixed point, i.e. $\rho(J(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})) = 1$ to see that this can be an indifferent fixed point. Usually calculating the spectral radius is more expensive than calculating k.

2. In contrast to this, the partition $\{1, 2\}$, $\{3\}$ leads to the following function

$$\begin{array}{rccc} F: & [0,1]^3 & \to & [0,1]^3 \\ & (x_1, x_2, x_3) & \mapsto & (\frac{1}{2}\frac{x_1+x_2}{x_1+2x_2}, \frac{1}{2}\frac{x_1+x_2}{x_1+2x_2}, \frac{x_2}{x_1+2x_2}) \end{array}$$

Looking again at the estimate for the contraction constant

$$k(x_1, x_2) = \frac{x_1 + x_2}{(x_1 + 2x_2)^2}$$

one sees in Fig. 4.3b that there is again a region where $k > 1$. Fortunately it is smaller than in the example before and $k(\frac{1}{3}, \frac{1}{3}) = \frac{2}{3} < 1$, so it is an attracting fixed point. Obviously it holds that after the first iteration step with F we have $x_1 = x_2$. The convergence after the second step can be verified directly. For all initial values, the result after the second step is $(1/3, 1/3, 1/3)$.

Alternatively, looking at $\rho(J(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})) = 0$ one also sees that this is a attracting fixed point. In dynamics, this is called a *superattracting* fixed point.

3. The third possible partition is $\{1, 3\}$, $\{2\}$. There the iteration function becomes constant:

$$\begin{array}{rccc} F: & [0,1]^3 & \to & [0,1]^3 \\ & (x_1, x_2, x_3) & \mapsto & (\frac{1}{3}, \frac{1}{3}, \frac{1}{3}) \end{array}$$

It is obvious that convergence is achieved after one step and the Jacobian is the zero matrix. Therefore $k = 0$ and the fixed point is superattracting. Of course, also $\rho(J(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})) = 0$.

4.1.5. Local convergence vs. global convergence

This example can be found in [42]. There local convergence is shown and it is proposed to use more than one power iteration in order to achieve convergence. Here we show that also only one power iteration guarantees global convergence.

The matrix B is given as

$$B := \begin{pmatrix} \frac{1}{12} & \frac{1}{12} & \frac{10}{12} \\ \frac{10}{12} & \frac{1}{12} & \frac{1}{12} \\ \frac{1}{12} & \frac{1}{12} & \frac{10}{12} \end{pmatrix}$$

And the partition used for the IAD algorithm is $\{1, 2\}$, $\{3\}$. The exact solution for the steady-state equation is $(\frac{7}{48}, \frac{1}{12}, \frac{37}{48}) \approx (0.146, 0.083, 0.771)$.

The iteration is given as:

$$\begin{array}{rccc} F: & [0,1]^3 & \to & [0,1]^3 \\ & (x_1, x_2, x_3) & \mapsto & (\frac{1}{12}\frac{4x_1+7x_2}{4x_1+x_2}, \frac{1}{12}, \frac{1}{3}\frac{10x_1+x_2}{4x_1+x_2}) \end{array}$$

Therefore the upper bound of the contraction constant is equal to

$$k(x_1, x_2) := \frac{2(x_1 + x_2)}{(4x_1 + x_2)^2} \tag{4.5}$$

which is plotted in Fig. 4.4

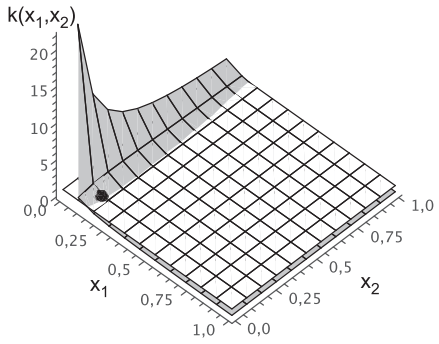


Figure 4.4.: Upper bound for the contraction constant

Unfortunately for the solution it holds that $k(\frac{7}{48}, \frac{1}{12}) = \frac{33}{32} > 1$, But this approximation is too coarse. As the spectral radius of the Jacobian at the fixed point is $\rho(J(\frac{7}{48}, \frac{1}{12}, \frac{37}{48})) = \frac{6}{25}$, at least local convergence can be expected.

4.1.5.1. Möbius transformation approach

Without loss of generality we can regard the second iteration step, where it is sure that $x_2 = \frac{1}{12}$. Then the iteration scheme becomes

$$\begin{array}{rccc} F: & [0,1]^3 & \to & [0,1]^3 \\ & (x_1, x_2, x_3) & \mapsto & (\frac{1}{12}\frac{48x_1+7}{48x_1+1}, \frac{1}{12}, \frac{1}{3}\frac{120x_1+1}{48x_1+1}) \end{array}$$

So we can focus on component one of the function (the other components are constant or functions of the first component) and therefore on the iteration

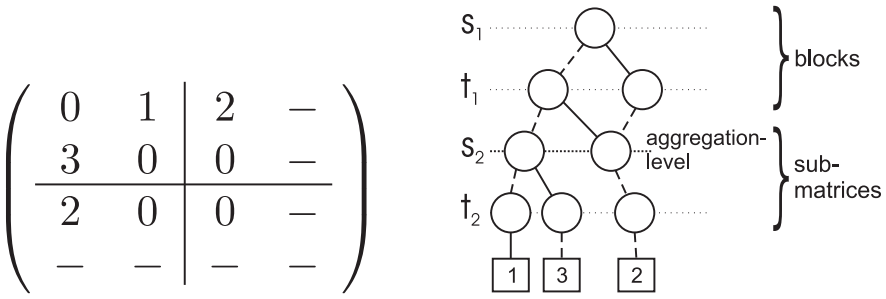
$$\begin{array}{rccc} f: & [0,1] & \to & [0,1] \\ & x & \mapsto & \frac{1}{12}\frac{48x+7}{48x+1} \end{array}$$

Obviously this is a Möbius-transformation $f : \hat{\mathbb{C}} \to \hat{\mathbb{C}}$, $x \mapsto \frac{ax+b}{cx+d}$ ($a = 48$, $b = 7$, $c = 576$, $d = 12$, $ad - bc \neq 0$, cf. [7]). Here we use $\hat{\mathbb{C}} = \mathbb{C} \cup \{\infty\}$ to denote the Riemann sphere. The transformation is loxodromic, i.e. it has two fixed points $\xi_1 = -\frac{1}{12}$, $\xi_2 = \frac{7}{48}$. We are looking for a nice conjugate representation of f, using another Möbius-transformation g such that gfg^{-1} in the following diagram becomes a multiplication.

$$\begin{array}{ccc} \hat{\mathbb{C}} & \xrightarrow{f} & \hat{\mathbb{C}} \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle g} \\ \hat{\mathbb{C}} & \xrightarrow[g\circ f\circ g^{-1}]{} & \hat{\mathbb{C}} \end{array}$$

According to [7], the transformation $g(x) = \frac{x-\xi_1}{x-\xi_2}$ maps ξ_1 to 0 and ξ_2 to ∞. Thus $gfg^{-1}(y) = -\frac{8}{3}y$, where $|-\frac{8}{3}| > 1$. So for $z \in \hat{\mathbb{C}} \setminus \{0\}$ it holds that $\lim gf^n g^{-1}(z) = \lim(gfg^{-1})^n(z) = \infty$. Looking at f^n this means that for every initial value $z \in \hat{\mathbb{C}} \setminus \{\xi_1\}$ the sequence $f^n z$ converges to ξ_2, so $\frac{7}{48}$ attracts all elements from $[0,1]$ and thus global convergence is guaranteed.

So in this case it is not necessary to increase the number of power method steps in order to achieve convergence as it was done in [42].



(a) Matrix (b) MTBDD

Figure 4.5.: Example Matrix and corresponding MTBDD

4.2. Multilevel in connection with MTBDDs

The Multilevel algorithm presented in the following is based on the MTBDD data structure. We assume that there is an MTBDD *trans* that stores the transition rate matrix and a BDD *reach* that stores the reachable states within the potential state space. The MTBDD *trans* is directly generated from the input language while the BDD *reach* is calculated by symbolic reachability analysis. Note that in the pure MTBDD setup (i.e. both iteration vectors and generator matrices are stored as MTBDDs) reachability analysis is not mandatory (in the iteration vectors all unreachable states are mapped to the same terminal node with zero value), but we use *reach* also in this case to explain the operations on the state space. The aim of the following sections is to introduce a variant of the Multilevel algorithm [31] for this data structure that does not change the basic shape of the (MT)BDDs and has only moderate additional memory demand.

4.2.1. Matrix aggregation in the context of MTBDDs

Given a state space $S = \{0, \ldots, (2^{(k_{agg}+k)} - 1)\}$ for finite $k, k_{agg} \in \mathbb{N}$ and subsets $S_i := \{2^{k_{agg}} \cdot i, 2^{k_{agg}} \cdot (i+1) - 1)\}$, $i \in \{0, \ldots, 2^k - 1\}$ defining a partition of the state space. Obviously it holds that $|S| = 2^{k_{agg}+k}$ and $|S_i| = 2^{k_{agg}}$. State spaces of arbitrary but fixed size not equal to a power of two can be embedded in bigger state spaces of size $2^{k_{agg}+k}$, with $k_{agg} + k$ sufficiently large. By the given partition $\{S_i\}_{i \in \{0,\ldots,2^k-1\}}$ every consecutive $2^{k_{agg}}$ neighbouring states are aggregated and the state space is reduced by a factor of $2^{k_{agg}}$. This means that for the transition matrix, row and column dimension are reduced by factor $2^{k_{agg}}$. In the context of Markov chains one can say that such an aggregation reduces the *potential* state space by factor $2^{k_{agg}}$. It depends on the structure of the Markov chain which reduction factor applies to the reachable state space. In the example in Fig. 4.5a the aggregation of two neighbouring states ($k = 1$, $k_{agg} = 1$) would correspond to the aggregation of the indicated 2×2 blocks. As the example matrix contains an unreachable state, the initial system of equations corresponds to the steady-state equations of a Markov chain with 3 reachable states, where the aggregated Markov chain has 2 reachable states. So the reduction factor for the potential state space is still 2 where the reachable state space is only reduced by factor $\frac{3}{2}$. As seen in this small example, an aggregation of k_{agg} MTBDD variables defined in this way can only group *up to* $2^{k_{agg}}$ reachable states (in the other extremal case a reachable state could be grouped with $2^{k_{agg}} - 1$ unreachable states, which does not reduce the state space at all). Looking at the MTBDD representation of the example matrix (cf. Sec. 2.8.1) which is given in Fig. 4.5b, one can reason about the correspondence between the aggregation defined for matrices above and the same aggregation in the MTBDD context. For this purpose the following definition is introduced:

Definition 37. *An* aggregation level *in BDD reach is a variable that defines the terminal variables of an aggregation. An* aggregation level *in MTBDD trans is a row variable that*

defines the terminal level for the aggregated system. A node in an aggregation level is also called pseudo terminal node.

As there is a one-to-one correspondence between row variable levels in the MTBDD *trans* and the BDD variable levels in *reach*, we will not distinguish the different definitions in the sequel (for details on the correspondence cf. Alg. 13). Considering level s_2 in the example as the aggregation level of the MTBDD, a splitting of the MTBDD is induced. If s_2 is the terminal level of the aggregated matrix, only the variables named *block variables* in the picture are needed to address the aggregated matrix. All variables below (including) s_2 are aggregated to single values which correspond to the terminal values of the aggregated MTBDD. Generally speaking, having defined an aggregation level it follows that in the aggregated MTBDD only the MTBDD variables above the aggregation level remain variables. Nodes corresponding to levels below the aggregation level do not play any role in the aggregated MTBDD. They are only used during the symbolic aggregation procedure. As the pseudo terminal nodes now play the role of the terminal nodes for the aggregated system, they have to be filled with the aggregated values by the aggregation routine. Similar to the matrix context introduced above, the symbolic multilevel algorithm can always aggregate a power of 2 *potential* states to one aggregated state.

4.2.2. Successive aggregations

Treating aggregated values in an aggregation level as terminal nodes it is straight-forward to define successive aggregations by introducing more than one aggregation level. Symbolic aggregations always are performed from the terminal nodes up in direction to the root node. To illustrate the successive aggregations within a V-cycle in the context of MTBDDs we show the MTBDDs corresponding to a 4-level approach (one fine system and 3 aggregated systems) to solve the Flexible Manufacturing System (cf. Sec. 4.7.2) model with parameter 1. The aggregation levels are 23,15,7. Its MTBDD is shown in Fig. 4.6a and the three aggregations that are used in the V-cycle are shown in Fig. 4.6b-4.6d. The roles of the MTBDD levels change during the cycle. For the first aggregation, part3 defines the submatrix levels, while the other parts define the blocks levels. In the second aggregation, the nodes within s_{23} are regarded as pseudo-terminal nodes, part12 defines the submatrix-levels while the other levels are block-levels, and so on.

This example motivates the following definition:

Definition 38. *Given an MTBDD with (row) variable levels $V = \{1, 2, \ldots, L\}$, $L \in \mathbb{N}$ and terminal level $L+1$. A V-cycle aggregation for MTBDDs is defined by an ordered set $(a_N, a_{N-1}, \ldots, a_0)$ of variable levels $L \geq a_N > a_{N-1} > \ldots > a_0 > 1$, $a_i \in V$. Every aggregation level a_i defines one aggregation. Starting from the terminal nodes, the V-cycle aggregations are uniquely defined in this case.*

Regarding the correspondence between row levels, column levels and levels in the BDD *reach*, the same definition could be made for a BDD. Note that there is an order-preserving one-to-one correspondence of (MT)BDD aggregation levels (including the terminal level) and levels of linear systems of equations in the sense of Sec. 4.1.1:

$$\begin{array}{rccc} index: & \{\texttt{aggregation levels}\} \cup \{\texttt{terminal level}\} & \rightarrow & \mathbb{N}_0 \\ & \texttt{MTBDD level} & \mapsto & \texttt{level in sense of Fig. 4.1} \end{array} \tag{4.6}$$

For the example above, the correspondence can be seen in Tab. 4.1. So if it is clear from the context it is not necessary to distinguish between levels in the sense of Sec. 4.1.1 and aggregation levels in an MTBDD. They will all be called *aggregation levels* in the sequel.

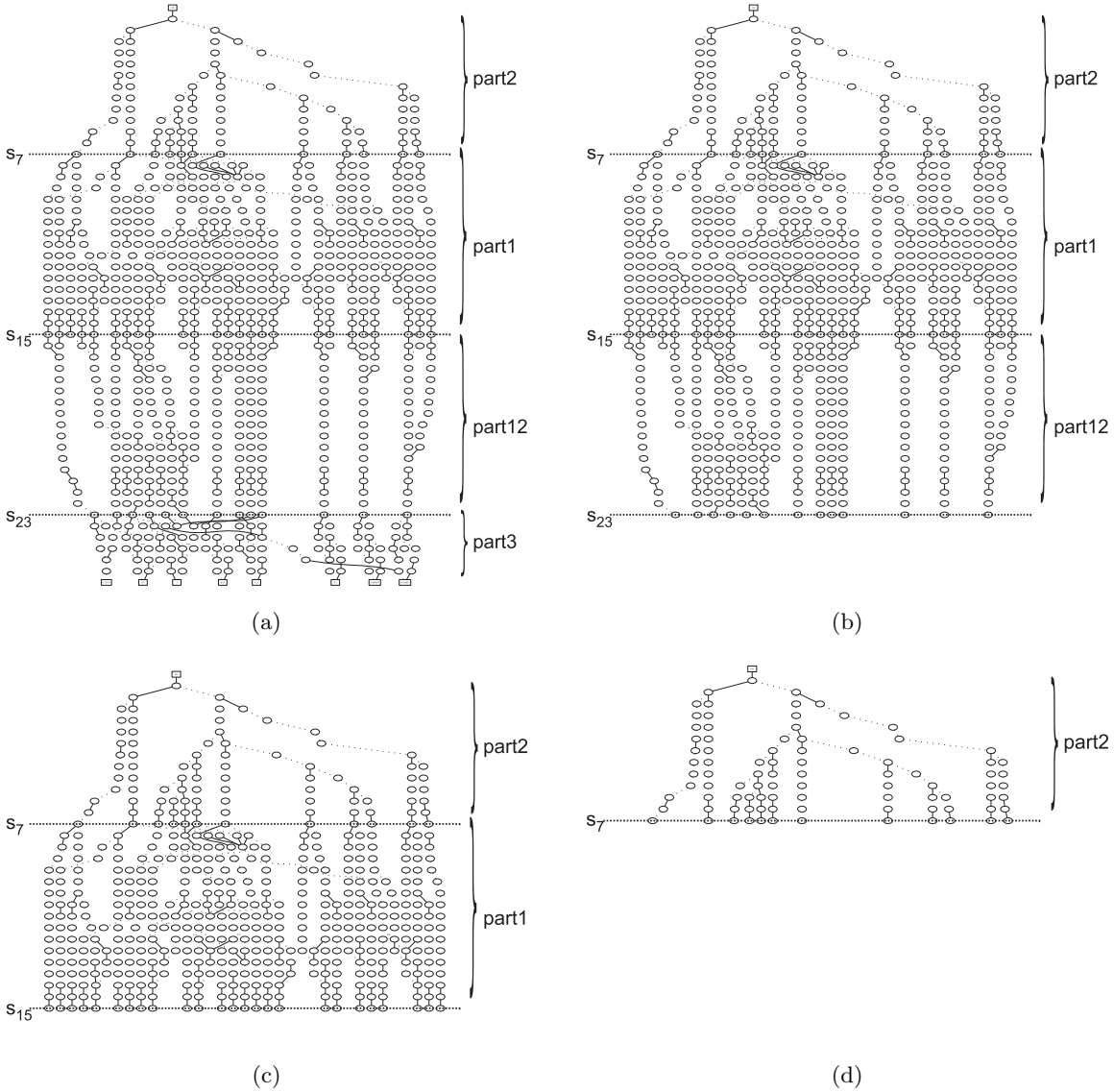


Figure 4.6.: Flexible Manufacturing System with parameter 1

4.3. Pure MTBDD approach

This section describes the first approach for a symbolic multilevel algorithm where both the transition matrix and the iteration vectors are stored as MTBDDs. This should be seen as a proof of concept and as a motivation towards the hybrid approach given in Sec. 4.4. Although it turned out that the efficiency of this approach was poor, we use it to explain the basic principle behind the MTBDD algorithm. This principle is also used in the hybrid variant of the algorithm.

Within the pure MTBDD context there is no need to distinguish between the potential and the reachable state space, as the MTBDDs for the state probability vectors share their zero-values. To be consistent with the other chapters we will anyway denote matrix entries that belong to unreachable states by '-' (not by '*0*'). Without loss of generality, we assume that all rows and columns belonging to unreachable states are also set to 0 and will be denoted by '-'.

4.3.1. An illustrative example

All steps for the aggregation of a system of linear equations are explained by means of the following small example (Fig. 4.7). It consists of 16 states. The aggregations to be made for the

MTBDD level	level of systems of linear equations (index)
7	0
15	1
23	2
26 (terminal)	3

Table 4.1.: MTBDD aggregation levels and corresponding levels of linear systems of equations

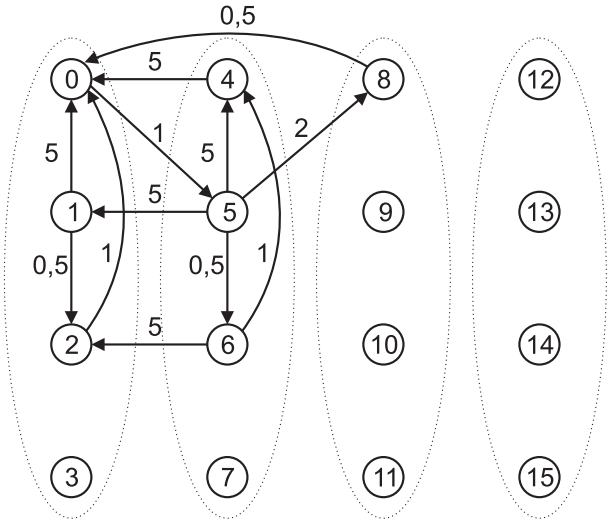
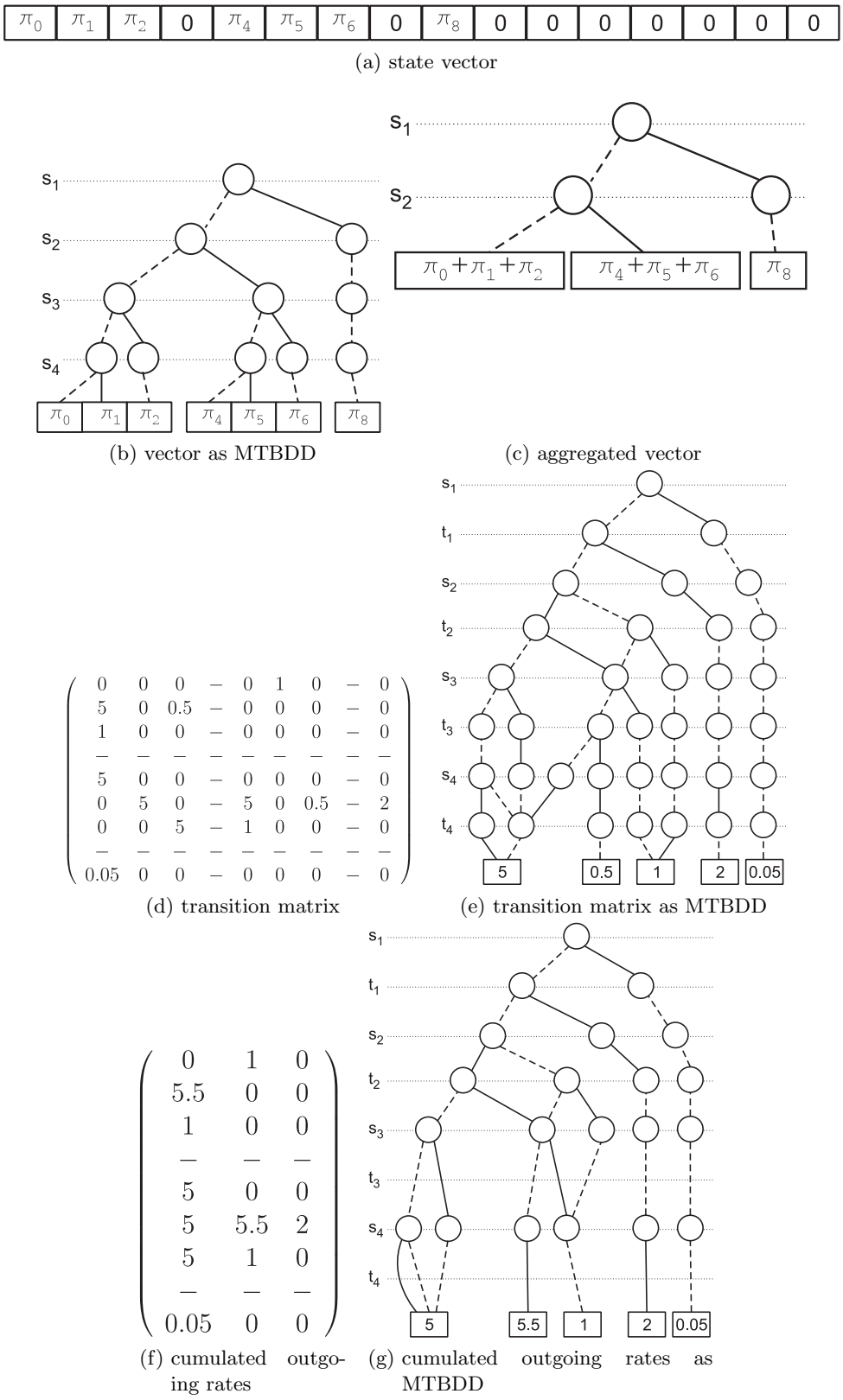


Figure 4.7.: Example Markov chain with indicated aggregations

example are indicated by the dotted ellipses (i.e. 2^2 states are aggregated). So in the terminology of Sec. 4.2.1 this means $k_{agg} = k = 2$. Note that the transition system has 9 unreachable states, so the state probabilities of states 3, 7 and 9-15 can be set to zero. The starting vector is shown in Fig. 4.8a, its MTBDD representation is given in 4.8b. The upper left portion of the rate matrix corresponding to the labelled transition system in Fig. 4.7 is shown in Fig. 4.8d, its MTBDD representation is given in Fig. 4.8e. As all submodels encoded by MTBDDs use a power of 2 as its state space (defined by the MTBDD variables) and compositional modelling may also lead to unreachable states, one has to deal with unreachable states in the matrix in practice. The variables π_i, $i \in \{0, \ldots 8\}$ are only introduced for illustration purpose. The actual calculations are done with arbitrary but fixed double precision values (modulo some arbitrary round-off introduced by the CUDD library [20]), not in a parameterised way.

4.3.2. Aggregation procedure

Let the Markov chain of interest be encoded by $k_{agg} + k$ MTBDD-variables (i.e. the state space is smaller than $2^{k_{agg}+k}$). As before, the variable levels are ordered from the most significant digit (level 1) to the least significant digit (level L). Recall that one observation of Sec. 4.2.1 was that an aggregation in the MTBDD context is uniquely determined by the levels of the aggregated variables, as the aggregations can only be done from the least significant bits up to the most significant bit. For the following MTBDD operations it is convenient to introduce the set AGG that is defined as the set of the k_{agg} consecutive aggregated variable levels ($|AGG| = k_{agg}$). As seen in Sec. 4.3.1 for the example given in Fig. 4.7 one has $k = k_{agg} = 2$. Whenever the set AGG is defined, we denote by s_{AGG} (t_{AGG}) the set of aggregated s-variables (t-variables). In the example: $AGG = \{4, 3\}$, $s_{AGG} = \{s_4, s_3\}$ and $t_{AGG} = \{t_4, t_3\}$. The algorithm for the matrix- and vector-aggregation is given in Alg. 11 and will be explained in the subsequent sections. It has a vector and a matrix as input and calculates a vector and a matrix representing the aggregated



(a) state vector

(b) vector as MTBDD

(c) aggregated vector

$$\begin{pmatrix}
0 & 0 & 0 & - & 0 & 1 & 0 & - & 0 \\
5 & 0 & 0.5 & - & 0 & 0 & 0 & - & 0 \\
1 & 0 & 0 & - & 0 & 0 & 0 & - & 0 \\
- & - & - & - & - & - & - & - & - \\
5 & 0 & 0 & - & 0 & 0 & 0 & - & 0 \\
0 & 5 & 0 & - & 5 & 0 & 0.5 & - & 2 \\
0 & 0 & 5 & - & 1 & 0 & 0 & - & 0 \\
- & - & - & - & - & - & - & - & - \\
0.05 & 0 & 0 & - & 0 & 0 & 0 & - & 0
\end{pmatrix}$$

(d) transition matrix

(e) transition matrix as MTBDD

$$\begin{pmatrix}
0 & 1 & 0 \\
5.5 & 0 & 0 \\
1 & 0 & 0 \\
- & - & - \\
5 & 0 & 0 \\
5 & 5.5 & 2 \\
5 & 1 & 0 \\
- & - & - \\
0.05 & 0 & 0
\end{pmatrix}$$

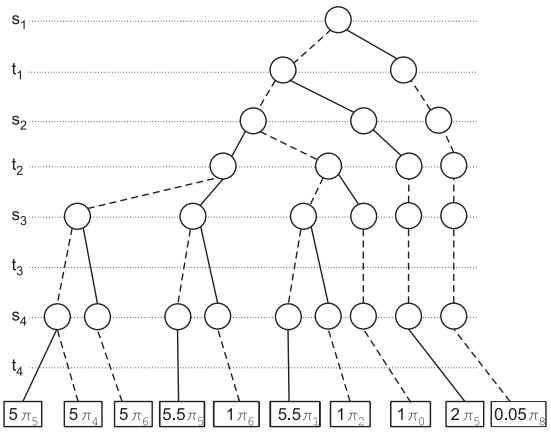
(f) cumulated outgoing rates

(g) cumulated outgoing rates as MTBDD

Figure 4.8.: Example aggregation procedure

$$\begin{pmatrix} 0 & 1\pi_0 & 0 \\ 5.5\pi_1 & 0 & 0 \\ 1\pi_2 & 0 & 0 \\ - & - & - \\ 5\pi_4 & 0 & 0 \\ 5\pi_5 & 5.5\pi_5 & 2\pi_5 \\ 5\pi_6 & 1\pi_6 & 0 \\ - & - & - \\ 0.05\pi_8 & 0 & 0 \end{pmatrix}$$

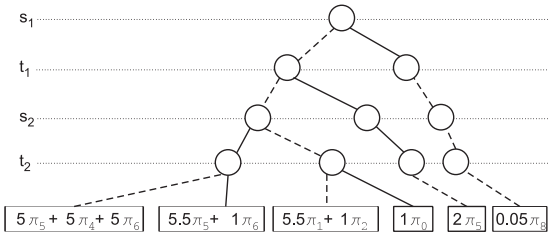
(a) weighted outgoing rates



(b) weighted outgoing rates as MTBDD

$$\begin{pmatrix} 5.5\pi_1 + 1\pi_2 & 1\pi_0 & 0 \\ 5 \cdot (\pi_4 + \pi_5 + \pi_6) & 5.5\pi_5 + 1\pi_6 & 2\pi_5 \\ 0.05\pi_8 & 0 & 0 \end{pmatrix}$$

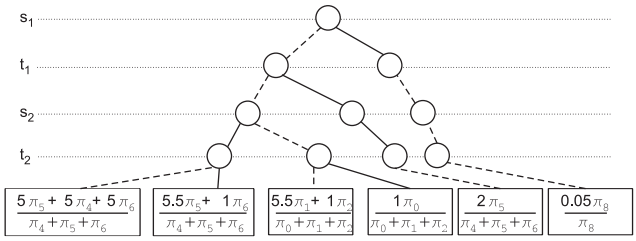
(c) cumulated incoming rates



(d) cumulated incoming rates as MTBDD

$$\begin{pmatrix} \frac{5.5\pi_1+1\pi_2}{\pi_0+\pi_1+\pi_2} & \frac{1\pi_0}{\pi_0+\pi_1+\pi_2} & 0 \\ \frac{5\cdot(\pi_4+\pi_5+\pi_6)}{\pi_4+\pi_5+\pi_6} & \frac{5.5\pi_5+1\pi_6}{\pi_4+\pi_5+\pi_6} & \frac{2\pi_5}{\pi_4+\pi_5+\pi_6} \\ \frac{0.05\pi_8}{\pi_8} & 0 & 0 \end{pmatrix}$$

(e) normalised rates



(f) normalised rates as MTBDD

Figure 4.9.: Example aggregation procedure - continued

Markov chain. Every input and output is stored as MTBDDs (or BDDs, respectively).

Algorithm 11 Aggregation in terms of MTBDD operations

1: vector$_{agg}$=ABSTRACT(vector, s_{AGG}, +)
2: rowsums=ABSTRACT(matrix, t_{AGG}, +)
3: weighted=rowsums·vector
4: newmatrix=ABSTRACT(weighted, s_{AGG}, +)
5: normalised=newmatrix/vector$_{agg}$
6: rates=ITE(ID,0,normalised)
7: diag=ABSTRACT(rates,T,+)
8: diag=diag·(-1)

Remark 28. *Having in mind the MTBDD representation of a matrix, s-variables (t-variables) correspond to the row (column) of the matrix. Therefore in the sequel, s-variables and row-variables will be used as synonyms (the same for t- and column-variables).*

4.3.3. Vector aggregation

The vector of state probabilities can be aggregated by a single abstraction operation. In this case the abstraction is used with the addition operation as the probability of the aggregate is the sum of the probabilities of the aggregated states. All aggregated variables are abstracted. The corresponding MTBDD operation is given in line 1 of Alg. 11. Fig. 4.8c shows the result of the abstraction of the s_{AGG}-variables in the example.

4.3.4. Matrix aggregation

The more interesting case is the aggregation of the matrix. It can be divided into the following steps.

4.3.4.1. Cumulative rates

In the first step, the cumulative rates from single states to certain aggregates are calculated. Therefore the $2^{k_{agg}+k} \times 2^{k_{agg}+k}$ matrix is transformed into a $2^{k_{agg}+k} \times 2^{k}$ matrix. This calculation in MTBDD-terms is given in line 2 of Alg. 11, namely abstraction from the aggregated t-variables by summation. The top left block of the result (for the example) can be seen in Fig. 4.8g. The least significant t-bit is now t_2 (t_{AGG} variables have been abstracted away). Because of only two remaining t-bits, the column dimension is $2^2 = 4$.

4.3.4.2. Weighting rates

Now the cumulative outgoing rates are weighted by the corresponding state probabilities. This is done symbolically by an APPLY operation in line 3 of Alg. 11 (Note that the source states in the matrix and the vector have to be encoded by the same variables). The resulting matrix is given in Fig. 4.9a, the corresponding MTBDD can be seen in Fig. 4.9b.

4.3.4.3. Grouping weighted rates

One can now abstract from the aggregated s-variables in the MTBDD in order to get a $2^k \times 2^k$ matrix. The suitable MTBDD operation is given on line 4 of Alg. 11: an additive abstraction of the aggregated source variables as every single aggregated state contributes to the cumulated weighted rate. The result as matrix is given in Fig. 4.9c and its corresponding MTBDD is given in Fig. 4.9d

4.3.4.4. Normalising

As the these rates are taken only under the *condition* that the Markov chain is in a certain aggregate, conditional probabilities have to be used. Therefore the previous result has to be normalised by the probability to be in the corresponding aggregate. This can be done by the APPLY operation in line 5 of Alg. 11, where we use the fact that in $vector_{agg}$ the least significant bits (corresponding to AGG) have already been abstracted away and therefore they are no longer significant for $vector_{agg}$ (i.e. these levels correspond to don't care nodes). The resulting MTBDD is given in Fig. 4.9f and its corresponding matrix is given in Fig. 4.9e

4.3.4.5. Remove self-loops

It remains to remove the diagonal entries (i.e. potential self-loops) to get the aggregated rate matrix. This can be done by an if-then-else statement, that has as a parameter *ID*, the identity matrix encoded as an MTBDD (of the same dimension as matrix *normalised*). The corresponding statement is given in line 6 of Alg. 11. The symbol 0 means the MTBDD corresponding to the terminal zero node.

4.3.4.6. Calculate diagonal vector

For the numerical steps it is convenient to calculate the diagonal entries in a way that the matrix `rates` can be enhanced to a generator matrix. Therefore of course the negative rowsums have to be calculated. Line 7 in Alg. 11 sums up all the rates per row (with T meaning all remaining t-variables, in the example this would be t_1 and t_2) and line 8 changes the sign of the calculated sums.

4.3.5. Discussion

Even though it has an appealing form, the algorithm has a major drawback. Compared to the smoothing iterations, the matrix and vector aggregations are relatively slow. Another issue is that if the iteration proceeds, the solution vector in general will have pairwise different values. Therefore the resulting MTBDD tends to grow (and in the limit is a vector of reachable state space size). Another drawback is that this version of the algorithm is not capable of using precalculations (i.e. if an aggregated rate can be determined in advance regardless of the actual state probabilities). This approach could be extended, but we did not follow this way any further. However, the pure MTBDD approach is the starting point of the work in the following sections.

4.4. Hybrid algorithm

In contrast to the full MTBDD approach presented in Sec. 4.3 this approach uses vectors of double-values to store the probabilities for the reachable part of the state space. All unreachable states have a state probability equal to zero, therefore no double value has to be stored for an unreachable state. The idea of the hybrid approach for standard numerical methods is to add offset information to the MTBDD *trans* in order to connect the potential state space in *trans* with the consecutive vector entries of the iteration vectors in a memory efficient way. The idea behind the hybrid approach for the multilevel algorithm is that for the different aggregated systems no new MTBDDs should be built and the structure of the MTBDD encoding the fine system should be used for the aggregated systems as well. The hybrid approach for the multilevel algorithm extends the offset-labelling scheme to the concept of multi-offset labelling in such a way, that also all the vectors for the aggregated systems can be compactly stored.

In the following subsections we will make use of the definition of aggregation levels from Sec. 4.2.1. Recall that a set of successive aggregations is given by a set of aggregation levels in

an MTBDD (this is unique and well-defined under the constraint that aggregations are always made starting from the terminal levels). In the following, level always means a variable level in an MTBDD (similar to the aggregated levels in the context of the pure MTBDD approach). Therefore the algorithms will use the term *level_to* to indicate the level that defines the current aggregation and *level_from* for the level where the aggregation started. As mentioned before, for the first aggregation level_from is equal to the terminal level, in all other cases, level_from corresponds to an aggregation level. Both for the vector and the matrix the aggregation levels are always defined in terms of s-variables.

As in [47] we assume that don't care nodes are explicitly present in the (MT)BDDs and that in MTBDD *trans* both rows and columns corresponding to unreachable states are set to 0.

4.4.1. Multi-offset labelling

In this section we reason about the basics for the symbolic multilevel algorithm in a hybrid setup, the multi-offset labelling concept. The offset labelling scheme was introduced in [47] for standard iterative algorithms and can be extended to a multi-offset-labelling scheme for the multi-level algorithm in a canonical way: Every matrix has (up to the terminal level or to the corresponding aggregation level) its own offset labelling scheme.

Assume a model has a certain potential (S_p) and reachable (S_r) state space. The BDD *reach* that encodes the set of reachable states can be used for a compact mapping $S_p \rightarrow S_r$ by offset-labelling. In addition to the BDD *reach* only as many integer values as there are non-terminal nodes in *reach* are needed to define the mapping. As described in Sec. 2.8.3 each minterm in a (MT)BDD encodes a binary number. In the case of *reach* this number is the number of a state in the potential state space (each variable encodes a certain power of two). Once the offset-labelling of *reach* is known, the corresponding number in the reachable state space is calculated as the sum of the offsets of nodes that are left via the then-edge (cf. [47]). As an example consider again the matrix given in Fig. 4.8d (with initial state 0). Symbolic reachability analysis leads to the offset-labelled BDD given in Fig. 4.11a. The rightmost path corresponds to the bitstring 1000. This path maps the potential state $1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 8$ to the reachable state $6 + 0 + 0 + 0 = 6$. Looking at the Fig. 4.8d and having in mind the aggregation procedure, one can make the following observation:

Remark 29. *If in a group of aggregated states at least one state is reachable, then the whole aggregate is reachable (no further reachability analysis on the aggregated Markov chain is needed). Equivalently, aggregates consisting only of unreachable states will be unreachable.*

An example of the two different situations in remark 29 is given in Fig. 4.10. The 16-dimensional vector at the bottom has to be aggregated. Let the zero probabilities in this vector correspond to unreachable states. By grouping four neighbouring states, four aggregates remain. Only those aggregates that exclusively contain unreachable states remain unreachable. All others are reachable. To be able to store the probability vectors for the aggregated systems also without containing any unreachable states, it is necessary to pre-calculate the offsets for the original and all aggregated systems. So in contrast to [47] we will use an array of (else-) offsets to store all the different reachability structures. There will be one entry in this array for the fine system and one for every aggregated system.

4.4.1.1. Calculating the offsets

Algorithm 12 presents an offset-labelling algorithm that can be seen as a variant of the offset-labelling algorithm given in [47]. Both algorithms produce the same offset-labelling for the non-aggregated system (the algorithm given in [47] is not defined for aggregated systems). For the calculation of the multi-offsets it is useful to reformulate remark 29 in terms of the BDD reach: A node in an aggregation level will be reachable if it is non-trivial, i.e. non-zero. The

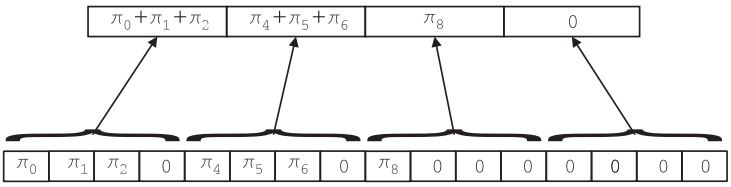


Figure 4.10.: Example for the different situations in vector aggregation

algorithm uses the *agg_level* for an arbitrary aggregation- or terminal level. As an abbreviation $agg\_index := index(agg\_level)$ is used (where index is the function defined in Eq. 4.6). In our case a multi-offset labelled BDD node has the following properties:

else a reference to the else-node below the current node

then a reference to the then-node below the current node

toff the then-offset of the current node

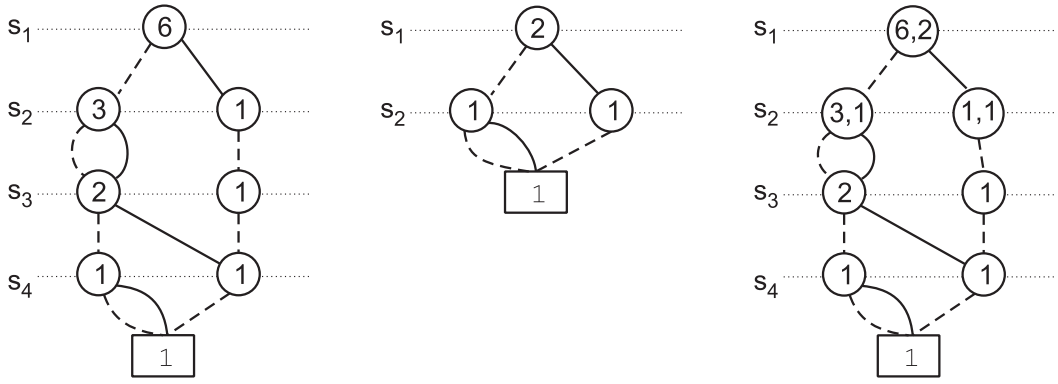
eoff an array of the else-offsets corresponding to the original system and all aggregates

The then-offsets are only used temporarily for the calculations, therefore they occur as single value, not as array (they can be re-used for different pairs of agg_level/agg_index). The description of the algorithm is as follows. The initial call to the algorithm is add_offsets(root,0) where root is the root node of BDD *reach*. Line 1-3 checks whether the terminal zero node is reached and returns offset 0 in this case. Line 4 checks for the other stopping criterion of the recursion, i.e. a non-zero node in the current agg_level (for zero-nodes, this line will never be reached). In this case the offset one is returned in line 5. Lastly, in line 7 we check whether the offsets of the child nodes already have been calculated and if not, they are recursively calculated in lines 8 and 9. Line 11 returns the sum of the child offsets as the offset of the current node. Note that in order to make the algorithm more general, we do not check for terminal-one-nodes (as done in [47]), but instead check for a non-zero node in the current stop-level of the recursion. Thus the algorithm can be used both for the original and the aggregated systems.

Algorithm 12 add_offsets(node, level)

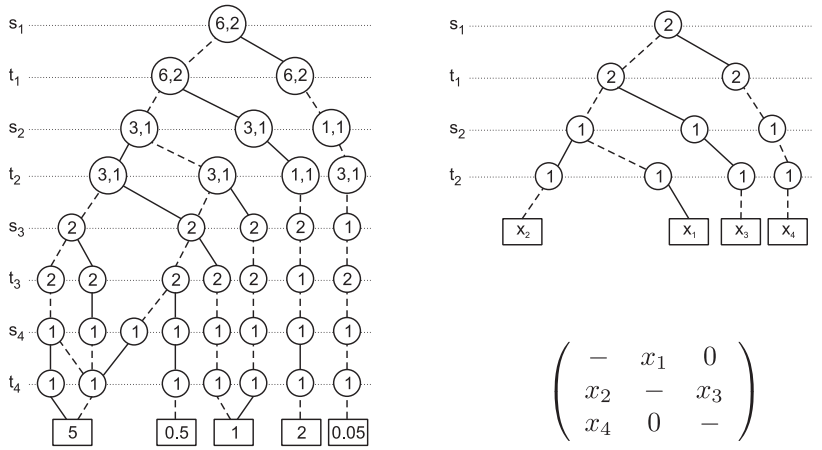
```
 1: if node = ZERO then
 2:    return 0
 3: end if
 4: if level = agg_level then
 5:    return 1
 6: end if
 7: if NotCalculated then
 8:    node→toff = add_offsets(node→then, level+1)
 9:    node→eoff[aggregate_index] = add_offsets(node→else, level+1)
10: end if
11: return node→toff+node→eoff[aggregate_index]
```

An example of the algorithm can be found in Fig. 4.11. According to the non-aggregated vector in Fig. 4.10, the corresponding BDD *reach* is offset-labelled in Fig. 4.11a (agg_level:= terminal level). Fig. 4.11b shows the offset-labelling for a BDD $reach_{agg}$ that corresponds to the reachable states of the system aggregated in s_3. The result for two runs of the $add\_offsets$ algorithm (agg_level$\in$ {terminal level, s_3}) is given in Fig. 4.11c. The single offset values and the left values within the offset tuples correspond to the non-aggregated system while the right values within the offset-tuples correspond to the aggregated system (cf. Fig. 4.11a, 4.11b). One could also store the different offset informations in different offset-labelled BDDs, but as the



(a) BDD *reach* (b) aggregation (c) Multi-offset

Figure 4.11.: Multi-offset labelling Example



(a) (b)

Figure 4.12.: Multi-offset labelling and aggregation

shared parts of the BDDs are the same, we chose to store all information in one single BDD. Such multi-offset labelled BDDs can be used for vector operations on the reachable state space and for labelling the MTBDD *trans* in order to map it to the reachable state space.

4.4.1.2. Offset-labelling of MTBDD *trans*

The MTBDD *trans* representing the rate matrix can be labelled according to the algorithm given in [47]. The only difference is that from the corresponding copies of *reach* multi-offsets instead of ordinary offsets are added to the MTBDD nodes (in the form of references to the corresponding offset arrays in BDD *reach* - in [47] the integer values are stored directly). The labelling procedure for *trans* is basically the interleaved traversal of two copies of reach. One for the row- and one for the column variables. Following this principle the offset-labelled MTBDD can be seen in Fig. 4.12. In Sec. 4.4.3.3 we give an extended version of the offset labelling algorithm presented in [47] that works in the context of characterisation of nodes.

4.4.2. Storage of the aggregated matrices

This section is about the compact storage of aggregated matrices when the MTBDD *trans* is known and should not be altered. The first two approaches are only capable of detecting diagonal entries in aggregated matrices (that do not have to be calculated), the third approach is more sophisticated. The basic problem in every storage scheme for aggregated matrices is

that the symmetries can be broken, i.e. aggregated values coming from the same submatrices can lead to different aggregates as they are weighted with different parts of the iteration vector. Therefore each non-diagonal node within an aggregation level has to represent at least one aggregated value. It has to represent at most as many aggregated values as there are non-diagonal minterms leading to this node. In the first two approaches for every aggregation level as many aggregated values are stored as there are minterms up to this aggregation level (minus the minterms corresponding to diagonal entries). In the first two approaches diagonal elements are detected on the fly by checking equality of row and column offsets for the corresponding aggregate (that have to be calculated anyway). In the third approach diagonals are detected by precalculations and the diagonal property is stored for every node.

4.4.2.1. First approach: Separate arrays

This is the approach presented in [54]. It largely uses pointer arithmetic to remember the correct positions of aggregated values. Every node in an aggregation level carries the following data structure.

counter carries the index of the aggregate that corresponds to the current visit of the node

aggregates an array that carries the aggregated values for the different minterms.

During the construction of the offset-labelled MTBDD $trans'$ the number of visits to every node in an aggregation level is counted and is used to determine the dimension of the different arrays. As a precomputation before reading or writing the matrix all counters in a certain aggregation level are reset. On every visit to the node in the corresponding aggregation level its counter is incremented and then the array at position counter is read/written. Fig. 4.13a shows the example from Fig. 4.12b as separate-array approach. Obviously for nodes that correspond to only one aggregate, there is a large overhead with additional counters.

4.4.2.2. One array per aggregation level

A slight optimisation for the matrix storage was used in [55]. It was driven by the observation that once a traversal scheme for the MTBDD is fixed the sequence of visited nodes within an aggregation level is uniquely determined. So one single counter is sufficient for every aggregation level. At most two aggregation levels are used during an aggregation procedure, therefore it can be deduced that it is sufficient to use two counters in total. At most one for the level to be aggregated (level_from, called from_counter - for the terminal level not necessary) and the second one for the aggregated level (level_to, called to_counter). The nodes do not have to carry any separate counters or arrays. One single array for the aggregated values is used per aggregation level. The schematic data structure can be seen in Fig. 4.13b, again for the example in Fig. 4.12b. In this example, the DFS (depth first search) traversal was used to generate the ordering of the aggregated values. Any other unique traversal procedure could be used as well.

4.4.2.3. Node Characterisation

The main difference of the last approach in contrast to the first two approaches is the fact that not all aggregations have to be computed. There is a number of redundant aggregations that are known a priori without knowing the actual iteration vector. In the running example, the result of one aggregation is a priori known, as the corresponding fraction of state probabilities shortens out. As shown in Fig. 4.13c, for the leftmost node in the aggregation level, the aggregated value 5 is already known (after some precalculations). Nodes corresponding to diagonal entries are marked with a D. For the remaining three nodes, the entries in the aggregates array are needed. This concept will be explained in full detail in the following Sec. 4.4.3.

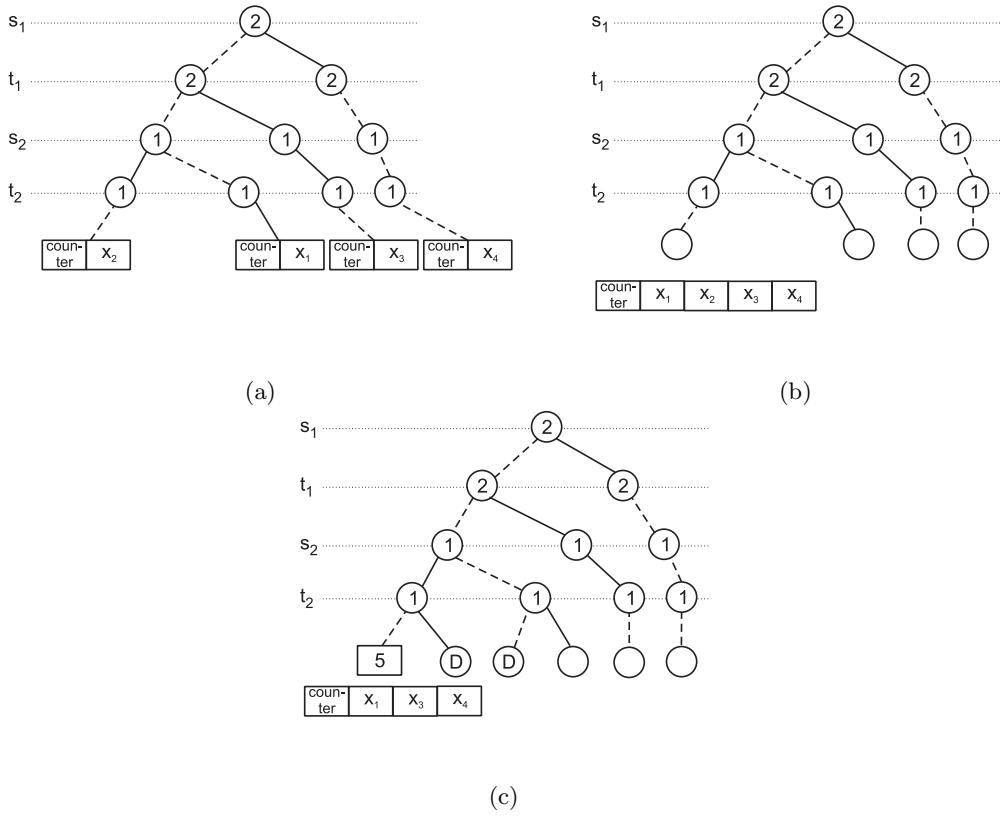
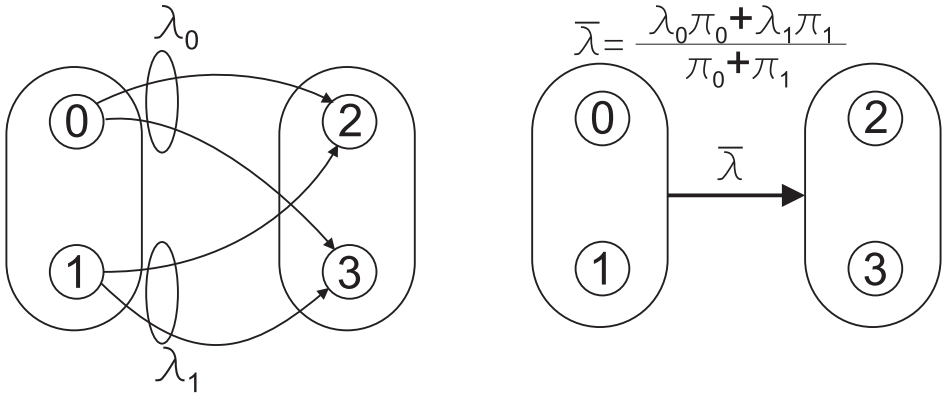


Figure 4.13.: Storage of aggregated matrices



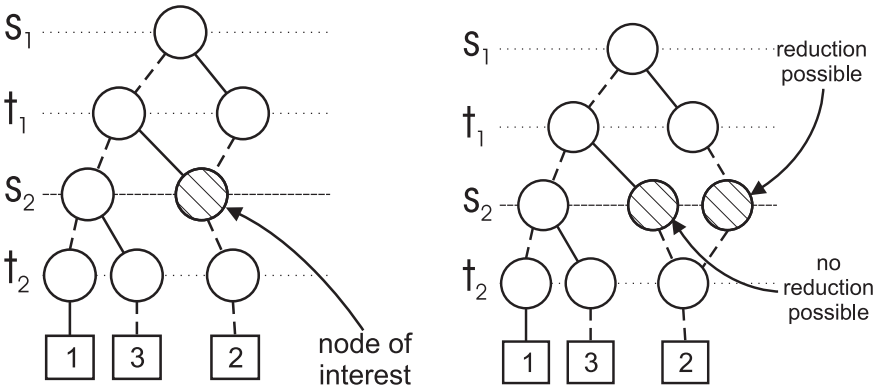
(a) Rates to be aggregated (b) Aggregated rate

Figure 4.14.: Aggregation of matrices and MTBDDs

4.4.3. Characterisation of nodes

Looking again at Fig. 4.5, for the characterisation we are only interested in the *aggregation nodes*, i.e. nodes that belong to the aggregation level (as already mentioned, in general there can be usually more than one aggregation level). Firstly, looking at the corresponding matrix in Fig. 4.5a we can distinguish between nodes belonging to diagonal blocks in the aggregated matrix and those that do not. As before we do not regard nodes belonging to diagonal elements in the aggregated matrix. In the sequel we focus on the nodes that belong to non-diagonal elements, i.e. nodes, where aggregations have to be performed.

Next, we introduce the notation of *reducible* and *non-reducible* nodes by means of the simple four-state-model given in Fig. 4.14a. It can be seen that the aggregated rate $\bar{\lambda}$ depends on the cumulative sums of the rates to be aggregated (λ_0 and λ_1, respectively) and the current



(a) MTBDD trans (b) Clash resolved

Figure 4.15.: Example: Aggregation clash

state probabilities (π_0 and π_1, respectively) of the states from which they emanate. Reducible nodes are those where the aggregation equation cancels out and the result can be precalculated, otherwise we call them non-reducible nodes. There are three different cases:

- $\lambda_0 = \lambda_1$: fraction cancels out $\Rightarrow$ *reducible*
- $\lambda_0 = 0$, $\lambda_1 \neq 0$ (without loss of generality):
  - cancels out for $\pi_0 = 0 \Rightarrow$ *reducible*
  - does not cancel out for $\pi_0 \neq 0 \Rightarrow$ *non-reducible*
- $\lambda_0 \neq \lambda_1$, both non-zero: no cancellation $\Rightarrow$ *non-reducible*

Note that this characterisation can easily be generalised to a larger number of states to be aggregated. Further we would like to stress that it is possible to do all these calculations by purely symbolic operations.

In order to generalise from the example one can state the following characterisations. Given that there is an arbitrary finite set $\{\lambda_i | i \in I\}$ of cumulative sums. For the general weighted sum $\frac{\sum_{i\in I} \lambda_i \cdot \pi_i}{\sum_{i\in I} \pi_i}$ the different cases in the example generalise as follows:

- If $\exists c \neq 0 : \forall i \in I : \lambda_i = c$ then obviously the fraction cancels out.
- If for a set $N \subset I$ it holds that $\forall i \in N : \lambda_i = 0$ and $\exists c \neq 0 : \forall i \in I \setminus N : \lambda_i = c$, then
  - fraction cancels out if $\forall i \in N : \pi_i = 0$
  - fraction does not cancel out if $\exists i \in N : \pi_i \neq 0$
- If $\exists i, j \in I : \lambda_i \neq \lambda_j$ fraction does not cancel out.

4.4.3.1. Aggregation clash

Similar to an offset clash introduced in [47], in the multilevel context, the notion of an aggregation clash is important. Consider once again the small example given by the matrix in Fig. 4.5.

It is an easy exercise to show that the offset-labelled BDD *reach′* looks like the one in Fig. 4.11b. As in one BDD level only the same offset values occur, no offset clash is possible in the sense of [47]. According to Sec. 4.2.1, the corresponding MTBDD representation is given in Fig. 4.15a. Suppose again that s_2 is the aggregation level, i.e. consider aggregations of 2×2 submatrices. The problem is that there is one node (marked by *node of interest*) representing both the top right $\begin{pmatrix} 2 & - \\ 0 & - \end{pmatrix}$ (non-reducible) and bottom left $\begin{pmatrix} 2 & 0 \\ - & - \end{pmatrix}$ (reducible) submatrix. In this case we speak of an *aggregation clash*. This clash has to be resolved by splitting this node into two separate nodes, as seen in Fig. 4.15b (during the offset labelling procedure for *trans*). As in the example no offset clashes occur we notice that {`aggregation clashes`} $\not\subseteq$ {`offset clashes`}.

4.4.3.2. Symbolic reducibility check

We would like to note at this point that the characterisation of nodes can be calculated using symbolic (i.e. MTBDD) operations. The reducibility check is performed during the offset labelling procedure of the MTBDD *trans*. In order to take care of reducible nodes, the standard generation routines [47] have to be modified. As the reducibility check is only relevant for nodes in aggregation levels, only the routine for the row-variables has to be modified. The function $LabelMTBDDCol$ remains more or less unchanged. The reducibility check algorithm is explained in detail in Sec. 4.4.3.4.

4.4.3.3. Offset labelling of MTBDD *trans* in the case of node characterisation

The description is focused on the row labelling part, as the column labelling is a straight-forward generalisation of the algorithm given in [47]. The two routines together produce an offset-labelled MTBDD *trans* out of a (non-offset-labelled) MTBDD *trans* and an offset-labelled BDD *reach*. The statement for the invocation of the routine is

$$LabelMTBDDRow(trans, reach, reach, \vec{0}, \vec{0})$$

where *trans* is the transition matrix (without offset labels) and *reach* is the offset-labelled BDD of reachable states (one copy is used for tracing the row, one for the column). The vectors $\vec{0}$ carry the array with the current offset information (for the fine system and all aggregates).

The term *hybrid node* will be used in the sequel for a node in the offset labelled MTBDD. Otherwise—without offsets—simply *node* will be used. The algorithm works as follows. If the current node is a terminal node, no offset labelling is necessary, so the corresponding hybrid node can immediately be returned (line 1-3). For the characterisation of nodes it is important to know whether a node is reducible or not. This is calculated in line 4 (this function is explained in Sec. 4.4.3.4). Lines 5-8 check if the node in the current row and column with the given reducibility property already has been created. If so, the node is returned in line 7. Note that in the naive setup it is necessary to visit all the nodes below the current node in order to keep the number of visits during a DFS traversal up to date (line 6). As explained later, this line can be omitted and the number of visits can be calculated in a postprocessing step with fewer calculation effort. Line 9-11 are used to maintain the correct system for the actual traversal. The recursion that calculates the hybrid nodes corresponding to the child nodes of the current node is performed in line 12 and 13. Depending on which type of node was detected, the different hybrid nodes are created in line 14-26. It can be a diagonal node with respect to the current aggregation (line 15-16), a reducible node (lines 18-19), a non-reducible node (lines 20-21) or it can be a node that is not in an aggregation level (line 24-25). Note that for reducible nodes an index for the current aggregated value is stored in the hybrid node. This index points to the actual aggregated value. Note that both CreateNode and LabelMTBDDRow functions return offset-labelled nodes while the variable dd is the node without offsets.

A similar algorithm is performed for the columns (given in Alg. 14). The difference is that a column variable cannot belong to a terminal level, therefore line 1 and 2 are different. Further no column level variable can be in an aggregation level, so all the reducibility checks are omitted. As the visitAll routine is only defined on row variables, it is performed on the children of the column node (line 5 and 6). Again, the visitAll commands can be omitted, if the number of visits per aggregated node are calculated as a postprocessing step.

4.4.3.4. Function isReducible

The criterion if the state probability is zero can be determined by looking at the BDD *reach* that carries the reachable set of the state space. The symbolic operations for a reducibility check are given in Alg. 15. In addition to the standard MTBDD operations it is convenient to use

Algorithm 13 LabelMTBDDRow(dd, row, col, rowOffsetArray, colOffsetArray)

```
1: if isTerminal(dd) then
2:    return CreateNode(dd,-,-,-,-,NON_AGG);
3: end if
4: reducible=isReducible(dd, row)
5: if isCachedRowNode(dd, row, col, reducible) then
6:    visitAll(cachedNode(dd, row, col, reducible))
7:    return cachedNode(dd, row, col, reducible)
8: end if
9: if dd ∈ nextAggLevel then
10:   aggregation_level:=true, update nextAggLevel
11: end if
12: e_node = LabelMTBDDCol(dd→else, row→e, col, rOffArray, cOffArray)
13: t_node = LabelMTBDDCol(dd→then, row→t, col, rOffArray+(row→eoff), cOffArray)
14: if aggregation_level then
15:    if isDiagonal then
16:       return CreateNode(dd,e_node,t_node, col, row, DIAG)
17:    else
18:       if reducible then
19:          return CreateNode(dd,e_node,t_node, col, row, REDUCIBLE(index))
20:       else
21:          return CreateNode(dd,e_node,t_node, col, row, NON_REDUCIBLE)
22:       end if
23:    end if
24: else
25:    return CreateNode(dd,e_node,t_node, col, row, NON_AGG)
26: end if
```

Algorithm 14 LabelMTBDDCol(dd, row, col, rowOffsetArray, colOffsetArray)

```
1: if dd = ZERO then
2:    return CreateNode(dd,-,-,-,-,NON_AGG);
3: end if
4: if isCachedColNode(dd, row, col) then
5:    visitAll(cachedNode(dd, row, col)→else)
6:    visitAll(cachedNode(dd, row, col)→then)
7:    return cachedNode(dd, row, col)
8: end if
9: e_node = LabelMTBDDRow(dd→else, row, col→e, rOffArray, cOffArray)
10: t_node = LabelMTBDDRow(dd→then, row, col→t, rOffArray, cOffArray+(col→eoff))
11: return CreateNode(dd,e_node,t_node, col, row, NON_AGG)
```

a function COUNTANDCHECKZERO that counts the number of different leaves (nLeaves), returns the first non-zero leaf value and sets a flag (*zerofound*) that indicates whether the terminal zero node was found or not. This function is a slight extension of the CUDD [20] function Cudd_CountLeaves. Its implementation is given in Sec. A.

Algorithm 15 isReducible(trans, row)

```
 1: partOfRowSum = ABSTRACT(trans, t_agg)
 2: (nLeaves, value, zerofound) = COUNTANDCHECKZERO(partOfRowSum)
 3: if nLeaves = 1 then
 4:    return TRUE
 5: else if nLeaves = 2 then
 6:    if zerofound = TRUE then
 7:       partOfRowSum_01 = THRESHOLD(partOfRowSum, 0)
 8:       pattern = partOfRowSum_01 ⊕ row
 9:       if pattern = ZERO then
10:          return TRUE
11:       else
12:          return FALSE
13:       end if
14:    end if
15: end if
16: return FALSE
```

The algorithm works on sub-MTBDDs of the MTBDD *trans* and *reach*. For the description of the algorithm we choose the local view of the MTBDD assuming *trans* to be the root node of the sub-MTBDD that corresponds to the sub-matrix that is to be aggregated. Further let *row* be the corresponding sub-MTBDD in *reach* that corresponds to the row of the sub-matrix. The algorithm primary works on rowsums which are generated in line 1. Line 2 determines some basic properties of the set of row-sums. If there is only one terminal node (either zero or non-zero), the corresponding aggregation is reducible. This is checked in line 3 and 4. Line 5 checks for the more interesting case where two different leaves are detected. If one of the two terminal nodes is equal to zero (zerofound=TRUE in line 6) the pattern of reachable states has to be checked. Therefore the zero/non-zero pattern of the rowsums is determined in line 7 and XORed with the reachability pattern in line 8. If every unreachable state corresponds to a zero-rowsum in the matrix (and every reachable state corresponds to a non-zero rowsum in the matrix), then the resulting XOR-pattern is zero and so the aggregation is reducible (lines 9-10), otherwise it is non-reducible (lines 11-12). Note that in the case where two terminal nodes exist and both are non-zero (zerofound is not TRUE in line 6) or when there are more than two terminal nodes, the resulting aggregation is non-reducible. That case is covered in line 16.

4.4.4. Counting the aggregations

For a node in an aggregation level it is an important invariant how many non-reducible nodes in the next aggregation level are below this node (in the sense that they are reached during further descend in DFS algorithm). This information is used to calculate the sum of nonreducible nodes per level (i.e. the memory that has to be allocated for each aggregated matrix) and it can be used to skip reducible nodes during the aggregation process. We present two variants of the algorithm

4.4.4.1. Standard approach

This is the visitAll routine that is called from Alg. 13 and Alg. 14 during the construction of the hybrid MTBDD. The initial call to the algorithm depends on the current situation while building the hybrid MTBDD *trans*. The algorithm is shown in Alg. 16. If *agg_index* corresponds to the

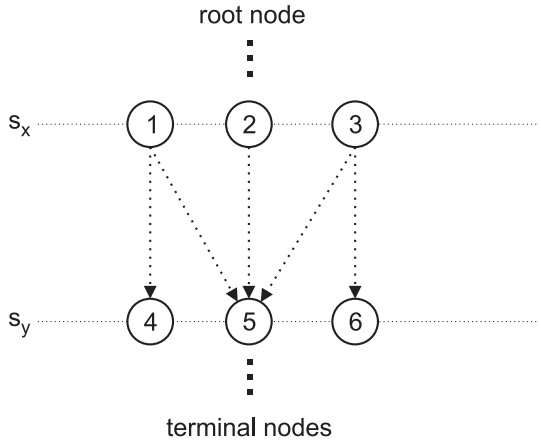


Figure 4.16.: Calculating the number of visits

fine system or the node is the zero node, the recursion bottoms out (lines 1-3). If a node in the aggregation level is reducible, the counter is updated in lines 5-7. Whenever an aggregation level is reached, the recursion parameters are reset to look for the next possible aggregation level (lines 4, 8 and 9). As the parameter level is the row variable level, the recursion is split up to four branches with the next possible (row-variable) nodes in lines 11-20.

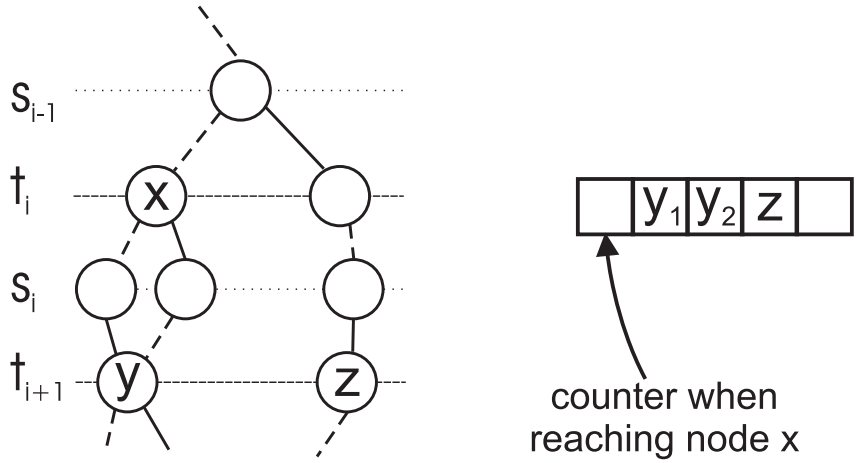
Algorithm 16 visitAll(node, level, agg_index, next_agg_level)

```
 1: if (agg_index=0) OR (node = ZERO) then
 2:    return;
 3: end if
 4: if level=next_agg_level then
 5:    if node→type = NON_REDUCIBLE then
 6:       nonreducible_per_level[agg_index]++;
 7:    end if
 8:    agg_index=agg_index-1;
 9:    next_agg_level=agg_level[agg_index];
10: end if
11: e_node = node→else;
12: if e_node != ZERO then
13:    visitAll(e_node→else, level+1, agg_index, next_agg_level);
14:    visitAll(e_node→then, level+1, agg_index, next_agg_level);
15: end if
16: t_node = node→then;
17: if t_node != ZERO then
18:    visitAll(t_node→else, level+1, agg_index, next_agg_level);
19:    visitAll(t_node→then, level+1, agg_index, next_agg_level);
20: end if
```

4.4.4.2. Optimised approach

As seen in Alg. 13 and Alg. 14 if a cached node is found, a visitAll function is called that updates all counters below the given node. A slight improvement can be made when the visitAll function calls are omitted during the hybrid MTBDD generation. It has to be replaced by a postprocessing call if one uses the visits per aggregation level and updates the visits in the next level according to the appropriate multiplication factors.

This observation is sketched in Fig. 4.16. Suppose the first aggregation takes place at s_y and the second aggregation at s_x. The levels above s_x, the root node, the levels between s_y and s_x, the levels below s_y and the terminal nodes are not explicitly depicted. Suppose further that



(a) Higher aggregations (b) Critical values

Figure 4.17.: Skipping reducible nodes

starting from the root node, the number of visits of a node i is denoted by n_i. Now let n_1, n_2 and n_3 be known (e.g. by a depth first traversal up to s_x). Every path from a node i in s_x to a node in s_y can now be weighted with multiplicity n_i. Thus the DFS traversal can be split into DFS traversals from the root node to the last aggregation level followed by DFS traversals (using the calculated weights) from the last aggregation level to the last but one aggregation level. In this way the visitAll routine can be omitted from the model generation and can be performed afterwards as a block-based aggregation scheme which successively calculates the number of visits from the root to the bottom.

4.4.5. Aggregation Offsets

Without exploiting the reducibility of nodes, the matrix aggregation amounts to a DFS traversal of the MTBDD *trans* up to the variable level of the system to be aggregated. Once the nodes are distinguished, aggregations of reducible nodes do not have to be calculated in every iteration, as they can be precalculated once as shown above. When an aggregated value is known in advance, it is desirable to be able to skip the further traversal of such a node. A slight problem arises when successive aggregations have to be performed, as all aggregated values of one aggregation level are stored sequentially: In Fig. 4.17a a part of an MTBDD representing a transition matrix is given. Assume there are two aggregation levels s_i and s_{i+1} and assume further that x is a diagonal (or reducible) node for the aggregation level s_i, and therefore might be skipped (i.e. its sub-MTBDD is not processed). Let y be an non-reducible node and let us further assume that during the aggregation in s_{i+1} the aggregated matrix values were stored as indicated in Fig. 4.17b (two paths leading from x to y, so two values y_1 an y_2 have to be stored for the aggregated values of node y). Now the point is that if during the aggregation at level s_i node x is skipped (i.e. its sub-MTBDD is not processed), the counter indicating the position of the aggregated value has to be updated as well. Otherwise in the example node z would get the value y_1 as two incrementations of the counter were missed. In the spirit of the offset labelling concept presented in [47] we therefore introduce the concept of aggregation offsets. That means for every node x at an aggregation level s_i we precalculate the number of non-reducible nodes in aggregation level s_{i+1} that are reached during a traversal of the sub-MTBDD below x. This is the offset to be added to the array pointer for the aggregated values when such a node is skipped.

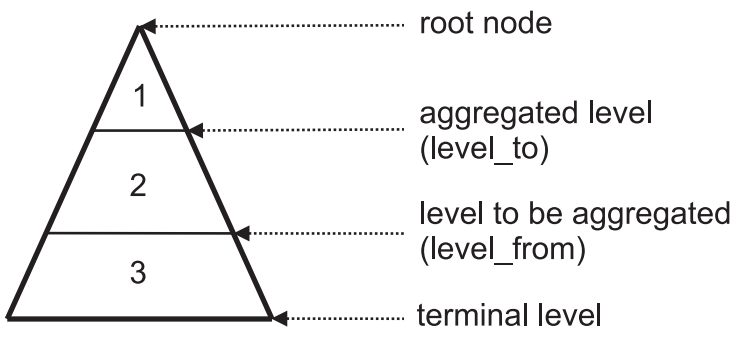


Figure 4.18.: Partition of BDD *reach*

4.4.6. Vector operations

Using the multi-offset labelled BDD *reach* it is possible to define recursive vector operations that can be used for operations like aggregation, disaggregation without any additional memory effort. The following notation will be used:

level_to (MTBDD) level that corresponds to the aggregated state space

level_from (MTBDD) level that corresponds to the state space before aggregation

Further it is convenient to have a notation for accessing the right multi_offset for a certain state space. Therefore the following abbreviations will be used (where index is the function defined by Eq. 4.6 in Sec. 4.2.2):

index_to index that corresponds to the aggregated state space, $index\_to := index(level\_to)$

index_from index that corresponds to the state space before aggregation,
$index\_from := index(level\_from)$

For the first aggregation level_from is equal to the level for the terminal nodes. For the next aggregation level_from is equal to level_to of the first aggregation and so on. As an example we present the algorithm for vector aggregation but the principle can be transferred easily to other vector operations as well. Every choice of level_from and level_to levels divide the BDD as sketched in Fig. 4.18 (Of course in the extremal case when level_from corresponds to the terminal nodes only two sections remain). When processing a multi-offset labelled BDD for a vector operation, two indices have to be updated. Firstly the vector index for the aggregated system (referenced as coarse offset, *c_off*) and secondly the vector index for the system to be aggregated (referenced as *off*). Section 1 in Fig. 4.18 are those nodes that affect both offsets (system to be aggregated and aggregated system). In section 2 in Fig. 4.18 only the offsets of the system to be aggregated are used for the calculations (the aggregated system has no nodes in the corresponding levels). Section 3 in Fig. 4.18 are unnecessary levels for the current aggregation.

The algorithm for the vector aggregation is given in Alg. 17. In line 1 the recursion bottoms out for terminal zero nodes. Line 3 and 4 are the situation where a node in the aggregated level is reached and its corresponding aggregated value is updated. In line 5-7 the case for section 1 in Fig. 4.18 is covered (recursion with updates on both kinds of offsets). Line 8-10 cover section 2 in Fig. 4.18 (recursion with updates only on the offsets for the system to be aggregated). Analogously the other vector operations can be performed. Obviously a vector operation corresponds to a DFS traversal of the BDD *reach* and is therefore dependent on the number of BDD variables to be processed. A faster algorithm with linear complexity has to precalculate the border indices between each aggregated set of state probabilities (one observes that only consecutive probabilities are aggregated according to the aggregation scheme). Then a vector operation reduces to looping through the array with respect to the precalculated indices.

Algorithm 17 aggvec(node, off, c_off, level)

```
 1: if node = ZERO then
 2:    return
 3: else if  level = level_from then
 4:    coarse_vector[c_off]=coarse_vector[coarse_offset]+vector[off]
 5: else if level < level_to then
 6:    aggvec(node→else, off, c_off, level+1)
 7:    aggvec(node→then, off+node→eoff[index_from], c_off+node→eoff[index_to], level+1)
 8: else if level < level_from then
 9:    aggvec(node→else, off, c_off, level+1)
10:    aggvec(node→then, off+node→eoff[index_from], c_off, level+1)
11: end if
```

4.4.7. Aggregation of an offset-labelled matrix

For an aggregation step without node characterisations it would be necessary to traverse the MTBDD *trans* up to the level that is to be aggregated. Using node characterisation, some paths can be skipped at the aggregation level (when reducible or diagonal nodes are reached). In this section, only the algorithm for the case with node characterisation is presented. The case without node characterisations is a subcase where no node is reducible. The corresponding branches could then be omitted. For the description the same terminology for level_from, index_from, level_to and index_to as in Sec. 4.4.6 is used. Additionally, the term to_counter (from_counter) is used as counter for the aggregate matrix entries (counter for the entries of the matrix to be aggregated). The term vector means the iteration vector of the system to be aggregated and coarse_vector is the vector in the aggregate (i.e. the aggregation of the iteration vector which was calculated in Sec. 4.4.6). For the description of the recursion it is useful to look at Fig. 4.18 (now for MTBDD *trans*). The meaning of the partition is the same as described in Sec. 4.4.6.

The algorithm is given in Alg. 18 and is explained as follows. As the traversal is skipped when section 3 in Fig. 4.18 is reached, it is sufficient to distinguish between section 1 and section 2 of the MTBDD. This will be done by a flag coarse_offsets_change that is true for section 1 and false for section 2. When a zero node is reached, the traversal can be skipped. This is done in line 1-2. If the traversal has reached the border between section 2 and section 3 of the MTBDD, namely level_to, the current matrix value is multiplied with the corresponding vector entry and added to the currently processed aggregate by the function addCurrent (cf. Alg. 19) called in line 4. Lines 5-14 distinguish section 1 and section 2 of the MTBDD and additionally update the to_counter when another non-reducible node in the aggregation level is reached. A further recursion step is necessary if either the current level is not equal level_to or it is a non-reducible node in the aggregation level (this is checked in line 15). The recursion is performed in the subroutine recurse (cf. Alg. 20). Lines 17-19 handle the case of skipped aggregation nodes. Then the from_counter has to be updated by the number fo non-reducible nodes that are below this node. After the recursion the diagonal vector can be updated. In the non-reducible case the normalisation step has to be made before updating the diagonal (line 22). For reducible nodes, the diagonal can be updated directly (line 25).

The addCurrent routine given in Alg. 19 is explained as follows. For a terminal node its value can be directly used as the matrix entry (line 1-2). In the case of a non-reducible node the corresponding counter is pre-incremented in line 5 and then the value is read from the array of aggregated values for this aggregation level in line 6. If the node is reducible, the precalculated value is taken (line 7-8). The actual update of the current aggregated value is done in line 11. there the value determined in line 1-10 is multiplied by the corresponding vector entry determined by roff.

The recursion for the matrix aggregation algorithm is given in Alg. 20. As the aggregation algorithm is defined on row (i.e. s) variables and the only offset parameters in the algorithm are

Algorithm 18 aggmatrix(node, roff, c_roff, level)

```
1: if node = ZERO then
2:    return
3: else if  level = level_from then
4:    addCurrent(node, roff)
5: else if level > level_to then
6:    coarse_offsets_change = FALSE
7: else if level = level_to then
8:    coarse_offsets_change = FALSE
9:    if type = NON_REDUCIBLE then
10:      to_counter = to_counter+1
11:   end if
12: else if level < level_to then
13:   coarse_offsets_change = TRUE
14: end if
15: if (level != level_to) or (type = NON_REDUCIBLE) then
16:   recurse(node, roff, c_roff, level)
17: else if type = REDUCIBLE or DIAG then
18:   adjust from_counter
19: end if
20: if level = level_to then
21:   if type = NON_REDUCIBLE then
22:      matrix_aggregations[index_to][to_counter]=
       matrix_aggregations[index_to][to_counter]/coarse_vector[coarse_row_offset]
23:      update diagonal vector
24:   else if type = REDUCIBLE then
25:      update diagonal vector
26:   end if
27: end if
```

Algorithm 19 addCurrent(node, roff)

```
1: if  level = terminal_level then
2:    value=(node→val)
3: else
4:    if type = NON_REDUCIBLE then
5:       from_counter = from_counter+1
6:       value=aggregations[index_from][from_counter]
7:    else if type = REDUCIBLE then
8:       value =node→reducible_value
9:    end if
10: end if
11: aggregations[index_to][to_counter]=
    aggregations[index_to][to_counter]+vector[roff]*value
```

Algorithm 20 recurse(node, roff, c_roff, level)

```
1: e_node = node→else
2: aggmatrix(e_node→else, level+1, roff, c_roff);
3: aggmatrix(e_node→then, level+1, roff, c_roff);
4: t_node = node→then
5: if coarse_offsets_change then
6:    c_roff = c_roff + node→off[index_to]
7: end if
8: aggmatrix(t_node→else, level+1, roff+node→off[index_from], c_roff);
9: aggmatrix(e_node→then, level+1, roff+node→off[index_from], c_roff);
```

row offsets, column (i.e. t) variables are not important for the offset calculations. Therefore, the offsets used in line 2 and 3 (line 8 and 9) coincide, only the MTBDD node is changed. The else case for the next row variable is processed in line 1-3. When the then edge of the row variable is taken, different offset updates depending on the current part of the MTBDD are possible. In section 1 in Fig. 4.18 (coarse_offsets_change is TRUE) c_roff is updated in line 6. For section 2 in Fig. 4.18 c_roff is left unchanged. All settings for the next column variable are processed in line 8 and 9.

4.5. MTBDD with sparse submatrix representation

4.5.1. Notation and MTBDD basics

Without loss of generality, it is assumed that all diagonal elements of the transition rate matrix R are zero (the row sums needed for the generator matrix Q are stored separately as a vector). The number of reachable states of the Markov chain is denoted by $S := |\mathcal{S}|$.

4.5.2. Sparse matrix representation

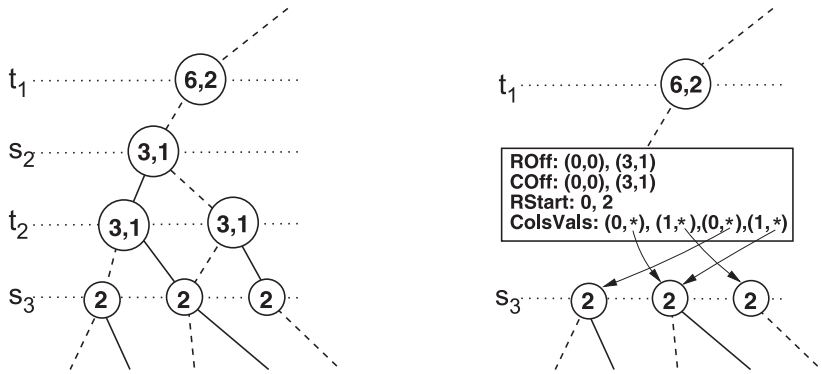
In order to speed up the traversal of the symbolic data structure trans, parts of it (as much as memory allows) may be replaced by sparse matrix structures. Replacing sub-MTBDDs below the first aggregation level a_N by sparse matrices is straight-forward (following the replacement scheme originally described in [47]), described in [54]. However, the achievable gain in speed was quite limited, as only the lowest parts of the MTBDD could be replaced. Therefore, in [55] we presented how to replace parts of the MTBDD that lie between two aggregation levels, which is more complicated, since offset information for more than one system has to be included in the sparse matrix representation.

Intermediate sparse matrix substitution can be seen as an approach that is orthogonal to the multilevel approach.

The following briefly recalls the principle of sparse matrix data structures [46, 59]: A common sparse representation for a square $S \times S$ matrix with NZ non-zero entries uses three arrays: Vals, Cols and RStart. The array Vals (of size NZ) contains all non-zero entries of the matrix, ordered by row. Array Cols is also of size NZ, its position i contains the column index of the corresponding entry in Vals. The array RStart of size S contains pointers into Vals and Cols, such that RStart[i] denotes the beginning of row i.

In the context of the multilevel method, every non-zero value of a sparse matrix block must be associated with an array of row offsets and an array of column offsets. These arrays may not coincide, since for non-diagonal blocks the reachability structure for rows and columns may be different. Therefore, the sparse matrix storage scheme is extended by additional data structures, whose identifiers and roles are as follows:

- ROff is a list of row multi-offsets. The length of the list equals the number of reachable rows in the current block.
- COff is a list of column multi-offsets. The length of the list equals the number of reachable columns in the current block.
- RStart is used in the standard way. It is an array of indices into the array ColsVals. Its length is the number of reachable rows in the current block.
- ColsVals combines the roles of the above mentioned Cols and Vals, but concerning the column index, there is one more degree of redirection than in the standard case. It is an array of index-pointer pairs. Its length is the number of non-zero entries in the current block. The index does not directly store the column index, it is an index into the array COff. The pointer is a pointer to a rate value or to the anchor node of a sub-block.



(a) before substitution (b) after substitution

Figure 4.19.: Part of matrix substitution process for the running example

This general scheme is now explained by the running example (Fig. 4.12a). The aggregation level is s_3. In this example, MTBDD levels s_2 and t_2 should be replaced by sparse multi-offset matrices (i.e. these matrices are of dimension 2×2). From Fig. 4.12a we conclude that the matrices have to contain multi-offset information for two systems, namely the fine system and the first (and in this case only) aggregated system. For the discussion, we focus on the part of the MTBDD shown in Fig. 4.19a. The replacement of s_2 and t_2 is given in Fig. 4.19b and reads as follows: $\mathsf{ROff} = (0,0);(3,1)$ means that there are two non-zero rows. For the fine system, the row offsets are 0 and 3, whereas for the aggregated system the row offsets are 0 and 1. $\mathsf{RStart} = 0,2$ means that the first non-zero row contains 2 entries, and the second non-zero row contains the remaining (in this case also 2) entries. $\mathsf{ColsVals} = (0,*),(1,*),(0,*),(1,*)$ means that the matrix contains 4 non-zero entries. The column offsets for a particular entry are determined by looking up the corresponding element of COff, i.e. for the value 0 the column offsets for the fine/aggregated system are 0/0, while for the value 1 the column offsets are 3/1. The actual entries can be found by following the corresponding arrows emanating from the $*$ symbol. In Fig. 4.20, the resulting overall data structure is shown.

4.6. Parallel implementation

A parallel version of the symbolic multilevel algorithm was developed under the constraint that only a minimum of synchronisations between the different threads should be necessary. All algorithms presented here are general enough to work with an arbitrary number of threads. The thread model for the implementation is given in Fig. 4.21. The dispatcher thread performs the V-cycle algorithm and delegates the time-consuming tasks to the worker threads (we assume there are MAX worker threads). The dispatcher uses barrier synchronisation to unite the program flow after a certain job has been performed.

The parallelisation of the symbolic multilevel algorithm has two different branches, namely parallel vector- and parallel matrix operations. The branches will be treated in the following sections.

4.6.1. Parallel vector operations

The following vector operations were parallelised (v_i denotes the vector element at index i):

- Reset ($v_i = 0$)
- Inversion ($v_i = 1/v_i$)
- Aggregation

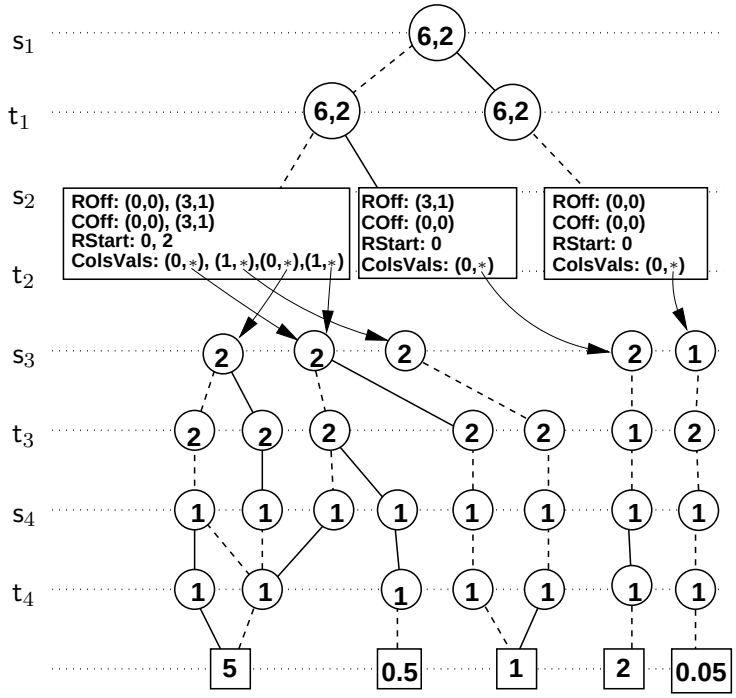


Figure 4.20.: Multi-offset-labelled MTBDD with sparse submatrices for the running example

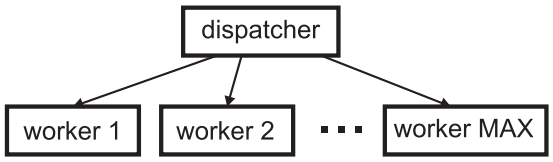
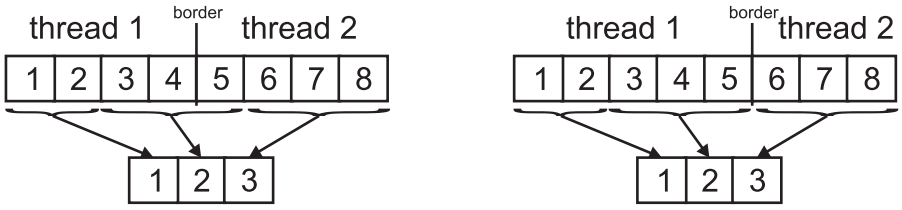


Figure 4.21.: Thread model of the parallelisation



(a) Symmetric case (b) Adjusted border

Figure 4.22.: Parallel vector aggregation

- Disaggregation
- Multiplication by inverted Diagonal and Overrelaxation

The parallelisation of the vector operations is straight-forward. The vector length of the current operation is divided up into MAX parts. Under the constraint that we want to have a minimum number of synchronisations, the situation shown in Fig. 4.22 has to be taken into account. Using the fraction to divide the vector into pieces might have the problem that data dependencies occur (aggregated value 2 in Fig. 4.22a) has to be updated both by thread 1 and 2, i.e. a write conflict occurs). To eliminate these dependencies we adjust the different parts of the vector according to the aggregations to be performed (as seen in Fig. 4.22b). This approach was taken as for large vectors slight differences between the lengths for the different threads are negligible and conflicts are not possible in this setup. All such borders between the subvectors for the different

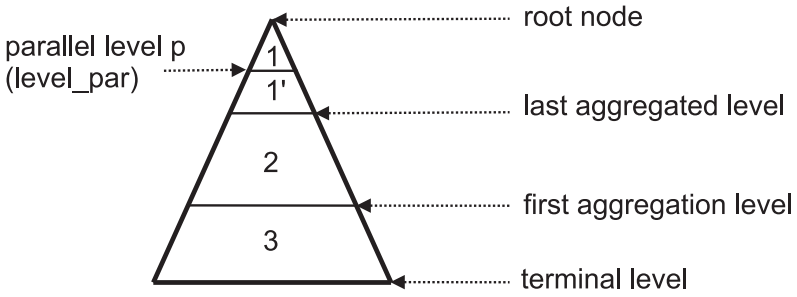


Figure 4.23.: Partition of MTBDD *trans*

threads can be pre-calculated, as there are only MAX values per aggregation to be stored. Of course, using this concept of borders, all other vector operations can be treated in a similar way. Although it would not always be necessary to use these borders (e.g. the reset operation would work with any choice for the borders), the same borders are used for all vector operations.

4.6.2. Parallel matrix operations

For every matrix aggregation, the MTBDD is traversed up to the aggregation level, the current aggregate is calculated by traversing the sub-MTBDDs below the aggregation level and the aggregates are stored in a vector to allow for a compact storage of the aggregated matrices. As long as the MTBDD is traversed in the same way, the order of the aggregated values will not change (e.g. always depth-first). This fact is also exploited in the algorithm for the smoothing (i.e. Jacobi) steps.

When different threads come into play, the ordering of the traversed aggregated nodes may change. Therefore the algorithm has to be changed to get a deterministic correct parallel version, still under the constraint to spend as little memory as possible for the arbitration of the different jobs to the different threads.

There are two matrix operations to be parallelised:

- matrix-vector-product (for the smoothing steps)
- matrix aggregation

The difference of the two regarding the parallelisation are the data collisions that may occur (cf. Sec. 4.6.2.2, 4.6.2.3). The basic idea in parallel matrix algorithms is to split the matrix in blocks that can be used for independent calculations. We first introduce a parameterised splitting method and later introduce a heuristics to adequately choose the splitting parameter.

As the MTBDD data structure represents a natural block structure of the matrix, one can use a corresponding MTBDD-level p (also called level_par) to induce a certain block structure of the matrix. The basic scheme is given in Fig. 4.23. The MTBDD is split into the following regions: 1 and 1' are above the last aggregation level, 2 is between the last and the first aggregation level and 3 is below the first aggregation level. The parallel level splits the region above the last aggregation level into part 1 which defines the block structure and the subblocks 1' to the last aggregation level.

Starting with such a block level p, the calculations in an MTBDD-setup are straight-forward:

- the dispatcher process starts at the MTBDD root and performs a depth-first search up to level p (i.e. part 1 in Fig. 4.23), calculating the offsets of the traversed path-prefixes.
- at level p, a job can be defined by the dispatcher process. It consists of the calculated offsets and the reference to the corresponding sub-MTBDD starting at the current node.

The dispatcher also has to take care at which position in the array of aggregated values the job is located. This is resolved in Sec. 4.6.2.4.

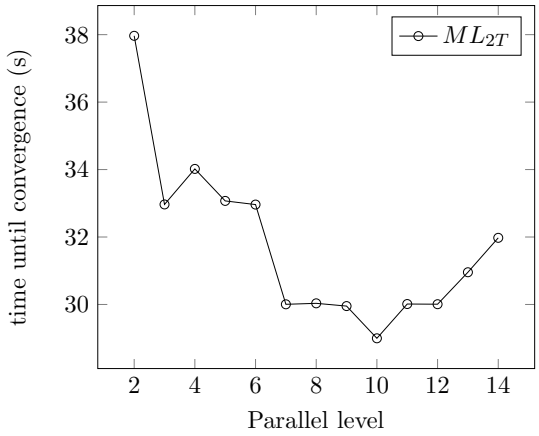
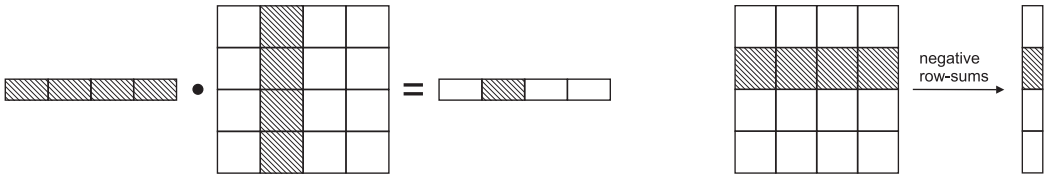


Figure 4.24.: Parallel level for FMS-6



(a) Vector matrix multiplication (b) On-the-fly diagonal

Figure 4.25.: Data collisions

4.6.2.1. Right choice of the parallel level

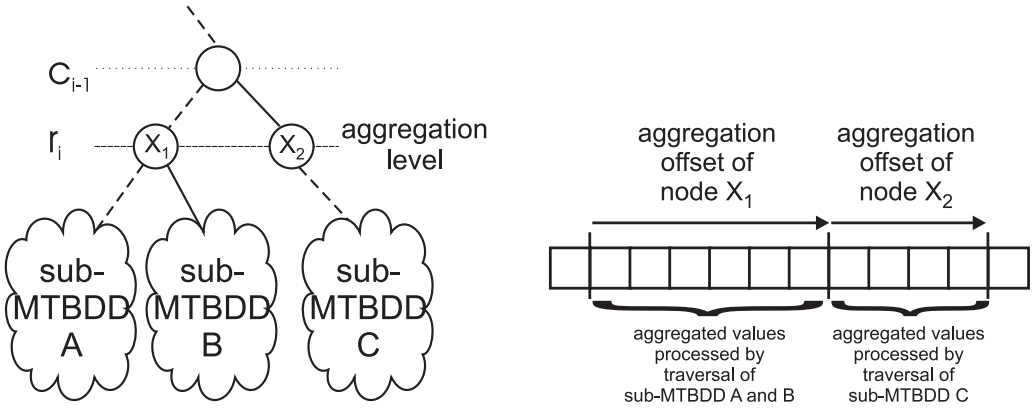
It has to be noted, that the right choice of the parallel level is not entirely trivial. If it is too close to the root node, the blocks are too coarse therefore the scheduling of the jobs to the worker threads might not be optimal. Otherwise, if it is too close to the last aggregation level, the different jobs on the last aggregation level are very small (i.e. only 2×2 matrices for one level above the last aggregation level), thus also leading to bad performance. The impact of the parallel level on the solution time is given in Fig. 4.24 for the Flexible Manufacturing System model (cf. Sec. 4.7.2) with parameter 6 and two worker threads (root node at level 1, last aggregation level at level 15). The basic picture stays the same for other models, so as a heuristics one can choose the parallel level such that the levels 1 and 1' are divided as 2:1.

4.6.2.2. Data collisions - vector matrix multiplication

The symbolic multilevel algorithm in its current version uses Jacobi iterations as smoothing steps. The vector-matrix multiplication part is given in Eq. 4.7, where R is the rate matrix, i.e. the generator matrix Q without the diagonal entries.

$$x' = x \cdot R \tag{4.7}$$

The situation under an assumed block structuring of the matrix in 16 blocks is given in Fig. 4.25a. Therefrom it is obvious that all blocks from the same block column of R will affect the same block in the result vector. To resolve data collisions it is therefore necessary to prevent concurrent threads from processing matrix blocks of the same block column. In terms of the currently processed system in the MTBDD it is therefore sufficient to look at the column offset of a block to determine whether a conflicting block is currently processed by another thread. In order to avoid of keeping track of the column offsets of the currently processed blocks, a different approach is used: a certain column offset can only be processed by a fixed task.



(a) MTBDD example (b) Vector of aggregates

Figure 4.26.: The use of aggregation offsets

4.6.2.3. Data collisions - matrix aggregation

During the aggregation of a matrix, the diagonal of the aggregated generator matrix is generated on the fly and is stored in a separate vector. As the diagonal elements are defined as negative row sums, the data collisions can be sketched as shown in Fig. 4.25b. They are different from those in Sec. 4.6.2.2 as they depend on the block row of the matrix. In MTBDD terms this corresponds to the row offset of the corresponding part of the MTBDD.

4.6.2.4. Scheduling jobs to different tasks

To solve the collision problems from Sec. 4.6.2.2 and Sec. 4.6.2.3, a queue-based system to communicate the job information from the dispatcher thread to the worker threads is introduced. Every worker thread has its own job queue, that only contains jobs that do not depend on the results of other threads. The queues are made in a double-blocking way: If the queue is full, the dispatcher blocks on writing a job to the queue and conversely if the queue is empty, the worker thread blocks on reading a job from the queue. Depending on the matrix operation used, either block columns or block rows in the matrix induce data collisions. Using a queue-based arbitration of the jobs, the question is how to schedule the matrix blocks to the different worker threads. Without having to maintain a list of the blocks currently processed by the different threads, an efficient way to avoid data collisions is to use hash functions on the block offset values (row offset for the aggregation, column offset for the matrix-vector-multiplication). A simple hash function is provided in Sec. 4.6.2.5. Every job for the aggregation or iteration needs the information, at which position in the array of aggregated values it should start. The dispatcher can provide this information using the concept of aggregation offsets. (i.e. the number of irreducible nodes below a certain aggregation node). An example is given in Fig. 4.26. Node x_1 has two sub-MTBDDs A and B, node x_2 has only one sub-MTBDD C. Let the dispatcher traverse the MTBDD only up to the aggregation level (which coincides in this case with the parallel level p). Reaching node x_1 a job can be defined and put into a worker's queue. For the dispatcher to continue independently of any worker it is important to know the offset induced by processing x_1. With this knowledge, the next job can be defined. As long as every worker thread maintains its own current position in the array of aggregated values and the starting positions are correct, no data collisions can occur (given that the dispatcher knows the right offsets).

In the more general case, the parallel level is above the last aggregation level. In this case processing a sub-MTBDD for a certain aggregation may affect both the array of aggregated values for the aggregated system (at level_to) and the array of aggregated values for the system to be aggregated (at level_from). The solution in this case is to add additional information to the nodes in the parallel level. For every aggregation they are annotated by two offsets: One offset

for the array of aggregated matrix values in level_to, one for the array of aggregated matrix values in level_from. These offsets can be calculated from the aggregation offsets introduced in Sec. 4.4.5 by Alg. 21. Suppose global variables offset_to and offset_from are initialised with 0, global variable agg_level is the MTBDD level of one of the aggregations. The initial call is $addOffsets(node, par\_level)$, where node is a node in the parallel level (these offsets have to be calculated for every node and aggregation pair). The algorithm works as follows. Lines 1-3 terminate the recursion if a zero-valued node is found. In lines 4-8 the actual update process takes place. If the recursion as reached the current aggregation level, then offset_to is incremented if the node is non-reducible (i.e. then there is a value for this node in the array of aggregated matrix values). In any case, offset_from is updated by the node's aggregation offset in line 8. Line 9-20 perform the recursive descent in such a way, that the parameter *node* is always a node in a row level of the MTBDD *trans*.

Algorithm 21 addOffsets(node, level)

```
 1: if node=ZERO then
 2:    return
 3: end if
 4: if level=agg_level then
 5:    if type = NON_REDUCIBLE then
 6:       offset_to = offset_to + 1
 7:    end if
 8:    offset_from = offset_from + node→aggregation_offset
 9: else
10:    e_node = node→else;
11:    if e_node != ZERO then
12:       addOffsets(e_node→else, level+1)
13:       addOffsets(e_node→then, level+1)
14:    end if
15:    t_node = node→then
16:    if t_node != ZERO then
17:       addOffsets(t_node→else, level+1)
18:       addOffsets(t_node→then, level+1)
19:    end if
20: end if
```

The worker threads main loop is sketched in Alg. 22.

Algorithm 22 work()

```
1: while not done do
2:    get operation
3:    process operation
4:    wait for other threads to finish
5: end while
```

4.6.2.5. Hashing function

If it can be assured, that the same block row/column is processed by the same thread, no further synchronisation has to be performed. As a simple hash function we use the following operation: `(offset>>bits)%MAX` where `>>` is the bit shift operator that shifts the binary representation of offset by the number of bits that is specified by the value of *bits* and `%` means the modulo operation. It remains to determine the number of bits to be shifted. As the matrix Q is used in a transposed way for the Jacobi routine but in the normal way for the aggregation routine, it is convenient to consider both the normal and the transposed case (in general there will be different results - as long as the matrix is not symmetric). The runtime for different bitshift parameters

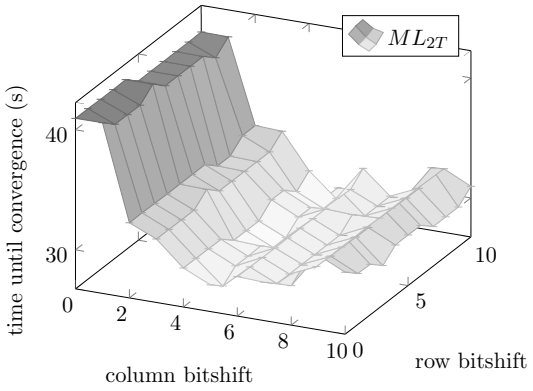


Figure 4.27.: Runtime for different bitshift parameters (FMS-6)

from 0 to 10 for the Flexible Manufacturing System model (cf. Sec. 4.7.2) with parameter 6 and two worker threads is shown in Fig. 4.27. The minimum runtime is achieved for column bitshift 5, row bitshift 9. The low impact of the row bitshift is due to the fact that in the experiment for each level eight Jacobi iterations and one aggregation have been performed.

Of course, it is not feasible to determine the best bitshift parameters by brute force. Therefore, the following heuristics is proposed: Under the assumption that every job needs the same average time for its processing, one can reason about the balance of the jobs for a different number of bits to be shifted. By counting the number of jobs assigned to the different worker threads, we get a certain sample for each number of bits.

The sample mean $\bar{x}$ is the MAXth fraction of the total number of jobs. The sample variance is then calculated as $\frac{1}{n-1}\sum_{i=1}^{\texttt{MAX}}(x_i - \bar{x})^2$.

The number of bits that correspond to the least sample variance will be used during the algorithm. Of course, the optimum would be a sample variance of 0. So we search the bit-shift that allows for the least sample variance. An example (FMS model, scaling parameter 6) for some calculated variances can be seen in Tab. 4.2

	sample variance	
bit shift	row-offset	column-offset
0	4900.5	4900.5
1	4900.5	4900.5
2	312.5	312.5
3	1984.5	3120.5
4	4.5	24.5
5	40.5	180.5
6	12.5	144.5
7	*0.5*	24.5
8	84.5	*12.5*
9	924.5	1012.5
10	264.5	544.5

Table 4.2.: Sample variance for some bit shifts

The first column shows the different bit shifts that have been applied. The second column shows the sample variance calculated from the shifted row offset (i.e. the shift used for the aggregation procedure) and the third column shows the sample variance calculated from the shifted column offset (i.e. the shift used for the Jacobi procedure). It can easily be seen that the best bit shift for the aggregation of the matrix is the bit shift of 7 bits, where the optimum for the Jacobi procedure is a bit shift of 8 bits. In Fig. 4.27 it can be seen that the parameters are not optimal, but still in the lower regions (optimal runtime is 28.05 s, runtime with the

heuristically determined parameters is 30.00 s). Note that, as the shape of the matrices does not change it is sufficient to calculate the variances only once as a precalculation step for the multilevel cycles.

4.7. Experimental results

In this section three standard case studies are treated. For each model, two tables are shown: One is the comparison between the multilevel algorithm and two standard numerical algorithms (measured on our Xeon server). The other shows the speedup of the parallel version of the algorithm for one, two, four and eight worker threads (measured on the SGI Altix).

Unless stated otherwise, the multilevel algorithm uses 4 pre- and 4 post-smoothing steps, respectively, on the fine system and 8 pre- and 8 post-smoothing steps, respectively, for the aggregated systems using V-cycles.

The following notation is used to denote the different versions of the algorithm:

ML The multilevel algorithm that does not use node characterisation (cf. Sec. 4.4.2.2). All nodes are aggregated, regardless of their possible reducibility. Vector operations use the BDD *reach* to determine the offsets. This is a bottleneck when *reach* grows and no sparse substitutions are made.

ML$_{red}$ The multilevel algorithm that uses precalculations (cf. Sec. 4.4.3) for the aggregated matrix values. Index vectors are allocated to perform faster vector operations (BDD *reach* is only traversed once to calculate the index information).

JOR Jacobi OverRelaxation. This is a standard algorithm shipped with the model checker PRISM. We use the default relaxation parameter $\omega = 0.9$

PSOR Pseudo Successive Overrelaxation. This is the overrelaxed variant of the Pseudo-Gauss-Seidel algorithm. Like for JOR $\omega = 0.9$ is used. This algorithm is also shipped with PRISM. It uses a block-fashioned variant of the overrelaxed Gauss-Seidel algorithm that needs only a small second iteration vector.

ML$_{xT}$ Parallel implementation of the multilevel algorithm, where x denotes the number of worker threads (cf. Sec. 4.6). As a default we used a queue size of 30 jobs per worker thread.

Every table has two subtables with different levels of sparse matrix substitutions. The number of MTBDD variable levels is given by the column *levels*. The *moderately sparse* subtable only substitutes relatively few variable levels by sparse matrices or intermediate sparse matrices while in the *full sparse* subtable virtually every variable level is replaced by sparse matrices. The number of substituted variables is given in the column *sparse levels*. If a sum is denoted, the left summand determines the sparse levels next to the terminal nodes, while the right summand determines the sum of intermediate sparse levels. Column *memory* is the total memory consumption of the algorithm. For all experiments the sparse substitutions next to the terminal levels are equal, so the difference in the memory consumption between moderately and full sparse is the memory used for intermediate sparse matrices. In the parallel algorithms current implementation it is not always possible to substitute all variable levels by sparse matrices: The levels from the parallel level to the last aggregation level cannot be replaced so far. Moreover, in the tables for the multithreaded algorithm, the column *time* is split into a column *iterations* and the *setup*, as the setup phase has not been parallelised up to now.

The tables for the sequential algorithms are structured as follows: Column *scaling* is the scaling parameter of the model (e.g. the queue size length for the tandem queueing network), column *states* and *transitions* show the model characteristics for the given parameter, that is the

term	explanation
matrix	MTBDD nodes for *trans*
sparse substitution	Sparse matrix replacements of MTBDD variables starting from terminal nodes
block substitution	Block matrix replacements of MTBDD variables starting from root node
inter. sparse substitution	Intermediate sparse matrices
diagonal	Diagonal elements of matrix Q
iteration vector	First and second iteration vector
temp vec.	Temporary iteration vector
matrix i	Matrix elements for i-th aggregate
diag i	Diagonal vector for i-th aggregate
sol i	First and second iteration vector for i-th aggregate

Table 4.3.: Notation used for the memory consumption

size of the reachable part of the state space and the number of reachable transitions. The fourth column is the *algorithm* used for the calculations. In case of the multilevel a tuple with the aggregation levels is added. Column *ML-cycles* is only relevant for the multilevel algorithms. It is the number of V-cycles until convergence. The column *steps* determines how many smoothing steps are performed on the fine (i.e. non-aggregated) system. To give an idea of the accuracy of the result, column *residual* is the maximum norm of the residual vector $\pi' \cdot Q$, where π' is the calculated solution vector. Column *MTBDD levels* gives the number of variables needed to store a state (or equivalently a row- or column index in the corresponding matrix).

The tables for the results of the parallel algorithm are structured in a similar way. As the states and transitions remain the same as for the sequential algorithm (with the corresponding scaling parameter), these two columns are omitted. An additional column *parallel level* was introduced. This is the level of the MTBDD from where the algorithm uses more than one worker thread, if possible (cf. Fig. 4.23).

Due to a bug in the algorithm used for [55], the results there show a different number of iterations (mainly more iterations and thus longer runtime until convergence). For the termination criteria, PRISMs standard was used, namely that the relative error between two consecutive iterations on the unaggregated system should be below $\epsilon = 1.0 \cdot 10^{-6}$.

4.7.1. Memory considerations

In this section we will qualitatively describe the memory consumption of the multilevel algorithm compared to the memory consumption of two standard algorithms. A description of the notation is given in Tab. 4.3.

The memory considerations sketched in Fig. 4.28 are explained as follows:

JOR The overrelaxed variant of the Jacobi algorithm needs, of course, the rate matrix stored as MTBDD (with optional sparse matrix substitutions of MTBDD variables starting from the terminal level), its negative rowsums (the diagonal vector) and two iteration vectors of reachable state space size.

PSOR In contrast to JOR, the overrelaxed variant of the Pseudo Gauss-Seidel algorithm needs only one iteration vector of full reachable state space size, the other iteration vector is considerably smaller. Additionally it uses substitutions of the upper MTBDD variables, the block substitutions.

ML The ML algorithms use the JOR algorithm as smoothing steps, so the memory consumption is based on the consumption of JOR. Additionally there is memory for intermediate

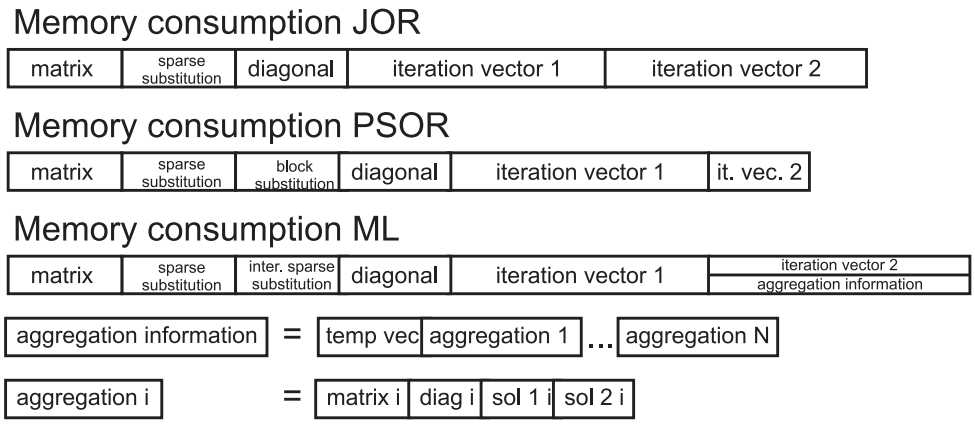


Figure 4.28.: Memory consumption

sparse matrix substitutions. The part of the memory labelled (iteration vector 2/aggregation information) is the maximum of both memory demands. Aggregation information consists of a temporary iteration vector of the size of the first aggregated system's reachable state space and additional aggregation information for each aggregated system. The aggregation information for one aggregated system consists of the aggregated matrix entries, the diagonal vector and two iteration vectors. ML_{red} has a lower memory demand for the aggregated matrix values than ML (due to the characterisation of reducible and non-reducible nodes). The parallel variants have a slightly higher memory demand for the partition of the iteration vectors into separate parts, the offset information for the nodes in the parallel level and the job queues for the matrix operations.

4.7.2. Flexible Manufacturing System (FMS)

This example is one of the standard benchmark case studies which are available from the PRISM web page [48] and is based on the model published in [18]. The model consists of three machines where one machine produces one certain part (denoted by part1), the second machine can produce two different parts (denoted by part2 and part3), and the third machine produces a new part (denoted by part12) out of part1 and part2 provided by the first and the second machine. The scaling parameter is the initial number of raw parts for each machine.

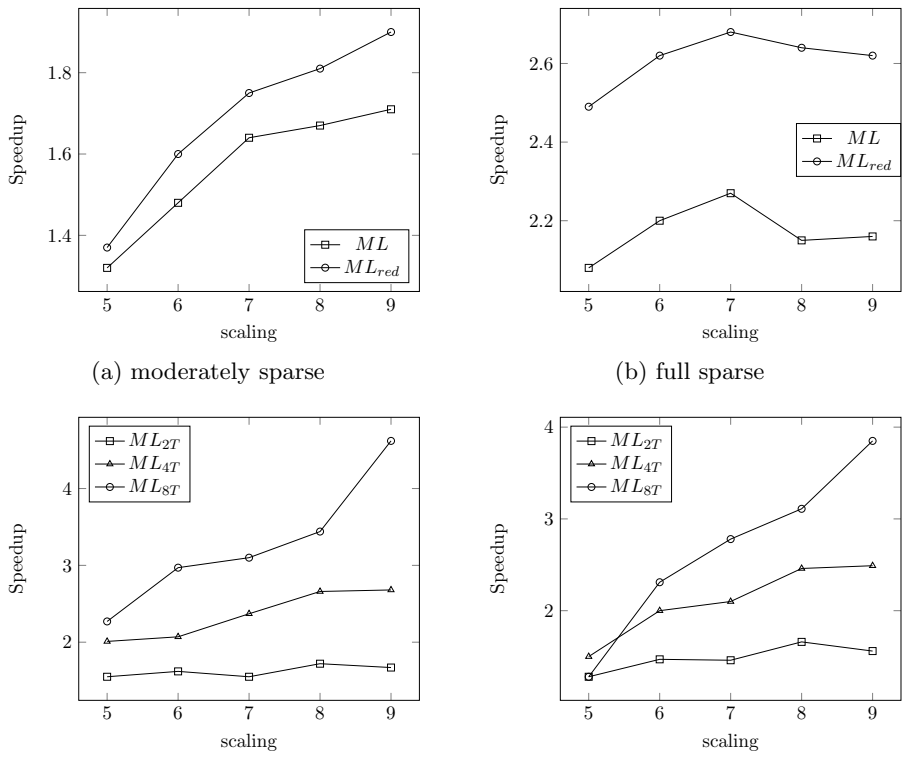
The current version of the multilevel algorithm uses a fixed ordering of the aggregations. For the experiments, the ordering of submodels is always (part2, part1, part12, part3) for the generation of the MTBDD (i.e. part2's MTBDD variables are located next to the root node, part3's variables are located next to the terminal nodes). The state variables of the model are ordered as given in Tab. 4.4.

A submodel-based aggregation is used, induced by the ordering of the submodels. The algorithm therefore will first aggregate part3, then part12, then part1 and finally part2. For example, the model with scaling parameter $n = 5$ has 55 variables in total, consisting of 14 for part2, 19 for part1, 13 for part12 and 9 for part3. Going up 9 variables from the terminal level (which is one below level 55), we end up at level 47 after the aggregation of part 3. Aggregating the 13 variables of part12 corresponds to the aggregation level 34 and aggregation level 15 corresponds to the aggregation of part1, which consists of 19 levels. Proceeding this way one sees that the aggregation levels for scaling parameter $n \in \{5, 6, 7\}$ are (47,34,15), while scaling parameters $n \in \{8, 9\}$ use aggregation levels (59,43,19).

The faster standard algorithm in this case is JOR, so the speedup between JOR the ML algorithms is depicted in Fig. 4.29a for the moderately sparse case and in Fig. 4.29b for the full sparse case.

submodel	variable	range	bits
part2	P2	[0..n]	$\lceil \log_2(n+1) \rceil$
	P2wM2	[0..n]	$\lceil \log_2(n+1) \rceil$
	P2M2	[0..1]	1
	P2s	[0..n]	$\lceil \log_2(n+1) \rceil$
	P2wP1	[0..n]	$\lceil \log_2(n+1) \rceil$
	M2	[0..1]	1
part1	P1	[0..n]	$\lceil \log_2(n+1) \rceil$
	P1wM1	[0..n]	$\lceil \log_2(n+1) \rceil$
	P1M1	[0..3]	2
	P1d	[0..n]	$\lceil \log_2(n+1) \rceil$
	P1s	[0..n]	$\lceil \log_2(n+1) \rceil$
	P1wP2	[0..n]	$\lceil \log_2(n+1) \rceil$
	M1	[0..3]	2
part12	P12	[0..n]	$\lceil \log_2(n+1) \rceil$
	P12wM3	[0..n]	$\lceil \log_2(n+1) \rceil$
	P12M3	[0..2]	2
	P12s	[0..n]	$\lceil \log_2(n+1) \rceil$
	M3	[0..2]	2
part3	P3	[0..n]	$\lceil \log_2(n+1) \rceil$
	P3M2	[0..n]	$\lceil \log_2(n+1) \rceil$
	P3s	[0..n]	$\lceil \log_2(n+1) \rceil$

Table 4.4.: FMS model - MTBDD variables



(a) moderately sparse

(b) full sparse

(c) moderately sparse (parallel)

(d) full sparse (parallel)

Figure 4.29.: FMS model: Speedup of ML versus JOR

								moderate sparse			full sparse		
scaling	states	transitions	algorithm	ML-cycles	steps	residual	variable levels	sparse levels	memory (kB)	time (s)	sparse levels	memory (kB)	time (s)
5	152712	1111482	ML(47,34,15)	19	152	$2.4834 \cdot 10^{-10}$	55	9	4055.5	26.41	9+46	4223.1	11.26
			ML_{red}(47,34,15)	19	152	$2.4834 \cdot 10^{-10}$	55	9	4544.3	25.40	9+46	4711.9	9.43
			JOR	-	996	$5.5554 \cdot 10^{-10}$	55	27	4789.8	34.87	55	8523.2	23.45
			PSOR	-	936	$5.5231 \cdot 10^{-10}$	55	27+17	3628.5	38.20	33+22	4303.7	33.82
6	537768	4205670	ML(47,34,15)	24	192	$2.4731 \cdot 10^{-11}$	55	9	11628.6	100.12	9+46	11944.6	48.03
			ML_{red}(47,34,15)	24	192	$2.4731 \cdot 10^{-11}$	55	9	12446.0	92.40	9+46	12762.0	40.36
			JOR	-	1189	$3.9511 \cdot 10^{-10}$	55	24	12586.2	147.74	55	28554.3	105.66
			PSOR	-	1124	$3.8807 \cdot 10^{-10}$	55	24+17	8453.2	163.49	33+22	11840.9	150.54
7	1639440	13552968	ML(47,34,15)	29	232	$2.7812 \cdot 10^{-12}$	55	9	32270.7	321.76	9+46	32824.5	173.92
			ML_{red}(47,34,15)	29	232	$2.7812 \cdot 10^{-12}$	55	9	33642.5	302.83	9+46	34196.4	147.55
			JOR	-	1385	$3.0842 \cdot 10^{-10}$	55	22	33017.4	528.45	55	86777.4	394.76
			PSOR	-	1315	$3.0422 \cdot 10^{-10}$	55	22+17	20341.0	579.38	33+22	31787.3	546.82
8	4459455	38533968	ML(59,43,19)	35	280	$1.7353 \cdot 10^{-12}$	70	12	84659.9	1056.31	12+58	85572.6	603.99
			ML_{red}(59,43,19)	35	280	$1.7353 \cdot 10^{-12}$	70	12	87227.7	974.49	12+58	88140.5	491.58
			JOR	-	1582	$2.4266 \cdot 10^{-10}$	70	24	85035.1	1759.28	70	239493.2	1297.75
			PSOR	-	1543	$2.3515 \cdot 10^{-10}$	70	24+17	50406.8	1963.72	42+28	79172.1	1734.65
9	11058190	99075405	ML(59,43,19)	41	328	$1.4684 \cdot 10^{-12}$	70	12	203503.3	2822.06	12+58	204941.2	1746.57
			ML_{red}(59,43,19)	41	328	$1.4684 \cdot 10^{-12}$	70	12	207497.8	2540.85	12+58	208935.7	1438.60
			JOR	-	1782	$1.9119 \cdot 10^{-10}$	70	24	204209.8	4834.75	70	601256.2	3775.74
			PSOR	-	1740	$1.8554 \cdot 10^{-10}$	70	24+17	118200.3	5389.44	34+28	146588.6	5022.77

Table 4.5.: FMS model - ML and standard numerical algorithms

					moderate sparse				full sparse			
scaling	algorithm	parallel level	ML-cycles	residual	sparse levels	memory (kB)	setup (s)	iterations (s)	sparse levels	memory (kB)	setup (s)	iterations (s)
5	$ML_{1T}(47,34,15)$	7	19	$2.4891 \cdot 10^{-10}$	9	5890.7	1.93	34.00	48	6080.0	1.03	9.02
	$ML_{2T}(47,34,15)$					5893.3	1.03	21.97		6082.6	1.92	7.06
	$ML_{4T}(47,34,15)$					5903.7	1.05	16.95		6093.0	1.93	6.01
	$ML_{8T}(47,34,15)$					5945.0	1.04	14.95		6134.3	1.92	7.02
6	$ML_{1T}(47,34,15)$	7	24	$2.4807 \cdot 10^{-11}$	9	14591.9	5.93	128.01	48	14951.5	4.96	44.02
	$ML_{2T}(47,34,15)$					14594.5	5.90	79.01		14954.1	4.98	30.04
	$ML_{4T}(47,34,15)$					14604.9	5.04	61.99		14964.4	4.95	21.98
	$ML_{8T}(47,34,15)$					14646.2	5.95	43.03		15005.8	4.98	19.02
7	$ML_{1T}(47,34,15)$	7	29	$2.7741 \cdot 10^{-12}$	9	37056.8	19.03	405.99	48	37691.0	18.98	163.99
	$ML_{2T}(47,34,15)$					37059.4	19.00	261.92		37693.6	18.08	111.95
	$ML_{4T}(47,34,15)$					37069.8	19.02	171.03		37704.0	18.95	78.06
	$ML_{8T}(47,34,15)$					37111.1	20.07	130.98		37745.3	18.94	59.00
8	$ML_{1T}(59,43,19)$	10	35	$1.7854 \cdot 10^{-12}$	12	93450.7	74.17	1402.96	59	94502.3	73.00	559.01
	$ML_{2T}(59,43,19)$					93453.3	74.02	814.02		94504.9	72.99	337.04
	$ML_{4T}(59,43,19)$					93463.7	74.94	528.04		94515.3	72.09	226.99
	$ML_{8T}(59,43,19)$					93505.0	75.00	408.02		94556.6	73.00	180.02
9	$ML_{1T}(59,43,19)$	10	41	$1.5229 \cdot 10^{-12}$	12	216556.6	194.61	3655.98	59	218220.8	190.96	1663.96
	$ML_{2T}(59,43,19)$					216559.2	192.93	2184.07		218223.4	189.92	1066.03
	$ML_{4T}(59,43,19)$					216569.6	192.05	1363.96		218233.8	188.05	668.03
	$ML_{8T}(59,43,19)$					216610.9	193.01	791.97		218275.1	188.03	432.00

Table 4.6.: FMS model - multithreaded algorithm

submodel	variable	range	bits
serverC	sc	[0..c]	$\lceil \log_2(c+1) \rceil$
	ph	[1..2]	1
serverM	sm	[0..c]	$\lceil \log_2(c+1) \rceil$

Table 4.7.: Tandem model - MTBDD variables

The speedup of the multilevel-algorithm is from 1.3 (1.3) to 1.7 (1.9) for the ML (ML_{red}) algorithm in the moderately sparse case and from 2.1 (2.5) to 2.3 (2.7) in the fully sparse case. For the moderately sparse case the speedup grows with the size of the MTBDD (as in the moderately sparse case iterations on the standard algorithms are slowed down by the low sparse level), in the full sparse case the speedup is much more constant.

The parallel speedup is shown in Fig. 4.29c for the moderately sparse case and in Fig. 4.29d for the full sparse case. Here, the situation for the moderately and full sparse setups are nearly the same: Speedup of ≈1.5 for two threads, more than 2 for four threads and more than three for eight threads (in the larger scalings). One exceptions occurs in the full sparse setup: parameter 5 of the four thread version is as slow as the two thread version. Here the total runtime for the algorithm with one single worker thread is only 9 seconds and therefore the possible speedup resolution is very coarse.

In Tab. 4.5 one sees that the number of total iterations needed is very small compared to the iterations needed for the standard algorithms. Regarding the memory consumption for the *moderately sparse* case, the multilevel algorithm has about the same memory consumption as the JOR algorithm while the PSOR algorithm has a considerably lower memory consumption. In the *fully sparse* setup the picture changes a bit: As it is more efficient to use intermediate sparse matrices than to use large sparse blocks at the bottom of the MTBDD, the memory consumption of PSOR and multilevel are of the same order of magnitude while the memory consumption of JOR is considerably higher.

4.7.3. Tandem Queueing Network

This example is also one of the standard benchmark case studies which are available from the PRISM web page [48] and is based on the model published in [30]. The model consists of two queues, the first one is a $M/Cox_2/1$, the second one a $M/M/1$ queue, both of the same capacity. The scaling parameter is the capacity of the queues.

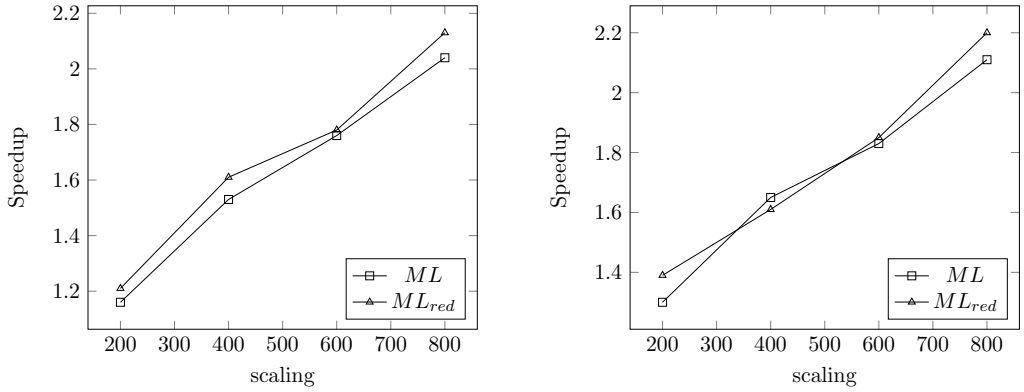
The PRISM specification consists of two submodels, namely the two queues. If only submodel-wise aggregation were applied, the multilevel algorithm would degenerate to a two-level algorithm. In this case, the aggregated matrix would be very small and the multilevel corrections could not efficiently contribute to the solution phase. Therefore, for every experiment with the tandem model, two aggregation levels were used which do *not* correspond to the submodel border given by the model (cf. Tab. 4.7).

An interesting observation for this example is that no significant speedup can be achieved by sparse matrix substitutions, i.e. the runtimes form the moderately sparse to the fully sparse setup do not change very much. This is due to the very small MTBDDs for this model. As the faster standard-algorithm in this case was the PSOR algorithm, the speedup of the multilevel algorithm is calculated with respect to PSOR.

The speedup in the moderately sparse setup is shown in Fig. 4.30a, the speedup in the full sparse setup is shown in Fig. 4.30b. As mentioned above, the moderately and fully sparse cases do not differ very much, so the figures both show the same behaviour.

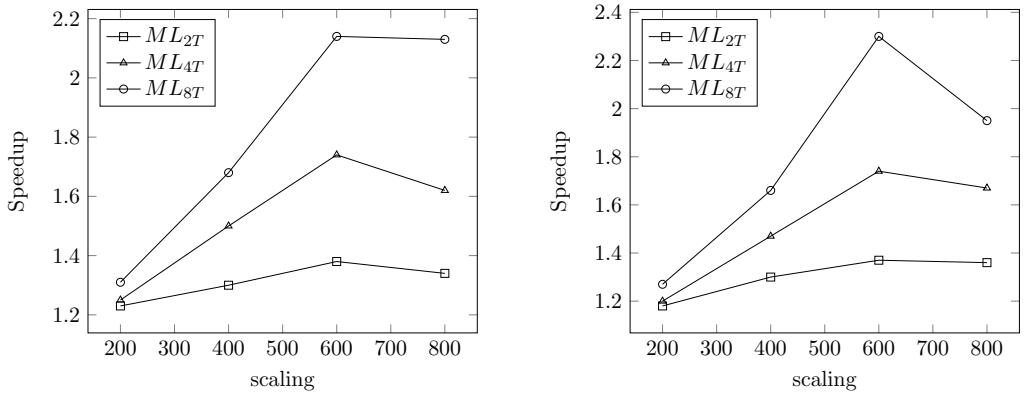
The speedup increases from 1.2 (1.2) to 2.0 (2.1) in the moderately (full) sparse setup. Note that the JOR algorithm, which does the smoothing steps in the multilevel algorithms is considerably slower than the PSOR algorithm.

For the parallel algorithm, the moderately (full) sparse cases in Fig. 4.30a (Fig. 4.30b) also



(a) moderately sparse (b) fully sparse

Figure 4.30.: Tandem model: Speedup of ML versus PSOR



(a) moderately sparse (b) fully sparse

Figure 4.31.: Tandem model: Speedup of multithreaded ML

show basically the same behaviour. The exception in this case is scaling parameter 800 for ML_{8T} there the fully sparse algorithm is slower than the moderately sparse variant.

Regarding the memory consumption, again for JOR and the multilevel algorithms the memory consumption is of the same order of magnitude, while the memory consumption for PSOR is considerably smaller. This is, in contrast to the FMS model, the case for both moderately and fully sparse case (as the MTBDD *trans* is quite small and therefore fully sparse substitution does not consume much memory). For scaling parameters 200 and 400, the aggregation information does not fit into the space provided by the second JOR iteration vector of the fine system, so slightly more memory as for the JOR algorithm is consumed by the ML algorithms.

								moderate sparse			fully sparse		
scaling	states	transitions	algorithm	ML-cycles	steps	residual	variable levels	sparse levels	memory (kB)	time (s)	sparse levels	memory (kB)	time (s)
200	80601	280599	ML(16,12)	110	880	$1.0408 \cdot 10^{-17}$	17	2	2412.5	64.72	2+15	2522.1	57.99
			ML_{red}(16,12)	110	880	$2.7755 \cdot 10^{-17}$	17	2	2072.1	61.98	2+15	2181.7	54.29
			JOR	-	3670	$2.2204 \cdot 10^{-16}$	17	2	1426.4	114.4	17	2601.1	84.36
			PSOR		2624	$2.2204 \cdot 10^{-16}$	17	11+6	868.4	75.23	11+6	868.4	75.23
400	321201	1121199	ML(17,13)	223	1784	$1.1102 \cdot 10^{-16}$	19	3	6364.9	447.30	3+16	6580.8	415.04
			ML_{red}(17,13)	223	1784	$5.5511 \cdot 10^{-17}$	19	3	5844.7	426.51	3+16	6060.6	426.50
			JOR	-	7389	$5.5511 \cdot 10^{-17}$	19	4	5657.2	949.0	21	23255.7	2998.17
			PSOR	-	5274	$1.1102 \cdot 10^{-16}$	19	12+7	3290.7	684.94	12+7	3290.7	684.94
600	721801	2521799	ML(17,13)	515	4120	$2.7755 \cdot 10^{-17}$	21	5	12713.6	1351.48	5+16	12866.9	1296.48
			ML_{red}(17,13)	515	4120	$1.1102 \cdot 10^{-16}$	21	5	12816.2	1333.36	5+16	12969.5	1284.35
			JOR	-	11130	$2.2204 \cdot 10^{-16}$	21	15	13517.5	3055.40	21	23255.7	2998.17
			PSOR	-	7941	$2.2204 \cdot 10^{-16}$	21	13+8	7275.8	2377.48	13+8	7275.8	2377.48
800	1282401	4482399	ML(17,13)	616	4928	$2.7755 \cdot 10^{-17}$	21	5	22567.0	2811.44	5+16	22771.4	2719.37
			ML_{red}(17,13)	617	4936	$1.1102 \cdot 10^{-16}$	21	5	22746.1	2683.75	5+16	22950.4	2599.74
			JOR	-	14880	$1.1102 \cdot 10^{-16}$	21	15	23443.7	7465.32	21	41315.7	7343.33
			PSOR	-	10614	$2.2204 \cdot 10^{-16}$	21	13+8	12821.6	5727.45	13+8	12821.6	5727.45

Table 4.8.: Tandem model - ML and standard numerical algorithms

					moderate sparse				fully sparse			
scaling	algorithm	parallel level	ML-cycles	residual	sparse levels	memory (kB)	setup (s)	iterations (s)	sparse levels	memory (kB)	setup (s)	iterations (s)
200	$ML_{1T}(16,12)$	9	110	$2.7756 \cdot 10^{-17}$	2	2082.9	0.00	131.98	13	2090.0	0.90	121.06
	$ML_{2T}(16,12)$	9				2085.5	0.00	106.99		2092.6	0.00	102.93
	$ML_{4T}(16,12)$	9				2095.9	0.00	105.92		2102.9	0.00	101.02
	$ML_{8T}(16,12)$	9				2137.2	0.00	101.03		2144.3	0.00	94.96
400	$ML_{1T}(17,13)$	9	223	$1.1102 \cdot 10^{-16}$	3	5856.7	0.01	840.00	14	5863.8	0.01	797.98
	$ML_{2T}(17,13)$	9				5859.3	0.01	646.00		5866.4	0.01	611.97
	$ML_{4T}(17,13)$	9				5869.7	0.01	560.96		5876.7	0.01	544.04
	$ML_{8T}(17,13)$	9				5911.0	0.01	501.04		5918.1	0.01	481.00
600	$ML_{1T}(17,13)$	9	515	$1.1102 \cdot 10^{-16}$	5	12829.6	0.02	2267.98	16	12835.8	0.02	2218.93
	$ML_{2T}(17,13)$	9				12832.2	0.02	1649.00		12838.4	0.02	1618.03
	$ML_{4T}(17,13)$	9				12842.6	0.02	1307.04		12848.8	0.02	1274.96
	$ML_{8T}(17,13)$	9				12883.9	0.02	1061.02		12890.1	0.02	964.06
800	$ML_{1T}(17,13)$	9	619	$1.7854 \cdot 10^{-12}$	5	22759.3	0.94	4036.07	16	22767.6	0.94	3947.06
	$ML_{2T}(17,13)$	9				22761.9	0.03	3004.93		22770.2	0.04	2911.95
	$ML_{4T}(17,13)$	9				22772.2	0.94	2486.02		22780.5	0.03	2363.95
	$ML_{8T}(17,13)$	9				22813.5	0.04	1894.99		22821.8	0.04	2022.99

Table 4.9.: Tandem model - multithreaded algorithm

submodel	variable	range	bits
highlevel	servers1	[0..2]	2
	servers2	[0..2]	2
	servers3	[0..2]	2
	servers4	[0..2]	2
	servers5	[0..2]	2
queue1	jobs1	[0..jobs]	$\lceil \log_2(jobs+1) \rceil$
	working1	[0..2]	2
	leaving1	[0..2]	2
queue2	jobs2	[0..jobs]	$\lceil \log_2(jobs+1) \rceil$
	working2	[0..2]	2
	leaving2	[0..2]	2
queue3	jobs3	[0..jobs]	$\lceil \log_2(jobs+1) \rceil$
	working3	[0..2]	2
	leaving3	[0..2]	2
queue4	jobs4	[0..jobs]	$\lceil \log_2(jobs+1) \rceil$
	working4	[0..2]	2
	leaving4	[0..2]	2
queue5	jobs5	[0..jobs]	$\lceil \log_2(jobs+1) \rceil$
	working5	[0..2]	2
	leaving5	[0..2]	2

Table 4.10.: MSMQ HLM model - MTBDD variables

4.7.4. Multi server multi queue model (MSMQ)

This third case study follows exactly the model published in [45]. A similar model had been used in the context of a multilevel algorithm for hierarchical Kronecker structures in [15]. It consists of five clients which are served by two servers in a round robin manner. We use the same parameter set as in [15], that is $\lambda = (0.075, 0.075, 0.225, 0.75, 1.2)$ for the arrival rates to the queues, service rate $\omega = 10.0$ and walk rate $\mu = 1.0$. Service and walk are of infinite server type, whereas the arrivals are of single server type.

Corresponding to [15] the queuesize of each job queue is equal to $jobs = 5$. The variable ordering for the HLM model is given in Tab. 4.10. The STD submodel has the same queues, but leaves out the highlevel model (this is redundant information). With aggregations of every single submodel and the last remaining submodel the high level model, HLM corresponds to the aggregation strategy given in [15] (but in [15] the ordering of the queue aggregations is changed after every V-cycle while in the MTBDD-case it is fixed by the variable ordering). For the MSMQ model different aggregation strategies are used: From the aggregation of one single submodel per aggregation step to the aggregation of two submodels at once with some variants in between. No aggregation is coarser than the highlevel (queue1) submodel in the HLM (STD) case,

As the number of servers per queue (here modelled as servers1, ..., servers5) is redundant information, the MTBDD levels increase for the high-level model while the state space remains the same. Processing the high-level model MTBDD in general will take longer.

The results are shown in Table 4.13, where the "scaling parameter" column has a special meaning: It distinguishes the HLM and STD case.

In both the HLM and STD case, when each submodel is aggregated separately, the memory consumption of the multilevel algorithms is about one megabyte higher than the memory consumption of JOR, as for this model the aggregation information cannot be "hidden" within the second iteration vector of the JOR smoother.

In the experiments, no parameter set has been found where the multilevel algorithms performed better than the standard PRISM methods. It is remarkable that the multilevel algorithms require fewer iterations than JOR and PSOR (as expected), but due to the multilevel overhead, they still perform more slowly than the standard algorithms. However, the positive

		moderately sparse			fully sparse		
model	aggregation	$1T \to 2T$	$1T \to 4T$	$1T \to 8T$	$1T \to 2T$	$1T \to 4T$	$1T \to 8T$
HLM	(39,32,25,18,11)	1.63	2.28	2.35	1.49	1.94	1.94
	(39,25,11)	1.62	2.35	2.43	1.46	2.10	2.03
	(39,25)	1.59	2.23	2.29	1.39	1.87	1.87
STD	(29,22,25,8)	1.45	2.35	2.87	1.34	2.07	2.13
	(29,15)	1.48	2.38	2.86	1.33	2.10	2.17

Table 4.11.: Parallel speedup for different aggregation strategies

Data structure	smoother	agg.-ordering	ML-cycles	residual	time (s)
MTBDD	JOR	fixed	154	$9.2590 \cdot 10^{-9}$	36.61
Kronecker	JOR	fixed	160	$9.9087 \cdot 10^{-9}$	32.24
Kronecker	SOR	fixed	60	$9.9653 \cdot 10^{-9}$	19.59
Kronecker	JOR	cyclic	116	$9.9153 \cdot 10^{-9}$	26.08
Kronecker	SOR	cyclic	42	$6.9808 \cdot 10^{-9}$	14.43

Table 4.12.: Comparison of MTBDD and Kronecker-based algorithm

results published in [15] are probably due to the fact that there a cyclic or dynamic change of the multilevel aggregation ordering was employed, a feature which is not currently available with our own implementation.

The fewest multilevel cycles are required by the (39,25,11) aggregation level set for HLM, where only the high-level submodel is not aggregated in the coarsest system. It seems that this model is too small to apply the MTBDD-based multilevel algorithm successfully.

The parallel speedup is given in Tab. 4.11. It can be deduced from there that for this model, the parallel speedup is (at least for the submodel-wise aggregations) independent from the actual aggregation scheme performed.

Comparison with Kronecker data structure

A variant of this model (6 queues, capacity 3 for each queueing sytem, 243456 states) also was available as an APNN model for a Kronecker-based solver (called MLsolve) kindly provided by Peter Buchholz [15]. We built a corresponding model where the last aggregation is also the high-level model. While the MTBDD-based solver is currently restricted to cycles with the same ordering of aggregated matrices in every iteration and JOR smoothing steps, the Kronecker-based approach is able to change the aggregation ordering from cycle to cycle and can use a variety of different smoothing algorithms. Table 4.12 shows some results for the same set of parameters (one pre- and post iteration for the aggregated and non-aggregated levels, V-cycles, overrelaxation-parameter 1.0). They were obtained on the Xeon-system. The table is organised as follows: The first column describes the data structure used, the second the smoothing algorithm used (SOR: successive overrelaxation). The aggregation policy is described in the third column. Cyclic means that the aggregation ordering is changed from step to step by the same cyclic permutation of all queues. The remaining columns present the results: multilevel-cycles needed until convergence, residual and time taken.

One sees that for the same parameters (JOR,fixed), the number of ML-cycles nearly coincides. The small deviation is due to the fact that the Kronecker-based algorithm uses the residual as a termination criterion, while the MTBDD-based algorithm uses the relative error. The cyclic change in the aggregation ordering helps to reduce the number of multilevel-cycles needed and therefore leads to faster solutions. Also the change from JOR to SOR smoothing steps leads to a significant improvement. Even if the SOR algorithm is not directly applicable to MTBDD data structure it seems to be beneficial to implement the PSOR algorithm as smoother for the symbolic multilevel algorithm. Also cyclic permutations of the aggregation ordering would be

								moderate sparse			fully sparse		
model	states	transitions	algorithm	ML-cycles	steps	residual	variable levels	sparse levels	memory (kB)	time (s)	sparse levels	memory (kB)	time (s)
HLM	358560	2135160	ML(39,32,25,18,11)	51	408	$2.3588 \cdot 10^{-9}$	45	7	8106.8	123.24	7+38	8166.9	64.28
			ML_{red}(39,32,25,18,11)	51	408	$2.3588 \cdot 10^{-9}$	45	7	6741.1	107.01	7+38	6801.2	51.03
			ML(39,25,11)	50	400	$2.5202 \cdot 10^{-9}$	45	7	7652.7	117.04	7+38	7827.2	58.14
			ML_{red}(39,25,11)	50	400	$2.5202 \cdot 10^{-9}$	45	7	6880.5	100.20	7+38	6880.5	46.52
			ML(39,25)	51	408	$2.2032 \cdot 10^{-9}$	45	7	7648.0	118.12	7+38	7788.9	60.00
			ML_{red}(39,25)	51	408	$2.2032 \cdot 10^{-9}$	45	7	6701.9	100.94	7+38	6842.8	48.09
			JOR	-	663	$8.9223 \cdot 10^{-9}$	45	26	7162.5	33.05	7+38	15134.2	29.87
			PSOR	-	588	$1.0097 \cdot 10^{-8}$	45	26+18	4391.1	37.63	27+18	5139.9	36.44
STD	358560	2135160	ML(29,22,15,8)	51	408	$2.3761 \cdot 10^{-9}$	35	7	7878.9	109.06	7+28	7906.9	62.43
			ML_{red}(29,22,15,8)	51	408	$2.3761 \cdot 10^{-9}$	35	7	6612.7	97.61	7+28	6640.7	50.59
			ML(29,15)	51	408	$2.1513 \cdot 10^{-9}$	35	7	7535.1	105.91	7+28	7640.3	59.44
			ML_{red}(29,15)	51	408	$2.1513 \cdot 10^{-9}$	35	7	6584.3	93.70	7+28	6689.5	47.88
			JOR	-	663	$8.9223 \cdot 10^{-9}$	35	25	6971.7	32.66	7+28	15047.2	30.09
			PSOR	-	587	$1.0975 \cdot 10^{-8}$	35	21+14	3735.6	36.09	21+14	3735.6	35.99

Table 4.13.: MSMQ model - ML and standard numerical algorithms

					moderate sparse				fully sparse			
model	algorithm	parallel level	ML-cycles	residual	sparse levels	memory (kB)	setup (s)	iterations (s)	sparse levels	memory (kB)	setup (s)	iterations (s)
HLM	$ML_{1T}(39,32,25,18,11)$	7	51	$2.3588 \cdot 10^{-9}$	7	6882.5	0.95	154.99	7+34	6949.6	0.94	63.99
	$ML_{2T}(39,32,25,18,11)$					6885.1	0.06	95.00		6952.2	0.04	43.01
	$ML_{4T}(39,32,25,18,11)$					6895.5	0.96	67.99		6962.6	0.04	32.93
	$ML_{8T}(39,32,25,18,11)$					6936.8	0.06	66.00		7003.9	0.04	33.03
	$ML_{1T}(39,25,11)$	7	51	$1.9780 \cdot 10^{-9}$	7	6847.3	0.96	146.04	7+34	7053.5	0.94	61.05
	$ML_{2T}(39,25,11)$					6849.9	0.96	89.98		7056.1	0.04	41.92
	$ML_{4T}(39,25,11)$					6860.3	0.95	62.07		7066.5	0.94	29.06
	$ML_{8T}(39,25,11)$					6901.6	0.96	59.98		7107.8	0.94	30.05
	$ML_{1T}(39,25)$	7	51	$2.4891 \cdot 10^{-9}$	7	6843.2	0.05	155.95	7+21	6944.6	0.04	70.93
	$ML_{2T}(39,25)$					6845.8	0.05	98.00		6947.2	0.04	50.95
	$ML_{4T}(39,25)$					6856.2	0.95	70.05		6957.5	0.94	38.01
	$ML_{8T}(39,25)$					6897.5	0.95	68.06		6998.8	0.04	38.02
STD	$ML_{1T}(29,22,15,8)$	7	51	$2.3760 \cdot 10^{-9}$	7	6667.2	0.94	129.05	7+28	6698.5	0.92	63.99
	$ML_{2T}(29,22,15,8)$					6669.8	0.04	87.02		6701.1	0.03	47.92
	$ML_{4T}(29,22,15,8)$					6680.2	0.04	54.96		6711.5	0.03	30.95
	$ML_{8T}(29,22,15,8)$					6721.6	0.04	44.96		6752.8	0.03	29.99
	$ML_{1T}(29,15)$	7	51	$2.1507 \cdot 10^{-9}$	7	6638.7	0.93	126.07	7+21	6732.9	0.93	65.03
	$ML_{2T}(29,15)$					6641.3	0.04	86.95		6735.5	0.03	48.98
	$ML_{4T}(29,15)$					6651.7	0.93	53.07		6745.9	0.93	31.00
	$ML_{8T}(29,15)$					6693.0	0.04	44.01		6787.2	0.03	30.01

Table 4.14.: MSMQ model - multithreaded algorithm

Type of cycle (i.e. V,W,F)
Number of pre-smoothing steps (aggregated systems)
Number of post-smoothing steps (aggregated systems)
Number of pre-smoothing steps (fine system)
Number of post-smoothing steps (fine system)
Number of smoothing steps for the coarsest aggregation
Number of aggregations (or -2 for submodel-wise aggregation)
If not submodel-wise aggregation: Consecutive aggregation levels in the MTBDD
Flags for the sparse matrix substitution of the different parts of the MTBDD

Table 4.15.: Parameters for the ML algorithm in the config file

an improvement of the MTBDD algorithm.

4.7.5. Configuration file

For practical reasons it is convenient to have a configuration file with all parameters needed for the ML algorithm. In the current implementation this file is called ml.config and it is located in the PRISM root directory. It consists of the information given in Tab. 4.15. When the configuration file is read, comments (i.e. lines starting with `#`) are ignored. Separators between the different values are linefeeds. For the parallel algorithm, two additional parameters are used: The parallel level and the size of the job queues for the matrix operations.

5. Path-based measures

In this chapter an alternative approach to the analysis of CTMCs is given. It is shown how path-based calculations can be performed in connection with symbolic data structures. Path-based calculations are useful to get an impression of the most probable failure causes, for the construction of counterexamples in model-checking and for the calculation of some approximate measures if the model has certain properties (e.g. if the model has fast repairs). All calculations given here are restricted to finite path lengths. The chapter is organised as follows: In Sec. 5.1, the standard Dijkstra algorithm is presented and some alternative formulations are given. In Sec. 5.2, a variant of the Dijkstra algorithm, called Flooding Dijkstra, is presented that calculates the most probable path in a transition system using set-theoretic operations that allow for MTBDD implementations. With this basic algorithm at hand, Sec. 5.3 shows how to calculate the path with the second highest probability. By an inductive argument, all k-most probable paths up to a certain fixed number k can be calculated. An approximation of the MTTFF (Mean Time To First Failure) and MTTFR (Mean Time To First Recovery) is given in Sec. 5.4. MTTFF will be used to approximate the MUT (Mean Up Time), MTTFR will be used to approximate the MDT (Mean Down Time), thus the availability can be approximated. Note that [12] uses the term MTTF to describe the mean time to *first* failure. Corresponding algorithms for transition systems encoded by MTBDDs are given in Sec. 5.5. Two case studies show the applicability of the algorithms. In the sequel, the term k-shortest path is often used as a synonym for k-most probable path.

5.1. Dijkstra's Algorithm

Suppose a directed graph (V, E) has weight functions $c : V \times V \to \mathbb{R}$. We assume that $\mathbb{R}$ is equipped with the usual ordering $(\mathbb{R}, <)$. Dijkstra's algorithm finds a shortest path from a vertex *init* to any other vertex, i.e. it builds a minimum-weight spanning tree. In order to describe the algorithm, we assume that every node's weight will be stored by $c(.)$, every node's predecessor along the shortest path will be stored by $pre(.)$ and U is the set of unprocessed states. The algorithm is given in Alg. 23. Lines 1-3 initialise every node except the initial node to an infinite

Algorithm 23 Dijkstra(V,E,init)

```
 1: for all v ∈ V, v ≠ init do
 2:    c(v) = ∞
 3: end for
 4: c(init) = 0
 5: U = V
 6: while U ≠ ∅ do
 7:    choose s from U with c(s) ≤ c(u) ∀u ∈ U
 8:    U = U − {s}
 9:    for all v ∈ U do
10:       if c(s) + c(s, v) < c(v) then
11:          c(v) = c(s) + c(s, v)
12:          pre(v) = s
13:       end if
14:    end for
15: end while
```

weight. The initial node's weight is set to 0 in line 4. As all nodes are currently unprocessed, U is set to V in line 5. Lines 6-15 is the main loop that is performed until all nodes are processed. A vertex with minimal weight is chosen in line 7 and removed from U in line 8. Now in lines 9-13 for every unprocessed node that is reachable from s it is checked whether there is an improved weight using s as predecessor. If so, the weight is updated in line 11, and the spanning tree has to be updated in line 12 (i.e. the predecessor of v is set to s).

We would like to stress that this algorithm works in a state-by-state manner that is not suitable for an implementation on the MTBDD data structure.

5.1.1. Additive and multiplicative formulations of Dijkstra's algorithm

Starting with the classical shortest path problem on a graph (V, E), we would like to mention that there are several dual problems regarding shortest path. One of them is the formulation of the most probable path. We look at some subsets of $\mathbb{R}$ with the induced ordering. In the following we will use that exponential function $e : [0, \infty) \to [1, \infty)$, $x \mapsto e^x$ is a group homomorphism $([0, \infty), +) \to ([1, \infty), \cdot)$, i.e. $e^{x+y} = e^x \cdot e^y \ \forall x, y \in (0, \infty)$ and strictly monotonically increasing, i.e. $x < y \Rightarrow e^x < e^y$. The inverse function ln is also a homomorphism and strictly monotonically increasing. The mapping $e^- : [0, \infty] \to [0, 1]$, $x \mapsto e^{-x}$ (where ∞ is mapped to 0) is a homomorphism $([0, \infty), +) \to ([0, 1], \cdot)$, but it is strictly monotonic decreasing, as $x < y \Rightarrow \frac{1}{e^x} > \frac{1}{e^y}$ (the same holds for the inverse function).

Suppose that for every vertex v there is a current cost $c(v)$ and for every edge (v_i, v_j) there is a cost named $c(v_i, v_j)$. A basic building block for Dijkstra's algorithm is the comparison $c(v_i) + c(v_i, v_j) < c(v_j)$ (cf. line 10 in Algorithm 23). If it is true, then $c(v_j)$ has to be updated to the new value $c(v_i) + c(v_i, v_j)$ (cf. line 11 in Alg. 23). Assuming non-negative costs, the following formulations are equivalent:

1. find the shortest (additive) path (comparison $c(v_i) + c(v_i, v_j) < c(v_j)$)
2. find the shortest (multiplicative) path with costs $c(v_i, v_j)$, $c(v_i) \in [1, \infty)$ (comparison $c(v_i) \cdot c(v_i, v_j) < c(v_j)$)
3. find the longest (multiplicative) path with costs $c(v_i, v_j)$, $c(v_i) \in [0, 1]$ (comparison $c(v_i) \cdot c(v_i, v_j) > c(v_j)$)

The updates (line 11 in Algorithm 23) have to be changed accordingly. As the initialisation of the additive variant is 0, the multiplicative variants are initialised by $e^0 = e^{-0} = 1$ (line 4 in Algorithm 23). One gets from 1. to 2. by exponentiation, in the other direction by logarithmisation. As the mappings are order-preserving (i.e. monotonically increasing), the comparison operator does not change. From formulation 1. to 3. one gets by e^-. In the opposite direction, one gets by $-ln$. As the operations are monotonically decreasing, the comparison operator has to be changed. In the following, we will always use formulation 3. with costs seen as probabilities.

5.2. Flooding Dijkstra algorithm

This Section introduces a variant of Dijkstra's algorithm which is used for calculating the most probable path. Our variant is a set-theoretic approach, in order to exploit the possibilities of symbolic data structure. A prototypical implementation was developed in [27]. This variant is called *flooding Dijkstra algorithm* in the sequel. The following description is a clarified version of [27] that was used as theoretical background for the re-implementation, included in the current version of the CASPA tool.

In the following a PSLTS $(S, L_I, \emptyset, \emptyset, Trans, Init)$ is fixed, i.e. a purely probabilistic PSLTS without Markovian transitions. As there are only probabilistic labels, we will use $L := L_I$ in

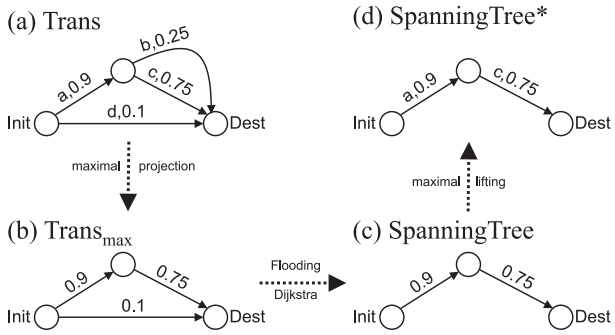


Figure 5.1.: maximal projection, Flooding Dijkstra and maximal lifting

the following. For a convenient setup it turned out to be useful to speak of projections. The *maximal projection* $Trans_{max}$ of $Trans$ is defined as

$$\begin{aligned} Trans_{max} \quad := \quad & \{(x,p,y) \in S \times [0,1] \times S | \\ & (\exists a \in L : (x,a,p,y) \in Trans) \wedge \\ & (\forall a \in L, p' \in [0,1] : \\ & ((x,a,p',y) \in Trans) \Rightarrow (p' \leq p))\}. \end{aligned}$$

The existence condition ensures that a lifting to $Trans$ exists and the second condition ensures that p is maximal. So these are the maximum transition probabilities one can get by choosing an arbitrary action from a source to a target state. As there are no Markovian transitions the abbreviation $x \xrightarrow{p} y$ is used as a synonym for the tuple (x,p,y). An example can be seen in Fig. 5.1 (a) to (b).

In contrast to Dijkstra's original algorithm there are not only optimal updates, i.e. there may also be some updates of probabilities that are re-updated later on. The resulting graph is not a real spanning tree, in the sense that in the result there may be paths that are equally probable. This issue will be resolved when the most probable paths are extracted from the result by making choices then. Nevertheless, in the sequel such a quasi-spanning tree will be called a spanning tree (alternatively one would have to ensure that in lines 26-28 of Alg. 24 only one transition for each target y is added, in order to get a real spanning tree).

The code of Alg. 24 is explained as follows: Lines 1-6 perform the necessary initialisations, i.e. all state probabilities are set to zero except the probability of state *Init* (the initial state) which is set to 1. The current *BorderSet* from where the updates start is set to state *Init*. As the algorithm just starts, the *SpanningTree* of course is the empty set. Lines 7-31 constitute the main loop that is carried out until there is no state in the *BorderSet* any more. First, in lines 8-10 the new probabilities of all states are set to zero. The loop from line 11 to 14 then calculates the maximum probability for every state $y \in S$ to be reached from the *BorderSet*. The *UpdateSet* is defined in line 16 to be those states that gained a higher probability in the current round. The update of the state probabilities to the higher probabilities gained by the *BorderSet* is done in lines 17 to 19. The remaining part is to update the spanning tree for those states that got a higher probability. Therefore the old transitions that reached the *UpdateSet* are deleted from the *SpanningTree* in lines 20-24 and the new transitions are added in lines 25-29. Note by looking at the condition in line 26 that by only looking at $Prob(y)$ the update in line 27 might not be unique and therefore a spanning tree that contains redundant, i.e. equally probable, paths is generated. This will be discussed also in Sec. 5.3. Finally, in line 30, the *BorderSet* is set to the *UpdateSet* and the loop can start again. Of course, the flooding Dijkstra algorithm shown in Alg. 24 also performs a reachability analysis, since only reachable states will be in the spanning tree. The result of a flooding Dijkstra run can be seen in Fig. 5.1 (b) to (c). Probabilities for each node are of course calculated but are not shown in the figure.

Algorithm 24 FloodingDijkstra(Trans)

1: **for all** $x \in S : x \neq 0$ **do**
2: $Prob(x) = 0$
3: **end for**
4: $Prob(Init) = 1$
5: $BorderSet = \{Init\} \subseteq S$
6: $SpanningTree = \emptyset \subseteq Trans_{max}$
7: **while** $BorderSet \neq \emptyset$ **do**
8: **for all** $y \in S$ **do**
9: $newProb(y) = 0$
10: **end for**
11: **for all** $(x, y) \in BorderSet \times S$ **do**
12: **if** $x \xrightarrow{p} y \in Trans_{max}$ **then**
13: $newProb(y) = max(Prob(x) \cdot p, newProb(y))$
14: **end if**
15: **end for**
16: $UpdateSet = \{x \in S | newProb(x) > Prob(x)\}$
17: **for all** $x \in UpdateSet$ **do**
18: $Prob(x) = newProb(x)$
19: **end for**
20: **for all** $(x, y) \in S \times UpdateSet$ **do**
21: **if** $x \xrightarrow{p} y \in SpanningTree$ **then**
22: $SpanningTree = SpanningTree \setminus \{x \xrightarrow{p} y\}$
23: **end if**
24: **end for**
25: **for all** $(x, y) \in S \times UpdateSet$ **do**
26: **if** $(x \xrightarrow{p} y \in Trans_{max}) \wedge (Prob(x) \cdot p = Prob(y))$ **then**
27: $SpanningTree = SpanningTree \cup \{x \xrightarrow{p} y\}$
28: **end if**
29: **end for**
30: $BorderSet = UpdateSet$
31: **end while**
32: return $SpanningTree$

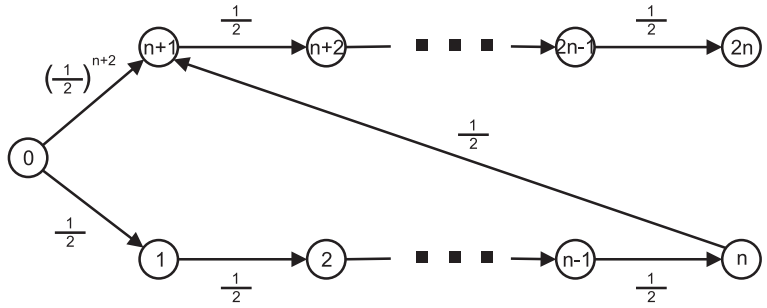


Figure 5.2.: Comparison of Dijkstra variants

5.2.1. Difference between flooding and standard Dijkstra

We only sketch the difference of the two algorithms by an example that causes arbitrarily many non-optimal updates. Suppose that $n < \infty$. We want to calculate the most probable path from state 0 to state $2n$ in Fig. 5.2. Suppose that there is an absorbing state, where all the missing transitions end (in order to get probability distributions for every outgoing set of transitions). Looking only at the states depicted in Fig. 5.2 one can observe a slight difference between the two algorithms. Obviously, the most probable path is $(0, 1, \ldots, 2n)$, as $\frac{1}{2^{n+2}} < \frac{1}{2^{n+1}}$. Both algorithms start at state 0 and calculate the probabilities for reaching states $n+1$ and 1. After that, the behaviour changes:

Dijkstra Successively state $1, 2, \ldots, n+1$ are processed. After that, states $n+2, \ldots, 2n$ are processed.

Flooding Dijkstra The border set is $\{i, n+i\}$ and it is processed for $1 \leq i \leq n$. After that, the border set $\{n+i\}$ is processed for $1 \leq i \leq n$.

Therefore, with the border set $\{i, n+i\}$, $1 \leq i \leq n$, the first updates of states $n+2, \ldots, 2n$ are non-optimal and could be omitted (standard Dijkstra also has to update state $n+1$ twice). However, regarding the MTBDD context it is costly to pick out single states and only perform their updates. That is the reason why the flooding variant is used.

5.3. Calculation of $k-$shortest paths

This Section shows how to calculate k-shortest-paths, thereby employing the flooding Dijkstra algorithm presented in Sec. 5.2. First the basic procedure of finding the correct action labels for a path within the spanning tree calculated by Flooding Dijkstra is shown. Next, the transformation of the original transition system to a new one is shown, such that the shortest path of the transformed transition system is the second shortest path of the original transition system. With this algorithms at hand, one can calculate the $k-$shortest paths up to a certain fixed k.

5.3.1. Reading the action labels from a path

Before proceeding with the algorithm, a *maximal lifting* is defined to extract corresponding action labels for a shortest path. The maximal lifting of a subset $T \subseteq Trans_{max}$ to $Trans$ is given by

$$T^* \quad := \quad \{(x, a, p, y) \in Trans | \exists (x, p, y) \in T\}.$$

One observes that this lifting is not unique: There can be more than one action fulfilling the maximality condition while p as the maximum is unique for a certain pair $(x, y) \in S^2$. Note that the algorithm for reading the most probable path works on the transition system $Trans$ and uses

the maximal lifting of the path found by flooding Dijkstra in $Trans_{max}$. An example is given in Fig. 5.1 (c) to (d). Let $Dest$ be the state which is the target of the path analysis and let $Init$ be the starting state of the analysis. The function $PickOne(X)$ is introduced that chooses one arbitrary element of a set X. Alg. 25 prints the shortest (reversed) path from $Dest$ to $Init$ by

Algorithm 25 getPath(SpanningTree)

```
 1: y = Dest
 2: while y ≠ Init do
 3:    print "←"
 4:    Predecessors =   {x ∈ S|∃(p) ∈ [0,1] : (x,p,y) ∈ SpanningTree}
 5:    x = PickOne(Predecessors)
 6:    for all (a,p) ∈ L × [0,1] do
 7:       if x -a,p-> y ∈ SpanningTree* then
 8:          print "<" a ">"
 9:          Path = Path ∪ {x -a,p-> y}
10:       end if
11:    end for
12:    y = x
13: end while
14: return Path
```

a traversal of SpanningTree choosing unique predecessor states. The explanation is as follows. Line 1 sets the current state to $Dest$. The main loop that jumps back to the current state's predecessor goes from line 2 to 13 as long as $Init$ has not been reached. Line 3 prints the arrow indicating the next step. Line 4 computes all predecessor states of y in $SpanningTree^*$ and line 5 arbitrarily chooses one of them. The loop from line 6 to 11 picks out the corresponding transitions from state x to state y in $SpanningTree^*$. In line 8 the corresponding action is printed and line 9 adds the corresponding transition to $Path$. Line 12 sets the current state to the predecessor x.

Note that there can still be concurring actions in the path generated. Of course due to the $SpanningTree^*$-property it holds that

$$\begin{aligned}\forall(x,y,a,a') \quad &\in S^2 \times L^2 : \\ &((x,a,p,y) \in SpanningTree^* \wedge \\ &(x,a',p',y) \in SpanningTree^*) \Rightarrow p = p'.\end{aligned}$$

So, since the probabilities of concurring actions in $Path$ are equal, there is no need to distinguish them for further processing. In the current implementation we print all possible alternatives.

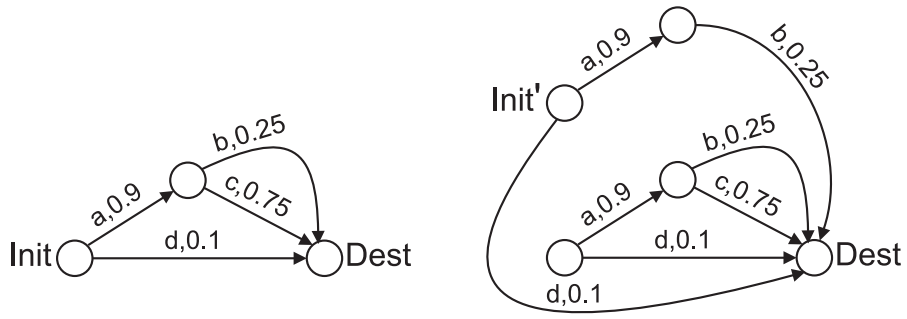
5.3.2. Second shortest path

This Subsection will show how to generate a transition system $Trans'$ out of a given transition system $Trans$ such that the shortest path of $Trans'$ corresponds to the second shortest path of $Trans$. Inductively it follows that by this concept the k-shortest paths can be calculated for an arbitrary but fixed k. The algorithm is due to Azevedo [3] using a refinement of Schmid [53].

Starting from a set of states S and a set of transitions $Trans$ as before, an additional set of states S' with $S \cap S' = \emptyset$ and $|S| = |S'|$ is introduced. A fixed bijection

$$' : \quad S \to S'$$

is used to identify elements x of S with their corresponding copy x' in S' and vice versa.



(a) $Trans$ (b) $Trans'$

Figure 5.3.: Trans and Trans'

Furthermore, the set of states

$$S_{new} := S \cup S'$$

for the new transition system $Trans' \subseteq S_{new} \times L \times [0,1] \times S_{new}$ is defined. Let $Path$ be the path calculated in Sec. 5.3.1.

Algorithm 26 ChangeTransitionSystem(Trans, Path)

1: $Trans' = Trans$
2: $PathCopy := \{(x', a, p, y') | \exists (x, a, p, y) \in Path \wedge y \neq Dest\}$
3: $Trans' = Trans' \cup PathCopy$
4: $SourceStates := \{x \in S | \exists (x, a, p, y) \in Path\}$
5: **for all** $(x, a, p, y) \in Trans$ **do**
6: **if** $x \in SourceStates$ **then**
7: **if** $x \overset{a,p}{\rightarrow} y \notin Path$ **then**
8: $Trans' = Trans' \cup \{x' \overset{a,p}{\rightarrow} y\}$
9: **end if**
10: **end if**
11: **end for**
12: return $(Trans', Init = Init')$

Alg. 26 works as follows: The seed of the new transition system $Trans'$ are the old transitions $Trans$ (of course the initial state has to be changed in order not to find the same path as before) which are set in line 1. The shortest path (including its parallel actions) is copied to S' without the last transition to $Dest$ in line 2. Line 3 adds this path to $Trans'$. Line 4 calculates the source states of the transitions in $Path$. The loop from line 5 to 11 picks all the transitions that emanate from a source state in $Path$ but are not on the shortest path and adds them as cross connections from S' to S to the transitions of $Trans'$. Finally the initial state is set to the copy of the initial state in S' in line 12. An example of the transformation is given in Fig. 5.3.

5.4. MTTFF and MTTFR

In this section, sojourn times of states have to be considered, therefore the purely probabilistic setup of the previous sections does not suffice. In the sequel will work on a PSLTS $(S, L_M, L_I, Trans_M, Trans_I, Init)$.

By the maximal progress assumption, Markovian transitions that compete with immediate transitions are removed from $Trans_M$. Using the transition probabilities of the immediate transitions and calculating the transition probabilities of the Markovian transitions (from the specified transition rates), a transition system $Trans$ like the one defined in Sec. 5.2 can be obtained easily.

For the following subsections a set of failure states of the system is required, which will be denoted by $Failure$. Using the exit rate (cf. Sec. 2.2), an associated average time can be calculated for every path in the Transition system $Trans_M \cup Trans_I$ starting at a certain state x_0. To determine the average time for a path, the function getTime is defined in a canonical way:

$$getTime(x_0, path) := \sum_{x \in S_M(path)} \frac{1}{\lambda(x)}$$

Here $S_M(path)$ ranges over all source states in $Trans_M \cap path$ and x_0 is the first state of $path$. Note that x_0 is redundant as $path$ contains all the information, but it makes calculations in Sec. 5.4.2 easier to read.

5.4.1. Calculation of MTTFF

The MTTFF is defined as the average time it takes, starting from the initial state (which is here the completely repaired system), to get to an erroneous state. Depending on the transition system, it could be the case that already the Mean Time To Second Failure (MTTSF) differs from the MTTFF. Only in the case that there is only one functional state and no state with limited functionality, MTTFF=MTTSF. A similar argument holds for the consecutive times. That is why the term MTTFF is used and not MTTF (Mean Time To Failure). In the sequel it is assumed that all models have *fast repair*, i.e. the repair rates are orders of magnitude higher than the failure rates. This assumption is quite natural and usually holds for a large group of dependability models. An approximation of the MTTFF for models with fast repair can be calculated as follows [12]:

$$MTTFF \approx \frac{1}{\alpha \cdot \lambda(Init)} \tag{5.1}$$

Being in the state $Init$, α is the probability of visiting at least one system failure state before returning to Init. The set of system failure states will be denoted by $Failure$. Clearly, α can be approximated by adding up the probabilities of the k-shortest paths from $Init$ to the set $Failure$.

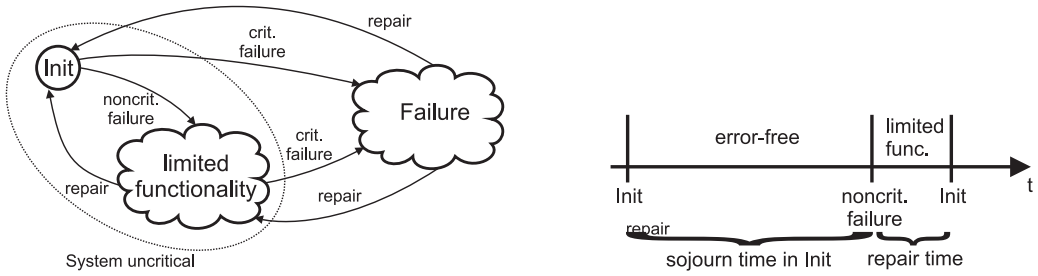
The approximation in Eq. 5.1 can be motivated as seen in Fig. 5.4. Starting from the state $Init$, we are interested in the mean time necessary to reach a critical failure before returning to the initial state (cf. Fig. 5.4(a)). An exemplary scenario is shown in Fig. 5.4(b), where a non-critical failure occurs and the system can recover to the error-free state Init. Under the fast repair assumption, the repair time is negligible in contrast to the sojourn time $T := \frac{1}{\lambda(Init)}$ in state Init, so the time for a loop from Init to Init (via the limited functionality set) is taken to be T. Therefore one can approximate

$$MTTFF \approx T \cdot \alpha + 2 \cdot T \cdot (1-\alpha) \cdot \alpha + \ldots = \alpha \cdot T \cdot \sum_{i=0}^{\infty} (i+1) \cdot (1-\alpha)^i$$

and from this one can deduce Eq. (5.1) by a geometric series argument. Note that if the system does not have fast repairs, an alternative approach similar to the calculation of MTTFR in Sec. 5.4.2 can be taken for the calculation of MTTFF.

5.4.2. Calculation of MTTFR

In the same way as seen for the MTTFF, the MTTFR might differ from the time for the second repair and so on. That is why the term MTTFR instead of MTTR (mean time to repair) is used. As the calculation of MTTFF exploits the fact that the system has fast repairs, a similar approximation cannot be applied for the calculation of the MTTFR. Therefore the following method, as originally proposed in [12], is applied: For a state $x \in S$ and a subset of target states



(a) Basic setup (b) Timeline

Figure 5.4.: Approximation of MTTFF

$T \subseteq S$ the term $PATH(x, T)$ is used to denote minimal paths from x to T. *Minimal* in this case means that only the last state of the path is in the set T. The probability of a certain path starting from state x is denoted by $P(x, path)$ (this is a by-product of the Flooding Dijkstra algorithm, cf. Sec. 5.2). Let the function $Target$ return the final state of a path. With this notation it is possible to define the MTTFR for paths starting in state x as

$$MTTFR(x) := \sum_{path \in PATH(x, S \setminus Failure)} P(x, path) \cdot getTime(x, path).$$

Further the cumulated probability of the explored paths

$$P_{expl} := \sum_{path \in PATH(Init, Failure)} P(Init, path)$$

is needed. The MTTFR is then approximated as the weighted sum

$$MTTFR := \frac{1}{P_{expl}} \cdot \sum_{path \in PATH(Init, Failure)} P(Init, path) \cdot MTTFR(Target(path)).$$

From this, the asymptotic unavailability can be calculated as follows:

$$\bar{A}(\infty) = \frac{MDT}{MUT + MDT} \approx \frac{MTTFR}{MTTFF + MTTFR}$$

Note that in this approximation it is also assumed that the system has fast repairs (then it holds that $MUT \approx MTTFF$, $MDT \approx MTTFR$). In systems with fast repair the fully repaired system has by far the highest probability (e.g. 0.996 in the example in Sec. 5.6).

5.4.3. Approximations in CASPA

An implementation of the path-based algorithms has been realised in the context of the tool CASPA [38]. The MTBDD code used is given in Sec. 5.5. In CASPA only trivial truncation criteria are implemented so far: Firstly, one can specify how many paths should be calculated (and therefore be used in the calculations of Sec. 5.4.1 and 5.4.2) and secondly, a path is cancelled if its probability is below a certain threshold. As stated in [12], one can only expect good approximation results if the model has fast repair, i.e. the repair rates are orders of magnitude bigger than the failure rates.

5.5. MTBDD code for path-based algorithms

In this Section, the MTBDD versions of the path-based algorithms presented in the preceding sections are given. In the description of the algorithms, the following name conventions for

MTBDD variables are used: $\vec{a}$ (action labels), $\vec{s}$ (source states) and $\vec{t}$ (target states). The variable ordering in the MTBDD is $a_1 \prec \ldots \prec a_n \prec s_1 \prec t_1 \ldots \prec s_m \prec t_m$ according to commonly accepted heuristics.

5.5.1. Spanning tree algorithm

For the algorithm, let us assume we are already given a purely probabilistic transition system $Trans$ where for every state s with at least one emanating transition the probabilities of all emanating transitions sum up to one, i.e. they define a discrete probability distribution. Whenever needed, subscripts show to which variable set a MTBDD belongs to (e.g. $Prob_s$ means that MTBDD $Prob$ is defined on s-variables. Let the initial state be stored in the variable $Init$ (by s-variables).

Algorithm 27 FloodingDijkstraSymbolic(Trans)

1: $Prob_s = Init$
2: $Border_s = Init$
3: $SpanningTree = 0$
4: $Trans_{max} = ABSTRACT(Trans, a, max)$
5: **while** TRUE **do**
6: $\quad Prob_s^{Border} = Border_s \cdot Prob_s$
7: $\quad Prob_{st}^{Trans} = Prob_s^{Border} \cdot Trans_{max}$
8: $\quad Prob_t^{new} = ABSTRACT(Prob_{st}^{Trans}, s, max)$
9: $\quad UpdateSet_t = (Prob_t^{new} > (Prob_s)_{s \to t})$
10: $\quad$ **if** $UpdateSet_t == 0$ **then**
11: $\quad\quad$ **break**
12: $\quad$ **end if**
13: $\quad Prob_s = (ITE(UpdateSet_t, Prob_t^{new}, (Prob_s)_{s \to t})))_{t \to s}$
14: $\quad Prob_t^{Updated} = UpdateSet_t \cdot Prob_t^{new}$
15: $\quad NewTransitions = \ (Prob_{st}^{Trans} == Prob_t^{Updated}) \cdot UpdateSet_t \cdot Trans_{max}$
16: $\quad SpanningTree = \ ITE(UpdateSet_t, NewTransitions, SpanningTree)$
17: $\quad Border_s = (UpdateSet_t)_{t \to s}$
18: **end while**

Alg. 27 works as follows. In line 1-3 some initialisation assignments are given. Line 4 calculates the maximal projection of $Trans$. The loop from line 5 to 18 calculates the updates. In line 6, the probabilities for the states in the current border set are calculated and in line 7 they are multiplied by the maximal transition probabilities. Line 8 calculates the new probabilities for successor states of the border set by abstraction of $Prob_{st}^{Trans}$. From this the $UpdateSet_t$ can be calculated that encodes the states that are reached with a higher probability by transitions from the border states. In lines 10-12 the exit condition is checked: Whenever there are no more states to be updated, the algorithm has finished. With the knowledge of the states to be updated, line 13 calculates the correct new probabilities. The new probabilities that have been updated are calculated in line 14 and are used in line 15 to get the transitions that provide the maximum probability for those states (not necessarily unique). Looking at the updated states the spanning tree can be updated in line 16: All transitions that lead to a state in $UpdateSet_t$ are taken from $NewTransitions$, the others remain the same. Finally $Border_s$ is updated in line 17.

5.5.2. Reading the action labels from a path

This code fragment reads one shortest path from the spanning tree calculated in 5.5.1. Note that the path read may include parallel actions, but the predecessor states have to be unique.

Suppose that $Init$ encodes the initial state, $Dest$ the destination state of the analysis (by s-variables). It is notable that two versions of the SpanningTree are used: The maximal projection

Algorithm 28 getPathSymbolic(SpanningTree, Trans)

```
1:  p = ABSTRACT(Prob_s · Dest, s, max)
2:  ShortestPath = 0
3:  SpanningTree* = (SpanningTree == Trans) · Trans
4:  CurrentState_s = Dest
5:  while CurrentState_s! = Init do
6:     Edges_st = (CurrentState_s)_{s→t} · SpanningTree
7:     Edge_st = PickOne(Edges_st)
8:     Edges_ast = (Edge_st == SpanningTree*) · SpanningTree*
9:     ShortestPath = ShortestPath + Edges_ast
10:    while Edges_ast! = 0 do
11:       Action = PickOne(Edges_ast)
12:       print getName(Action)
13:       Edges_ast = Edges_ast − Action
14:    end while
15:    print "←"
16:    CurrentState_s = ABSTRACT(Edge_st, t, +)
17: end while
```

and the maximal lifting. All paths are printed from $Dest$ to $Init$. The code in Alg. 28 reads as follows: Line 1 calculates the probability of $Dest$ by abstracting over all s-variables. In lines 2-4 some initialisations are done, most notably the maximal lifting of SpanningTree in line 3. The main loop starts at line 5 and terminates when $Init$ is reached. All incoming edges of $CurrentState$ are read from $SpanningTree$ in line 6. Out of them a single edge is picked in line 7. Line 8 calculates the maximal lifting of $Edge_{st}$ in $SpanningTree^*$. The current $Edge_{ast}$ (with possible parallel actions) is added to $ShortestPath$ in line 9. The loop from line 10-14 prints all parallel actions belonging to $Edge_{ast}$. Line 16 switches to the next predecessor of CurrentState given by $Edge_{st}$.

5.5.3. Changing the transition system

In this Subsection the symbolic operations for changing the transition system are shown. The transition system is altered in such a way that the shortest path of the new transition system is the second shortest path of the old transition system. Suppose $Prob_s$ contains the probabilities of states and $ShortestPath$ contains a shortest path with possible parallel actions as calculated in 5.5.2. The Expand function used in the code is just a short-hand notation, given as follows:

$$\begin{aligned} Expand(Trans, s, t) &:= NewVariable_s(s) \cdot \\ &\quad NewVariable_t(t) \cdot Trans \\ Expand(State, s) &:= NewVariable_s(s) \cdot State \end{aligned}$$

Two new variables are introduced (here between action and state variables) in order to encode original and copied states in source and target variables. $NewVariable_s$ encodes whether the source state lies in the original or the copied set of states. If it is set to 0 it means that the source state lies in the original set of states, if it is set to 1 the state is a copied one. A similar statement applies to the target states. Alg. 29 works as follows. First the reachability analysis done by the spanning tree calculation is used to minimise the $Trans$ by leaving out unreached states (i.e. states that have a maximal probability of 0). This is done in line 1 and 2. Next, the shortest path without the last transition to $Dest$ is calculated in line 3. Line 4 calculates all

Algorithm 29 ChangeTransitionSystemSymbolic(Trans, ShortestPath)

1: $NonZeroProb_s = (Prob_s > 0)$
2: $Trans = NonZeroProb_s \cdot Trans$
3: $PathWithoutDest = ITE((Dest)_{s \to t}, 0, ShortestPath)$
4: $Deviations = ITE(ShortestPath, 0, Trans) \cdot \ (ABSTRACT(ShortestPath, t, +) > 0)$
5: $PathWithoutDest = Expand(PathWithoutDest, 1, 1)$
6: $Deviations = Expand(Deviations, 1, 0)$
7: $Trans = Expand(Trans, 0, 0)$
8: $Trans' = Trans + Deviations + PathWithoutDest$
9: $Init' = Expand(Init, 1)$
10: $Dest' = Expand(Dest, 0)$

allowed deviations, i.e. those transitions that emanate from states on the shortest path but do not lie on the shortest path. In lines 5-7 some expansions are done. From the given parameters one sees that $PathWithoutDest$ lies in the copied states, while $Deviations$ lead from copies to original states and $Trans$ still lies in the original states. The new transition system is built in line 8 as the union of the three precalculations and finally in line 9 and 10 $Init$ and $Dest$ are copied.

5.5.4. MTTFF and MTTFR

In order to calculate the mean times it is necessary to know the sojourn times for every state. They are calculated by Alg. 30 as follows. Line 1 and 2 abstract `t` and `a` variables from the Markovian transitions. In line 3 a 0-1-MTBDD is generated that encode states with emanating Markovian transitions. Finally, in line 4 the actual Sojourn times are calculated as 1 divided by the exit rate for every state.

Algorithm 30 Symbolic Calculation of sojourn times

1: $Trans_{Mas} = ABSTRACT(Trans_M, t, +)$
2: $Trans_{Ms} = ABSTRACT(Trans_{Mas}, a, +)$
3: $Trans_{M01} = THRESHOLD(Trans_{Ms}, 0)$
4: $SojournTime_s = Trans_{M01}/Trans_{Ms}$

Algorithm 31 getDurationSymbolic(ShortestPath, SojournTime$_s$)

1: $ShortestPath_{as} = ABSTRACT(ShortestPath, t, +)$
2: $ShortestPath_s = ABSTRACT(ShortestPath_{as}, a, +)$
3: $ShortestPath_{s01} = THRESHOLD(ShortestPath_s, 0)$
4: $Meantime_s = ShortestPath_{s01} \cdot SojournTime_s$
5: $Meantime = ABSTRACT(Meantime_s, s, +)$

In a similar way, line 1-3 of Alg. 31 calculate the states in the shortest path encoded by `s` variables. Now, the calculation of the mean time of the path amounts to multiply the states in the path with their corresponding sojourn times, which is done in line 4, and abstracting over all `s` variables to get the sum of all sojourn times of the different states.

It is possible to perform this algorithm only once (and not for every new Trans') as of course the sojourn time of a copied state is the same as for the original state. The current implementation is as follows: Alg. 30 is only performed once (i.e. for $Trans$ without modifications) and a modification to Alg. 31 is added that maps all copied states back to S. This can be done by inserting the instruction $ShortestPath_{s01} = ABSTRACT(ShortestPath_{s01}, s_{copy}, +)$ between

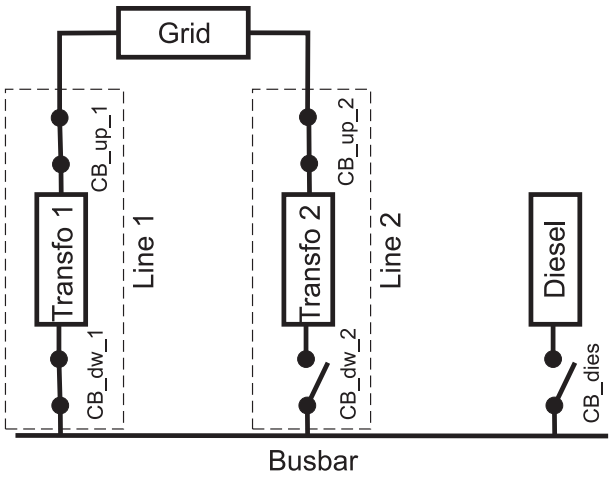


Figure 5.5.: Sketch of the busbar model

line 3 and 4. The symbol s_{copy} here means the set of `s` bits that define the new variables introduced in 5.5.3.

5.6. Case study I: Electric power supply

To show the applicability of the algorithms, a model of a busbar given in [10] is studied. After a discussion with the first author of [10], we present a slightly modified model that to the best of our knowledge produces the desired paths.

5.6.1. Description of the model

The model is shown in Fig. 5.5. The aim of the system is to provide the busbar with electrical energy. Each of the main lines consists of upper and lower circuit breakers and a transformer. They route electrical energy from the grid to the busbar. If the lines fail or the grid does, the diesel generator has to be used. The initial configuration is as seen in Fig. 5.5 where only CB_dw_2 and CB_dies are in the open position, the other switches are closed. The following constraints for the operation and dynamic behaviour are given:

- States of the components can be WORKING, STANDBY or FAILED
- Either line 1, line 2 or the diesel engine is used. Mode switches can only be line 1 $\leftrightarrow$ line 2 $\leftrightarrow$ diesel, no direct switches from line 1 to diesel and vice versa are allowed.
- The transfos and the grid are hot spares and always fail with the same rate, no matter if they are active or not.
- The circuit breakers CB_up and CB_dw are cold spares (i.e. they do not fail as long as no current runs over them) and they can produce on-demand-failures.
- CB_dies only fails on-demand, it does not fail internally.
- An on-demand-failure of a circuit breaker does not change its internal state. Reconfiguration can change the internal state of a component.
- Switching from transfo 1 to transfo 2 means trying to open CB_up_1 and to close CB_dw_2. Note that when transfo 1 fails and CB_up_1 fails to open, one must try to close CB_dw_2 even if it's clear that the diesel engine has to be used, as the grid is short-circuited by line 1 in this case.
- Switching on the diesel engine means closing CB_dies and trying to start the engine. Both operations have on-demand failures
- Switching back after a repair always works without on-demand failures

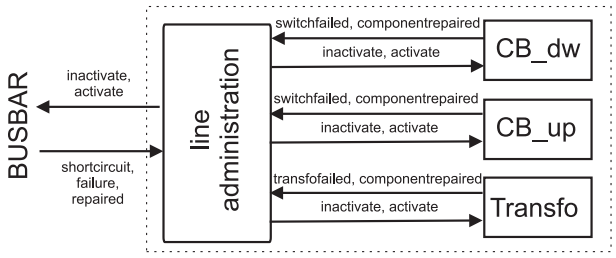


Figure 5.6.: Sketch of the model of a transfo line

- When a transfo fails, its upper circuit breaker has to be opened, otherwise a shortcircuit will make the grid unavailable for the other transfo.

Since CASPA does not allow nondeterminism, a total ordering of all switching actions has been introduced as follows: (CB_up_1 > CB_dw_1 > CB_up_2 > CB_dw_2 > CB_dies > Start_dies). Whenever there are more switching operations to be done in our model, the greater operation will always be switched first.

Following the philosophy of shortest paths to error states it is natural that whenever a greater switching action fails, the path is not explored any further (except for the short circuit situation given above). All intended switchings succeed with probability $\frac{999}{1000}$, all failure rates are equal to 10^{-4} per hour and all repair rates equal to 10^{-1} per hour.

5.6.2. CASPA implementation of the model

The model has been built in CASPA using a compositional modelling approach with synchronisations. Each line is the parallel composition of two switches, a transformer and a line administration. Its synchronised actions are given in Fig. 5.6. The line administration process has in addition to its internal state a counter variable that keeps track of how many components of the line are currently in the FAILED state. From this it can be determined, when the entire line has been repaired. A top-level process synchronises with two line subprocesses and takes care of the grid and the entire diesel line. As the immediate actions used only for synchronising are of no interest for the resulting paths, they are eliminated. The elimination is done by four semisymbolic elimination rounds [4] and the resulting model has 774 reachable states.

5.6.3. Experimental results

Table 5.1 shows the 6 most probable paths calculated by our algorithm (more paths can be found in [28]). The first column is the ordering of the paths given by CASPA, the second one the corresponding sequence of actions. The third column shows the numerical result for the probability provided by our algorithm, the fourth one shows the exact result calculated by hand using elementary probability theory. The last column shows the mean time the corresponding path takes. In column 2, the prefix `Failure_` always means a failure caused by an exponential distribution. The prefix `OK_` means that an on-demand failure did not occur, while prefix `Occurrence_` means that an on-demand failure did occur. The following abbreviations are used: RO (Request to Open), RC (Request to Close) and RS (Request to Start). For the by-hand calculation one only has to take care which failure event(s) and repair event(s) can occur for a certain state. For example, the most probable path can be calculated as follows: In the initial configuration no component has to be repaired and Transfo1, CB_up_1, CB_dw_1, Transfo2 or Grid can fail. Therefore

$$P\left(\begin{array}{l}\texttt{Failure\_of\_GRID} \rightarrow \\ \texttt{Occurrence\_of\_RC\_CB\_dies}\end{array}\right) = \frac{10^{-4}}{5 \cdot 10^{-4}} \cdot \frac{1}{1000} = \frac{1}{5} \cdot \frac{1}{1000}.$$

Due to the fact that CASPA uses the CUDD library [20] with a default Cudd_Epsilon of $1.0 \cdot$

Nr.	Path	numerical result	theoretical result	mean time (h)
1	Failure_of_GRID Occurrence_of_RC_CB_dies	$2.0e-04$	$\frac{1}{5}\frac{1}{1000}$	2.000e+03
2	Failure_of_GRID OK_of_RC_CB_dies Occurrence_of_RStart_dies	1.998e-04	$\frac{1}{5}\frac{999}{1000}\frac{1}{1000}$	2.000e+03
3	Failure_of_GRID OK_of_RC_CB_dies OK_of_RStart_dies Failure_of_dies_generator	1.990e-04	$\frac{1}{5}\left(\frac{999}{1000}\right)^2\frac{10^{-4}}{10^{-1}+3\cdot10^{-4}}$	2.010e+03
4	Failure_of_Transfo2 Occurrence_of_RO_CB_up_2 Occurrence_of_RC_CB_dies	2.000e-07	$\frac{1}{5}\left(\frac{1}{1000}\right)^2$	2.000e+03
5	Failure_of_CB_dw_1 Occurrence_of_RC_CB_dw_2 Occurrence_of_RC_CB_dies			
6	Failure_of_CB_up_1 Occurrence_of_RC_CB_dw_2 Occurrence_of_RC_CB_dies			

Table 5.1.: Start of the list of most probable paths

10^{-12}, this is the maximum accuracy one can expect from the results. For example the tool calculates

$$P\left(\begin{array}{l}\texttt{Failure\_of\_Transfo2}\rightarrow\\ \texttt{OK\_of\_RO\_CB\_up\_2}\rightarrow\\ \texttt{Failure\_of\_GRID}\rightarrow\\ \texttt{Occurrence\_of\_RC\_CB\_dies}\end{array}\right)=P\left(\begin{array}{l}\texttt{Failure\_of\_Transfo2}\rightarrow\\ \texttt{Occurrence\_of\_RO\_CB\_up\_2}\rightarrow\\ \texttt{OK\_of\_RC\_CB\_dies}\rightarrow\\ \texttt{OK\_of\_RStart\_dies}\rightarrow\\ \texttt{Failure\_of\_dies\_generator}\end{array}\right),$$

but the exact values differ:

$$\frac{1}{5}\cdot\frac{999}{1000}\cdot\frac{10^{-4}}{10^{-1}+4\cdot10^{-4}}\cdot\frac{1}{1000}\neq\frac{1}{5}\cdot\frac{1}{1000}\left(\frac{999}{1000}\right)^2\cdot\frac{10^{-4}}{10^{-1}+3\cdot10^{-4}}$$

As the difference is below $1.0\cdot10^{-12}$, the values are taken as equal. Reducing Cudd_Epsilon would improve accuracy but also slow down the calculations.

5.6.3.1. Path-based measures and discussion

The asymptotic unavailability can be calculated using the following characterisation of a non-critical state:

$$\left(\left(\texttt{line 1 and }\overline{\texttt{sc line 2}}\right)\texttt{ or }\left(\texttt{line 2 and }\overline{\texttt{sc line 1}}\right)\right)\texttt{ and grid }\texttt{) or diesel}.$$

The first part of the term is the condition that line 1 is working and no short-circuit at line 2 is present (and so on). For the measure of a failure state, the term was negated and converted into disjunctive normal form (DNF) to fit the CASPA syntax of a measure.

Table 5.2 shows a comparison of the results calculated on the Xeon-machine. Column one determines the measures of interest, column two shows the results of the algorithms described in Sec. 5.2-5.4 (using the two most probable paths into every failure state and for every failure state the two most probable paths out of it), column three contains the results given in [10] and

	MTBDD-path-based	results in [10]	MTBDD-steady-state
MTTFF (h)	$3.2870 \cdot 10^6$	$3.29 \cdot 10^6$	-
MTTFR (h)	4.9792	4.95	-
Unavailability	$1.5148 \cdot 10^{-6}$	$1.51 \cdot 10^{-6}$	$1.5153 \cdot 10^{-6}$
Solution time (s)	92.22	n.a.	3.5

Table 5.2.: Experimental results

the last column shows the result by steady-state analysis. The results of [10] and our results are quite similar. To verify our path-based algorithms we compared the result for the asymptotic unavailability to the result of steady-state analysis, which fits very well. With the Gauss-Seidel algorithm the result was calculated within only 19 iterations in 0.04 seconds. The rest of the 3.5 seconds given in Tab. 5.2 was spent for the model generation, reachability analysis and so on. This showed that being only interested in the asymptotic unavailability, it is much faster using standard numerical methods and calculating the state-measure, than performing the path-based calculations. However, path-based calculations are very valuable for generating counterexamples in model-checking and for debugging purpose.

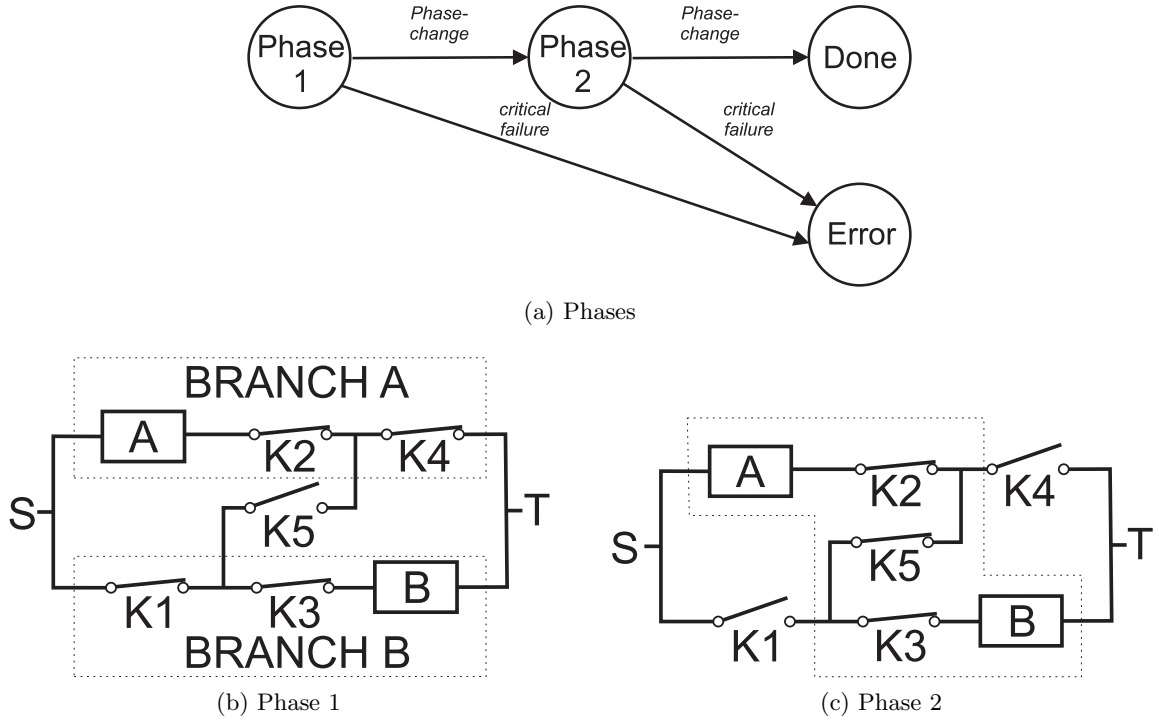
5.7. Case study II: Phased Mission System

The second case study is the Phased Mission System (PMS) from [11], sketched in Fig. 5.7. It consists of two non-repairable components A and B and five switches *K1* to *K5*. The aim is to keep the connection between S and T over the lifetime of the system. The basic functionality of the system is shown in Fig. 5.7a. Its desired functionality is, after starting in phase 1, to make a phase change to phase 2 and a second phase change to done. If the system fails during the mission time, it enters the state fail. The phases are described as follows: In phase 1 two parallel branches, namely *BRANCH A* (A-K2-K4) and *BRANCH B* (K1-K3-B) are used (cf. Fig. 5.7b). On failure of A or B both switches of its branch have to be opened. Phase 2 is the serial connection (A-K2-K5-K3-B) (cf. Fig. 5.7c). The phase change from phase 1 to phase 2 has to be done by first opening *K1* and *K4* and then closing *K5*. After an exponentially distributed time phase 2 is left and the mission is accomplished. The following errors with exponential failure rate $\lambda_{\texttt{err}} = 1.0 \cdot 10^{-4}\,1/\text{h}$ have to be taken into account: Failure of A, B, and of currently closed switches. Furthermore, on-demand failures (probability $p_{\texttt{err}} = \frac{5}{1000}$) can occur when trying to close or open the switches. The rate for the phase change from phase 1 to phase 2 is $\frac{1}{100}\,1/\text{h}$ and for the phase change from phase 2 to done $\frac{1}{50}\,1/\text{h}$. One could also think of a modified model that performs the serial phase 2 before the parallel phase 1. This would be a more natural approach (perform the redundant part after the non-redundant part).

The PMS system can be modelled with CASPA in a natural compositional way: One sequential process (i.e. submodel) for each switch and each non-repairable component. A monitoring process *PMS* keeps track of the errors and the current phase of the system. Fig. 5.8 shows the overall model, which is the synchronised parallel composition of the submodels. The synchronising action `broken` is used to notify all submodels when the system has failed (thus reducing the state space), whereas SYNCSET is an abbreviation for all actions allowing for a bi-directional communication between the PMS process and the other components (e.g. `failK2` if switch K2 has failed, `open2` if switch K2 has to be opened). The submodels will be defined in the following sections. Note that the initial states of the switches are according to phase 1 in Fig. 5.7.

5.7.1. Switch process

In Fig. 5.9 the CASPA model of switch *K2* is shown. The switches are in some sense *memoryless*, that means if an inadvertent open occurs, one can simply close it again and it will work as before.



(a) Phases

(b) Phase 1

(c) Phase 2

Figure 5.7.: PMS scheme and mission

```
(1) System :=
(2)  ( (K1(CLOSED) |[broken]| K2(CLOSED) |[broken]|
(3)     K3(CLOSED) |[broken]| K4(CLOSED) |[broken]|
(4)     K5(OPEN))
(5)     |[broken]|
(6)     (A(OK) |[broken]| B(OK)) )
(7)  |[SYNCSET]|
(8) PMS(PARALLEL, OK, OK)
```

Figure 5.8.: System process

So a switch only has two states, *OPEN* and *CLOSED*. Line 1 of the model description means that the parameterised process *K2* has a parameter range of $\{0, 1\}$. In line 2, the inadvertent opening of a switch is reflected: The action *failureK2* is Markovian with rate *SWITCHFAILRATE*. It is followed by an immediate action *failK2* (which indicates to the PMS process that the switch has failed) and ends up at process *K2* with *state=OPEN*. The on-demand failure is covered in line 3, namely when by the immediate action *open2* the switch is requested to open, a choice between stuck-at-closed (*fail_k2_open*) and successful open (*k2_open*) must be made. Again, the process is continued in an open or closed state respectively. Line 4 covers the case where an open request occurs and the switch is already open. Lines 5 and 6 are used by the *PMS* process to check if all components for phase 2 still work. If at least one component answers with *unavailable*, the mission has failed. Finally, to keep the state space small, line 7 is used to avoid further activity after the system is broken (the broken event is broadcast by the *PMS* process).

5.7.2. Unreliable component

The components A (and similarly B) are defined as shown in Fig. 5.10. In line 2 the component failure is modelled. The immediate action `failA` is used to notify the monitoring process. The immediate action `useA`, given in line 3 and 4 is used by the monitoring process to indicate that

```
(1) K2(state[1]) :=
(2) [state=CLOSED]  -> (failureK2, SWITCHFAILRATE); (*failK2,1*); K2(OPEN)
(3) [state=CLOSED]  -> (*open2,1*); ( (*fail_k2_open, FAILUREPROB*);
                                          (*openfailed,1*); K2(CLOSED)
                                    + (*k2_open, SUCCESSPROB*);
                                          (*open, 1*);      K2(OPEN) )
(4) [state!=CLOSED] -> (*open2,1*);(*open,1*);K2(state)
(5) [state=CLOSED]  -> (*use2,1*); K2(CLOSED)
(6) [state!=CLOSED] -> (*use2,1*); (*unavailable,1*); K2(state)
(7) [*]             -> (*broken, 1*); stop
```

Figure 5.9.: Model of K2

```
(1) A(state[1]) :=
(2)   [state=OK]     -> (failureA, AFAILRATE); (*failA,1*); A(FAILED)
(3)   [state=FAILED] -> (*useA,1*); (*unavailable,1*); A(FAILED)
(4)   [state=OK]     -> (*useA,1*); A(OK)
(5)   [*]            -> (*broken, 1*); stop
```

Figure 5.10.: Model of A

component A is needed. If the component has already failed and cannot be used, the PMS process is notified by the immediate action `unavailable` in line 3. As already mentioned above, line 5 is used to avoid further activity after the system is broken.

5.7.3. The PMS process

The monitoring process of the PMS system is sketched in Fig. 5.11. As seen in line 1 the process has three parameters: The current phase, the state of *BRANCH A* and the state of *BRANCH B*. Line 2 shows an example of a switch failure when both branches are working in parallel. Again, the other switches have to be covered, as well. More interesting, line 3 shows the case when *K1* fails but *BRANCH A* does not work any more. Then the system breaks and *PMS* changes to the *FAIL* state. Line 4 shows the detection of a failure of component *A* during phase 1 and does the reconfiguration: *K2* and *K4* are requested to open (actions *open2* and *open4*). If both switches get stuck at closed (PMS process receives *openfailed* twice), *PMS* stops all switches and components by the immediate *broken* action and sets *phase* and both branches to *FAIL*. If at least one switch can be opened, the system continues in the *PARALLEL* phase but with *brancha=FAIL*. A similar line has to be given for the case of a failure of component *B*. Of course, some other cases, the phase change and the entire serial phase are missing, but it should be enough to get the idea.

5.7.4. Experimental results

For determining the probability that the mission will be accomplished successfully, both CASPA's path-based solver and its uniformisation solver were used (each running on our Xeon system). CASPA directly generates an MTBDD representation of the model specification. The model has 117 reachable states, and path-based analysis leads in 1.1 seconds (including model generation and reachability analysis, $\simeq 0.08$ seconds per path) to three fulfilling paths (given in Tab. 5.3). The third path is the dual to the second one with K1 and K4 interchanged. The probability for a successful completion of the mission is $9.24006 \cdot 10^{-1}$. The same probability can be calculated in CASPA specifying the measure `statemeasure survival PMS(phase=DONE)` and using the uniformisation algorithm. For this approach CASPA reduces the model to only 19 tangible states. The result for a sufficiently large T (e.g. T>41000 hours) is $9.24006 \cdot 10^{-1}$ and the calculation takes 0.91 seconds (again, including model generation, elimination and reachability analysis).

```
(1) PMS(phase[PHASES], brancha[1], branchb[1]) :=
(2)  [phase=PARALLEL, brancha=OK, branchb=OK]  -> (*failK2,1*);
                                                  PMS(PARALLEL,FAIL,branchb)
(3)  [phase=PARALLEL, brancha!=OK, branchb=OK] -> (*failK1,1*); (*broken,1*);
                                                  PMS(FAIL,FAIL,FAIL)
(4)  [phase=PARALLEL] -> (*failA,1*);(*open2,1*); (
                                        (*openfailed,1*);(*open4,1*);(
                                            (*openfailed,1*); (*broken,1*);
                                                              pms(FAIL,FAIL,FAIL)
                                          + (*open,1*); pms(PARALLEL,FAIL,branchb))
                                       +(*open,1*);(*open4,1*);(
                                            (*openfailed,1*);
                                               pms(PARALLEL,FAIL,branchb)
                                           +(*open,1*); pms(PARALLEL,FAIL,branchb)
                                        )
                                    )
```

Figure 5.11.: Administration process (sketch)

Nr.	Path	numerical result	theoretical result	mean time (h)
1	phasechange12 k1_open k4_open k5_close phasechange2done	$9.066497 \cdot 10^{-1}$	$\frac{\frac{1}{100}}{\frac{1}{100}+6\cdot10^{-4}} \cdot \left(\frac{995}{1000}\right)^3 \cdot \frac{\frac{1}{50}}{\frac{1}{50}+5\cdot10^{-4}}$	$1.431 \cdot 10^2$
2	Failure_OF_K1 phasechange12 k4_open k5_close phasechange2done			
3	Failure_OF_K4 phasechange12 k1_open k5_close phasechange2done	$8.678150 \cdot 10^{-3}$	$\frac{10^{-4}}{\frac{1}{100}+6\cdot10^{-4}} \cdot \frac{\frac{1}{100}}{\frac{1}{100}+5\cdot10^{-4}} \cdot \left(\frac{995}{1000}\right)^2 \frac{\frac{1}{50}}{\frac{1}{50}+5\cdot10^{-4}}$	$2.384 \cdot 10^2$

Table 5.3.: Successful paths

6. Conclusion

This work provides advances in different topics of model-based performance and dependability analysis in the context of MTBDD data structure.

The major part of this work was devoted to the symbolic multilevel algorithm. It provides an alternative means to calculating the steady-state distribution of large CTMCs stored by MTBDDs. It has been shown how the nodes in the MTBDD representing the transition matrix can be altered such that the basic shape of the MTBDD stays the same for all aggregated matrices and the aggregates can be stored in a compact way using the structure of the given MTBDD. Although some heuristics for the partition of the state space can be given, the question of finding an optimal partition remains open in this work.

The second part treats the MTBDD-based elimination of vanishing states. A two-phased approach is proposed. The combination of fully symbolic and semi-symbolic elimination phases leads to a practicable and fast way to reduce the state space of large systems containing both Markovian and immediate transitions. When no timeless traps are present, the models can be reduced to CTMCs. The current restriction of the tau-elimination to systems without compositionally vanishing states is nevertheless quite useful in practical modelling: The vanishing states only used for synchronisation purpose can be eliminated and if the elimination fails, the model specification has to be wrong.

Path-based analysis in the MTBDD context has been introduced as a proof of concept. It has proven to be very useful in model debugging and for calculating the MTTFF and MTTFR. Being only interested in the unavailability when the system is in the steady-state, it seems to be faster to analyse the system with the standard numerical methods.

A. Modified CUDD routine

All MTBDD algorithms presented in this work were implemented with the Colorado University Decision Diagram (CUDD) library [20]. Only one additional routine has been added. It is needed for the node characterisation in the symbolic multilevel algorithm (cf. Sec. 4.4.3). The original Cudd_CountLeaves routine does not provide the value of the first non-zero terminal node nor the information whether the terminal zero node has been found.

```
/**Function********************************************************************

  Synopsis    [Counts the number of leaves in a DD.]

  Description [Counts the number of leaves in a DD. Returns the number
  of leaves in the DD rooted at node if successful; CUDD_OUT_OF_MEM
  otherwise.
  Sets the variable zerofound if the zero terminal node
  has been detected,
  sets the variable value to the first non-zero terminal value that is
  found.
  Note: For a correct function value has to be initialised with -1.]

  SideEffects [None]

  SeeAlso     [Cudd_PrintDebug]

******************************************************************************/
int
Cudd_CountLeavesAndCheckZero(
  DdNode * node, int *zerofound, double *value)
{
    int i;

    *zerofound = 0;

    i = ddLeavesIntAndCheckZero(Cudd_Regular(node), zerofound, value);
    ddClearFlag(Cudd_Regular(node));
    return(i);

} /* end of Cudd_CountLeavesAndCheckZero */

/**Function********************************************************************

  Synopsis    [Performs the recursive step of Cudd_CountLeavesAndCheckZero.]

  Description [Performs the recursive step of Cudd_CountLeavesAndCheckZero.
  Returns the number of leaves in the DD rooted at n. And sets a global
  variable if the zero terminal has been detected]
```

```
  SideEffects [None]

  SeeAlso     [Cudd_CountLeavesAndCheckZero]

******************************************************************************/
static int
ddLeavesIntAndCheckZero(
  DdNode * n, int *zerofound, double *value)
{
    int tval, eval;
    double val;

    if (Cudd_IsComplement(n->next)) {
      return(0);
    }

    n->next = Cudd_Not(n->next);
    if (cuddIsConstant(n)) {
        // detect the zero leaf within a MTBDD
        val = Cudd_V(n);
        if(val==0.0) {
          *zerofound = 1;
          if(*value == -1.0) {
            *value = 0.0;
          }
        }
        else {  // for the first nonterminal node not equal to zero,
                // we return the value to the caller
          if ( *value <= 0.0) {
            *value = val;
          }
        }
        return(1);
    }
    tval = ddLeavesIntAndCheckZero(cuddT(n), zerofound, value);
    eval = ddLeavesIntAndCheckZero(Cudd_Regular(cuddE(n)), zerofound, value);
    return(tval + eval);

} /* end of ddLeavesIntAndCheckZero */
```

Bibliography

[1] H.W. Alt. *Lineare Funktionalanalysis: Eine anwendungsorientierte Einführung*. Springer Lehrbuch. Springer, 2002.

[2] S.W.M. Au-Yeung, P.G. Harrison, and W.J. Knottenbelt. A Queueing Network Model of Patient Flow in an Accident and Emergency Department. In *Proc. 20th Ann. European Simulation and Modelling Conf.*, pages 60–67, Toulouse, France, 2006.

[3] J. Azevedo, J. Madeira, E. Martins, and F. Pires. A Shortest Paths Ranking Algorithm. In *Proc. of the Annual Conference AIRO'90 Operational Research Society of Italy*, pages 1001–1011, IEEE, 1990.

[4] J. Bachmann, M. Riedl, J. Schuster, and M. Siegle. An Efficient Symbolic Elimination Algorithm for the Stochastic Process Algebra tool CASPA. In *SOFSEM 2009: Theory and Practice of Computer Science*, pages 485–496, Špindlerův Mlýn, Czech Republic, 2009. Springer, LNCS 5404.

[5] R.I. Bahar, E.A. Frohm, C.M.Ganoa, 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, 1997.

[6] F. Bause. No Way Out ∞ The Timeless Trap. *Petri Net Newsletters*, 37:4–8, 1990.

[7] A.F. Beardon. *Iteration of rational functions: complex analytic dynamical systems*. Graduate texts in mathematics. Springer, 2000.

[8] M. Bernardo and R. Gorrieri. A Tutorial on EMPA: A Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time. *Theoretical Computer Science*, 202(1-2):1–54, 1998.

[9] M. Bernardo and R. Gorrieri. Corrigendum to "A Tutorial on EMPA: A Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time". `http://www.sti.uniurb.it/bernardo/documents/tcs202.corrigendum.pdf`, (last checked September 2011).

[10] M. Bouissou and J.-L. Bon. A new formalism that combines advantages of fault-trees and Markov models: Boolean logic driven Markov processes. *Reliability Engineering and System Safety*, 82:149–163, 2003.

[11] M. Bouissou, Y. Dutuit, and S. Maillard. Reliability analysis of a dynamic phased mission system: Comparison of Two Approaches. *Modern Statistical and Mathematical Methods in Reliability*, pages 87–104, 2005.

[12] M. Bouissou and Y. Lefebvre. A Path-Based Algorithm to Evaluate Asymptotic Unavailability for Large Markov Models. *Proc. of the Annual Reliability and Maintainability Symposium*, pages 32–39, 2002.

[13] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C - 35(8):677 – 691, August 1986.

[14] R. E. Bryant. On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication. *IEEE Transactions on Computers*, 40(2):205–213, February 1991.

[15] P. Buchholz and T. Dayar. Comparison of Multilevel Methods for Kronecker-based Markovian Representations. *Computing*, 73(4):349–371, 2004.

[16] P. Buchholz and T. Dayar. On the convergence of a class of multilevel methods for large sparse markov chains. *SIAM J. Matrix Analysis Applications*, pages 1025–1049, 2007.

[17] G. Chiola, S. Donatelli, and G. Franceschinis. Gspns versus spns: What is the actual role of immediate transitions? In *PNPM*, pages 20–31, 1991.

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

[19] E. M. Clarke, M. Fujita, P. C. McGeer, K. McMillan, J. Yang, and X. Zhao. Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation. In *IWLS: International Workshop on Logic Synthesis*, pages 1–15, Tahoe City, 1993.

[20] CUDD website. `http://vlsi.colorado.edu/~fabio/CUDD/`, (last checked September 2011).

[21] P.R. D'Argenio, H. Hermanns, and J.-P. Katoen. On generative parallel composition. *Electronic Notes in Theoretical Computer Science*, 22, 1999.

[22] C. Eisentraut, H. Hermanns, and L. Zhang. Concurrency and composition in a stochastic world. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, volume 6269 of *Lecture Notes in Computer Science*, pages 21–39. Springer Berlin / Heidelberg, 2010.

[23] C. Eisentraut, H. Hermanns, and L. Zhang. On probabilistic automata in continuous time. In *Proceedings of the 2010 25th Annual IEEE Symposium on Logic in Computer Science*, LICS '10, pages 342–351, Washington, DC, USA, 2010. IEEE Computer Society.

[24] ISO/IEC 14977 : Information technology - Syntactic metalanguage - Extended BNF. `http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip`, (last checked September 2011).

[25] E. Frank. Erweiterung eines MTBDD-basierten Werkzeugs für die Analyse stocchastischer Transitionssysteme. Int. Report, Univ. of Erlangen, Computer Science 7 (in German), 2000.

[26] L. Griebl and J. Schuster. Some notes on the abstraction operation for Multi-Terminal Binary Decision Diagrams. to be published, 2011.

[27] M. Günther. Pfadbasierte Algorithmen zur Zuverlässigkeitsanalyse. Master's thesis, Univ. der Bundeswehr München, Fakultät für Informatik (in German), 2009.

[28] M. Günther, J. Schuster, and M. Siegle. Symbolic calculation of k-shortest paths and related measures with the stochastic process algebra tool CASPA. In *Proc. of the First Workshop on DYnamic Aspects in DEpendability Models for Fault-Tolerant Systems*, pages 13–18, 2010.

[29] H. Hermanns. *Interactive Markov Chains : The Quest for Quantified Quality*. Springer LNCS 2428, 2002.

[30] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi terminal binary decision diagrams to represent and analyse continuous-time Markov chains. *Proc. 3rd Int. Workshop on the Num. Sol. of Markov Chains*, pages 188–207, 1999.

[31] G. Horton and S. Leutenegger. A Multi-Level Solution Algorithm for Steady-State Markov Chains. *ACM Performance Evaluation Review*, 22(1):191–200, May 1994.

[32] Ilse C. F. Ipsen and Steve Kirkland. Convergence Analysis of a PageRank Updating Algorithm by Langville and Meyer. *SIAM J. Matrix Anal. Appl.*, 27:952–967, December 2005.

[33] JINC package. `http://www.jossowski.de`, (last checked September 2011).

[34] D.E. Knuth. *The art of computer programming*, volume 1. Fundamental algorithms. Addison-Wesley, 3rd edition, 1997.

[35] M. Kracht. Strict Compositionality and Literal Movement Grammars. In *Selected papers from the Third International Conference on Logical Aspects of Computational Linguistics*, LACL '98, pages 126–142, London, UK, 2001. Springer-Verlag.

[36] M. Kuntz. *Symbolic Semantics and Verification of Stochastic Process Algebras*. PhD thesis, Technische Fakultät, Universität Erlangen-Nürnberg, 2006.

[37] M. Kuntz, M. Siegle, J. Schuster, and E. Werner. CASPA Handbuch. in german. Please contact M. Siegle `Markus.Siegle@unibw.de` for a digital copy.

[38] M. Kuntz, M. Siegle, and E. Werner. Symbolic Performance and Dependability Evaluation with the Tool CASPA. In *Proc. of First European Performance Engineering Workshop (EPEW), FORTE'04 Workshop*, pages 293–307. Springer LNCS 3236, 2004.

[39] R. Lal and U.N. Bhat. Reduced systems in Markov chains and their applications in queueing theory. *Queueing Systems*, 2(2):147–172, 1987.

[40] I. Marek. Iterative aggregation/disaggregation methods for computing some characteristics of Markov chains. II. Fast convergence. *Applied Numerical Mathematics*, 45:11–28, 2003.

[41] I. Marek and P. Mayer. Convergence analysis of an iterative aggregation/disaggregation method for computing stationary probability vectors of stochastic matrices. *Numerical Linear Algebra with Applications*, 5(4):253–274, 1998.

[42] I. Marek and I. Pultarová. A note on local and global convergence analysis of iterative aggregation-disaggregation methods. *Linear Algebra and its Applications*, 413(2-3):327 – 341, 2006. Special Issue on the 11th Conference of the International Linear Algebra Society, Coimbra, 2004.

[43] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Wiley Series in Parallel Computing, 1995.

[44] M. Ajmone Marsan and G. Conte. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 2:93–122, 1984.

[45] M. Ajmone Marsan, S. Donatelli, and F. Neri. GSPN models of Markovian multiserver multiqueue systems. *Performance Evaluation*, 11:227–240, May 1990.

[46] R. Mehmood. Serial disk-based analysis of large stochastic models. In C. Baier, B. Haverkort, H. Hermanns, J-P. Katoen, and M. Siegle, editors, *Validation of Stochastic Systems: A Guide to Current Research*, volume 2925 of *Lecture Notes in Computer Science (Tutorial Volume)*, pages 230–255. Springer, 2004.

[47] D. Parker. *Implementation of symbolic model checking for probabilistic systems*. PhD thesis, School of Computer Science, Faculty of Science, University of Birmingham, 2002.

[48] PRISM website. `http://www.prismmodelchecker.org/`, (last checked September 2011).

[49] I. Pultarová. Necessary and sufficient local convergence condition of one class of iterative aggregation–disaggregation methods. *Numerical Linear Algebra with Applications*, 15(4):339–354, 2008.

[50] I. Pultarová and I. Marek. Physiology and pathology of iterative aggregation-disaggregation methods. *submitted (2011). Available at* `http://mat.fsv.cvut.cz/nales/preprints`, (last checked September 2011).

[51] G. Rubino and B. Sericola. Sojourn Times in Finite Markov Processes. *Journal of Applied Probability*, 26(4):744–756, Dec 1989.

[52] M. Růžička. *Nichtlineare Funktionalanalysis: Eine Einführung*. Springer Lehrbuch. Springer, 2004.

[53] W. Schmid. *Berechnung kürzester Wege in Straßennetzen mit Wegeverboten*. PhD thesis, Universität Stuttgart, Fakultät für Bauingenieur- und Vermessungswesen, 2000.

[54] J. Schuster and M. Siegle. A Multilevel Algorithm based on Binary Decision Diagrams. In K. Al-Begain, A. Heindl, and M. Telek, editors, *14th Int. Conf. on Analytical and Stochastic Modelling Techniques and Applications (ASMTA'07)*, pages 129–136, June 2007.

[55] J. Schuster and M. Siegle. A symbolic multilevel method with sparse submatrix representation for memory-speed tradeoff. In *14. GI/ITG Conf. Measurement, Modelling and Evaluation of Comp. and Communic. Systems (MMB08)*, pages 191–205. VDE Verlag, 2008.

[56] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1995.

[57] M. Siegle. Advances in model representation. In L. de Alfaro and S. Gilmore, editors, *Process Algebra and Probabilistic Methods, Joint Int. Workshop PAPM-PROBMIV 2001*, pages 1–22. Springer, LNCS 2165, September 2001.

[58] M. Siegle. *Behaviour analysis of communication systems: Compositional modelling, compact representation and analysis of performability properties*. Shaker-Verlag, 2002.

[59] W.J. Stewart. *Introduction to the numerical solution of Markov chains*. Princeton University Press, 1994.

[60] R. Strecker. Über entropische Gruppoide. *Mathematische Nachrichten*, 64(1):363–371, 1974.

[61] V. Volenec. Extension of Toyoda's theorem on entropic groupoids. *Mathematische Nachrichten*, 102(1):183–188, 1981.

[62] D.M. Young. *Iterative Solution of Large Linear Systems*. Academic Press, 1971.

Index